IPython

An enhanced Interactive Python

User Manual, v. 0.5.0

Fernando Pérez

25th August 2003

Contents

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 GNU LGPL[1]) 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 Mathcad inspired its design, but similar ideas can be useful in many fields.

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 ('?').
- Numbered input/output prompts with command history (persistent across sessions), full searching in this history and caching of all input and output.
- Macro system for quickly re-executing multiple lines of previous input with a single name.
- Session logging (you can then later use these logs as code in your programs).
- Session restoring: logs can be replayed to restore a previous session to the state where you left it.
- 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.

[1]IPython is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. Its full text is included in the file GNU-LGPL or can be obtained directly from the Free Software Foundation at: `http://www.gnu.org/copyleft/lesser.html`.

- 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.
- Automatic indentation (optional) of code as you type (trhough the readline library).
- 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 ',' as the first character forces auto-quoting of the rest of the line: `',my_function a b'` becomes automatically `'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 data analysis situations, for example).
- Easy debugger access. You can set IPython to call up 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 its possible to navigate the stack to rapidly isolate the source of a bug.
- Profiler support. You can run single statements (similar to `profile.run()`) or complete programs under the profiler's control.

1.2 Portability and Python requirements

Developed under **Linux**, should work under most unices (tested OK under Solaris).

Mac OS X: it works, apparently without any problems (thanks to Jim Boyle at Lawrence Livermore for the information).

CygWin: I would guess this environment is Unix enough for IPython to work unchanged (any comments welcome).

Windows: It works reasonably well under Windows XP, and I suspect NT and Win2000 should work similarly. Windows 9x support has been added but has seen very little testing, as I don't have access to a machine with that operating system.

Please note, however, that I have very little access to and experience with Windows development. For this reason, Windows-specific bugs tend to linger far longer than I would like, and often I just can't find a satisfactory solution. If any Windows user wants to join in with development help, all hands are always welcome.

MacOS Classic: it may work (I have no idea), and if not it should be reasonably easy to port it. But someone else will have to do that, since I have no access to a Macintosh.

IPython requires Python version 2.1 or newer. It has been tested with Python 2.2 and showed no problems.

1.3 Location

IPython is generously hosted at `http://ipython.scipy.org` by the SciPy project. This site offers downloads, CVS access, mailing lists and a bug tracking system. I am very grateful to the SciPy team for their contribution.

2 Installation

2.1 Instant instructions

If you are of the impatient kind, simply untar/unzip the download, install with `'python setup.py install'` under Linux/Unix or by double-clicking the `setup.py` file under Windows, and 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 Under Unix-type operating systems (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 simply by 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. For example:

```
$ python setup.py install --home $HOME/local
```

will install[2] IPython into `$HOME/local` and its subdirectories (creating them if necessary).
You can type

```
$ python setup.py --help
```

for more details.

2.2.1 RedHat 7.x notes

The problems discussed in this section do *not* apply to RedHat 8.0 and newer versions, only to the 7.x series.

RedHat made the 'wise' choice of using Python 1.5.2 as the default standard even for users (not just for internal system stuff). Since they couldn't be bothered to make things right, now you need to manually play around to get things to work with Python 2.x (which IPython requires).

First, your system administrator may have fixed things so that as a user you automagically see python 2.x. Test this by typing `'python'` at the prompt. If you get a Python 2.x prompt, you're safe. Otherwise you'll need to explicitly call Python2.

Start by making sure you did install Python 2.x. The rpm for it is named `python2....rpm`. You can check by typing `'python2'` at the command prompt and seeing if you get a python prompt with 2.x as the version. If you don't have it, install the Python 2.x rpm now.

Once you have confirmed you have Python 2.x installed, call the IPython setup routine as

```
$ python2 setup.py install
```

Hopefully, things will work. If they don't, go yell at RedHat, not me. One possible manual fix you may try is to edit `/usr/bin/ipython` and rename the `#!/usr/bin/python` line at the top to `#!/usr/bin/python2`.

