IPython

An enhanced Interactive Python

User Manual, v. 0.8.2

Fernando Pérez*

November 29, 2007

Contents

*Department of Applied Mathematics, University of Colorado at Boulder. <Fernando.Perez@colorado.edu>

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.

IPython is a free software project (released under the BSD license) which tries to:

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.

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.

1.1 Main features

- 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, 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.

- 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 !! 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 conveninence 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.
- 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.

1.2 Portability and Python requirements

Python requirements: IPython requires with Python version 2.3 or newer. If you are still using Python 2.2 and can not upgrade, the last version of IPython which worked with Python 2.2 was 0.6.15, so you will have to use that.

IPython is developed under **Linux**, but it should work in any reasonable Unix-type system (tested OK under Solaris and the *BSD family, for which a port exists thanks to Dryice Liu).

Mac OS X: it works, apparently without any problems (thanks to Jim Boyle at Lawrence Livermore for the information). Thanks to Andrea Riciputi, Fink support is available.

CygWin: it works mostly OK, though some users have reported problems with prompt coloring. No satisfactory solution to this has been found so far, you may want to disable colors permanently in the `ipythonrc` configuration file if you experience problems. If you have proper color support under cygwin, please post to the IPython mailing list so this issue can be resolved for all users.

Windows: it works well under Windows XP/2k, and I suspect NT should behave similarly. Section 2.3 describes installation details for Windows, including some additional tools needed on this platform.

Windows 9x support is present, and has been reported to work fine (at least on WinME).

Note, that I have very little access to and experience with Windows development. However, an excellent group of Win32 users (led by Ville Vainio), consistently contribute bugfixes and platform-specific enhancements, so they more than make up for my deficiencies on that front. In fact, Win32 users report using IPython as a system shell (see Sec. 12 for details), as it offers a level of control and features which the default `cmd.exe` doesn't provide.

1.3 Location

IPython is generously hosted at `http://ipython.scipy.org` by the Enthought, Inc and the SciPy project. This site offers downloads, subversion access, mailing lists and a bug tracking system. I am very grateful to Enthought (`http://www.enthought.com`) and all of the SciPy team for their contribution.

2 Installation

2.1 Instant instructions

If you are of the impatient kind, under Linux/Unix simply untar/unzip the download, then install with 'python setup.py install'. Under Windows, double-click on the provided .exe binary installer.

Then, take a look at Sections 3 for configuring things optimally and 4 for quick tips on efficient use of IPython. You can later refer to the rest of the manual for all the gory details.

See the notes in sec. 2.4 for upgrading IPython versions.

2.2 Detailed Unix instructions (Linux, Mac OS X, etc.)

For RPM based systems, simply install the supplied package in the usual manner. If you download the tar archive, the process is:

1. Unzip/untar the ipython-XXX.tar.gz file wherever you want (XXX is the version number). It will make a directory called ipython-XXX. Change into that directory where you will find the files README and setup.py. Once you've completed the installation, you can safely remove this directory.

2. If you are installing over a previous installation of version 0.2.0 or earlier, first remove your $HOME/.ipython directory, since the configuration file format has changed somewhat (the '=' were removed from all option specifications). Or you can call ipython with the -upgrade option and it will do this automatically for you.

3. IPython uses distutils, so you can install it by simply typing at the system prompt (don't type the $)
   ```
   $ python setup.py install
   ```
   Note that this assumes you have root access to your machine. If you don't have root access or don't want IPython to go in the default python directories, you'll need to use the --home option (or --prefix). For example:
   ```
   $ python setup.py install --home $HOME/local
   ```
   will install IPython into $HOME/local and its subdirectories (creating them if necessary).
   You can type
   ```
   $ python setup.py --help
   ```
   for more details.
   Note that if you change the default location for --home at installation, IPython may end up installed at a location which is not part of your $PYTHONPATH environment variable. In this case, you'll need to configure this variable to include the actual directory where the IPython/ directory ended (typically the value you give to --home plus /lib/python).

2.2.1 Mac OSX information

Under OSX, there is a choice you need to make. Apple ships its own build of Python, which lives in the core OSX filesystem hierarchy. You can also manually install a separate Python, either

purely by hand (typically in `/usr/local`) or by using Fink, which puts everything under `/sw`. Which route to follow is a matter of personal preference, as I've seen users who favor each of the approaches. Here I will simply list the known installation issues under OSX, along with their solutions.

This page: http://geosci.uchicago.edu/~tobis/pylab.html contains information on this topic, with additional details on how to make IPython and matplotlib play nicely under OSX.

GUI problems

The following instructions apply to an install of IPython under OSX from unpacking the `.tar.gz` distribution and installing it for the default Python interpreter shipped by Apple. If you are using a fink install, fink will take care of these details for you, by installing IPython against fink's Python.

IPython offers various forms of support for interacting with graphical applications from the command line, from simple Tk apps (which are in principle always supported by Python) to interactive control of WX, Qt and GTK apps. Under OSX, however, this requires that ipython is installed by calling the special `pythonw` script at installation time, which takes care of coordinating things with Apple's graphical environment.

So when installing under OSX, it is best to use the following command:

```
$ sudo pythonw setup.py install --install-scripts=/usr/local/bin
```

or

```
$ sudo pythonw setup.py install --install-scripts=/usr/bin
```

depending on where you like to keep hand-installed executables.

The resulting script will have an appropriate shebang line (the first line in the script whic begins with `#!...`) such that the ipython interpreter can interact with the OS X GUI. If the installed version does not work and has a shebang line that points to, for example, just `/usr/bin/python`, then you might have a stale, cached version in your `build/scripts-<python-version>` directory. Delete that directory and rerun the `setup.py`.

It is also a good idea to use the special flag `--install-scripts` as indicated above, to ensure that the ipython scripts end up in a location which is part of your `$PATH`. Otherwise Apple's Python will put the scripts in an internal directory not available by default at the command line (if you use `/usr/local/bin`, you need to make sure this is in your `$PATH`, which may not be true by default).

