IPython Documentation

Release 0.10.1

The IPython Development Team

October 11, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Overview

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

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

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

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

1.2 Enhanced interactive Python shell

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

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

1.2.1 Main features of the interactive shell

- Dynamic object introspection. One can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke ('?', and using '??' provides additional detail).
- Searching through modules and namespaces with '*' wildcards, both when using the '?' system and via the '%psearch' command.
- Completion in the local namespace, by typing TAB at the prompt. This works for keywords, modules, methods, variables and files in the current directory. This is supported via the readline library, and full access to configuring readline's behavior is provided. Custom completers can be implemented easily for different purposes (system commands, magic arguments etc.)
- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.
- User-extensible 'magic' commands. A set of commands prefixed with '%' is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with '!' are passed directly to the system shell, and using '!!' or 'var = !cmd' captures shell output into python variables for further use.
- Background execution of Python commands in a separate thread. IPython has an internal job manager called jobs, and a convenience backgrounding magic function called '%bg'.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with '$' is expanded. A double '$$' allows passing a literal '$' to the shell (for access to shell and environment variables like PATH).
- Filesystem navigation, via a magic '%cd' command, along with a persistent bookmark system (using '%bookmark') for fast access to frequently visited directories.
- A lightweight persistence framework via the '%store' command, which allows you to save arbitrary Python variables. These get restored automatically when your session restarts.
- Automatic indentation (optional) of code as you type (through the readline library).
- Macro system for quickly re-executing multiple lines of previous input with a single name. Macros can be stored persistently via '%store' and edited via '%edit'.
- Session logging (you can then later use these logs as code in your programs). Logs can optionally timestamp all input, and also store session output (marked as comments, so the log remains valid Python source code).
- Session restoring: logs can be replayed to restore a previous session to the state where you left it.
- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information (basically a terminal version of the cgitb module).
- Auto-parentheses: callable objects can be executed without parentheses: 'sin 3' is automatically converted to 'sin(3)'.

- Auto-quoting: using '`,`', or '`;`' as the first character forces auto-quoting of the rest of the line: '`,my_function a b`' becomes automatically '`my_function("a","b")`', while '`;my_function a b`' becomes '`my_function("a b")`'.
- Extensible input syntax. You can define filters that pre-process user input to simplify input in special situations. This allows for example pasting multi-line code fragments which start with '`>>>`' or '`...`' such as those from other python sessions or the standard Python documentation.
- Flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.
- Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).
- Easy debugger access. You can set IPython to call up an enhanced version of the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The '`%run`' magic command (with the '`-d`' option) can run any script under pdb's control, automatically setting initial breakpoints for you. This version of pdb has IPython-specific improvements, including tab-completion and traceback coloring support. For even easier debugger access, try '`%debug`' after seeing an exception. winpdb is also supported, see ipy_winpdb extension.
- Profiler support. You can run single statements (similar to '`profile.run()`') or complete programs under the profiler's control. While this is possible with standard cProfile or profile modules, IPython wraps this functionality with magic commands (see '`%prun`' and '`%run -p`') convenient for rapid interactive work.
- Doctest support. The special '`%doctest_mode`' command toggles a mode that allows you to paste existing doctests (with leading '`>>>`' prompts and whitespace) and uses doctest-compatible prompts and output, so you can use IPython sessions as doctest code.

1.3 Interactive parallel computing

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

The main features of this system are:

- Quickly parallelize Python code from an interactive Python/IPython session.
- A flexible and dynamic process model that be deployed on anything from multicore workstations to supercomputers.
- An architecture that supports many different styles of parallelism, from message passing to task farming. And all of these styles can be handled interactively.
- Both blocking and fully asynchronous interfaces.

- High level APIs that enable many things to be parallelized in a few lines of code.
- Write parallel code that will run unchanged on everything from multicore workstations to supercomputers.
- Full integration with Message Passing libraries (MPI).
- Capabilities based security model with full encryption of network connections.
- Share live parallel jobs with other users securely. We call this collaborative parallel computing.
- Dynamically load balanced task farming system.
- Robust error handling. Python exceptions raised in parallel execution are gathered and presented to the top-level code.

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

1.3.1 Portability and Python requirements

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

IPython is known to work on the following operating systems:

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

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

CHAPTER

TWO

INSTALLATION

2.1 Overview

This document describes the steps required to install IPython. IPython is organized into a number of subpackages, each of which has its own dependencies. All of the subpackages come with IPython, so you don't need to download and install them separately. However, to use a given subpackage, you will need to install all of its dependencies.

Please let us know if you have problems installing IPython or any of its dependencies. Officially, IPython requires Python version 2.5 or 2.6. We have *not* yet started to port IPython to Python 3.0.

> **Warning:** Officially, IPython supports Python versions 2.5 and 2.6.
> IPython 0.10 has only been well tested with Python 2.5 and 2.6. Parts of it may work with Python 2.4, but we do not officially support Python 2.4 anymore. If you need to use 2.4, you can still run IPython 0.9.

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

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

2.2 Quickstart

If you have `setuptools` installed and you are on OS X or Linux (not Windows), the following will download and install IPython *and* the main optional dependencies:

```
$ easy_install ipython[kernel,security,test]
```

This will get Twisted, zope.interface and Foolscap, which are needed for IPython's parallel computing features as well as the nose package, which will enable you to run IPython's test suite. To run IPython's test suite, use the **iptest** command:

```
$ iptest
```

Read on for more specific details and instructions for Windows.

2.3 Installing IPython itself

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

2.3.1 Installation using easy_install

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

```
$ easy_install ipython
```

That's it.

2.3.2 Installation from source

If you don't want to use **easy_install**, or don't have it installed, just grab the latest stable build of IPython from here. Then do the following:

```
$ tar -xzf ipython.tar.gz
$ cd ipython
$ python setup.py install
```

If you are installing to a location (like `/usr/local`) that requires higher permissions, you may need to run the last command with **sudo**.

2.3.3 Windows

There are a few caveats for Windows users. The main issue is that a basic `python setup.py install` approach won't create `.bat` file or Start Menu shortcuts, which most users want. To get an installation with these, you can use any of the following alternatives:

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

IPython by default runs in a termninal window, but the normal terminal application supplied by Microsoft Windows is very primitive. You may want to download the excellent and free Console application instead, which is a far superior tool. You can even configure Console to give you by default an IPython tab, which is very convenient to create new IPython sessions directly from the working terminal.

2.3.4 Installing the development version

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

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

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

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

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

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

```
$ bzr pull
```

2.4 Basic optional dependencies

There are a number of basic optional dependencies that most users will want to get. These are:

- readline (for command line editing, tab completion, etc.)
- nose (to run the IPython test suite)
- pexpect (to use things like irunner)

If you are comfortable installing these things yourself, have at it, otherwise read on for more details.

2.4.1 readline

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

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

On OS X, the built-in Python doesn't not have `readline` because of license issues. Starting with OS X 10.5 (Leopard), Apple's built-in Python has a BSD-licensed not-quite-compatible readline replacement. As of IPython 0.9, many of the issues related to the differences between readline and libedit seem to have been resolved. While you may find libedit sufficient, we have occasional reports of bugs with it and several developers who use OS X as their main environment consider libedit unacceptable for productive, regular use with IPython.

Therefore, we *strongly* recommend that on OS X you get the full `readline` module. We will *not* consider completion/history problems to be bugs for IPython if you are using libedit.

To get a working `readline` module, just do (with `setuptools` installed):

```
$ easy_install readline
```

Note: Other Python distributions on OS X (such as fink, MacPorts and the official python.org binaries) already have readline installed so you likely don't have to do this step.

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

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

2.4.2 nose

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

```
$ easy_install nose
```

Another way of getting this is to do:

```
$ easy_install ipython[test]
```

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

```
$ iptest
```

2.4.3 pexpect

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

```
$ easy_install pexpect
```

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

2.5 Dependencies for IPython.kernel (parallel computing)

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

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

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

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

2.5.1 zope.interface and Twisted

Twisted [Twisted] and zope.interface [ZopeInterface] are used for networking related things. On Unix style platforms (including OS X), the simplest way of getting the these is to use **easy_install**:

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

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

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

2.5.2 Foolscap

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

On all platforms a simple:

```
$ easy_install foolscap
```

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

2.5.3 pyOpenSSL

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

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

2.6 Dependencies for IPython.frontend (the IPython GUI)

2.6.1 wxPython

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

CHAPTER

THREE

USING IPYTHON FOR INTERACTIVE WORK

3.1 Quick IPython tutorial

IPython can be used as an improved replacement for the Python prompt, and for that you don't really need to read any more of this manual. But in this section we'll try to summarize a few tips on how to make the most effective use of it for everyday Python development, highlighting things you might miss in the rest of the manual (which is getting long). We'll give references to parts in the manual which provide more detail when appropriate.

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

3.1.1 Highlights

Tab completion

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

Explore your objects

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

The *%run* magic command

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

Debug a Python script

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

Use the output cache

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

Suppress output

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

Input cache

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

Use your input history

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

Define your own system aliases

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

Call system shell commands

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

Use Python variables when calling the shell

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

Use profiles

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

Embed IPython in your programs

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

Use the Python profiler

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

Use IPython to present interactive demos

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

Run doctests

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

3.1.2 Source code handling tips

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

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

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

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

where the command given can be:

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

While %macro saves input lines into memory for interactive re-execution, sometimes you'd like to save your input directly to a file. The %save magic does this: its input sytnax is the same as %macro, but it saves your input directly to a Python file. Note that the %logstart command also saves input, but it logs all input to disk

(though you can temporarily suspend it and reactivate it with %logoff/%logon); %save allows you to select which lines of input you need to save.

3.1.3 Lightweight 'version control'

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

```
In [1]: %edit

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

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

hello - this is a temporary file

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

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

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

```
In [2]: edit -p

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

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

hello - now I made some changes

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

In [3]: edit _1

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

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

hello - this is a temporary file

IPython version control at work :)

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

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

3.1.4 Effective logging

A very useful suggestion sent in by Robert Kern follows:

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

```
include ipythonrc

# cancel earlier logfile invocation:

logfile ''

execute import time

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

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

I also added a shell alias for convenience:

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

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

3.1.5 Logging to a file

Here is an alternative logging solution that lets you record your sessions in a daily time-stamped log-files.

Add the following lines or make it a-like to your ipy_user_conf.py:

```
from time import strftime

def main():

    try:
        ldir = '/home/$YOUR_USERNAME_HERE/.ipython/'
        filename = os.path.join(ldir, strftime('%Y-%m-%d')+".py")
        notnew = os.path.exists(filename)
        ip.IP.logger.logstart(logfname=filename, logmode='append')
        log_write = ip.IP.logger.log_write
        if notnew:
            log_write("# ================================")
        else:
            log_write("#!/usr/bin/env python \n# %s.py \n"
                      "# IPython automatic logging file" %
                      strftime('%Y-%m-%d'))
        log_write("# %s \n# ================================" %
                  strftime('%H:%M'))
        print " Logging to "+filename

    except RuntimeError:
        print " Already logging to "+ip.IP.logger.logfname
```

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

3.2 IPython reference

3.2.1 Command-line usage

You start IPython with the command:

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

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

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

Special Threading Options

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

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

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

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

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

-tk

The -g/q/q4/wthread options, and -pylab (if matplotlib is configured to use GTK, Qt3, Qt4 or WX), will normally block Tk graphical interfaces. This means that when either GTK, Qt or WX threading is active, any attempt to open a Tk GUI will result in a dead window, and possibly cause the Python interpreter to crash. An extra option, -tk, is available to address this issue. It can only be given as a second option after any of the above (-gthread, -wthread or -pylab).

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

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

Regular Options

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

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

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

-help

print a help message and exit.

-pylab

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-editor <name> Which editor to use with the %edit command. By default, IPython will honor your EDITOR environment variable (if not set, vi is the Unix default and notepad the Windows one). Since this editor is invoked on the fly by IPython and is meant for editing

small code snippets, you may want to use a small, lightweight editor here (in case your default EDITOR is something like Emacs).

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

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

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

-logplay, lp <name>

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

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

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

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

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

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

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

-profile, p <name>

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

1. $HOME/.ipython/ipythonrc : load basic things you always want.

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

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

-prompt_in1, pi1 <string>

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

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

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

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

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

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

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

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

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

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

-separate_in, si <string>

separator before input prompts. Default: 'n'

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

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

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

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

-Version print version information and exit.

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

-xmode <modename>

Mode for exception reporting.

Valid modes: Plain, Context and Verbose.

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

3.2.2 Interactive use

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

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

Caution for Windows users

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

Magic command system

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

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

If you have 'automagic' enabled (in your ipythonrc file, via the command line option -automagic or with the %automagic function), you don't need to type in the % explicitly. IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type 'cd mydir' to go to directory 'mydir'. The automagic system has the lowest possible precedence in name searches, so defining an identifier with the same name as an existing magic function will shadow it for automagic use. You can still access the shadowed magic function by explicitly using the % character at the beginning of the line.

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

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

/home/fperez/ipython

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

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

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

    File "<console>", line 1

      cd ..

          ^

SyntaxError: invalid syntax

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

/home/fperez

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

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

/home/fperez/ipython
```

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

```
import IPython.ipapi

ip = IPython.ipapi.get()

def doimp(self, arg):
```

```
    ip = self.api

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

    arg,arg,arg)

    )

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

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

```
execute __IP.magic_cl = __IP.magic_clear
```

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

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

The API documentation for the `IPython.Magic` module contains the full docstrings of all currently available magic commands.

Access to the standard Python help

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

Dynamic object information

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

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

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

- **%pdoc <object>**: Print (or run through a pager if too long) the docstring for an object. If the given object is a class, it will print both the class and the constructor docstrings.
- **%pdef <object>**: Print the definition header for any callable object. If the object is a class, print the constructor information.

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

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

Readline-based features

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

Command line completion

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

Search command history

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

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

Persistent command history across sessions

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

Autoindent

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

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

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

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

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

Customizing readline behavior

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

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

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

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

Session logging and restoring

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

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

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

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

%logstart [log_name [log_mode]]

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

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

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

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

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

System shell access

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

Manual capture of command output

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

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

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

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

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

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

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

```
In [5]: x=10
In [6]: y=20
In [13]: !echo $x+y
10+y
In [7]: !echo ${x+y}
30
```

Even object attributes can be expanded:

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

System command aliases

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

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

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

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

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

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

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

Recursive reload

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

Verbose and colored exception traceback printouts

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

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

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

Input caching system

IPython offers numbered prompts (In/Out) with input and output caching (also referred to as 'input history'). All input is saved and can be retrieved as variables (besides the usual arrow key recall), in addition to the %rep magic command that brings a history entry up for editing on the next command line.

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

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

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

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

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

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

You can also search ('grep') through your history by typing '%hist -g somestring'. This also searches through the so called *shadow history*, which remembers all the commands (apart from multiline code blocks) you have ever entered. Handy for searching for svn/bzr URL's, IP adrresses etc. You can bring shadow history entries listed by '%hist -g' up for editing (or re-execution by just pressing ENTER) with %rep command. Shadow history entries are not available as _iNUMBER variables, and they are identified by the

'0' prefix in %hist -g output. That is, history entry 12 is a normal history entry, but 0231 is a shadow history entry.

Shadow history was added because the readline history is inherently very unsafe - if you have multiple IPython sessions open, the last session to close will overwrite the history of previountly closed session. Likewise, if a crash occurs, history is never saved, whereas shadow history entries are added after entering every command (so a command executed in another IPython session is immediately available in other IPython sessions that are open).

To conserve space, a command can exist in shadow history only once - it doesn't make sense to store a common line like "cd .." a thousand times. The idea is mainly to provide a reliable place where valuable, hard-to-remember commands can always be retrieved, as opposed to providing an exact sequence of commands you have entered in actual order.

Because shadow history has all the commands you have ever executed, time taken by %hist -g will increase oven time. If it ever starts to take too long (or it ends up containing sensitive information like passwords), clear the shadow history by *%clear shadow_nuke*.

Time taken to add entries to shadow history should be negligible, but in any case, if you start noticing performance degradation after using IPython for a long time (or running a script that floods the shadow history!), you can 'compress' the shadow history by executing *%clear shadow_compress*. In practice, this should never be necessary in normal use.

Output caching system

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

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

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

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

These global variables are all stored in a global dictionary (not a list, since it only has entries for lines which returned a result) available under the names _oh and Out (similar to _ih and In). So the output from line 12 can be obtained as _12, Out[12] or _oh[12]. If you accidentally overwrite the Out variable you can recover it by typing 'Out=_oh' at the prompt.

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

Directory history

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

Automatic parentheses and quotes

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

Automatic parentheses

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

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

and the input will be translated to this:

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

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

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

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

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

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

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

but this will work:

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

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

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

Automatic quoting

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

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

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

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

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

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

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

3.2.3 IPython as your default Python environment

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

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

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

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

3.2.4 Embedding IPython

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

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

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

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

```
from IPython.Shell import IPShellEmbed

ipshell = IPShellEmbed()

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

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

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

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

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

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

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

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

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

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

# The basics to get you going:

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

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

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

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

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

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

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

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

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

# IPShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
# messages) is available as <instance>.IP.BANNER in case you want it.

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

# - A global startup banner.

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

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

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

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

# The code below illustrates all this.
```

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

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

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

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

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

# Reactivate the IPython shell
ipshell.set_dummy_mode(0)

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

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

print 'Main program finished. Bye!'

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

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

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

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

#-----------------------------------------------------------------------------
# This code loads IPython but modifies a few things if it detects it's running
```

```
# embedded in another IPython session (helps avoid confusion)

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

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

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

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

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

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

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

3.2.5 Using the Python debugger (pdb)

Running entire programs via pdb

pdb, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of pdb,

regardless of whether you have wrapped it into a 'main()' function or not. For this, simply type '%run -d myscript' at an IPython prompt. See the %run command's documentation (via '%run?' or in Sec. magic for more details, including how to control where pdb will stop execution first.

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

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

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

Automatic invocation of pdb on exceptions

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

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

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

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

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

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

3.2.6 Extensions for syntax processing

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

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

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

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

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

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

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

```
IPython profile: tutorial

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

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

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

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

Input of physical quantities with units

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

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

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

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

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

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

3.2.7 Threading support

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

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

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

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

Tk issues

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

I/O pitfalls

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

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

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

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

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

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

3.2.8 Interactive demos with IPython

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

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

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

Any python script can be run as a demo, but that does little more than showing
it on-screen, syntax-highlighted in one shot.  If you add a little simple
markup, you can stop at specified intervals and return to the ipython prompt,
resuming execution later.
"""

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

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

x = 1
y = 2

# <demo> --- stop ---

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

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

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

print 'z=',x

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

print 'bye!'
```

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

```
from IPython.demo import Demo

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

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

```
mydemo
```

and IPython will call it, executing each block. Demo objects can be restarted, you can move forward or back skipping blocks, re-execute the last block, etc. Simply use the Tab key on a demo object to see its methods, and call '?' on them to see their docstrings for more usage details. In addition, the demo module itself contains a comprehensive docstring, which you can access via:

```
from IPython import demo

demo?
```

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

3.2.9 Plotting with matplotlib

The matplotlib library (http://matplotlib.sourceforge.net http://matplotlib.sourceforge.net) provides high quality 2D plotting for Python. Matplotlib can produce plots on screen using a variety of GUI toolkits, including Tk, GTK and WXPython. It also provides a number of commands useful for scientific computing, all with a syntax compatible with that of the popular Matlab program.

IPython accepts the special option -pylab (see *here*). This configures it to support matplotlib, honoring the settings in the .matplotlibrc file. IPython will detect the user's choice of matplotlib GUI backend, and

automatically select the proper threading model to prevent blocking. It also sets matplotlib in interactive mode and modifies %run slightly, so that any matplotlib-based script can be executed using %run and the final show() command does not block the interactive shell.

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

3.3 IPython as a system shell

3.3.1 Overview

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

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

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

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

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

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

3.3.2 Aliases

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

3.3.3 Directory management

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

IPython provides its own builtin '%cd' magic command to move in the filesystem (the % is not required with automagic on). It also maintains a list of visited directories (use %dhist to see it) and allows direct switching to any of them. Type 'cd?' for more details.

%pushd, %popd and %dirs are provided for directory stack handling.

3.3.4 Enabled extensions

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

envpersist

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

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

ipy_which

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

Example:

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

ipy_app_completers

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

ipy_rehashdir

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

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

Now, you can execute any of those commams directly:

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

(the putty window opens).

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

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

mglob

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

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

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

3.3.5 Prompt customization

The sh profile uses the following prompt configurations:

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

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

3.3.6 String lists

String lists (IPython.genutils.SList) are handy way to process output from system commands. They are produced by `var = !cmd` syntax.

First, we acquire the output of 'ls -l':

```
[Q:doc/examples]|2> lines = !ls -l
 ==
['total 23',
 '-rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py',
 '-rw-rw-rw- 1 ville None 1927 Sep 30  2006 example-embed-short.py',
 '-rwxrwxrwx 1 ville None 4606 Sep  1 17:15 example-embed.py',
 '-rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py',
 '-rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py',
 '-rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py',
 '-rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc']
```

Now, let's take a look at the contents of 'lines' (the first number is the list element number):

```
[Q:doc/examples]|3> lines
                <3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py
2: -rw-rw-rw- 1 ville None 1927 Sep 30  2006 example-embed-short.py
3: -rwxrwxrwx 1 ville None 4606 Sep  1 17:15 example-embed.py
4: -rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py
5: -rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py
6: -rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py
7: -rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc
```

Now, let's filter out the 'embed' lines:

```
[Q:doc/examples]|4> l2 = lines.grep('embed',prune=1)
[Q:doc/examples]|5> l2
                <5> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py
2: -rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py
3: -rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py
4: -rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py
5: -rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc
```

Now, we want strings having just file names and permissions:

```
[Q:doc/examples]|6> l2.fields(8,0)
                <6> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total
1: example-demo.py -rw-rw-rw-
2: example-gnuplot.py -rwxrwxrwx
3: extension.py -rwxrwxrwx
4: seteditor.py -rwxrwxrwx
5: seteditor.pyc -rwxrwxrwx
```

Note how the line with 'total' does not raise IndexError.