2.2.2 Mac OSX notes

Apparently the problems which Mac OSX users may encounter with in the terminal window are due to poor VT100 emulation on Apple's part.

I don't have access to a Mac, so I rely on the helpful users from the OSX community for feedback on this issue. The information below was graciously provided by Andrea Riciputti from the Fink project, and I reproduce it unaltered hoping that it will be useful to others. If you find a mistake/update to this information, please let me know so that I can include it in future releases.

Note: I don't know if this information applies to 10.2 (Jaguar). It is *possible* that 10.2 fixes these problems, but this information has not been confirmed. Please let me know of any details concerning Jaguar which should be added to this documentation.

Many thanks to Andrea for taking the time to do this. His Mini-HOWTO follows.

Mac OSX Terminal Mini-HOWTO

[2] If you are reading these instructions in HTML format, please note that the option is –home, with *two* dashes. The automatic HTML conversion program seems to eat up one of the dashes, unfortunately (it's ok in the PDF version).

From: Andrea Riciputi <andrea.riciputi@libero.it>

Date: Thu, 28 Nov 2002 19:07:20 +0100

1) In order to get IPython works smoothly on MacOSX you have to reset the TERM env variable as follow:

```
% setenv TERM xterm
```

2) Done. Open a new terminal window and start ipython setting `color` and `color_info` to 1. Everything will go well!!

3) If someone wants to know more about this topics please look at these links:

`http://www.nyangau.fsnet.co.uk/terminfo/terminfo.htm` (not so easy)

`http://www.cs.utk.edu/~shuford/terminal/vt100_colorized_termcap.tx` (not easy at all)

2.3 Under Windows

Please note that for the automatic installer to work you need Mark Hammond's PythonWin extensions (and they're great for anything Windows-related anyway, so you might as well get them). If you don't have them, get them at:

`http://starship.python.net/crew/mhammond/`

From the download directory grab the `IPython-XXX.zip` file (but the popular WinZip handles `.tar.gz` files perfectly, so use that if you have WinZip and want a smaller download).

Unzip it and double-click on the `setup.py` file. A text console should open and proceed to install IPython in your system. If all goes well, that's all you need to do. You should now have an IPython entry in your Start Menu with links to IPython and the manuals.

If you don't have PythonWin, you can:

- Copy the `doc\` directory wherever you want it (it contains the manuals in HTML and PDF).
- Create a shortcut to the main IPython script, located in the `Scripts` subdirectory of your Python installation directory.

These steps are basically what the auto-installer does for you.

IPython tries to install the configuration information in a directory named `.ipython` located in your 'home' directory, which it determines by joining the environment variables `HOMEDRIVE` and `HOMEPATH`. 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.

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`, 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 2.1 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-2.1.1/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 wo work, you need to set your `EDITOR` environment variable to `'emacsclient'`. The code below, supplied by François 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 which typically don't set 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. But not all terminals out there have reliable support for on-screen color, so these options are not enabled by default to avoid confusing new users with on-screen garbage.

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.

These have shown problems:

- Windows command prompt in Win2k/XP logged into a Linux machine via telnet or ssh.

- Windows native command prompt in Win2k/XP for local execution. Colors do not work at all. The installer is set up to disable colors by default. If you have a terminal replacement which can handle colors, you can turn them back on. Test it by typing '`colors Linux`' at the prompt: if you get garbage on screen, go back with '`colors NoColor`'.
  Note that under Windows, if you use the CygWin environment, coloring (and readline-related features) all work correctly.

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. 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. 6.3. 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, currently (X)Emacs and IPython get along very well. You will need to use a recent version of `python-mode.el`, along with the file `ipython.el`. At the IPython website's download section, you will find a tarball containing both of these files. Once you put them 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.

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).