Readline problems

By default, the Python version shipped by Apple does *not* include the readline library, so central to IPython's behavior. If you install IPython against Apple's Python, you will not have arrow keys, tab completion, etc. For Mac OSX 10.3 (Panther), you can find a prebuilt readline library here: http://pythonmac.org/packages/readline-5.0-py2.3-macosx10.3.zip

If you are using OSX 10.4 (Tiger), after installing this package you need to either:

1. move `readline.so` from `/Library/Python/2.3` to `/Library/Python/2.3/site-packages`, or

2. install http://pythonmac.org/packages/TigerPython23Compat.pkg.zip

Users installing against Fink's Python or a properly hand-built one should not have this problem.

DarwinPorts

I report here a message from an OSX user, who suggests an alternative means of using IPython under this operating system with good results. Please let me know of any updates that may be useful for this section. His message is reproduced verbatim below:

> From: Markus Banfi <markus.banfi-AT-mospheira.net>
>
> As a MacOS X (10.4.2) user I prefer to install software using DawinPorts instead of Fink. I had no problems installing ipython with DarwinPorts. It's just:
>
> `sudo port install py-ipython`
>
> It automatically resolved all dependencies (python24, readline, py-readline). So far I did not encounter any problems with the DarwinPorts port of ipython.

2.3 Windows instructions

Some of IPython's very useful features are:

- Integrated readline support (Tab-based file, object and attribute completion, input history across sessions, editable command line, etc.)
- Coloring of prompts, code and tracebacks.

These, by default, are only available under Unix-like operating systems. However, thanks to Gary Bishop's work, Windows XP/2k users can also benefit from them. His readline library originally implemented both GNU readline functionality and color support, so that IPython under Windows XP/2k can be as friendly and powerful as under Unix-like environments.

This library, now named `PyReadline`, has been absorbed by the IPython team (Jörgen Stenarson, in particular), and it continues to be developed with new features, as well as being distributed directly from the IPython site.

The `PyReadline` extension requires `CTypes` and the windows IPython installer needs `PyWin32`, so in all you need:

1. `PyWin32` from http://sourceforge.net/projects/pywin32.
2. `PyReadline` for Windows from http://ipython.scipy.org/moin/PyReadline/Intro. That page contains further details on using and configuring the system to your liking.
3. Finally, *only* if you are using Python 2.3 or 2.4, you need `CTypes` from http://starship.python.net/crew/theller/ctypes (you *must* use version 0.9.1 or newer). This package is included in Python 2.5, so you don't need to manually get it if your Python version is 2.5 or newer.

Warning about a broken readline-like library: several users have reported problems stemming from using the pseudo-readline library at http://newcenturycomputers.net/projects/readline.html. This is a broken library which, while called readline, only implements an incomplete subset of the readline API. Since it is still called readline, it fools IPython's detection mechanisms and causes unpredictable crashes later. If you wish to use IPython under Windows, you must NOT use this library, which for all purposes is (at least as of version 1.6) terminally broken.

2.3.1 Installation procedure

Once you have the above installed, from the IPython download directory grab the `ipython-XXX.win32.exe` file, where `XXX` represents the version number. This is a regular windows executable installer, which you can simply double-click to install. It will add an entry for IPython to your Start Menu, as well as registering IPython in the Windows list of applications, so you can later uninstall it from the Control Panel.

IPython tries to install the configuration information in a directory named `.ipython` (`_ipython` under Windows) located in your 'home' directory. IPython sets this directory by looking for a `HOME` environment variable; if such a variable does not exist, it uses `HOMEDRIVE\HOMEPATH` (these are always defined by Windows). This typically gives something like `C:\Documents and Settings\YourUserName`, but your local details may vary. In this directory you will find all the files that configure IPython's defaults, and you can put there your profiles and extensions. This directory is automatically added by IPython to `sys.path`, so anything you place there can be found by `import` statements.

Upgrading For an IPython upgrade, you should first uninstall the previous version. This will ensure that all files and directories (such as the documentation) which carry embedded version strings in their names are properly removed.

Manual installation under Win32 In case the automatic installer does not work for some reason, you can download the `ipython-XXX.tar.gz` file, which contains the full IPython source distribution (the popular WinZip can read `.tar.gz` files). After uncompressing the archive, you can install it at a command terminal just like any other Python module, by using `'python setup.py install'`.

After the installation, run the supplied `win32_manual_post_install.py` script, which creates the necessary Start Menu shortcuts for you.

2.4 Upgrading from a previous version

If you are upgrading from a previous version of IPython, after doing the routine installation described above, you should call IPython with the `-upgrade` option the first time you run your new copy. This will automatically update your configuration directory while preserving copies of your old files. You can then later merge back any personal customizations you may have made into the new files. It is a good idea to do this as there may be new options available in the new configuration files which you will not have.

Under Windows, if you don't know how to call python scripts with arguments from a command line, simply delete the old config directory and IPython will make a new one. Win2k and WinXP users will find it in `C:\Documents and Settings\YourUserName\_ipython`, and Win 9x users under `C:\Program Files\IPython\_ipython`.

3 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 in Sec. 7.1. This file is very commented and has many variables you can change to suit your taste, you can find more details in Sec. 7. Here we discuss the basic things you will want to make sure things are working properly from the beginning.