If you want to split these (yielding lists), call fields() without arguments:

```
[Q:doc/examples]|7> _.fields()
                <7>
[['total'],
 ['example-demo.py', '-rw-rw-rw-'],
 ['example-gnuplot.py', '-rwxrwxrwx'],
 ['extension.py', '-rwxrwxrwx'],
 ['seteditor.py', '-rwxrwxrwx'],
 ['seteditor.pyc', '-rwxrwxrwx']]
```

If you want to pass these separated with spaces to a command (typical for lists if files), use the .s property:

```
[Q:doc/examples]|13> files = l2.fields(8).s
[Q:doc/examples]|14> files
                <14> 'example-demo.py example-gnuplot.py extension.py seteditor.py setedit
[Q:doc/examples]|15> ls $files
example-demo.py  example-gnuplot.py  extension.py  seteditor.py  seteditor.pyc
```

SLists are inherited from normal python lists, so every list method is available:

```
[Q:doc/examples]|21> lines.append('hey')
```

3.3.7 Real world example: remove all files outside version control

First, capture output of "hg status":

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

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

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

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

3.3.8 The .s, .n, .p properties

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

3.4 IPython extension API

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

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

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

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

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

3.4.1 Getting started

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

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

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

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

3.4.2 Getting a handle to the api

Put this in the start of your module:

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

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

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

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

3.4.3 Changing options

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

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

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

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

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

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

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


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

3.4.5 Accessing the IPython namespace

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

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

3.4.6 Defining new magic commands

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

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

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

Things to observe in this example:

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

3.4.7 Calling magic functions and system commands

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

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

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

3.4.8 Launching IPython instance from normal python code

Use ipapi.launch_new_instance() with an argument that specifies the namespace to use. This can be useful for trivially embedding IPython into your program. Here's an example of normal python program test.py

(‘‘without’’ an existing IPython session) that launches an IPython interpreter and regains control when the interpreter is exited:

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

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

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

3.4.9 Accessing unexposed functionality

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

For example:

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

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

3.4.10 Provided extensions

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

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

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

...
```

```
Docstring:
    File system operations

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

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

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

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

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

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

CHAPTER

FOUR

USING IPYTHON FOR PARALLEL COMPUTING

4.1 Overview and getting started

4.1.1 Introduction

This section gives an overview of IPython's sophisticated and powerful architecture for parallel and distributed computing. This architecture abstracts out parallelism in a very general way, which enables IPython to support many different styles of parallelism including:

- Single program, multiple data (SPMD) parallelism.
- Multiple program, multiple data (MPMD) parallelism.
- Message passing using MPI.
- Task farming.
- Data parallel.
- Combinations of these approaches.
- Custom user defined approaches.

Most importantly, IPython enables all types of parallel applications to be developed, executed, debugged and monitored *interactively*. Hence, the `I` in IPython. The following are some example usage cases for IPython:

- Quickly parallelize algorithms that are embarrassingly parallel using a number of simple approaches. Many simple things can be parallelized interactively in one or two lines of code.
- Steer traditional MPI applications on a supercomputer from an IPython session on your laptop.
- Analyze and visualize large datasets (that could be remote and/or distributed) interactively using IPython and tools like matplotlib/TVTK.
- Develop, test and debug new parallel algorithms (that may use MPI) interactively.
- Tie together multiple MPI jobs running on different systems into one giant distributed and parallel system.

- Start a parallel job on your cluster and then have a remote collaborator connect to it and pull back data into their local IPython session for plotting and analysis.
- Run a set of tasks on a set of CPUs using dynamic load balancing.

4.1.2 Architecture overview

The IPython architecture consists of three components:

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

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

IPython engine

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

IPython controller

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

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

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

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

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

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

Controller clients

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

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

Security

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

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

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

Currently, there are three FURL files that the controller creates:

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

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

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

More details of how these FURL files are used are given below.

A detailed description of the security model and its implementation in IPython can be found *here*.

4.1.3 Getting Started

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. Initially, it is best to simply start a controller and engines on a single host using the **ipcluster** command. To start a controller and 4 engines on you localhost, just do:

```
$ ipcluster local -n 4
```

More details about starting the IPython controller and engines can be found *here*

Once you have started the IPython controller and one or more engines, you are ready to use the engines to do something useful. To make sure everything is working correctly, try the following commands:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()

In [4]: mec.get_ids()
Out[4]: [0, 1, 2, 3]

In [5]: mec.execute('print "Hello World"')
Out[5]:
<Results List>
[0] In [1]: print "Hello World"
[0] Out[1]: Hello World

[1] In [1]: print "Hello World"
[1] Out[1]: Hello World

[2] In [1]: print "Hello World"
[2] Out[1]: Hello World

[3] In [1]: print "Hello World"
[3] Out[1]: Hello World
```

Remember, a client also needs to present a FURL file to the controller. How does this happen? When a multiengine client is created with no arguments, the client tries to find the corresponding FURL file in the local `~./ipython/security` directory. If it finds it, you are set. If you have put the FURL file in a different location or it has a different name, create the client like this:

```
mec = client.MultiEngineClient('/path/to/my/ipcontroller-mec.furl')
```

Same thing hold true of creating a task client:

```
tc = client.TaskClient('/path/to/my/ipcontroller-tc.furl')
```

You are now ready to learn more about the *MultiEngine* and *Task* interfaces to the controller.

Note: Don't forget that the engine, multiengine client and task client all have *different* furl files. You must move *each* of these around to an appropriate location so that the engines and clients can use them to connect to the controller.

4.2 Starting the IPython controller and engines

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. The controller and each engine can run on different machines or on the same machine. Because of this, there are many different possibilities.

Broadly speaking, there are two ways of going about starting a controller and engines:

- In an automated manner using the **ipcluster** command.
- In a more manual way using the **ipcontroller** and **ipengine** commands.

This document describes both of these methods. We recommend that new users start with the **ipcluster** command as it simplifies many common usage cases.

4.2.1 General considerations

Before delving into the details about how you can start a controller and engines using the various methods, we outline some of the general issues that come up when starting the controller and engines. These things come up no matter which method you use to start your IPython cluster.

Let's say that you want to start the controller on `host0` and engines on hosts `host1-hostn`. The following steps are then required:

1. Start the controller on `host0` by running **ipcontroller** on `host0`.
2. Move the FURL file (`ipcontroller-engine.furl`) created by the controller from `host0` to hosts `host1-hostn`.
3. Start the engines on hosts `host1-hostn` by running **ipengine**. This command has to be told where the FURL file (`ipcontroller-engine.furl`) is located.

At this point, the controller and engines will be connected. By default, the FURL files created by the controller are put into the `~/.ipython/security` directory. If the engines share a filesystem with the controller, step 2 can be skipped as the engines will automatically look at that location.

The final step required required to actually use the running controller from a client is to move the FURL files `ipcontroller-mec.furl` and `ipcontroller-tc.furl` from `host0` to the host where the clients will be run. If these file are put into the `~/.ipython/security` directory of the client's host, they will be found automatically. Otherwise, the full path to them has to be passed to the client's constructor.

4.2.2 Using ipcluster

The **ipcluster** command provides a simple way of starting a controller and engines in the following situations:

1. When the controller and engines are all run on localhost. This is useful for testing or running on a multicore computer.
2. When engines are started using the **mpirun** command that comes with most MPI [MPI] implementations

3. When engines are started using the PBS [PBS] batch system.
4. When engines are started using the SGE [SGE] batch system.
5. When engines are started using the LSF [LSF] batch system.
6. When the controller is started on localhost and the engines are started on remote nodes using **ssh**.

Note: It is also possible for advanced users to add support to **ipcluster** for starting controllers and engines using other methods (like Sun's Grid Engine for example).

Note: Currently **ipcluster** requires that the `~/.ipython/security` directory live on a shared filesystem that is seen by both the controller and engines. If you don't have a shared file system you will need to use **ipcontroller** and **ipengine** directly. This constraint can be relaxed if you are using the **ssh** method to start the cluster.

Underneath the hood, **ipcluster** just uses **ipcontroller** and **ipengine** to perform the steps described above.

Using ipcluster in local mode

To start one controller and 4 engines on localhost, just do:

```
$ ipcluster local -n 4
```

To see other command line options for the local mode, do:

```
$ ipcluster local -h
```

Using ipcluster in mpiexec/mpirun mode

The mpiexec/mpirun mode is useful if you:

1. Have MPI installed.
2. Your systems are configured to use the **mpiexec** or **mpirun** commands to start MPI processes.

Note: The preferred command to use is **mpiexec**. However, we also support **mpirun** for backwards compatibility. The underlying logic used is exactly the same, the only difference being the name of the command line program that is called.

If these are satisfied, you can start an IPython cluster using:

```
$ ipcluster mpiexec -n 4
```

This does the following:

1. Starts the IPython controller on current host.
2. Uses **mpiexec** to start 4 engines.

On newer MPI implementations (such as OpenMPI), this will work even if you don't make any calls to MPI or call `MPI_Init()`. However, older MPI implementations actually require each process to call `MPI_Init()` upon starting. The easiest way of having this done is to install the mpi4py [mpi4py] package and then call ipcluster with the `--mpi` option:

```
$ ipcluster mpiexec -n 4 --mpi=mpi4py
```

Unfortunately, even this won't work for some MPI implementations. If you are having problems with this, you will likely have to use a custom Python executable that itself calls `MPI_Init()` at the appropriate time. Fortunately, mpi4py comes with such a custom Python executable that is easy to install and use. However, this custom Python executable approach will not work with **ipcluster** currently.

Additional command line options for this mode can be found by doing:

```
$ ipcluster mpiexec -h
```

More details on using MPI with IPython can be found *here*.

Using ipcluster in PBS mode

The PBS mode uses the Portable Batch System [PBS] to start the engines.

To start an ipcluster using the Portable Batch System:

```
$ ipcluster pbs -n 12
```

The above command will launch a PBS job array with 12 tasks using the default queue. If you would like to submit the job to a different queue use the -q option:

```
$ ipcluster pbs -n 12 -q hpcqueue
```

By default, ipcluster will generate and submit a job script to launch the engines. However, if you need to use your own job script use the -s option:

```
$ ipcluster pbs -n 12 -q hpcqueue -s mypbscript.sh
```

For example the default autogenerated script looks like:

```
#PBS -q hpcqueue
#!/bin/sh
#PBS -V
#PBS -t 1-12
#PBS -N ipengine
eid=$(($PBS_ARRAYID - 1))
ipengine --logfile=ipengine${eid}.log
```

Note: ipcluster relies on using PBS job arrays to start the engines. If you specify your own job script without specifying the job array settings ipcluster will automatically add the job array settings (#PBS -t 1-N) to your script.

Additional command line options for this mode can be found by doing:

```
$ ipcluster pbs -h
```

Using ipcluster in SGE mode

The SGE mode uses the Sun Grid Engine [SGE] to start the engines.

To start an ipcluster using Sun Grid Engine:

```
$ ipcluster sge -n 12
```

The above command will launch an SGE job array with 12 tasks using the default queue. If you would like to submit the job to a different queue use the -q option:

```
$ ipcluster sge -n 12 -q hpcqueue
```

By default, ipcluster will generate and submit a job script to launch the engines. However, if you need to use your own job script use the -s option:

```
$ ipcluster sge -n 12 -q hpcqueue -s mysgescript.sh
```

For example the default autogenerated script looks like:

```
#$ -q hpcqueue
#$ -V
#$ -S /bin/sh
#$ -t 1-12
#$ -N ipengine
eid=$(($SGE_TASK_ID - 1))
ipengine --logfile=ipengine${eid}.log    #$ -V
```

Note: ipcluster relies on using SGE job arrays to start the engines. If you specify your own job script without specifying the job array settings ipcluster will automatically add the job array settings (#$ -t 1-N) to your script.

Additional command line options for this mode can be found by doing:

```
$ ipcluster sge -h
```

Using ipcluster in LSF mode

The LSF mode uses the Load Sharing Facility [LSF] to start the engines.

To start an ipcluster using the Load Sharing Facility:

```
$ ipcluster lsf -n 12
```

The above command will launch an LSF job array with 12 tasks using the default queue. If you would like to submit the job to a different queue use the -q option:

```
$ ipcluster lsf -n 12 -q hpcqueue
```

By default, ipcluster will generate and submit a job script to launch the engines. However, if you need to use your own job script use the -s option:

```
$ ipcluster lsf -n 12 -q hpcqueue -s mylsfscript.sh
```

For example the default autogenerated script looks like:

```
#BSUB -q hpcqueue
#!/bin/sh
#BSUB -J ipengine[1-12]
```

```
eid=$(($LSB_JOBINDEX - 1))
ipengine --logfile=ipengine${eid}.log
```

Note: ipcluster relies on using LSF job arrays to start the engines. If you specify your own job script without specifying the job array settings ipcluster will automatically add the job array settings (#BSUB -J ipengine[1-N]) to your script.

Additional command line options for this mode can be found by doing:

```
$ ipcluster lsf -h
```

Using ipcluster in SSH mode

The SSH mode uses **ssh** to execute **ipengine** on remote nodes and the **ipcontroller** on localhost.

When using using this mode it highly recommended that you have set up SSH keys and are using ssh-agent [SSH] for password-less logins.

To use this mode you need a python file describing the cluster, here is an example of such a "clusterfile":

```
send_furl = True
engines = { 'host1.example.com' : 2,
            'host2.example.com' : 5,
            'host3.example.com' : 1,
            'host4.example.com' : 8 }
```

Since this is a regular python file usual python syntax applies. Things to note:

- The *engines* dict, where the keys is the host we want to run engines on and the value is the number of engines to run on that host.
- send_furl can either be *True* or *False*, if *True* it will copy over the furl needed for **ipengine** to each host.

The `--clusterfile` command line option lets you specify the file to use for the cluster definition. Once you have your cluster file and you can **ssh** into the remote hosts with out an password you are ready to start your cluster like so:

```
$ ipcluster ssh --clusterfile /path/to/my/clusterfile.py
```

Two helper shell scripts are used to start and stop **ipengine** on remote hosts:

- sshx.sh
- engine_killer.sh

Defaults for both of these are contained in the source code for **ipcluster**. The default scripts are written to a local file in a tmep directory and then copied to a temp directory on the remote host and executed from there. On most Unix, Linux and OS X systems this is /tmp.

The default sshx.sh is the following:

```
#!/bin/sh
"$@" &> /dev/null &
echo $!
```

If you want to use a custom sshx.sh script you need to use the `--sshx` option and specify the file to use. Using a custom sshx.sh file could be helpful when you need to setup the environment on the remote host before executing **ipengine**.

For a detailed options list:

```
$ ipcluster ssh -h
```

Current limitations of the SSH mode of **ipcluster** are:

- Untested on Windows. Would require a working **ssh** on Windows. Also, we are using shell scripts to setup and execute commands on remote hosts.
- **ipcontroller** is started on localhost, with no option to start it on a remote node.

4.2.3 Using the ipcontroller and ipengine commands

It is also possible to use the **ipcontroller** and **ipengine** commands to start your controller and engines. This approach gives you full control over all aspects of the startup process.

Starting the controller and engine on your local machine

To use **ipcontroller** and **ipengine** to start things on your local machine, do the following.

First start the controller:

```
$ ipcontroller
```

Next, start however many instances of the engine you want using (repeatedly) the command:

```
$ ipengine
```

The engines should start and automatically connect to the controller using the FURL files in `~./ipython/security`. You are now ready to use the controller and engines from IPython.

> **Warning:** The order of the above operations is very important. You *must* start the controller before the engines, since the engines connect to the controller as they get started.

Note: On some platforms (OS X), to put the controller and engine into the background you may need to give these commands in the form `(ipcontroller &)` and `(ipengine &)` (with the parentheses) for them to work properly.

Starting the controller and engines on different hosts

When the controller and engines are running on different hosts, things are slightly more complicated, but the underlying ideas are the same:

1. Start the controller on a host using **ipcontroller**.
2. Copy `ipcontroller-engine.furl` from `~./ipython/security` on the controller's host to the host where the engines will run.
3. Use **ipengine** on the engine's hosts to start the engines.

The only thing you have to be careful of is to tell **ipengine** where the `ipcontroller-engine.furl` file is located. There are two ways you can do this:

- Put `ipcontroller-engine.furl` in the `~./ipython/security` directory on the engine's host, where it will be found automatically.
- Call **ipengine** with the `--furl-file=full_path_to_the_file` flag.

The `--furl-file` flag works like this:

```
$ ipengine --furl-file=/path/to/my/ipcontroller-engine.furl
```

Note: If the controller's and engine's hosts all have a shared file system (`~./ipython/security` is the same on all of them), then things will just work!

Make FURL files persistent

At fist glance it may seem that that managing the FURL files is a bit annoying. Going back to the house and key analogy, copying the FURL around each time you start the controller is like having to make a new key every time you want to unlock the door and enter your house. As with your house, you want to be able to create the key (or FURL file) once, and then simply use it at any point in the future.

This is possible, but before you do this, you **must** remove any old FURL files in the `~/.ipython/security` directory.

Warning: You **must** remove old FURL files before using persistent FURL files.

Then, The only thing you have to do is decide what ports the controller will listen on for the engines and clients. This is done as follows:

```
$ ipcontroller -r --client-port=10101 --engine-port=10102
```

These options also work with all of the various modes of **ipcluster**:

```
$ ipcluster local -n 2 -r --client-port=10101 --engine-port=10102
```

Then, just copy the furl files over the first time and you are set. You can start and stop the controller and engines any many times as you want in the future, just make sure to tell the controller to use the *same* ports.

Note: You may ask the question: what ports does the controller listen on if you don't tell is to use specific ones? The default is to use high random port numbers. We do this for two reasons: i) to increase security through obscurity and ii) to multiple controllers on a given host to start and automatically use different ports.

Log files

All of the components of IPython have log files associated with them. These log files can be extremely useful in debugging problems with IPython and can be found in the directory `~/.ipython/log`. Sending the log files to us will often help us to debug any problems.

4.3 IPython's multiengine interface

The multiengine interface represents one possible way of working with a set of IPython engines. The basic idea behind the multiengine interface is that the capabilities of each engine are directly and explicitly exposed to the user. Thus, in the multiengine interface, each engine is given an id that is used to identify the engine and give it work to do. This interface is very intuitive and is designed with interactive usage in mind, and is thus the best place for new users of IPython to begin.

4.3.1 Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **ipcluster** command:

```
$ ipcluster local -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

4.3.2 Creating a `MultiEngineClient` instance

The first step is to import the IPython `IPython.kernel.client` module and then create a `MultiEngineClient` instance:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()
```

This form assumes that the `ipcontroller-mec.furl` is in the `~./ipython/security` directory on the client's host. If not, the location of the FURL file must be given as an argument to the constructor:

```
In [2]: mec = client.MultiEngineClient('/path/to/my/ipcontroller-mec.furl')
```

To make sure there are engines connected to the controller, use can get a list of engine ids:

```
In [3]: mec.get_ids()
Out[3]: [0, 1, 2, 3]
```

Here we see that there are four engines ready to do work for us.

4.3.3 Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. The multiengine interface provides two simple ways of accomplishing this: a parallel version of `map()` and `@parallel` function decorator.

Parallel map

Python's builtin `map()` functions allows a function to be applied to a sequence element-by-element. This type of code is typically trivial to parallelize. In fact, the multiengine interface in IPython already has a parallel version of `map()` that works just like its serial counterpart:

```
In [63]: serial_result = map(lambda x:x**10, range(32))

In [64]: parallel_result = mec.map(lambda x:x**10, range(32))

In [65]: serial_result==parallel_result
Out[65]: True
```

Note: The multiengine interface version of `map()` does not do any load balancing. For a load balanced version, see the task interface.

See Also:

The `map()` method has a number of options that can be controlled by the `mapper()` method. See its docstring for more information.

Parallel function decorator

Parallel functions are just like normal function, but they can be called on sequences and *in parallel*. The multiengine interface provides a decorator that turns any Python function into a parallel function:

```
In [10]: @mec.parallel()
   ....: def f(x):
   ....:     return 10.0*x**4
   ....:

In [11]: f(range(32))     # this is done in parallel
Out[11]:
[0.0,10.0,160.0,...]
```

See the docstring for the `parallel()` decorator for options.

4.3.4 Running Python commands

The most basic type of operation that can be performed on the engines is to execute Python code. Executing Python code can be done in blocking or non-blocking mode (blocking is default) using the `execute()` method.

Blocking execution

In blocking mode, the `MultiEngineClient` object (called `mec` in these examples) submits the command to the controller, which places the command in the engines' queues for execution. The `execute()` call then blocks until the engines are done executing the command:

```
# The default is to run on all engines
In [4]: mec.execute('a=5')
Out[4]:
<Results List>
[0] In [1]: a=5
[1] In [1]: a=5
[2] In [1]: a=5
[3] In [1]: a=5

In [5]: mec.execute('b=10')
Out[5]:
<Results List>
[0] In [2]: b=10
[1] In [2]: b=10
[2] In [2]: b=10
[3] In [2]: b=10
```

Python commands can be executed on specific engines by calling execute using the `targets` keyword argument:

```
In [6]: mec.execute('c=a+b',targets=[0,2])
Out[6]:
<Results List>
[0] In [3]: c=a+b
[2] In [3]: c=a+b


In [7]: mec.execute('c=a-b',targets=[1,3])
Out[7]:
<Results List>
[1] In [3]: c=a-b
[3] In [3]: c=a-b


In [8]: mec.execute('print c')
Out[8]:
<Results List>
[0] In [4]: print c
[0] Out[4]: 15

[1] In [4]: print c
[1] Out[4]: -5

[2] In [4]: print c
[2] Out[4]: 15

[3] In [4]: print c
```

```
[3] Out[4]: -5
```

This example also shows one of the most important things about the IPython engines: they have a persistent user namespaces. The `execute()` method returns a Python `dict` that contains useful information:

```
In [9]: result_dict = mec.execute('d=10; print d')

In [10]: for r in result_dict:
   ....:     print r
   ....:
   ....:
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 0, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 1, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 2, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 3, 's
```

Non-blocking execution

In non-blocking mode, `execute()` submits the command to be executed and then returns a `PendingResult` object immediately. The `PendingResult` object gives you a way of getting a result at a later time through its `get_result()` method or `r` attribute. This allows you to quickly submit long running commands without blocking your local Python/IPython session:

```
# In blocking mode
In [6]: mec.execute('import time')
Out[6]:
<Results List>
[0] In [1]: import time
[1] In [1]: import time
[2] In [1]: import time
[3] In [1]: import time

# In non-blocking mode
In [7]: pr = mec.execute('time.sleep(10)',block=False)

# Now block for the result
In [8]: pr.get_result()
Out[8]:
<Results List>
[0] In [2]: time.sleep(10)
[1] In [2]: time.sleep(10)
[2] In [2]: time.sleep(10)
[3] In [2]: time.sleep(10)

# Again in non-blocking mode
In [9]: pr = mec.execute('time.sleep(10)',block=False)

# Poll to see if the result is ready
In [10]: pr.get_result(block=False)

# A shorthand for get_result(block=True)
In [11]: pr.r
```

```
Out[11]:
<Results List>
[0] In [3]: time.sleep(10)
[1] In [3]: time.sleep(10)
[2] In [3]: time.sleep(10)
[3] In [3]: time.sleep(10)
```

Often, it is desirable to wait until a set of `PendingResult` objects are done. For this, there is a the method `barrier()`. This method takes a tuple of `PendingResult` objects and blocks until all of the associated results are ready:

```
In [72]: mec.block=False

# A trivial list of PendingResults objects
In [73]: pr_list = [mec.execute('time.sleep(3)') for i in range(10)]

# Wait until all of them are done
In [74]: mec.barrier(pr_list)

# Then, their results are ready using get_result or the r attribute
In [75]: pr_list[0].r
Out[75]:
<Results List>
[0] In [20]: time.sleep(3)
[1] In [19]: time.sleep(3)
[2] In [20]: time.sleep(3)
[3] In [19]: time.sleep(3)
```

The `block` and `targets` keyword arguments and attributes

Most methods in the multiengine interface (like `execute()`) accept `block` and `targets` as keyword arguments. As we have seen above, these keyword arguments control the blocking mode and which engines the command is applied to. The `MultiEngineClient` class also has `block` and `targets` attributes that control the default behavior when the keyword arguments are not provided. Thus the following logic is used for `block` and `targets`:

- If no keyword argument is provided, the instance attributes are used.
- Keyword argument, if provided override the instance attributes.

The following examples demonstrate how to use the instance attributes:

```
In [16]: mec.targets = [0,2]

In [17]: mec.block = False

In [18]: pr = mec.execute('a=5')

In [19]: pr.r
Out[19]:
<Results List>
[0] In [6]: a=5
```

```
[2] In [6]: a=5

# Note targets='all' means all engines
In [20]: mec.targets = 'all'

In [21]: mec.block = True

In [22]: mec.execute('b=10; print b')
Out[22]:
<Results List>
[0] In [7]: b=10; print b
[0] Out[7]: 10

[1] In [6]: b=10; print b
[1] Out[6]: 10

[2] In [7]: b=10; print b
[2] Out[7]: 10

[3] In [6]: b=10; print b
[3] Out[6]: 10
```

The `block` and `targets` instance attributes also determine the behavior of the parallel magic commands.

Parallel magic commands

We provide a few IPython magic commands (`%px`, `%autopx` and `%result`) that make it more pleasant to execute Python commands on the engines interactively. These are simply shortcuts to `execute()` and `get_result()`. The `%px` magic executes a single Python command on the engines specified by the `targets` attribute of the `MultiEngineClient` instance (by default this is `'all'`):

```
# Make this MultiEngineClient active for parallel magic commands
In [23]: mec.activate()

In [24]: mec.block=True

In [25]: import numpy

In [26]: %px import numpy
Executing command on Controller
Out[26]:
<Results List>
[0] In [8]: import numpy
[1] In [7]: import numpy
[2] In [8]: import numpy
[3] In [7]: import numpy


In [27]: %px a = numpy.random.rand(2,2)
Executing command on Controller
Out[27]:
<Results List>
```

```
[0] In [9]: a = numpy.random.rand(2,2)
[1] In [8]: a = numpy.random.rand(2,2)
[2] In [9]: a = numpy.random.rand(2,2)
[3] In [8]: a = numpy.random.rand(2,2)


In [28]: %px print numpy.linalg.eigvals(a)
Executing command on Controller
Out[28]:
<Results List>
[0] In [10]: print numpy.linalg.eigvals(a)
[0] Out[10]: [ 1.28167017  0.14197338]

[1] In [9]: print numpy.linalg.eigvals(a)
[1] Out[9]: [-0.14093616  1.27877273]

[2] In [10]: print numpy.linalg.eigvals(a)
[2] Out[10]: [-0.37023573  1.06779409]

[3] In [9]: print numpy.linalg.eigvals(a)
[3] Out[9]: [ 0.83664764 -0.25602658]
```

The `%result` magic gets and prints the stdin/stdout/stderr of the last command executed on each engine. It is simply a shortcut to the `get_result()` method:

```
In [29]: %result
Out[29]:
<Results List>
[0] In [10]: print numpy.linalg.eigvals(a)
[0] Out[10]: [ 1.28167017  0.14197338]

[1] In [9]: print numpy.linalg.eigvals(a)
[1] Out[9]: [-0.14093616  1.27877273]

[2] In [10]: print numpy.linalg.eigvals(a)
[2] Out[10]: [-0.37023573  1.06779409]

[3] In [9]: print numpy.linalg.eigvals(a)
[3] Out[9]: [ 0.83664764 -0.25602658]
```

The `%autopx` magic switches to a mode where everything you type is executed on the engines given by the `targets` attribute:

```
In [30]: mec.block=False

In [31]: %autopx
Auto Parallel Enabled
Type %autopx to disable

In [32]: max_evals = []
<IPython.kernel.multiengineclient.PendingResult object at 0x17b8a70>

In [33]: for i in range(100):
   ....:     a = numpy.random.rand(10,10)
```

```
   ....:     a = a+a.transpose()
   ....:     evals = numpy.linalg.eigvals(a)
   ....:     max_evals.append(evals[0].real)
   ....:
   ....:
<IPython.kernel.multienginecient.PendingResult object at 0x17af8f0>

In [34]: %autopx
Auto Parallel Disabled

In [35]: mec.block=True

In [36]: px print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
Executing command on Controller
Out[36]:
<Results List>
[0] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[0] Out[13]: Average max eigenvalue is:  10.1387247332

[1] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[1] Out[12]: Average max eigenvalue is:  10.2076902286

[2] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[2] Out[13]: Average max eigenvalue is:  10.1891484655

[3] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[3] Out[12]: Average max eigenvalue is:  10.1158837784
```

4.3.5 Moving Python objects around

In addition to executing code on engines, you can transfer Python objects to and from your IPython session and the engines. In IPython, these operations are called `push()` (sending an object to the engines) and `pull()` (getting an object from the engines).

Basic push and pull

Here are some examples of how you use `push()` and `pull()`:

```
In [38]: mec.push(dict(a=1.03234,b=3453))
Out[38]: [None, None, None, None]

In [39]: mec.pull('a')
Out[39]: [1.03234, 1.03234, 1.03234, 1.03234]

In [40]: mec.pull('b',targets=0)
Out[40]: [3453]

In [41]: mec.pull(('a','b'))
Out[41]: [[1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453]]
```

```
In [42]: mec.zip_pull(('a','b'))
Out[42]: [(1.03234, 1.03234, 1.03234, 1.03234), (3453, 3453, 3453, 3453)]

In [43]: mec.push(dict(c='speed'))
Out[43]: [None, None, None, None]

In [44]: %px print c
Executing command on Controller
Out[44]:
<Results List>
[0] In [14]: print c
[0] Out[14]: speed

[1] In [13]: print c
[1] Out[13]: speed

[2] In [14]: print c
[2] Out[14]: speed

[3] In [13]: print c
[3] Out[13]: speed
```

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

```
In [47]: mec.block=False

In [48]: pr = mec.pull('a')

In [49]: pr.r
Out[49]: [1.03234, 1.03234, 1.03234, 1.03234]
```

Push and pull for functions

Functions can also be pushed and pulled using `push_function()` and `pull_function()`:

```
In [52]: mec.block=True

    In [53]: def f(x):
       ....:     return 2.0*x**4
       ....:

    In [54]: mec.push_function(dict(f=f))
    Out[54]: [None, None, None, None]

    In [55]: mec.execute('y = f(4.0)')
    Out[55]:
    <Results List>
    [0] In [15]: y = f(4.0)
    [1] In [14]: y = f(4.0)
    [2] In [15]: y = f(4.0)
    [3] In [14]: y = f(4.0)
```

```
In [56]: px print y
Executing command on Controller
Out[56]:
<Results List>
[0] In [16]: print y
[0] Out[16]: 512.0

[1] In [15]: print y
[1] Out[15]: 512.0

[2] In [16]: print y
[2] Out[16]: 512.0

[3] In [15]: print y
[3] Out[15]: 512.0
```

Dictionary interface

As a shorthand to `push()` and `pull()`, the `MultiEngineClient` class implements some of the Python dictionary interface. This make the remote namespaces of the engines appear as a local dictionary. Underneath, this uses `push()` and `pull()`:

```
In [50]: mec.block=True

In [51]: mec['a']=['foo','bar']

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

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `MultiEngineClient` class, `scatter()` is from the interactive IPython session to the engines and `gather()` is from the engines back to the interactive IPython session. For scatter/gather operations between engines, MPI should be used:

```
In [58]: mec.scatter('a',range(16))
Out[58]: [None, None, None, None]

In [59]: px print a
Executing command on Controller
Out[59]:
<Results List>
[0] In [17]: print a
[0] Out[17]: [0, 1, 2, 3]

[1] In [16]: print a
[1] Out[16]: [4, 5, 6, 7]
```

```
[2] In [17]: print a
[2] Out[17]: [8, 9, 10, 11]

[3] In [16]: print a
[3] Out[16]: [12, 13, 14, 15]


In [60]: mec.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

4.3.6 Other things to look at

How to do parallel list comprehensions

In many cases list comprehensions are nicer than using the map function. While we don't have fully parallel list comprehensions, it is simple to get the basic effect using `scatter()` and `gather()`:

```
In [66]: mec.scatter('x',range(64))
Out[66]: [None, None, None, None]

In [67]: px y = [i**10 for i in x]
Executing command on Controller
Out[67]:
<Results List>
[0] In [19]: y = [i**10 for i in x]
[1] In [18]: y = [i**10 for i in x]
[2] In [19]: y = [i**10 for i in x]
[3] In [18]: y = [i**10 for i in x]


In [68]: y = mec.gather('y')

In [69]: print y
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824,...]
```

Parallel exceptions

In the multiengine interface, parallel commands can raise Python exceptions, just like serial commands. But, it is a little subtle, because a single parallel command can actually raise multiple exceptions (one for each engine the command was run on). To express this idea, the MultiEngine interface has a `CompositeError` exception class that will be raised in most cases. The `CompositeError` class is a special type of exception that wraps one or more other types of exceptions. Here is how it works:

```
In [76]: mec.block=True

In [77]: mec.execute('1/0')
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)
```

```
/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multienginecient.pyc in execute(self, lines, target
    432         targets, block = self._findTargetsAndBlock(targets, block)
    433         result = blockingCallFromThread(self.smultiengine.execute, lines,
--> 434             targets=targets, block=block)
    435         if block:
    436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
     72             result.raiseException()
     73         except Exception, e:
---> 74             raise e
     75     return result
     76

CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero
```

Notice how the error message printed when `CompositeError` is raised has information about the individual exceptions that were raised on each engine. If you want, you can even raise one of these original exceptions:

```
In [80]: try:
   ....:     mec.execute('1/0')
   ....: except client.CompositeError, e:
   ....:     e.raise_exception()
   ....:
   ....:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/error.pyc in raise_exception(self, excid)
    156             raise IndexError("an exception with index %i does not exist"%excid)
    157         else:
--> 158             raise et, ev, etb
    159
    160 def collect_exceptions(rlist, method):

ZeroDivisionError: integer division or modulo by zero
```

If you are working in IPython, you can simple type `%debug` after one of these `CompositeError` exceptions is raised, and inspect the exception instance:

```
In [81]: mec.execute('1/0')
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)
```

```
/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in execute(self, lines, target
    432         targets, block = self._findTargetsAndBlock(targets, block)
    433         result = blockingCallFromThread(self.smultiengine.execute, lines,
--> 434             targets=targets, block=block)
    435         if block:
    436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
     72             result.raiseException()
     73         except Exception, e:
---> 74             raise e
     75     return result
     76

CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero

In [82]: %debug
>

/ipython1-client-r3021/ipython1/kernel/twistedutil.py(74)blockingCallFromThread()
     73         except Exception, e:
---> 74             raise e
     75     return result

ipdb> e.
e.__class__          e.__getitem__       e.__new__           e.__setstate__      e.args
e.__delattr__        e.__getslice__      e.__reduce__        e.__str__           e.elist
e.__dict__           e.__hash__          e.__reduce_ex__     e.__weakref__       e.message
e.__doc__            e.__init__          e.__repr__          e._get_engine_str   e.print_tra
e.__getattribute__   e.__module__        e.__setattr__       e._get_traceback    e.raise_exc
ipdb> e.print_tracebacks()
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero
```

```
[2:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero

[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero
```

Note: The above example appears to be broken right now because of a change in how we are using Twisted.

All of this same error handling magic even works in non-blocking mode:

```
In [83]: mec.block=False

In [84]: pr = mec.execute('1/0')

In [85]: pr.r
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in _get_r(self)
    170
    171     def _get_r(self):
--> 172         return self.get_result(block=True)
    173
    174     r = property(_get_r)

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_result(self, default, b
    131                 return self.result
    132         try:
--> 133             result = self.client.get_pending_deferred(self.result_id, block)
    134         except error.ResultNotCompleted:
    135             return default

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_pending_deferred(self,
    385
    386     def get_pending_deferred(self, deferredID, block):
--> 387         return blockingCallFromThread(self.smultiengine.get_pending_deferred, defe
    388
    389     def barrier(self, pendingResults):

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
     72             result.raiseException()
     73         except Exception, e:
```

```
---> 74             raise e
     75     return result
     76

CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero
```

4.4 The IPython task interface

The task interface to the controller presents the engines as a fault tolerant, dynamic load-balanced system or workers. Unlike the multiengine interface, in the task interface, the user have no direct access to individual engines. In some ways, this interface is simpler, but in other ways it is more powerful.

Best of all the user can use both of these interfaces running at the same time to take advantage or both of their strengths. When the user can break up the user's work into segments that do not depend on previous execution, the task interface is ideal. But it also has more power and flexibility, allowing the user to guide the distribution of jobs, without having to assign tasks to engines explicitly.

4.4.1 Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **ipcluster** command:

```
$ ipcluster local -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

4.4.2 Creating a `TaskClient` instance

The first step is to import the IPython `IPython.kernel.client` module and then create a `TaskClient` instance:

```
In [1]: from IPython.kernel import client

In [2]: tc = client.TaskClient()
```

This form assumes that the `ipcontroller-tc.furl` is in the `~./ipython/security` directory on the client's host. If not, the location of the FURL file must be given as an argument to the constructor:

```
In [2]: mec = client.TaskClient('/path/to/my/ipcontroller-tc.furl')
```

4.4.3 Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. Like the multiengine interface, the task interface provides two simple ways of accomplishing this: a parallel version of `map()` and `@parallel` function decorator. However, the verions in the task interface have one important difference: they are dynamically load balanced. Thus, if the execution time per item varies significantly, you should use the versions in the task interface.

Parallel map

The parallel `map()` in the task interface is similar to that in the multiengine interface:

```
In [63]: serial_result = map(lambda x:x**10, range(32))

In [64]: parallel_result = tc.map(lambda x:x**10, range(32))

In [65]: serial_result==parallel_result
Out[65]: True
```

Parallel function decorator

Parallel functions are just like normal function, but they can be called on sequences and *in parallel*. The multiengine interface provides a decorator that turns any Python function into a parallel function:

```
In [10]: @tc.parallel()
   ....: def f(x):
   ....:     return 10.0*x**4
   ....:

In [11]: f(range(32))     # this is done in parallel
Out[11]:
[0.0,10.0,160.0,...]
```

4.4.4 More details

The `TaskClient` has many more powerful features that allow quite a bit of flexibility in how tasks are defined and run. The next places to look are in the following classes:

- `IPython.kernel.client.TaskClient`
- `IPython.kernel.client.StringTask`
- `IPython.kernel.client.MapTask`

The following is an overview of how to use these classes together:

1. Create a `TaskClient`.
2. Create one or more instances of `StringTask` or `MapTask` to define your tasks.
3. Submit your tasks to using the `run()` method of your `TaskClient` instance.

4. Use `TaskClient.get_task_result()` to get the results of the tasks.

We are in the process of developing more detailed information about the task interface. For now, the docstrings of the `TaskClient`, `StringTask` and `MapTask` classes should be consulted.

4.5 Using MPI with IPython

Often, a parallel algorithm will require moving data between the engines. One way of accomplishing this is by doing a pull and then a push using the multiengine client. However, this will be slow as all the data has to go through the controller to the client and then back through the controller, to its final destination.

A much better way of moving data between engines is to use a message passing library, such as the Message Passing Interface (MPI) [MPI]. IPython's parallel computing architecture has been designed from the ground up to integrate with MPI. This document describes how to use MPI with IPython.

4.5.1 Additional installation requirements

If you want to use MPI with IPython, you will need to install:

- A standard MPI implementation such as OpenMPI [OpenMPI] or MPICH.
- The mpi4py [mpi4py] package.

Note: The mpi4py package is not a strict requirement. However, you need to have *some* way of calling MPI from Python. You also need some way of making sure that `MPI_Init()` is called when the IPython engines start up. There are a number of ways of doing this and a good number of associated subtleties. We highly recommend just using mpi4py as it takes care of most of these problems. If you want to do something different, let us know and we can help you get started.

4.5.2 Starting the engines with MPI enabled

To use code that calls MPI, there are typically two things that MPI requires.

1. The process that wants to call MPI must be started using **mpiexec** or a batch system (like PBS) that has MPI support.
2. Once the process starts, it must call `MPI_Init()`.

There are a couple of ways that you can start the IPython engines and get these things to happen.

Automatic starting using mpiexec and ipcluster

The easiest approach is to use the *mpiexec* mode of **ipcluster**, which will first start a controller and then a set of engines using **mpiexec**:

```
$ ipcluster mpiexec -n 4
```

This approach is best as interrupting **ipcluster** will automatically stop and clean up the controller and engines.

Manual starting using mpiexec

If you want to start the IPython engines using the **mpiexec**, just do:

```
$ mpiexec -n 4 ipengine --mpi=mpi4py
```

This requires that you already have a controller running and that the FURL files for the engines are in place. We also have built in support for PyTrilinos [PyTrilinos], which can be used (assuming is installed) by starting the engines with:

```
mpiexec -n 4 ipengine --mpi=pytrilinos
```

Automatic starting using PBS and ipcluster

The **ipcluster** command also has built-in integration with PBS. For more information on this approach, see our documentation on *ipcluster*.

4.5.3 Actually using MPI

Once the engines are running with MPI enabled, you are ready to go. You can now call any code that uses MPI in the IPython engines. And, all of this can be done interactively. Here we show a simple example that uses mpi4py [mpi4py].

First, lets define a simply function that uses MPI to calculate the sum of a distributed array. Save the following text in a file called `psum.py`:

```
from mpi4py import MPI
import numpy as np

def psum(a):
    s = np.sum(a)
    return MPI.COMM_WORLD.Allreduce(s,MPI.SUM)
```

Now, start an IPython cluster in the same directory as `psum.py`:

```
$ ipcluster mpiexec -n 4
```

Finally, connect to the cluster and use this function interactively. In this case, we create a random array on each engine and sum up all the random arrays using our `psum()` function:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()

In [3]: mec.activate()

In [4]: px import numpy as np
Parallel execution on engines: all
Out[4]:
<Results List>
[0] In [13]: import numpy as np
```

```
[1] In [13]: import numpy as np
[2] In [13]: import numpy as np
[3] In [13]: import numpy as np

In [6]: px a = np.random.rand(100)
Parallel execution on engines: all
Out[6]:
<Results List>
[0] In [15]: a = np.random.rand(100)
[1] In [15]: a = np.random.rand(100)
[2] In [15]: a = np.random.rand(100)
[3] In [15]: a = np.random.rand(100)

In [7]: px from psum import psum
Parallel execution on engines: all
Out[7]:
<Results List>
[0] In [16]: from psum import psum
[1] In [16]: from psum import psum
[2] In [16]: from psum import psum
[3] In [16]: from psum import psum

In [8]: px s = psum(a)
Parallel execution on engines: all
Out[8]:
<Results List>
[0] In [17]: s = psum(a)
[1] In [17]: s = psum(a)
[2] In [17]: s = psum(a)
[3] In [17]: s = psum(a)

In [9]: px print s
Parallel execution on engines: all
Out[9]:
<Results List>
[0] In [18]: print s
[0] Out[18]: 187.451545803