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 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. 6.4 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. Note that this feature does not work on platforms lacking readline support, such as Windows.

- 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. 6.3 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. 6.1 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. 6.8 for a recursive reload command.

- Use the Python debugger, pdb[3]. The `@pdb` command allows you to toggle on and off the automatic invocation of the pdb debugger 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`.

- 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. 6.11 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 ant 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. 6.10 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

[3]Thanks to Christian Hart for the suggestions leading to this feature and the profiler support.

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 macros with `@macro`. This can be useful for automating sequences of expressions when working interactively.
- 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 `@ds`) and via direct `@cd`. The latter keeps a history of visited directories and allows you to go to any previously visited one.
- 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. 7.2 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).

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 Options

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

`-help`: print a help message and exit.

`-no|automagic:` make magic commands automatic (without needing their first character to be `@`). Type `@magic` at the IPython prompt for more information.

`-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. Defaults to `$IPYTHONDIR/log`. 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 '`%n`' in the string. Don't forget to quote strings with spaces embedded in them. Default: '`In [%n]:`'

`-prompt_in2|pi2 <string>`: Similar to the previous option, but used for the continuation prompts. In this case, the number (`%n`) is replaced by as many dots as there are digits in the number (so you can have your continuation prompt aligned with your input prompt). Default: '`   .%n.:`' (note three spaces at the start for alignment with '`In [%n]`')

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

`-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.

`-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 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
/usr/local/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
/usr/local/home/fperez
In [5]:  del cd # if you remove the cd variable
In [6]:  cd ipython # automagic can work again
/usr/local/home/fperez/ipython
```

You can define your own magic functions to extend the system. The following is a snippet of code which shows how to do it. It is provided as file `example-magic.py` in the examples directory:

```
"""Example of how to define a magic function for extending IPython.

The name of the function *must* begin with magic_. IPython mangles it so
that magic_foo() becomes available as @foo.

The argument list must be *exactly* (self,parameter_s='').

The single string parameter_s will have the user's input. It is the magic
function's responsability to parse this string.

That is, if the user types
>>>@foo a b c

The followinng internal call is generated:
   self.magic_foo(parameter_s='a b c').

To have any functions defined here available as magic functions in your
IPython environment, import this file in your configuration file with an
execfile = this_file.py statement. See the details at the end of the sample
ipythonrc file.  """

# fisrt define a function with the proper form:
def magic_foo(self,parameter_s=''):
    """My very own magic!. (Use docstrings, IPython reads them)."""
    print 'Magic function. Passed parameter is between < >: <'+parameter_s+'>'
    print 'The self object is:',self

# Add the new magic function to the class dict:
from IPython.iplib import InteractiveShell
InteractiveShell.magic_foo = magic_foo

# And remove the global name to keep global namespace clean.  Don't worry, the
# copy bound to IPython stays, we're just removing the global name.
del magic_foo

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

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

6.1.1 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.

`@Quit`: Exit IPython without confirmation (like `@Exit`).

`@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).

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'

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

`@autocall`: Make functions callable without having to type parentheses.

This toggles the autocall command line option on and off.

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

`@automagic`: Make magic functions callable without having to type the initial @.

Toggles on/off (when off, you must call it as `@automagic`, of course). 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.

`@cat`: Alias to the system command 'cat'

`@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.

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

cd - changes to the last visited directory.

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

`@clear`: Alias to the system command 'clear'

`@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 the prompts and exception handlers.

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

`@config`: Show IPython's internal configuration.

`@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>.

`@dirs`: Return the current directory stack.

`@ed`: Alias to `@edit`.

`@edit`: Bring up an editor and execute the resulting code.

Usage: `@edit` [options] [args]

`@edit` will use the editor you have configured in your environment as the EDITOR variable. If this isn't found, it will default to vi under Linux/Unix and to notepad under Windows.

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 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:

-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.

-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.