3.1 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-docs-2.3.4/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.dat acks.html dist/ ext/ index.html lib/ modindex.html
stdabout.dat tut/ about.html api/ doc/ icons/ inst/ mac/ ref/ style.css
```

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.

3.2 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).

3.3 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 sec.3.4 for more details on using IPython with (X)Emacs.
- A Windows (XP/2k) command prompt *with Gary Bishop's support extensions*. Gary's extensions are discussed in Sec. 2.3.
- 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.

3.3.1 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.

3.3.2 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 in Sec. 6.4. 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.

3.4 (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.

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).

4 Quick tips

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

- The TAB key. 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 sec. 6.5 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 sec. 6.4 for more.

- 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 sec. 6.2 for more on this and other magic commands, or type the name of any magic command and ? to get details on it. See also sec. 6.9 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.

- Use the Python debugger, `pdb`[1]. 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`.

[1] Thanks to Christian Hart and Matthew Arnison for the suggestions leading to IPython's improved debugger and profiler support.

Note also that '`%run -d`' activates `pdb` and automatically sets initial breakpoints for you to step through your code, watch variables, etc. See Sec. 6.12 for 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 Sec. 6.12 for more.

- Put a '`;`' at the end of a line to supress 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.

- 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 sec. 6.11 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.

- 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.

- 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 sec. 6.7 for more.

- 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. See sec. 7.3 for more.

- 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 sec. 9 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 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 sec. 14 for more.
- Run your doctests from within IPython for development and debugging. The special `%doctest_mode` command toggles a mode where the prompt, output and exceptions display matches as closely as possible that of the default Python interpreter. In addition, this mode allows you to directly paste in code that contains leading '>>>' prompts, even if they have extra leading whitespace (as is common in doctest files). This combined with the '`%history -t`n' call to see your translated history (with these extra prompts removed and no line numbers) allows for an easy doctest workflow, where you can go from doctest to interactive execution to pasting into valid Python code as needed.

4.1 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.

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 conti
```

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

4.2 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!

5 Command-line use

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`.

5.1 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.

5.2 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.net http://matplotlib.sourceforge.net), 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 Sec. 15 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, and set sys.argv to `['c']`. 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.

-[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>` (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 [\#]:'`. Sec. 7.2 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 <string>:` 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.

-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).

6 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.

6.1 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, issuing many of the commands discussed below (especially magics which affect the filesystem) with '`\`' in them will cause strange errors.

A partial solution is to use instead the '`/`' character as a path separator, which Windows recognizes in *most* situations. 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
```

There is no sensible thing IPython can do to truly work around this flaw in Windows[2].

6.2 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)
```

[2] If anyone comes up with a *clean* solution which works consistently and does not negatively impact other platforms at all, I'll gladly accept a patch.

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 for information on the `'?'` system) to get information about any particular magic function you are interested in.

6.2.1 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'.

`%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.

%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.

%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.

`%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.

`%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.

`%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)

%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.

6.3 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 in sec. 3.1, you need to properly configure your environment variable `PYTHONDOCS` for this feature to work correctly.

6.4 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??`.

6.5 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.

6.5.1 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.

6.5.2 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.

6.5.3 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.

6.5.4 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`).

6.5.5 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.

6.6 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 sec. 5.2)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.

6.7 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.

6.7.1 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.

See Sec. 6.2 for details on the magics `%sc` and `%sx`, or use IPython's own help (`sc?` and `sx?`) for further details.

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]
```

6.8 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 for further details).

6.9 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.

6.10 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.

6.11 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 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.

6.12 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.

6.13 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.

6.14 Automatic parentheses and quotes

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

6.14.1 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)
```

6.14.2 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
```

7 Customization

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`). Sec 5.2 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.

7.1 Sample ipythonrc file

The default rcfile, called `ipythonrc` and supplied in your `IPYTHONDIR` directory contains lots of comments on all of these options. We reproduce it here for reference:

```
# -*- Mode: Shell-Script -*-  Not really, but shows comments correctly
# $Id: ipythonrc 2156 2007-03-19 02:32:19Z fperez $

#***************************************************************************
#
# Configuration file for IPython -- ipythonrc format
#
# ===========================================================
# Deprecation note: you should look into modifying ipy_user_conf.py (located
# in ~/.ipython or ~/_ipython, depending on your platform) instead, it's a
# more flexible and robust (and better supported!) configuration
# method.
# ===========================================================
#
# The format of this file is simply one of 'key value' lines.
# Lines containing only whitespace at the beginning and then a # are ignored
# as comments. But comments can NOT be put on lines with data.
```

```
# The meaning and use of each key are explained below.

#---------------------------------------------------------------------------
# Section: included files

# Put one or more *config* files (with the syntax of this file) you want to
# include. For keys with a unique value the outermost file has precedence. For
# keys with multiple values, they all get assembled into a list which then
# gets loaded by IPython.

# In this file, all lists of things should simply be space-separated.

# This allows you to build hierarchies of files which recursively load
# lower-level services. If this is your main ~/.ipython/ipythonrc file, you
# should only keep here basic things you always want available. Then you can
# include it in every other special-purpose config file you create.
include

#---------------------------------------------------------------------------
# Section: startup setup

# These are mostly things which parallel a command line option of the same
# name.

# Keys in this section should only appear once. If any key from this section
# is encountered more than once, the last value remains, all earlier ones get
# discarded.


# Automatic calling of callable objects.  If set to 1 or 2, callable objects
# are automatically called when invoked at the command line, even if you don't
# type parentheses.  IPython adds the parentheses for you.  For example:

#In [1]: str 45
#------> str(45)
#Out[1]: '45'

# IPython reprints your line with '---->' indicating that it added
# parentheses.  While this option is very convenient for interactive use, it
# may occasionally cause problems with objects which have side-effects if
# called unexpectedly.

# The valid values for autocall are:

# autocall 0 -> disabled (you can toggle it at runtime with the %autocall
    ...magic)

# autocall 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'

autocall 1

# Auto-edit syntax errors.  When you use the %edit magic in ipython to edit
# source code (see the 'editor' variable below), it is possible that you save
# a file with syntax errors in it.  If this variable is true, IPython will ask
# you whether to re-open the editor immediately to correct such an error.

autoedit_syntax 0

# Auto-indent. 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 indent/unindenting more
# convenient (M-i indents, M-u unindents):

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

# The feature is potentially a bit dangerous, because it can cause problems
# with pasting of indented code (the pasted code gets re-indented on each
# line).  But it's a huge time-saver when working interactively.  The magic
# function %autoindent allows you to toggle it on/off at runtime.

autoindent 1

# Auto-magic. This gives you access to all the magic functions without having
# to prepend them with an % sign. If you define a variable with the same name
# as a magic function (say who=1), you will need to access the magic function
# with % (%who in this example). However, if later you delete your variable
# (del who), you'll recover the automagic calling form.
```

```
# Considering that many magic functions provide a lot of shell-like
# functionality, automagic gives you something close to a full Python+system
# shell environment (and you can extend it further if you want).

automagic 1

# Size of the output cache. After this many entries are stored, the cache will
# get flushed. Depending on the size of your intermediate calculations, you
# may have memory problems if you make it too big, since keeping things in the
# cache prevents Python from reclaiming the memory for old results. Experiment
# with a value that works well for you.