[1] In [18]: print s
[1] Out[18]: 187.451545803

[2] In [18]: print s
[2] Out[18]: 187.451545803

[3] In [18]: print s
[3] Out[18]: 187.451545803
```

Any Python code that makes calls to MPI can be used in this manner, including compiled C, C++ and Fortran libraries that have been exposed to Python.

4.6 Security details of IPython

IPython's `IPython.kernel` package exposes the full power of the Python interpreter over a TCP/IP network for the purposes of parallel computing. This feature brings up the important question of IPython's security model. This document gives details about this model and how it is implemented in IPython's architecture.

4.6.1 Processs and network topology

To enable parallel computing, IPython has a number of different processes that run. These processes are discussed at length in the IPython documentation and are summarized here:

- The IPython *engine*. This process is a full blown Python interpreter in which user code is executed. Multiple engines are started to make parallel computing possible.
- The IPython *controller*. This process manages a set of engines, maintaining a queue for each and presenting an asynchronous interface to the set of engines.
- The IPython *client*. This process is typically an interactive Python process that is used to coordinate the engines to get a parallel computation done.

Collectively, these three processes are called the IPython *kernel*.

These three processes communicate over TCP/IP connections with a well defined topology. The IPython controller is the only process that listens on TCP/IP sockets. Upon starting, an engine connects to a controller and registers itself with the controller. These engine/controller TCP/IP connections persist for the lifetime of each engine.

The IPython client also connects to the controller using one or more TCP/IP connections. These connections persist for the lifetime of the client only.

A given IPython controller and set of engines typically has a relatively short lifetime. Typically this lifetime corresponds to the duration of a single parallel simulation performed by a single user. Finally, the controller, engines and client processes typically execute with the permissions of that same user. More specifically, the controller and engines are *not* executed as root or with any other superuser permissions.

4.6.2 Application logic

When running the IPython kernel to perform a parallel computation, a user utilizes the IPython client to send Python commands and data through the IPython controller to the IPython engines, where those commands are executed and the data processed. The design of IPython ensures that the client is the only access point for the capabilities of the engines. That is, the only way of addressing the engines is through a client.

A user can utilize the client to instruct the IPython engines to execute arbitrary Python commands. These Python commands can include calls to the system shell, access the filesystem, etc., as required by the user's application code. From this perspective, when a user runs an IPython engine on a host, that engine has the same capabilities and permissions as the user themselves (as if they were logged onto the engine's host with a terminal).

4.6.3 Secure network connections

Overview

All TCP/IP connections between the client and controller as well as the engines and controller are fully encrypted and authenticated. This section describes the details of the encryption and authentication approached used within IPython.

IPython uses the Foolscap network protocol [Foolscap] for all communications between processes. Thus, the details of IPython's security model are directly related to those of Foolscap. Thus, much of the following discussion is actually just a discussion of the security that is built in to Foolscap.

Encryption

For encryption purposes, IPython and Foolscap use the well known Secure Socket Layer (SSL) protocol [RFC5246]. We use the implementation of this protocol provided by the OpenSSL project through the pyOpenSSL [pyOpenSSL] Python bindings to OpenSSL.

Authentication

IPython clients and engines must also authenticate themselves with the controller. This is handled in a capabilities based security model [Capability]. In this model, the controller creates a strong cryptographic key or token that represents each set of capability that the controller offers. Any party who has this key and presents it to the controller has full access to the corresponding capabilities of the controller. This model is analogous to using a physical key to gain access to physical items (capabilities) behind a locked door.

For a capabilities based authentication system to prevent unauthorized access, two things must be ensured:

- The keys must be cryptographically strong. Otherwise attackers could gain access by a simple brute force key guessing attack.
- The actual keys must be distributed only to authorized parties.

The keys in Foolscap are called Foolscap URL's or FURLs. The following section gives details about how these FURLs are created in Foolscap. The IPython controller creates a number of FURLs for different purposes:

- One FURL that grants IPython engines access to the controller. Also implicit in this access is permission to execute code sent by an authenticated IPython client.
- Two or more FURLs that grant IPython clients access to the controller. Implicit in this access is permission to give the controller's engine code to execute.

Upon starting, the controller creates these different FURLS and writes them files in the user-read-only directory `$HOME/.ipython/security`. Thus, only the user who starts the controller has access to the FURLs.

For an IPython client or engine to authenticate with a controller, it must present the appropriate FURL to the controller upon connecting. If the FURL matches what the controller expects for a given capability, access is granted. If not, access is denied. The exchange of FURLs is done after encrypted communications channels have been established to prevent attackers from capturing them.

Note: The FURL is similar to an unsigned private key in SSH.

Details of the Foolscap handshake

In this section we detail the precise security handshake that takes place at the beginning of any network connection in IPython. For the purposes of this discussion, the SERVER is the IPython controller process and the CLIENT is the IPython engine or client process.

Upon starting, all IPython processes do the following:

1. Create a public key x509 certificate (ISO/IEC 9594).
2. Create a hash of the contents of the certificate using the SHA-1 algorithm. The base-32 encoded version of this hash is saved by the process as its process id (actually in Foolscap, this is the Tub id, but here refer to it as the process id).

Upon starting, the IPython controller also does the following:

1. Save the x509 certificate to disk in a secure location. The CLIENT certificate is never saved to disk.
2. Create a FURL for each capability that the controller has. There are separate capabilities the controller offers for clients and engines. The FURL is created using: a) the process id of the SERVER, b) the IP address and port the SERVER is listening on and c) a 160 bit, cryptographically secure string that represents the capability (the "capability id").
3. The FURLs are saved to disk in a secure location on the SERVER's host.

For a CLIENT to be able to connect to the SERVER and access a capability of that SERVER, the CLIENT must have knowledge of the FURL for that SERVER's capability. This typically requires that the file containing the FURL be moved from the SERVER's host to the CLIENT's host. This is done by the end user who started the SERVER and wishes to have a CLIENT connect to the SERVER.

When a CLIENT connects to the SERVER, the following handshake protocol takes place:

1. The CLIENT tells the SERVER what process (or Tub) id it expects the SERVER to have.
2. If the SERVER has that process id, it notifies the CLIENT that it will now enter encrypted mode. If the SERVER has a different id, the SERVER aborts.
3. Both CLIENT and SERVER initiate the SSL handshake protocol.
4. Both CLIENT and SERVER request the certificate of their peer and verify that certificate. If this succeeds, all further communications are encrypted.
5. Both CLIENT and SERVER send a hello block containing connection parameters and their process id.
6. The CLIENT and SERVER check that their peer's stated process id matches the hash of the x509 certificate the peer presented. If not, the connection is aborted.
7. The CLIENT verifies that the SERVER's stated id matches the id of the SERVER the CLIENT is intending to connect to. If not, the connection is aborted.
8. The CLIENT and SERVER elect a master who decides on the final connection parameters.

The public/private key pair associated with each process's x509 certificate are completely hidden from this handshake protocol. There are however, used internally by OpenSSL as part of the SSL handshake protocol. Each process keeps their own private key hidden and sends its peer only the public key (embedded in the certificate).

Finally, when the CLIENT requests access to a particular SERVER capability, the following happens:

1. The CLIENT asks the SERVER for access to a capability by presenting that capabilities id.
2. If the SERVER has a capability with that id, access is granted. If not, access is not granted.
3. Once access has been gained, the CLIENT can use the capability.

4.6.4 Specific security vulnerabilities

There are a number of potential security vulnerabilities present in IPython's architecture. In this section we discuss those vulnerabilities and detail how the security architecture described above prevents them from being exploited.

Unauthorized clients

The IPython client can instruct the IPython engines to execute arbitrary Python code with the permissions of the user who started the engines. If an attacker were able to connect their own hostile IPython client to the IPython controller, they could instruct the engines to execute code.

This attack is prevented by the capabilities based client authentication performed after the encrypted channel has been established. The relevant authentication information is encoded into the FURL that clients must present to gain access to the IPython controller. By limiting the distribution of those FURLs, a user can grant access to only authorized persons.

It is highly unlikely that a client FURL could be guessed by an attacker in a brute force guessing attack. A given instance of the IPython controller only runs for a relatively short amount of time (on the order of hours). Thus an attacker would have only a limited amount of time to test a search space of size 2**320. Furthermore, even if a controller were to run for a longer amount of time, this search space is quite large (larger for instance than that of typical username/password pair).

Unauthorized engines

If an attacker were able to connect a hostile engine to a user's controller, the user might unknowingly send sensitive code or data to the hostile engine. This attacker's engine would then have full access to that code and data.

This type of attack is prevented in the same way as the unauthorized client attack, through the usage of the capabilities based authentication scheme.

Unauthorized controllers

It is also possible that an attacker could try to convince a user's IPython client or engine to connect to a hostile IPython controller. That controller would then have full access to the code and data sent between the

IPython client and the IPython engines.

Again, this attack is prevented through the FURLs, which ensure that a client or engine connects to the correct controller. It is also important to note that the FURLs also encode the IP address and port that the controller is listening on, so there is little chance of mistakenly connecting to a controller running on a different IP address and port.

When starting an engine or client, a user must specify which FURL to use for that connection. Thus, in order to introduce a hostile controller, the attacker must convince the user to use the FURLs associated with the hostile controller. As long as a user is diligent in only using FURLs from trusted sources, this attack is not possible.

4.6.5 Other security measures

A number of other measures are taken to further limit the security risks involved in running the IPython kernel.

First, by default, the IPython controller listens on random port numbers. While this can be overridden by the user, in the default configuration, an attacker would have to do a port scan to even find a controller to attack. When coupled with the relatively short running time of a typical controller (on the order of hours), an attacker would have to work extremely hard and extremely *fast* to even find a running controller to attack.

Second, much of the time, especially when run on supercomputers or clusters, the controller is running behind a firewall. Thus, for engines or client to connect to the controller:

- The different processes have to all be behind the firewall.

or:

- The user has to use SSH port forwarding to tunnel the connections through the firewall.

In either case, an attacker is presented with addition barriers that prevent attacking or even probing the system.

4.6.6 Summary

IPython's architecture has been carefully designed with security in mind. The capabilities based authentication model, in conjunction with the encrypted TCP/IP channels, address the core potential vulnerabilities in the system, while still enabling user's to use the system in open networks.

4.6.7 Other questions

About keys

Can you clarify the roles of the certificate and its keys versus the FURL, which is also called a key?

The certificate created by IPython processes is a standard public key x509 certificate, that is used by the SSL handshake protocol to setup encrypted channel between the controller and the IPython engine or client. This public and private key associated with this certificate are used only by the SSL handshake protocol in setting up this encrypted channel.

The FURL serves a completely different and independent purpose from the key pair associated with the certificate. When we refer to a FURL as a key, we are using the word "key" in the capabilities based security model sense. This has nothing to do with "key" in the public/private key sense used in the SSL protocol.

With that said the FURL is used as an cryptographic key, to grant IPython engines and clients access to particular capabilities that the controller offers.

Self signed certificates

Is the controller creating a self-signed certificate? Is this created for per instance/session, one-time-setup or each-time the controller is started?

The Foolscap network protocol, which handles the SSL protocol details, creates a self-signed x509 certificate using OpenSSL for each IPython process. The lifetime of the certificate is handled differently for the IPython controller and the engines/client.

For the IPython engines and client, the certificate is only held in memory for the lifetime of its process. It is never written to disk.

For the controller, the certificate can be created anew each time the controller starts or it can be created once and reused each time the controller starts. If at any point, the certificate is deleted, a new one is created the next time the controller starts.

SSL private key

How the private key (associated with the certificate) is distributed?

In the usual implementation of the SSL protocol, the private key is never distributed. We follow this standard always.

SSL versus Foolscap authentication

Many SSL connections only perform one sided authentication (the server to the client). How is the client authentication in IPython's system related to SSL authentication?

We perform a two way SSL handshake in which both parties request and verify the certificate of their peer. This mutual authentication is handled by the SSL handshake and is separate and independent from the additional authentication steps that the CLIENT and SERVER perform after an encrypted channel is established.

4.7 IPython/Vision Beam Pattern Demo

Note: This page has not been updated to reflect the recent work on ipcluster. This work makes it much easier to use IPython on a cluster.

4.7.1 Installing and testing IPython at OSC systems

All components were installed from source and I have my environment set up to include ~/usr/local in my various necessary paths ($PATH, $PYTHONPATH, etc). Other than a slow filesystem for unpacking tarballs, the install went without a hitch. For each needed component, I just downloaded the source tarball, unpacked it via:

```
tar xzf (or xjf if it's bz2) filename.tar.{gz,bz2}
```

and then installed them (including IPython itself) with:

```
cd dirname/  # path to unpacked tarball
python setup.py install --prefix=~/usr/local/
```

The components I installed are listed below. For each one I give the main project link as well as a direct one to the file I actually dowloaded and used.

- nose, used for testing:

http://somethingaboutorange.com/mrl/projects/nose/ http://somethingaboutorange.com/mrl/projects/nose/nose-0.10.3.tar.gz

- Zope interface, used to declare interfaces in twisted and ipython. Note:

you must get this from the page linked below and not fro the defaul one(http://www.zope.org/Products/ZopeInterface) because the latter has an older version, it hasn't been updated in a long time. This pypi link has the current release (3.4.1 as of this writing): http://pypi.python.org/pypi/zope.interface http://pypi.python.org/packages/source/z/zope.interface/zope.interface-3.4.1.tar.gz

- pyopenssl, security layer used by foolscap. Note: version 0.7 *must* be

used: http://sourceforge.net/projects/pyopenssl/ http://downloads.sourceforge.net/pyopenssl/pyOpenSSL-0.6.tar.gz?modtime=1212595285&big_mirror=0

- Twisted, used for all networking:

http://twistedmatrix.com/trac/wiki/Downloads http://tmrc.mit.edu/mirror/twisted/Twisted/8.1/Twisted-8.1.0.tar.bz2

- Foolscap, used for managing connections securely:

http://foolscap.lothar.com/trac http://foolscap.lothar.com/releases/foolscap-0.3.1.tar.gz

- IPython itself:

http://ipython.scipy.org/ http://ipython.scipy.org/dist/ipython-0.9.1.tar.gz

I then ran the ipython test suite via:

```
iptest -vv
```

and it passed with only:

```
======================================================================
ERROR: testGetResult_2
----------------------------------------------------------------------
```

```
DirtyReactorAggregateError: Reactor was unclean.
Selectables:
<Negotiation #0 on 10105>

-------------------------------------------------------------------
Ran 419 tests in 33.971s

FAILED (SKIP=4, errors=1)
```

In three more runs of the test suite I was able to reproduce this error sometimes but not always; for now I think we can move on but we need to investigate further. Especially if we start seeing problems in real use (the test suite stresses the networking layer in particular ways that aren't necessarily typical of normal use).

Next, I started an 8-engine cluster via:

```
perez@opt-login01[~]> ipcluster -n 8
Starting controller: Controller PID: 30845
^X      Starting engines:    Engines PIDs:   [30846, 30847, 30848, 30849,
30850, 30851, 30852, 30853]
Log files: /home/perez/.ipython/log/ipcluster-30845-*

Your cluster is up and running.

