IPython

An enhanced Interactive Python

User Manual, v. 0.2.10

Fernando Pérez

29th April 2002

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.
- 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.

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 fairly 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. Comments 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

Currently IPython can be found at `http://www-hep.colorado.edu/~fperez/ipython`.

2 Installation

Please see the notes in sec. 3 for upgrading IPython versions.

2.1 Under Unix-type operating systems (Linux, Mac OS X, etc.)

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=$HOME/local`
   will install IPython into `$HOME/local` and its subdirectories (creating them if necessary).
   You can type
   `$ python setup.py --help`
   for more details.

Note that when installing, you will see some `SyntaxError` messages go by rapidly. Please ignore them, they are completely harmless (the result of an ugly but necessary hack around some limitations of distutils).

2.1.1 RedHat 7.x notes

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.

2.2 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.

2.3 Help access

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 Python documentation lives. In my system it's `/usr/share/doc/python-docs-2.1.1`, check your local details or ask your systems administrator.

You really want to set this variable correctly so that Python's pydoc-based help system works (it's very powerful).

Under Windows it seems that pydoc finds the documentation automatically, so no extra setup appears necessary.

2.4 Initial configuration comments

All of ipython's configuration information, history, logs, 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. Go poking around in there to learn more about configuring the system. As we said, this copy by default will be called `$HOME/.ipython`

If there is a problem, these are the instructions for manual installation:

1. `mkdir $HOME/.ipython`
2. Copy all the files in `IPython/UserConfig` to `$HOME/.ipython/,` except for the files named `__init__*`
3. In `$HOME/.ipython`, rename all the `ipythonrc*.py` files by removing the `.py` extension. They aren't really Python files, this is a workaround for a distutils limitation, and normally is done for you by the auto-installer.

Information on how to further customize the `ipythonrc*` files and how to build a hierarchy of them to manage IPython 'profiles' is provided in the sample files.

The default configuration has most bells and whistles turned on (they're pretty safe). But there's one that *may* cause problems on some systems: colored prompts and exception handlers.

If when you start IPython the input prompt shows garbage like:
`[0;32mIn [[1;32m1[0;32m]:  [0;00m`
instead of
`In [1]:`
this means that your terminal doesn't properly handle color escape sequences.

You can either try using a different terminal emulator program or switching coloring off completely. To do the latter, edit the file `$HOME/.ipython/ipythonrc` and set the `colors` option to the value `'NoColor'` (without quotes).

Terminals that seem to handle the color sequences fine:

- Linux main text console, KDE Konsole, Gnome Terminal, E-term, rxvt.
- CDE terminal (tested under Solaris). This one boldfaces light colors.
- (X)Emacs buffers. See sec.2.5 for more details on using IPython with (X)Emacs.

Terminals with problems:

- Old xterms (at least the version shipped with Mandrake 8.1). They display garbage for certain colors, haven't been able to fix it reliably.
- Windows command prompt in Win2k/XP logged into a Linux machine via telnet or ssh. Same problems as the xterms above.

- 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`'.

Currently available color schemes for prompts and exceptions:

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

In the future, more schemes may be implemented (if you add one, send it in. They are easy to write, look at the code in `ultraTB.py` and `Prompts.py`).

2.5 (X)Emacs users

To the best of my knowledge, the comments below apply both to GNU Emacs and to XEmacs[2].

In X/Emacs we get the following types of terminals (as returned by `os.environ['TERM']`):

- Term buffers (`M-x term`): 'eterm' terminal type.
- Shell buffers (`M-x shell`): 'emacs' terminal type (or 'dumb' in some versions).
- Python interpreter (comint) buffers (`C-c !`): 'emacs' terminal type (or 'dumb' in some versions).