# If you choose cache_size 0 IPython will revert to python's regular >>>
# unnumbered prompt. You will still have _, __ and ___ for your last three
# results, but that will be it.  No dynamic _1, _2, etc. will be created. If
# you are running on a slow machine or with very limited memory, this may
# help.

cache_size 1000

# Classic mode: Setting 'classic 1' you lose many of IPython niceties,
# but that's your choice! Classic 1 -> same as IPython -classic.
# Note that this is _not_ the normal python interpreter, it's simply
# IPython emulating most of the classic interpreter's behavior.
classic 0

# colors - Coloring option for prompts and traceback printouts.

# Currently available schemes: NoColor, Linux, LightBG.

# This option allows coloring the prompts and traceback printouts. This
# requires a terminal which can properly handle color escape sequences. If you
# are having problems with this, use the NoColor scheme (uses no color escapes
# at all).

# The Linux option works well in linux console type environments: dark
# background with light fonts.

# LightBG is similar to Linux but swaps dark/light colors to be more readable
# in light background terminals.

# keep uncommented only the one you want:
colors Linux
#colors LightBG
#colors NoColor

########################
# Note to Windows users
#
# Color and readline support is avaialble to Windows users via Gary Bishop's
# readline library.  You can find Gary's tools at
# http://sourceforge.net/projects/uncpythontools.
# Note that his readline module requires in turn the ctypes library, available
# at http://starship.python.net/crew/theller/ctypes.
```

```
########################

# 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. This information is passed through a
# pager (it defaults to 'less' if $PAGER is not set).

# If your pager has problems, try to setting it to properly handle escapes
# (see the less manpage for detail), or disable this option.  The magic
# function %color_info allows you to toggle this interactively for testing.

color_info 1

# confirm_exit: set to 1 if you want IPython to confirm when you try to exit
# with an EOF (Control-d in Unix, Control-Z/Enter in Windows). Note that using
# the magic functions %Exit or %Quit you can force a direct exit, bypassing
# any confirmation.

confirm_exit 1

# Use deep_reload() as a substitute for reload() by default. deep_reload() is
# still available as dreload() and appears as a builtin.

deep_reload 0

# Which editor to use with the %edit command. If you leave this at 0, IPython
# will honor your EDITOR environment variable. 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.

# For Emacs users, setting up your Emacs server properly as described in the
# manual is a good idea. An alternative is to use jed, a very light editor
# with much of the feel of Emacs (though not as powerful for heavy-duty work).

editor 0

# log 1 -> same as ipython -log. This automatically logs to ./ipython.log
log 0

# Same as ipython -Logfile YourLogfileName.
# Don't use with log 1 (use one or the other)
logfile ''

# banner 0 -> same as ipython -nobanner
banner 1

# messages 0 -> same as ipython -nomessages
messages 1

# 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.
pdb 0
```

```
# Enable the pprint module for printing. pprint tends to give a more readable
# display (than print) for complex nested data structures.
pprint 1

# Prompt strings

# Most bash-like escapes can be used to customize IPython's prompts, as well
    ...as
# a few additional ones which are IPython-specific.  All valid prompt escapes
# are described in detail in the Customization section of the IPython HTML/PDF
# manual.

# Use \# to represent the current prompt number, and quote them to protect
# spaces.
prompt_in1 'In [\#]: '

# \D is replaced by as many dots as there are digits in the
# current value of \#.
prompt_in2 '   .\D.: '

prompt_out 'Out[\#]: '

# Select whether to left-pad the output prompts to match the length of the
# input ones.  This allows you for example to use a simple '>' as an output
# prompt, and yet have the output line up with the input.  If set to false,
# the output prompts will be unpadded (flush left).
prompts_pad_left 1

# Pylab support: when ipython is started with the -pylab switch, by default it
# executes 'from matplotlib.pylab import *'.  Set this variable to false if
    ...you
# want to disable this behavior.

# For details on pylab, see the matplotlib website:
# http://matplotlib.sf.net
pylab_import_all 1


# quick 1 -> same as ipython -quick
quick 0

# Use the readline library (1) or not (0). Most users will want this on, but
# if you experience strange problems with line management (mainly when using
# IPython inside Emacs buffers) you may try disabling it. Not having it on
# prevents you from getting command history with the arrow keys, searching and
# name completion using TAB.

readline 1

# Screen Length: 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 paged with the less command 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. If for some reason this isn't
# working well (it needs curses support), specify it yourself. Otherwise don't
# change the default.

screen_length 0

# Prompt separators for input and output.
# Use \n for newline explicitly, without quotes.
# Use 0 (like at the cmd line) to turn off a given separator.

# The structure of prompt printing is:
# (SeparateIn)Input....
# (SeparateOut)Output...
# (SeparateOut2),   # that is, no newline is printed after Out2
# By choosing these you can organize your output any way you want.

separate_in \n
separate_out 0
separate_out2 0

# 'nosep 1' is a shorthand for '-SeparateIn 0 -SeparateOut 0 -SeparateOut2 0'.
# Simply removes all input/output separators, overriding the choices above.
nosep 0

# Wildcard searches - IPython has a system for searching names using
# shell-like wildcards; type %psearch? for details.  This variables sets
# whether by default such searches should be case sensitive or not.  You can
# always override the default at the system command line or the IPython
# prompt.

wildcards_case_sensitive 1

# Object information: at what level of detail to display the string form of an
# object.  If set to 0, ipython will compute the string form of any object X,
# by calling str(X), when X? is typed.  If set to 1, str(X) will only be
# computed when X?? is given, and if set to 2 or higher, it will never be
# computed (there is no X??? level of detail).  This is mostly of use to
# people who frequently manipulate objects whose string representation is
# extremely expensive to compute.