Note: opening at an exact line is only supported under Unix, and some editors (like kedit and gedit) 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(): print "foo() was defined in an editing session" '

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'"

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'"

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'"

`@env`: List environment variables.

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

`@hist` [-n] -> print at most 40 inputs (some may be multi-line)
`@hist` [-n] n -> print at most n inputs
`@hist` [-n] n1 n2 -> print inputs between n1 and n2 (n2 not included)

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

If option -n is used, input numbers are not printed. This is useful if you want to get a printout of many lines which can be directly pasted into a text editor.

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

`@lc`: Alias to the system command 'ls -F -o –color'

`@ld`: List (in color) things which are directories or links to directories.

`@less`: Alias to the system command 'less'

`@lf`: List (in color) things which are normal files.

`@ll`: List (in color) things which are symbolic links.

`@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` [log_name [log_mode]]

If no name is given, it defaults to a file named 'ipython.log' 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):
over: overwrite existing log.
backup: rename (if exists) to name and start name.
append: well, that says it.
rotate: create rotating logs name.1 , name.2 , etc.

`@logstate`: Print the status of the logging system.

`@ls`: Alias to the system command 'ls -F'

`@lsmagic`: List currently available magic functions.

`@lx`: List (in color) things which are executable.

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

Usage:
`@macro` name n1:n2 n3:n4 ... n5 .. n6 ...

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.

Note that the slices use the standard Python slicing notation (5:8 means include lines numbered 5,6,7).

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 44:48 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.

`@mkdir`: Alias to the system command 'mkdir'

`@mv`: Alias to the system command 'mv'

`@p`: Just a short alias for Python's 'print'.

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

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

`@pdb`: Control the 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.

`@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.

Since magic functions have a particular form of calling which prevents you from writing something like:
In [1]: p = @prun -r print 4 invalid!
you must instead use IPython's automatic variables to assign this:
In [1]: @prun -r print 4
Out[1]: <pstats.Stats instance at 0x8222cec>
In [2]: stats = _

If you really need to assign this value via an explicit function call, you can always tap directly into the true name of the magic function with:
In [3]: stats = __IP.magic_prun('-r print 4')

-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 'stdname'.

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 [opts] filename.py [args to program]' and then any profile specific options as described here.

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

`@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']

`@pushd` with no arguments does a `@pushd` to your home directory.

`@pwd`: Return the current working directory path.

`@r`: Repeat previous input.

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.

`@reset`: Resets the namespace by removing all names defined by the user.

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

`@rm`: Alias to the system command 'rm -i'

`@rmdir`: Alias to the system command 'rmdir'

`@rmf`: Alias to the system command 'rm -f'

`@run`: Run the named file inside IPython as a program.

Usage:
`@run` [-n -i -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 has 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. 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.

-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.

`@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` filename n1:n2 n3:n4 ... n5 .. n6 ...

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.

`@who`: Print all interactive variables, with some minimal formatting.

This 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 list of all interactive variables.

`@whos`: Like `@who`, but gives some extra information about each variable.

For all variables, the type is printed. Additionally it prints:
- For ,[],(): their length.
- 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.

6.2 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.

6.3 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??`.

6.4 Readline-based features

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

6.4.1 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.

6.4.2 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.

6.4.3 Persistent command history across sessions

IPython will save your input history when it leaves and reload it next time you restart it.

6.4.4 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.

The feature is off by default because it can cause problems with pasting of indented code (the pasted code gets re-indented on each line). But a magic function `@autoindent` allows you to toggle it on/off at runtime. You can also set it permanently on in your `ipythonrc` file (set `autoindent 1`), and disable it only when needed via the magic function.

6.4.5 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.

6.5 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. 5.1)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.

6.6 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 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.

6.8 Recursive reload

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

6.9 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.

6.10 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. 6.1 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.

6.11 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.

6.12 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.