2.5.1 Color support

All of these terminal types will support coloring of prompts and tracebacks, even though only light versions of colors seem to be displayed. I may write an Emacs-specific color scheme in the future (contributions welcome, look at `ultraTB.py` for details). But for this color support to work properly, you must first copy the following lines to your `.emacs` file:

```
; Customizations for IPython

(defun activate-ansi-colors ()
(require 'ansi-color)
(ansi-color-for-comint-mode-on))

(add-hook 'comint-mode-hook 'activate-ansi-colors)

(add-hook 'shell-mode-hook 'activate-ansi-colors)
```

[2]This section, and the lisp code in it, owes a lot to the kind help and comments by Milan Zamazal <pdm@zamazal.org>. Many thanks to him.

```
; Set IPython to be the python command and give it arguments

(setq py-python-command "ipython")

(setq py-python-command-args
(cond
((eq frame-background-mode 'dark)
'("--colors" "Linux"))
((eq frame-background-mode 'light)
'("--colors" "LightBG"))
(t  ; default (backg-mode isn't always set by XEmacs)
'("--colors" "LightBG"))
) )
```

You can customize the above to suit your personal preferences.

2.5.2 Name completion

'eterm' buffers support TAB name completion like a normal terminal, but 'emacs' and 'dumb' ones do not support it well, because Emacs itself takes control of line input. You will thus lose full name completion in an IPython buffer started via `C-c !`. All other features of IPython will work normally.

Still, name completion works to some extent: the completions are not printed when you hit TAB, instead you must hit Return after TAB (giving a `SyntaxError`, of course). At this point you get a printout of the possible completions and you can get your previous line with `Ctrl-UpArrow`. Not perfect, but better than nothing.

I know it's clumsy, but so far efforts to fix this have failed. Any Emacs gurus out there who can think of a clean way to fix this are encouraged to contact the developers at the addresses given in sec. 14. The issue (I think) boils down to the following: Emacs only prints the contents of a command received by its sub-process in a comint buffer when it gets an EOL (or more precisely, the result of `comint-send-input`). At that point it prints out all the buffered contents from the subprocess. But for normal TAB completion to work, one needs to print the list of completions at the current cursor position (possibly completing part of the line) *before* the command is finished. So a kind of catch-22 situation arises. As I said, all ideas/fixes are welcome from the Emacs gurus out there.

3 Upgrading from a previous version

If you are upgrading from a previous version of IPython, after doing the routine installation described in sec.2, you may want to 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`.

4 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`.

4.1 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).

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

`-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()`].

`-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 and Verbose. See the sample ipythonrc file for details on this option.

5 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.

5.1 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 your ipython configuration directory, typically `$HOME/.ipython/`:

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

5.1.1 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.

`@Pprint`: Toggle pretty printing on/off.

`@abort`: Raise a SystemExit exception.

In a normal IPython session this just exits IPython, but if IPython is running embedded inside other Python code, the SystemExit exception may propagate all the way up and fully exit the enclosing program.

`@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.

`@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'

`@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.

`@doc`: Print the docstring for an object.

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

`@env`: List environment variables.

`@file`: View the source file for an object through a pager.

The file opens at the line 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.

`@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 '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.
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:

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' will re-execute all this code in one pass.

The macro is a simple object which holds its value in an attribute, but the printing system checks for macros and executes them as code instead of printing them.

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

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

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

`@oinfo`: Provide detailed information about an object.

oinfo object is just a synonym for object? or ?object.

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

`@pdb`: Toggle the calling of the pdb interactive debugger.

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.

`@pfile`: Same as `@file`.

In Python 2.2 file() is now a builtin, so with automagic on, 'file' doesn't work anymore. This alias is provided for convenience.

`@popd`: Change to directory popped off the top of the stack.

`@profile`: Print your currently active profile.

`@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.

`@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] 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.

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 start in.

If the -n option is used, __name__ is NOT set to '__main__'. This allows running scripts and reloading the definitions in them without triggering a call to testing routines which are often wrapped in an 'if __name__=="__main__" clause.

`@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.

`@source`: Show the source code for an object.

`@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, Verbose.

If called without arguments, acts as a toggle.

5.2 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. 2.3, you need to properly configure your environment variable `PYTHONDOCS` for this feature to work correctly.

5.3 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:

`@doc <object>`: Print 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.