object_info_string_level 0

# xmode - Exception reporting mode.

# 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).

#xmode Plain
xmode Context
#xmode Verbose

# multi_line_specials: if true, allow magics, aliases and shell escapes (via
# !cmd) to be used in multi-line input (like for loops).  For example, if you
# have this active, the following is valid in IPython:
#
#In [17]: for i in range(3):
#    ....:     mkdir $i
#    ....:     !touch $i/hello
#    ....:     ls -l $i

multi_line_specials 1


# System calls: When IPython makes system calls (e.g. via special syntax like
# !cmd or !!cmd, or magics like %sc or %sx), it can print the command it is
# executing to standard output, prefixed by a header string.

system_header "IPython system call: "

system_verbose 1

# wxversion: request a specific wxPython version (used for -wthread)

# Set this to the value of wxPython you want to use, but note that this
# feature requires you to have the wxversion Python module to work.  If you
# don't have the wxversion module (try 'import wxversion' at the prompt to
# check) or simply want to leave the system to pick up the default, leave this
# variable at 0.

wxversion 0

#---------------------------------------------------------------------------
# Section: Readline configuration (readline is not available for MS-Windows)

# This is done via the following options:

# (i) 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.

# The TAB key can be used to complete names at the command line in one of two
# ways: 'complete' and 'menu-complete'. The difference is that 'complete' only
# completes as much as possible while 'menu-complete' cycles through all
```

```
# possible completions. Leave the one you prefer uncommented.

readline_parse_and_bind tab: complete
#readline_parse_and_bind tab: menu-complete

# This binds Control-l to printing the list of all possible completions when
# there is more than one (what 'complete' does when hitting TAB twice, or at
# the first TAB if show-all-if-ambiguous is on)
readline_parse_and_bind "\C-l": possible-completions

# This forces readline to automatically print the above list when tab
# completion is set to 'complete'. You can still get this list manually by
# using the key bound to 'possible-completions' (Control-l by default) or by
# hitting TAB twice. Turning this on makes the printing happen at the first
# TAB.
readline_parse_and_bind set show-all-if-ambiguous on

# If you have TAB set to complete names, you can rebind any key (Control-o by
# default) to insert a true TAB character.
readline_parse_and_bind "\C-o": tab-insert

# These commands allow you to indent/unindent easily, with the 4-space
# convention of the Python coding standards.  Since IPython's internal
# auto-indent system also uses 4 spaces, you should not change the number of
# spaces in the code below.
readline_parse_and_bind "\M-i": "    "
readline_parse_and_bind "\M-o": "\d\d\d\d"
readline_parse_and_bind "\M-I": "\d\d\d\d"

# Bindings for incremental searches in the history. These searches use the
# string typed so far on the command line and search anything in the previous
# input history containing them.
readline_parse_and_bind "\C-r": reverse-search-history
readline_parse_and_bind "\C-s": forward-search-history

# Bindings for completing the current line in the history of previous
# commands. This allows you to recall any previous command by typing its first
# few letters and hitting Control-p, bypassing all intermediate commands which
# may be in the history (much faster than hitting up-arrow 50 times!)
readline_parse_and_bind "\C-p": history-search-backward
readline_parse_and_bind "\C-n": history-search-forward

# I also like to have the same functionality on the plain arrow keys. If you'd
# rather have the arrows use all the history (and not just match what you've
# typed so far), comment out or delete the next two lines.
readline_parse_and_bind "\e[A": history-search-backward
readline_parse_and_bind "\e[B": history-search-forward

# These are typically on by default under *nix, but not win32.
readline_parse_and_bind "\C-k": kill-line
readline_parse_and_bind "\C-u": unix-line-discard

# (ii) 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.

readline_remove_delims -/~

# (iii) readline_merge_completions: whether to merge the result of all
# possible completions or not.  If true, IPython will complete filenames,
# python names and aliases and return all possible completions.  If you set it
# to false, each completer is used at a time, and only if it doesn't return
# any completions is the next one used.

# The default order is: [python_matches, file_matches, alias_matches]

readline_merge_completions 1

# (iv) readline_omit__names: normally hitting <tab> after a '.' in a name
# will complete all attributes of an object, including all the special methods
# whose names start with single or double underscores (like __getitem__ or
# __class__).

# This variable allows you to control this completion behavior:

# readline_omit__names 1 -> completion will omit showing any names starting
# with two __, but it will still show names starting with one _.

# readline_omit__names 2 -> completion will omit all names beginning with one
# _ (which obviously means filtering out the double __ ones).

# 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.

readline_omit__names 0

#-----------------------------------------------------------------------------
# Section: modules to be loaded with 'import ...'

# List, separated by spaces, the names of the modules you want to import

# Example:
# import_mod sys os
# will produce internally the statements
# import sys
# import os

# Each import is executed in its own try/except block, so if one module
# fails to load the others will still be ok.

import_mod

#-----------------------------------------------------------------------------
# Section: modules to import some functions from: 'from ... import ...'
```

```
# List, one per line, the modules for which you want only to import some
# functions. Give the module name first and then the name of functions to be
# imported from that module.

# Example:

# import_some IPython.genutils timing timings
# will produce internally the statement
# from IPython.genutils import timing, timings

# timing() and timings() are two IPython utilities for timing the execution of
# your own functions, which you may find useful.  Just commment out the above
# line if you want to test them.

# If you have more than one modules_some line, each gets its own try/except
# block (like modules, see above).

import_some

#-----------------------------------------------------------------------
# Section: modules to import all from : 'from ... import *'

# List (same syntax as import_mod above) those modules for which you want to
# import all functions. Remember, this is a potentially dangerous thing to do,
# since it is very easy to overwrite names of things you need. Use with
# caution.

# Example:
# import_all sys os
# will produce internally the statements
# from sys import *
# from os import *

# As before, each will be called in a separate try/except block.

import_all

#-----------------------------------------------------------------------
# Section: Python code to execute.

# Put here code to be explicitly executed (keep it simple!)
# Put one line of python code per line. All whitespace is removed (this is a
# feature, not a bug), so don't get fancy building loops here.
# This is just for quick convenient creation of things you want available.