6.13 Automatic parentheses and quotes

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

6.13.1 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)]
```

6.13.2 Automatic quoting

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

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

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

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

6.13.3 Notes on usage of these two features

1. 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)
```

2. Whitespace is more important than usual (even for Python!) Arguments to auto-quote functions cannot have embedded whitespace.

```
In [21]:  ,string.split a b
-------> string.split ("a", "b")
Out[21]= ['a'] # probably not what you wanted
In [22]:  string.split 'a b'
-------> string.split ('a b')
Out[22]= ['a', 'b'] # quote explicitly and it works.
```

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`). 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,v 1.4 2003/05/16 06:53:42 fperez Exp $

#***************************************************************************
#
# Configuration file for IPython -- ipythonrc format
#
# 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 true, 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.  Set it to 0 if you want to disable it.

# 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-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 off by default because it can cause problems with pasting of
# indented code (the pasted code gets re-indented on each line).  But a magic
# function @autoindent allows you to toggle it on/off at runtime.

autoindent 0

# 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

# I haven't been able to ever find a way to make color work reliably under
# Windows. If you find a solution, please let me know so I can include it in
# future releases.

########################


# 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 (see ipython --help for more details).
# Use %n to represent the current prompt number, and quote them to protect
# spaces.
prompt_in1 'In [%n]:'

# In prompt_in2, %n is replaced by as many dots as there are digits in the
# current value of %n.
prompt_in2 '   .%n.:'

prompt_out 'Out[%n]:'


# 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

# 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

#---------------------------------------------------------------------------
# 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

# (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 '"[]{}-/~

#"' -- just to fix emacs coloring which gets confused by unmatched quotes.

# (iii) 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 inlclude 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.

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 struct pack unpack
# will produce internally the statement
# from struct import pack,unpack

# 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 __IP.magic_cl = __IP.magic_clear

# defines @cl as a new name for @clear.

#-----------------------------------------------------------------------------
# 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'.
# If automagic is on, you can just type myls like you would at a system shell
# prompt.  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 IPython profiles

As we already mentioned, IPython supports the `-profile` command-line option (see sec. 5.1). 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 12 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 Using 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[4].

[4]Based on an idea by Holger Krekel.

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 in other programs

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

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

It can also be useful in scientific computing situations where it is common to need to do some automatic, computationally intensive part and then stop to look at data, plots, etc[5]. 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 IPythonShellEmbed
ipshell = IPythonShellEmbed()
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.
```

[5]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@gmx.de> on Dec. 06/01 concerning similar uses of pyrepl.

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

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

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

# The basics to get you going:

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

# Try running this code both at the command line and from inside IPython (with
# @run example-embed.py)
try:
    __IPYTHON__
except NameError:
    nested = 0
    args = ['']
else:
    print "Running nested copies of IPython."
    print "The prompts for the nested copy have been modified"
    nested = 1
    # what the embedded instance will see as sys.argv:
    args = ['-pi1','In <%n>:','-po','Out<%n>:','-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<%n>'
else:
    args = ['-pi1','In2<%n>:','-po','Out<%n>:','-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.')

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 <%n>:','-po','Out<%n>:']
    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)

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[6]. This feature can also be turned on and off 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'`

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

in the argument string and automatically `pdb` will be called if an uncaught exception is triggered by your code.

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

```
import sys,IPython.ultraTB
sys.excepthook = IPython.ultraTB.FormattedTB(mode='Verbose', color_scheme='Linux',
call_pdb=1)
```

The `mode` keyword can be either `'Verbose'` or `'Plain'`, giving either very detailed or normal tracebacks respectively. The `color_scheme` keyword can be one of `'NoColor'`, `'Linux'` (default) or `'LightBG'`. These are the same options which can be set in IPython with `-colors` and `-xmode`.

This will give any of your programs detailed, colored tracebacks with automatic invocation of `pdb`.

If you want more information on the use of the `pdb` debugger, read the included `pdb.doc` file (part of the standard Python distribution). On a stock Mandrake Linux system it is located at `/usr/lib/python2.2/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.

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 two 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 fragments 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://starship.python.net/crew/hinsen/scientific.html`).

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 Access to Gnuplot

Through the magic extension system described in sec. 6.1, 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`.

12.1 Proper Gnuplot configuration

'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
```

12.2 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 systems makes the resulting PostScript files not be created.