`@source <object>`: Show 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 (`?/??, @doc, @file, @pdef, @source`) 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??`. This feature can be extremely useful.

5.4 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.

5.4.1 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.

5.4.2 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.

5.4.3 Persistent command history across sessions

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

5.4.4 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.

5.5 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. 4.1)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.

5.6 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.

5.7 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.

5.8 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.

5.9 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.

5.10 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`.

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

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

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

You can re-execute multiple lines of input easily by assigning a macro name to them (which also allows you to build a macro out of any number of input lines even if they weren't contiguous in the first place). Type `@macro?` or see sec. 5.1 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. Note that inputs which contain magic functions (`@`) appear in the history with a prepended comment. This is because they aren't really valid Python code, so you can't exec them.

5.11 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`)

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 system is completely disabled and the prompts revert to the classic `'>>>'` of normal Python.

5.12 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.

5.13 Automatic parentheses and quotes

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

5.13.1 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)]
```

5.13.2 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
```

5.13.3 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.
```

6 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.

6.1 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
#***************************************************************************
#
# 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: prompt control and 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.


# 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

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

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

# nobanner 1 -> same as ipython -nobanner
```

```
nobanner 0

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

# 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

# Same as ipython -session YourSessionName
session ''

# xmode - Exception reporting mode. Valid modes: Plain, Verbose.

# Plain is similar to python's normal traceback printing.

# Verbose is a very detailed mode (a terminal port of Ka-Ping Yee's cgitb
# module, standard as of Python 2.2) which gives gobs of internal
# information. It basically dissects and dumps the entire state of Python when
# an exception is triggered. This can be very useful if you are doing tricky
# debugging work.

#xmode Plain
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

# 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. Do not change the default value
# unless you know what you're doing.
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> ************************
```

6.2 IPython profiles

As we already mentioned, IPython supports the `-profile` command-line option (see sec. 4.1). 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 9 and 10 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.

7 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[3]. 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 your `IPYTHONDIR` directory as `example-embed.py`. It should be fairly self-explanatory:

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

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

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

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

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

# The basics to get you going:

# Interactive runs of IPython set 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:
    if __IPYTHON__:
```

[3]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.

```
        print "Running nested copies of IPython."
        print "The prompts for the nested copy have been modified"
        # Remember not to use quotes and to use %s to represent spaces:
        nested = "-pi1 In%s<%n>: -po Out<%n>:"
except:
    nested = ''

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

# 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 = IPythonShellEmbed('-nosep '+nested,
                            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 = '-nosep '+nested.replace('In%s','In2')
else:
    args = '-nosep -pi1 In2<%n>: -po Out<%n>:'
ipshell2 = IPythonShellEmbed(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:

# IPythonShellEmbed 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.

# IPythonShellEmbed 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 (this file is also in your `IPYTHONDIR` directory as `example-embed-short.py`):

```
"""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)

# Command-line options for IPython (no quotes, %s for spaces)
argv = ''
try:
    if __IPYTHON__:
        # Remember not to use quotes and to use %s to represent spaces:
        nested = "-pi1 In%s<%n>: -po Out<%n>:"
        banner = '*** Nested interpreter ***'
        exit_msg = '*** Back in main IPython ***'
except:
    nested = banner = exit_msg = ''

# First import the embeddable shell class
from IPython.Shell import IPythonShellEmbed
ipshell = IPythonShellEmbed(argv+' '+nested,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 IPythonShellEmbed
ipshell = IPythonShellEmbed()
# 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:
    from IPython.Shell import IPythonShellEmbed
    ipshell = IPythonShellEmbed()
    # 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> ******************
```

8 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[4]. 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. 7), simply call the constructor with `'-pdb'` in the argument string and automatically `pdb` will be called if an uncaught exception is triggered by your code.

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

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

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

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

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.

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

9 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.

9.1 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.