# Example:
# execute x = 1
# execute print 'hello world'; y = z = 'a'
# will produce internally
# x = 1
# print 'hello world'; y = z = 'a'
# and each *line* (not each statement, we don't do python syntax parsing) is
# executed in its own try/except block.
```

```
execute

# Note for the adventurous: you can use this to define your own names for the
# magic functions, by playing some namespace tricks:

# execute __IPYTHON__.magic_pf = __IPYTHON__.magic_profile

# defines %pf as a new name for %profile.

#----------------------------------------------------------------------------
# Section: Pyhton files to load and execute.

# Put here the full names of files you want executed with execfile(file).  If
# you want complicated initialization, just write whatever you want in a
# regular python file and load it from here.

# Filenames defined here (which *must* include the extension) are searched for
# through all of sys.path. Since IPython adds your .ipython directory to
# sys.path, they can also be placed in your .ipython dir and will be
# found. Otherwise (if you want to execute things not in .ipyton nor in
# sys.path) give a full path (you can use ~, it gets expanded)

# Example:
# execfile file1.py ~/file2.py
# will generate
# execfile('file1.py')
# execfile('_path_to_your_home/file2.py')

# As before, each file gets its own try/except block.

execfile

# If you are feeling adventurous, you can even add functionality to IPython
# through here. IPython works through a global variable called __ip which
# exists at the time when these files are read. If you know what you are doing
# (read the source) you can add functions to __ip in files loaded here.

# The file example-magic.py contains a simple but correct example. Try it:

# execfile example-magic.py

# Look at the examples in IPython/iplib.py for more details on how these magic
# functions need to process their arguments.

#----------------------------------------------------------------------------
# Section: aliases for system shell commands

# Here you can define your own names for system commands. The syntax is
# similar to that of the builtin %alias function:

# alias alias_name command_string

# The resulting aliases are auto-generated magic functions (hence usable as
```

```
# %alias_name)

# For example:

# alias myls ls -la

# will define 'myls' as an alias for executing the system command 'ls -la'.
# This allows you to customize IPython's environment to have the same aliases
# you are accustomed to from your own shell.

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

# alias parts echo first %s second %s

# will give you in IPython:
# >>> %parts A B
# first A second B

# Use one 'alias' statement per alias you wish to define.

# alias

#************************ end of file <ipythonrc> ************************
```

7.2 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

\X*n* where $n = 0 \ldots 5$. The current working directory, with `$HOME` replaced by ~, and filtered out to contain only n path elements

\Y*n* Similar to \X*n*, but with the $n + 1$ element included if it is ~ (this is similar to the behavior of the %c*n* escapes in `tcsh`)

\u the username of the current user

\$ if the effective UID is 0, a #, otherwise a $

\h the hostname up to the first '.'

\H the hostname

\n a newline

\r a carriage return

\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).

7.2.1 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
```

The following shows the usage of dynamic expression evaluation:

7.3 IPython profiles

As we already mentioned, IPython supports the `-profile` command-line option (see sec. 5.2). 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>`, 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.

In sections 11 and 16 we discuss some particular profiles which come as part of the standard IPython distribution. You may also look in your `IPYTHONDIR` directory, any file whose name begins with `ipythonrc-` is a profile. You can use those as examples for further customizations to suit your own needs.

8 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[3].

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.

9 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

[3] Based on an idea by Holger Krekel.

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[4]. 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
```

[4] This functionality was inspired by IDL's combination of the `stop` keyword and the `.continue` executive command, which I have found very useful in the past, and by a posting on comp.lang.python by cmkl <cmkleffner-AT-gmx.de> on Dec. 06/01 concerning similar uses of pyrepl.

```
# 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> ********************
```

10 Using the Python debugger (pdb)

10.1 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. 6.2) 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.

10.2 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[5]. 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. 9), 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:

[5] Many thanks to Christopher Hart for the request which prompted adding this feature to IPython.

```
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`.

11 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.

11.1 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.

11.2 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
```

12 IPython as a system shell

IPython ships with a special profile called `pysh`, which you can activate at the command line as `'ipython -p pysh'`. This loads `InterpreterExec`, along with some additional facilities and a prompt customized for filesystem navigation.

Note that this does *not* make IPython a full-fledged system shell. In particular, it has no job control, so if you type Ctrl-Z (under Unix), you'll suspend pysh itself, not the process you just started.

What the shell profile allows you to do is to use the convenient and powerful syntax of Python to do quick scripting at the command line. Below we describe some of its features.

12.1 Aliases

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

12.2 Special syntax

Any lines which begin with `'~'`, `'/'` and `'.'` will be executed as shell commands instead of as Python code. The special escapes below are also recognized. `!cmd` is valid in single or multi-line input, all others are only valid in single-line input:

`!cmd` pass 'cmd' directly to the shell

`!!cmd` execute 'cmd' and return output as a list (split on '\n')

`$var=cmd` capture output of cmd into var, as a string

`$$var=cmd` capture output of cmd into var, as a list (split on '\n')

The `$/$$` syntaxes make Python variables from system output, which you can later use for further scripting. The converse is also possible: when executing an alias or calling to the system via `!/!!`, you can expand any python variable or expression by prepending it with `$`. Full details of the allowed syntax can be found in Python's PEP 215.

A few brief examples will illustrate these (note that the indentation below may be incorrectly displayed):

```
fperez[~/test]|3> !ls *s.py
scopes.py strings.py
```

ls is an internal alias, so there's no need to use `!`:

```
fperez[~/test]|4> ls *s.py
scopes.py* strings.py
```

!!ls will return the output into a Python variable:

```
fperez[~/test]|5> !!ls *s.py
             <5> ['scopes.py', 'strings.py']
fperez[~/test]|6> print _5
['scopes.py', 'strings.py']
```

`$` and `$$` allow direct capture to named variables:

```
fperez[~/test]|7> $astr = ls *s.py
fperez[~/test]|8> astr
             <8> 'scopes.py\nstrings.py'
```