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> *********************
```

12.3 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 GnuplotInteractive import *`
at the IPython prompt. This will allow you to keep your objects intact and start using Gnuplot to view them.

13 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. 16 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://www.scipy.net/roundup/ipython`. 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.

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

14 Brief history

14.1 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 the other two systems.

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. 16) 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 I had initially planned.

14.2 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 almost 11000 lines of pure python code, of which roughly 50% are fairly clean. The other 50% are fragile, 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.

14.3 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.

15 License

Unless indicated otherwise, files in this project are covered by the GNU Lesser General Public License (LGPL). Its full text is included in the file GNU-LGPL or can be obtained directly from the Free Software Foundation at: `http://www.gnu.org/copyleft/lesser.html`.

16 Credits

The main authors of the code are:

Fernando Pérez <fperez@colorado.edu> (currently main contact)

Janko Hauser <jhauser@comunit.de>

Nathan Gray <n8gray@caltech.edu>

And we are very grateful to:

Bill Bumgarner <bbum@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@lfw.org>: for providing the Itpl module for convenient and powerful string interpolation with a much nicer syntax than formatting through the '%' operator.

Arnd Bäcker <arnd.baecker@physik.uni-ulm.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.

Fernando would also like to thank Stephen Figgins <fig@monitor.net>, an O'Reilly Python editor. His Oct/11/01 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 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@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.

Mike Heeter `<korora@SDF.LONESTAR.ORG>`

Christopher Hart `<hart@caltech.edu>` PDB integration.

Milan Zamazal `<pdm@zamazal.org>` Emacs info.

Philip Hisley `<compsys@starpower.net>`

Holger Krekel `<pyth@devel.trillke.net>` Tab completion, lots more.

Alexander Schmolck `<a.schmolck@gmx.net>` Emacs work, bug reports, bug fixes, ideas, lots more.

Robin Siebler `<robinsiebler@starband.net>`

Ralf Ahlbrink `<ralf_ahlbrink@web.de>`

Andrea Riciputi `<andrea.riciputi@libero.it>` Mac OSX information.

Thorsten Kampe `<thorsten@thorstenkampe.de>`

Fredrik Kant `<fredrik.kant@front.com>` Windows setup.

Syver Enstad `<syver-en@online.no>` Windows setup.

Richard `<rxe@renre-europe.com>` Global embedding.

Hayden Callow `<h.callow@elec.canterbury.ac.nz>` Gnuplot.py 1.6 compatibility.

Leonardo Santagada `<retype@terra.com.br>` Fixes for Windows installation.

Christopher Armstrong `<radix@twistedmatrix.com>` Bugfixes.

François Pinard `<pinard@iro.umontreal.ca>` Code and documentation fixes.

Cory Dodt `<cdodt@fcoe.k12.ca.us>` Bug reports and Windows ideas.

Olivier Aubert `<oaubert@bat710.univ-lyon1.fr>` New magics.

Jeffrey Collins `<Jeff.Collins@vexcel.com>` Bug reports.

King C. Shu `<kingshu@myrealbox.com>` Autoindent patch.

Chris Drexler `<chris@ac-drexler.de>` Readline packages for Win32/CygWin.

Gustavo Córdova Avila `<gcordova@sismex.com>` EvalDict code for nice, lightweight string interpolation.

Gary Bishop `<gb@cs.unc.edu>` Bug reports, and patches to work around the exception handling idiosyncracies of WxPython.

Kasper Souren `<Kasper.Souren@ircam.fr>` Bug reports, ideas.