9.2 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
```

10 Access to Gnuplot

Through the magic extension system described in sec. 5.1, IPython incorporates a mechanism for conveniently interfacing with the Gnuplot system (`http://www.gnuplot.vt.edu`). 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`.

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 `GnuplotMagic` module. You can also load this module at any time while using IPython via an `'import GnuplotMagic'` command. This will create the globals `Gnuplot` (the standard Gnuplot module) and `g` (a Gnuplot active instance) and the new magic commands `@gp` and `@gp_set_instance`.

Below is a copy of the file which implements these[5]. This file can be used as an example for writing sub-systems for interfacing to any other program from within IPython, such as talking to Mathematica.

In the `numutils` module (available via `'import IPython.numutils'`) you'll also find the `gnuplot_exec()` function which is useful for passing multi-line strings (such as those generated in Gnuplot via the `save` command) to a Gnuplot instance. This function is *not* in `GnuplotMagic` because that module *only* works inside IPython.

Gnuplot extensions file (loaded in the `numeric` profile by importing the `GnuplotMagic` module):

[5]This file is automatically copied to your `IPYTHONDIR` so you can modify it to suit your needs.

```
"""Magic function system for convenient interfacing with Gnuplot.

Warning: This module can ONLY be imported inside IPython or through an
embedded IPython instance.

This requires the Gnuplot.py module for interfacing python with gnuplot. It
can be downloaded from:

http://gnuplot-py.sourceforge.net/

This system builds the magic functions:

@gp -> pass one command to gnuplot and execute them or open a gnuplot shell
where each line of input is executed.

@gp_set_instance -> see below.

and global variables:

- g: pointing to a running gnuplot instance.
- Gnuplot: the Gnuplot module.

The docstrings below contain more details.

Please note that all commands are executed in the gnuplot namespace, so no
python variables exist there. Any data communication has to be done via files,
or using the methods of the gnuplot instance directly. For convenience, a copy
of the internal gnuplot instance called 'g' is created for you at startup. You
can call whichever methods you need on g (try typing g.<TAB> to see a list of
g's methods, or see the documentation for the Gnuplot.py module for details).

Configure this file's behavior by setting the global variable 'gnuplot_mouse'.

Inspired by a suggestion/request from Arnd Baecker."""

# If you have a mouse-enabled gnuplot, you can set gnuplot_mouse to 1,
# otherwise leave it at 0. Unfortunately there's no way to currently catch the
# resulting error message as a python exception if this doesn't work, so I
# can't leave it permanently on.

# Please note that this feature may cause some hiccups in communication with
# the gnuplot process (requiring sometimes a command to be issued twice). This
# is discussed in the gnuplot help and is an issue beyond my control. If it
# becomes a problem, resign yourself to not using the mouse features in
# gnuplot through IPython.

gnuplot_mouse = 0
```

```
# Create the Gnuplot instance we need
import Gnuplot

__IPYTHON__.gnuplot = Gnuplot.Gnuplot()
__IPYTHON__.gnuplot.shell_first_time = 1
g = __IPYTHON__.gnuplot  # alias for interactive convenience

# Force g as a global in IPython's namespace if we're being imported
if __name__ == 'GnuplotMagic':
    __IPYTHON__.user_ns['g'] = g
    __IPYTHON__.user_ns['Gnuplot'] = Gnuplot

if gnuplot_mouse:
    g('set mouse')
    print "*** g is a global alias to the internal Gnuplot instance (mouse enabled)."
else:
    print "*** g is a global alias to the internal Gnuplot instance."
print "*** Gnuplot instance access via @gp, or by calling methods of g directly."
print "*** Gnuplot module has been exported as a global."

# Define the magic functions for communicating with the above gnuplot instance.
def magic_gp(self,parameter_s=''):
    """Execute a gnuplot command or open a gnuplot shell.

    Usage (omit the @ if automagic is on). There are two ways to use it:

      1) @gp 'command' -> passes 'command' directly to the gnuplot instance.

      2) @gp -> will open up a prompt (gnuplot>) which takes input until '.'
      is entered (without quotes), or ^C or ^D are pressed. Each line of input
      is given to gnuplot as a command to be executed. If you need to type a
      multi-line command, use \\ at the end of each intermediate line.