```
fperez[~/test]|9> $$alist = ls *s.py
fperez[~/test]|10> alist
              <10> ['scopes.py', 'strings.py']
```

alist is now a normal python list you can loop over. Using `$` will expand back the python values when alias calls are made:

```
fperez[~/test]|11> for f in alist:
              |..>     print 'file',f,
              |..>     wc -l $f
              |..>
file scopes.py 13 scopes.py
file strings.py 4 strings.py
```

Note that you may need to protect your variables with braces if you want to append strings to their names. To copy all files in alist to `.bak` extensions, you must use:

```
fperez[~/test]|12> for f in alist:
              |..>     cp $f ${f}.bak
```

If you try using `$f.bak`, you'll get an AttributeError exception saying that your string object doesn't have a `.bak` attribute. This is because the `$` expansion mechanism allows you to expand full Python expressions:

```
fperez[~/test]|13> echo "sys.platform is:  $sys.platform"
sys.platform is:  linux2
```

IPython's input history handling is still active, which allows you to rerun a single block of multi-line input by simply using exec:

```
fperez[~/test]|14> $$alist = ls *.eps
fperez[~/test]|15> exec _i11
file image2.eps 921 image2.eps
file image.eps 921 image.eps
```

While these are new special-case syntaxes, they are designed to allow very efficient use of the shell with minimal typing. At an interactive shell prompt, conciseness of expression wins over readability.

12.3 Useful functions and modules

The os, sys and shutil modules from the Python standard library are automatically loaded. Some additional functions, useful for shell usage, are listed below. You can request more help about them with '`?`'.

`shell` - execute a command in the underlying system shell

`system` - like `shell()`, but return the exit status of the command

`sout` - capture the output of a command as a string

`lout` - capture the output of a command as a list (split on '\n')

`getoutputerror` - capture (output,error) of a shell commandss

`sout/lout` are the functional equivalents of `$/$$`. They are provided to allow you to capture system output in the middle of true python code, function definitions, etc (where `$` and `$$` are invalid).

12.4 Directory management

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

Pysh 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.

12.5 Prompt customization

The supplied `ipythonrc-pysh` profile comes with an example of a very colored and detailed prompt, mainly to serve as an illustration. The valid escape sequences, besides color names, are:

\# - Prompt number, wrapped in the color escapes for the input prompt (determined by the current color scheme).

\N - Just the prompt counter number, *without* any coloring wrappers. You can thus customize the actual prompt colors manually.

\D - Dots, as many as there are digits in \# (so they align).

\w - Current working directory (cwd).

\W - Basename of current working directory.

\X*N* - Where *N*=0..5. N terms of the cwd, with $HOME written as ~.

\Y*N* - Where *N*=0..5. Like *XN*, but if ~ is term *N*+1 it's also shown.

\u - Username.

\H - Full hostname.

\h - Hostname up to first '.'

\$ - Root symbol ($ or #).

\t - Current time, in H:M:S format.

\v - IPython release version.

\n - Newline.

\r - Carriage return.

**** - An explicitly escaped '\'.

You can configure your prompt colors using any ANSI color escape. Each color escape sets the color for any subsequent text, until another escape comes in and changes things. The valid color escapes are:

\C_Black

\C_Blue

\C_Brown

\C_Cyan

\C_DarkGray

\C_Green

\C_LightBlue

\C_LightCyan

\C_LightGray

\C_LightGreen

\C_LightPurple

\C_LightRed

\C_Purple

\C_Red

\C_White

\C_Yellow

\C_Normal Stop coloring, defaults to your terminal settings.

13 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. 5.1), 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.

13.1 Tk issues

As indicated in Sec. 5.1, 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.

13.2 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.

14 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.

15 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` (Sec. 5.2). 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 (Sec. 7.3): "`ipython -pylab -p myprofile`" will load the profile defined in `ipythonrc-myprofile` after configuring matplotlib.

16 Plotting with `Gnuplot`

Through the magic extension system described in sec. 6.2, IPython incorporates a mechanism for conveniently interfacing with the Gnuplot system (`http://www.gnuplot.info`). Gnuplot is a very complete 2D and 3D plotting package available for many operating systems and commonly included in modern Linux distributions.

Besides having Gnuplot installed, this functionality requires the `Gnuplot.py` module for interfacing python with Gnuplot. It can be downloaded from: `http://gnuplot-py.sourceforge.net`.

16.1 Proper Gnuplot configuration

As of version 4.0, Gnuplot has excellent mouse and interactive keyboard support. However, as of `Gnuplot.py` version 1.7, a new option was added to communicate between Python and Gnuplot via FIFOs (pipes). This mechanism, while fast, also breaks the mouse system. You must therefore set the variable `prefer_fifo_data` to `0` in file `gp_unix.py` if you wish to keep the interactive mouse and keyboard features working properly (`prefer_inline_data` also must be `0`, but this is the default so unless you've changed it manually you should be fine).

'Out of the box', Gnuplot is configured with a rather poor set of size, color and linewidth choices which make the graphs fairly hard to read on modern high-resolution displays (although they work fine on old 640x480 ones). Below is a section of my `.Xdefaults` file which I use for having a more convenient Gnuplot setup. Remember to load it by running `'xrdb .Xdefaults'`:

```
!*************************************************************
!  gnuplot options
!  modify this for a convenient window size
gnuplot*geometry:  780x580

!  on-screen font (not for PostScript)
gnuplot*font:  -misc-fixed-bold-r-normal--15-120-100-100-c-90-iso8859-1

!  color options
gnuplot*background:  black
gnuplot*textColor:  white
gnuplot*borderColor:  white
gnuplot*axisColor:  white
gnuplot*line1Color:  red
gnuplot*line2Color:  green
gnuplot*line3Color:  blue
gnuplot*line4Color:  magenta
gnuplot*line5Color:  cyan
gnuplot*line6Color:  sienna
gnuplot*line7Color:  orange
gnuplot*line8Color:  coral