      IPython traps the following forms of gnuplot's termination command:
      'quit' and 'exit', and simply returns to IPython without killing the
      gnuplot process. But if you type another form accepted by gnuplot,
      you'll terminate gnuplot (see below for how to restart it). So in order
      to simply return to IPython, use only one of the following exit options:
      '.', 'quit','exit', ^D (EOF) or ^C.

      Upon exiting of the gnuplot sub-shell, you return to your IPython
      session (the gnuplot sub-shell can be invoked as many times as needed).

    This system relies on the existence of a user-level variable called
    __gnuplot pointing to an active Gnuplot session. This variable is
    automatically created at startup, but if you overwrite it the system will
    stop functioning. You can create it again with a call like:
```

```
    In [1]: __gnuplot = Gnuplot.Gnuplot()    """

    from IPython.genutils import raw_input_ext

    if parameter_s.strip() == '':
        try:
            self.shell.gnuplot.has_run
        except AttributeError:
            print \
"""To exit gnuplot and return to IPython: '.', 'quit', 'exit', ^C or ^D (EOF).
Use \\ to break multi-line commands."""
            self.shell.gnuplot.has_run = 1
        try:
            cmd = raw_input_ext('gnuplot> ','more...> ')
            while cmd.strip() not in ['.','quit','exit']:
                self.shell.gnuplot(cmd)
                cmd = raw_input_ext('gnuplot> ','more...> ')
        except (EOFError,KeyboardInterrupt):
            pass
    else:
        self.shell.gnuplot(parameter_s)


def magic_gp_set_instance(self,parameter_s=''):
    """Set the global gnuplot instance accessed by the @gp magic function.

    @gp_set_instance name

    Call with the name of the new instance at the command line. If you want to
    set this instance in your own code (using an embedded IPython, for
    example), simply set the variable __IP.gnuplot to your own gnuplot
    instance object."""

    self.shell.gnuplot = eval(parameter_s)


# Add the new magic functions to the class dict
from IPython.iplib import InteractiveShell
InteractiveShell.magic_gp = magic_gp
InteractiveShell.magic_gp_set_instance = magic_gp_set_instance

# Keep global namespace clean
del magic_gp,magic_gp_set_instance,gnuplot_mouse

#********************** End of file <GnuplotMagic.py> **********************
```

You can also use Gnuplot as part of your normal Python programs. 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 in your `IPYTHONDIR` as `example-gnuplot.py`:

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

import Gnuplot
from Numeric import *
from IPython.numutils import *
from IPython.Shell import IPythonShellEmbed

# Arguments to start IPython shell with. Load numeric profile.
ipargs = '-profile numeric'
ipshell = IPythonShellEmbed(ipargs)

# Compute sin(x) over the 0..2pi range at 100 points
x = frange(0,2*pi,npts=200)
y = sin(x)

# Make a gnuplot instance
g2 = Gnuplot.Gnuplot()

# Change some defaults
g2('set data style lines')

# Or also call a multi-line set of gnuplot commands on it:
gnuplot_exec(g2,"""
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)'
""")

# Set the gnuplot instance accessed by @gp to our customized one. This way the
# interactive system @gp will see our instance with whatever changes we've
# made to it.
ipshell.IP.gnuplot = g2

# Now start an embedded ipython. g2 is a visible global, but it's also the one
# accessed by @gp now
ipshell('Starting the embedded IPyhton.\n'
        'Try calling g2.plot(zip(x,y)), or "@gp plot x**2"\n')

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

11 Reporting bugs

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

Were such an unlikely event to occur :), 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. 14 for names and addresses), it will help us *a lot* in understanding the cause of the problem and fixing it sooner.

12 Brief history

12.1 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. 14) 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.

12.2 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 8000 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.

12.3 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.

13 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`.

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.

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

Some files (`DPyGetOpt.py`, for example) may be licensed under different conditions. Ultimately each file indicates clearly the conditions under which its author/authors have decided to publish the code.

14 Credits

The main authors of the code are:

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

Janko Hauser <jhauser@ifm.uni-kiel.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.