!  multiplicative factor for point styles
gnuplot*pointsize:  2

!  line width options (in pixels)
gnuplot*borderWidth:  2
gnuplot*axisWidth:  2
gnuplot*line1Width:  2
gnuplot*line2Width:  2
gnuplot*line3Width:  2
gnuplot*line4Width:  2
gnuplot*line5Width:  2
gnuplot*line6Width:  2
gnuplot*line7Width:  2
gnuplot*line8Width:  2
```

16.2 The IPython.GnuplotRuntime module

IPython includes a module called Gnuplot2.py which extends and improves the default Gnuplot.py (which it still relies upon). For example, the new plot function adds several improvements to the original making it more convenient for interactive use, and hardcopy fixes a bug in the original which under some circumstances blocks the creation of PostScript output.

For scripting use, GnuplotRuntime.py is provided, which wraps Gnuplot2.py and creates a series of global aliases. These make it easy to control Gnuplot plotting jobs through the Python language.

Below is some example code which illustrates how to configure Gnuplot inside your own programs but have it available for further interactive use through an embedded IPython instance. Simply run this file at a system prompt. This file is provided as example-gnuplot.py in the examples directory:

```
#!/usr/bin/env python
"""
Example code showing how to use Gnuplot and an embedded IPython shell.
"""

from Numeric import *
from IPython.numutils import *
from IPython.Shell import IPShellEmbed

# Arguments to start IPython shell with. Load numeric profile.
ipargs = ['-profile','numeric']
ipshell = IPShellEmbed(ipargs)

# Compute sin(x) over the 0..2pi range at 200 points
x = frange(0,2*pi,npts=200)
y = sin(x)

# In the 'numeric' profile, IPython has an internal gnuplot instance:
g = ipshell.IP.gnuplot

# Change some defaults
g('set style data lines')

# Or also call a multi-line set of gnuplot commands on it:
g("""
set xrange [0:pi]     # Set the visible range to half the data only
set title 'Half sine' # Global gnuplot labels
set xlabel 'theta'
set ylabel 'sin(theta)'
""")

# Now start an embedded ipython.
ipshell('Starting the embedded IPyhton.\n'
        'Try calling plot(x,y), or @gpc for direct access to Gnuplot"\n')

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

16.3 The numeric profile: a scientific computing environment

The numeric IPython profile, which you can activate with 'ipython -p numeric' will automatically load the IPython Gnuplot extensions (plus Numeric and other useful things for numerical computing), contained in the IPython.GnuplotInteractive module. This will create the globals Gnuplot (an alias to the improved Gnuplot2 module), gp (a Gnuplot active instance), the new magic commands %gpc and %gp_set_instance and several other convenient globals. Type gphelp() for further details.

This should turn IPython into a convenient environment for numerical computing, with all the functions in the NumPy library and the Gnuplot facilities for plotting. Further improvements can be obtained by loading the SciPy libraries for scientific computing, available at http://scipy.org.

If you are in the middle of a working session with numerical objects and need to plot them but you didn't start the numeric profile, you can load these extensions at any time by typing
from IPython.GnuplotInteractive import *
at the IPython prompt. This will allow you to keep your objects intact and start using Gnuplot to view them.

17 Reporting bugs

Automatic crash reports

Ideally, IPython itself shouldn't crash. It will catch exceptions produced by you, but bugs in its internals will still crash it.

In such a situation, IPython will leave a file named IPython_crash_report.txt in your IPYTHONDIR directory (that way if crashes happen several times it won't litter many directories, the post-mortem file is always located in the same place and new occurrences just overwrite the previous one). If you can mail this file to the developers (see sec. 20 for names and addresses), it will help us *a lot* in understanding the cause of the problem and fixing it sooner.

The bug tracker

IPython also has an online bug-tracker, located at http://projects.scipy.org/ipython/ipython/report/1. In addition to mailing the developers, it would be a good idea to file a bug report here. This will ensure that the issue is properly followed to conclusion. To report new bugs you will have to register first.

You can also use this bug tracker to file feature requests.

18 Brief history

18.1 Origins

The current IPython system 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.

When I found out (see sec. 20) 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.

18.2 Current status

The above listed features work, and quite well for the most part. But until a major internal restructuring is done (see below), only bug fixing will be done, no other features will be added (unless very minor and well localized in the cleaner parts of the code).

IPython consists of some 18000 lines of pure python code, of which roughly two thirds is reasonably clean. The rest is, messy code which needs a massive restructuring before any further major work is done. Even the messy code is fairly well documented though, and most of the problems in the (non-existent) class design are well pointed to by a PyChecker run. So the rewriting work isn't that bad, it will just be time-consuming.

18.3 Future

See the separate `new_design` document for details. Ultimately, I would like to see IPython become part of the standard Python distribution as a 'big brother with batteries' to the standard Python interactive interpreter. But that will never happen with the current state of the code, so all contributions are welcome.

19 License

IPython is released under the terms of the BSD license, whose general form can be found at: http://www.opensource.org/licenses/bsd-license.php. The full text of the IPython license is reproduced below:

```
IPython is released under a BSD-type license.

Copyright (c) 2001, 2002, 2003, 2004 Fernando Perez
<fperez@colorado.edu>.
```

Individual authors are the holders of the copyright for their code and are listed in each file.

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.

20 Credits

IPython is mainly developed by Fernando Pérez `<Fernando.Perez@colorado.edu>`, but the project was born from mixing in Fernando's code with the IPP project by Janko Hauser `<jhauser-AT-zscout.de>` and LazyPython by Nathan Gray `<n8gray-AT-caltech.edu>`. For all IPython-related requests, please contact Fernando.

As of early 2006, 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 <bgranger-AT-scu.edu>: extending IPython to allow support for interactive parallel computing.

Ville Vainio <vivainio-AT-gmail.com>: Ville is the new maintainer for the main trunk of IPython after version 0.7.1.

User or development help should be requested via the IPython mailing lists:

User list: http://scipy.net/mailman/listinfo/ipython-user

Developer's list: http://scipy.net/mailman/listinfo/ipython-dev

The IPython project is also very grateful to[6]:

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.org by 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 me know if I have ommitted your name by accident:

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.

[6] I've mangled email addresses to reduce spam, since the IPython manuals can be accessed online.

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.