[... etc]
```

and in a separate ipython session checked that the cluster is running and I can access all the engines:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()

In [3]: mec.get_ids()
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7]
```

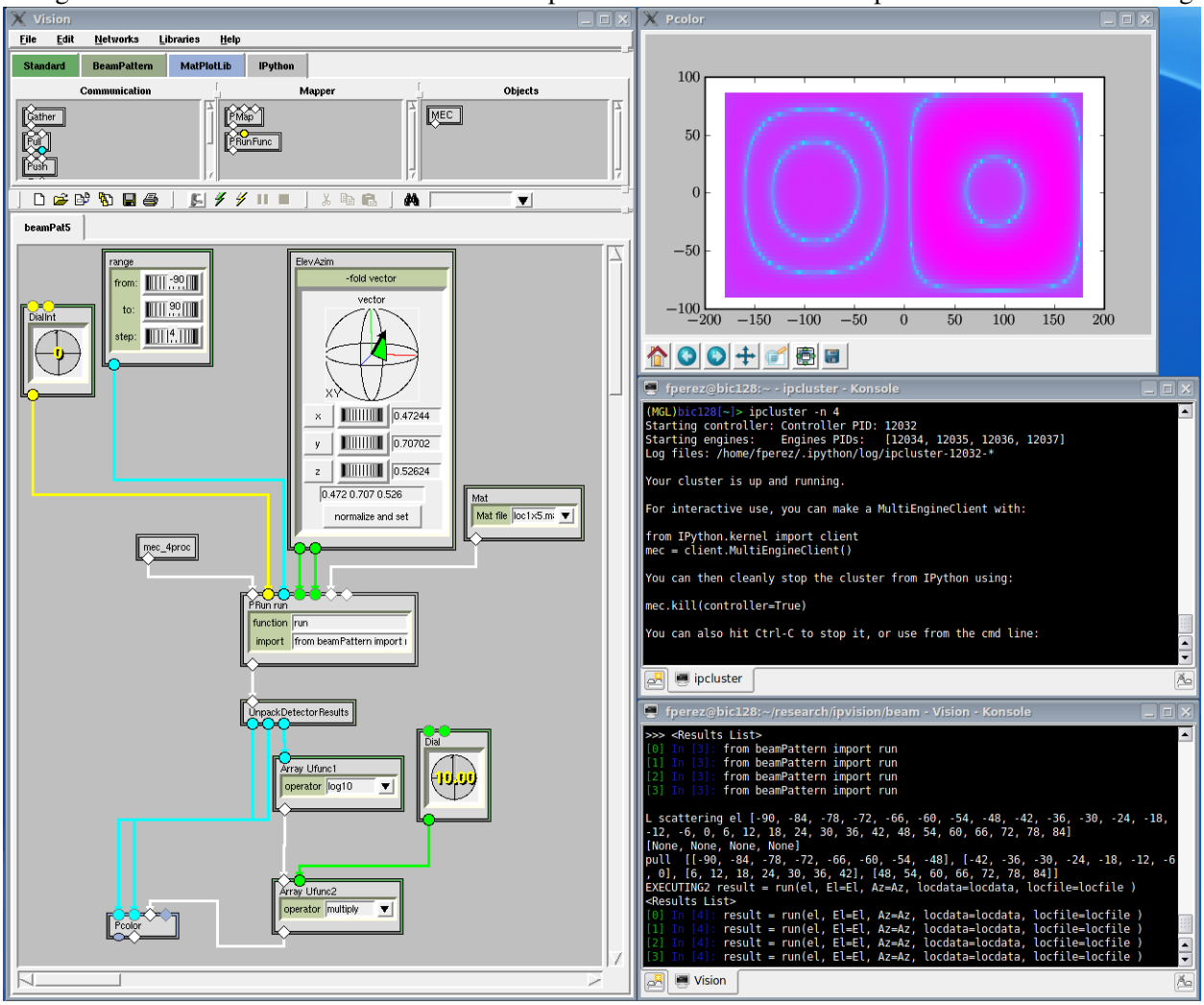
and run trivial code in them (after importing the `random` module in all engines):

```
In [11]: mec.execute("x=random.randint(0,10)")
Out[11]:
<Results List>
[0] In [3]: x=random.randint(0,10)
[1] In [3]: x=random.randint(0,10)
[2] In [3]: x=random.randint(0,10)
[3] In [3]: x=random.randint(0,10)
[4] In [3]: x=random.randint(0,10)
[5] In [3]: x=random.randint(0,10)
[6] In [3]: x=random.randint(0,10)
[7] In [3]: x=random.randint(0,10)

In [12]: mec.pull('x')
Out[12]: [10, 0, 8, 10, 2, 9, 10, 7]
```

We'll continue conducting more complex tests later, including instaling Vision locally and running the beam demo.

4.7.2 Michel's original instructions

I got a Vision network that reproduces the beam pattern demo working:



I created a package called beamPattern that provides the function run() in its __init__.py file.

A subpackage beamPattern/VisionInterface provides Vision nodes for:

- computing Elevation and Azimuth from a 3D vector
- Reading .mat files
- taking the results gathered from the engines and creating the output that a single engine would have had produced

The Mec node connect to a controller. In my network it was local but an furl can be specified to connect to a remote controller.

The PRun Func node is from the IPython library of nodes. the import statement is used to get the run function from the beamPattern package and bu puting "run" in the function entry of this node we push this function to the engines. In addition to the node will create input ports for all arguments of the function being pushed (i.e. the run function)

The second input port on PRun Fun take an integer specifying the rank of the argument we want to scatter. All other arguments will be pushed to the engines.

The ElevAzim node has a 3D vector widget and computes the El And Az values which are passed into the PRun Fun node through the ports created automatically. The Mat node allows to select the .mat file, reads it and passed the data to the locdata port created automatically on PRun Func

The calculation is executed in parallel, and the results are gathered and output. Instead of having a list of 3 vectors we nd up with a list of n*3 vectors where n is the number of engines. unpackDectorResults will turn it into a list of 3. We then plot x, y, and 10*log10(z)

Installation

- inflate beamPattern into the site-packages directory for the MGL tools.
- place the appended IPythonNodes.py and StandardNodes.py into the Vision package of the MGL tools.
- place the appended items.py in the NetworkEditor package of the MGL tools
- run vision for the network beamPat5_net.py:

```
vision beamPat5_net.py
```

Once the network is running, you can:

- double click on the MEC node and either use an emptty string for the furl to connect to a local engine or cut and paste the furl to the engine you want to use
- click on the yellow lighting bold to run the network.
- Try modifying the MAT file or change the Vector used top compute elevation and Azimut.

4.7.3 Fernando's notes

- I had to install IPython and all its dependencies for the python used by the MGL tools.
- Then I had to install scipy 0.6.0 for it, since the nodes needed Scipy. To do this I sourced the mglenv.sh script and then ran:

```
python setup.py install --prefix=~/usr/opt/mgl
```

4.7.4 Using PBS

The following PBS script can be used to start the engines:

```
#PBS -N bgranger-ipython
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l nodes=4:ppn=4

cd $PBS_O_WORKDIR
```

```
export PATH=$HOME/usr/local/bin
export PYTHONPATH=$HOME/usr/local/lib/python2.4/site-packages
/usr/local/bin/mpiexec -n 16 ipengine
```

If this file is called `ipython_pbs.sh`, then the in one login windows (i.e. on the head-node – `opt-login01.osc.edu`), run `ipcontroller`. In another login window on the same node, run the above script:

```
qsub ipython_pbs.sh
```

If you look at the first window, you will see some diagnostic output from ipcontroller. You can then get the furl from your own `~/.ipython/security` directory and then connect to it remotely.

You might need to set up an SSH tunnel, however; if this doesn't work as advertised:

```
ssh -L 10115:localhost:10105 bic
```

4.7.5 Links to other resources

- http://www.osc.edu/~unpingco/glenn_NewLynx2_Demo.avi

CHAPTER

FIVE

CONFIGURATION AND CUSTOMIZATION

5.1 Initial configuration of your environment

This section will help you set various things in your environment for your IPython sessions to be as efficient as possible. All of IPython's configuration information, along with several example files, is stored in a directory named by default $HOME/.ipython. You can change this by defining the environment variable IPYTHONDIR, or at runtime with the command line option -ipythondir.

If all goes well, the first time you run IPython it should automatically create a user copy of the config directory for you, based on its builtin defaults. You can look at the files it creates to learn more about configuring the system. The main file you will modify to configure IPython's behavior is called ipythonrc (with a .ini extension under Windows), included for reference *here*. This file is very commented and has many variables you can change to suit your taste, you can find more details *here*. Here we discuss the basic things you will want to make sure things are working properly from the beginning.

5.1.1 Access to the Python help system

This is true for Python in general (not just for IPython): you should have an environment variable called PYTHONDOCS pointing to the directory where your HTML Python documentation lives. In my system it's `/usr/share/doc/python-doc/html`, check your local details or ask your systems administrator.

This is the directory which holds the HTML version of the Python manuals. Unfortunately it seems that different Linux distributions package these files differently, so you may have to look around a bit. Below I show the contents of this directory on my system for reference:

```
[html]> ls
about.html  dist/  icons/      lib/           python2.5.devhelp.gz  whatsnew/
acks.html   doc/   index.html  mac/           ref/
api/        ext/   inst/       modindex.html  tut/
```

You should really make sure this variable is correctly set so that Python's pydoc-based help system works. It is a powerful and convenient system with full access to the Python manuals and all modules accessible to you.

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

5.1.2 Editor

The %edit command (and its alias %ed) will invoke the editor set in your environment as EDITOR. If this variable is not set, it will default to vi under Linux/Unix and to notepad under Windows. You may want to set this variable properly and to a lightweight editor which doesn't take too long to start (that is, something other than a new instance of Emacs). This way you can edit multi-line code quickly and with the power of a real editor right inside IPython.

If you are a dedicated Emacs user, you should set up the Emacs server so that new requests are handled by the original process. This means that almost no time is spent in handling the request (assuming an Emacs process is already running). For this to work, you need to set your EDITOR environment variable to 'emacsclient'. The code below, supplied by Francois Pinard, can then be used in your .emacs file to enable the server:

```
(defvar server-buffer-clients)
(when (and (fboundp 'server-start) (string-equal (getenv "TERM") 'xterm))
  (server-start)
  (defun fp-kill-server-with-buffer-routine ()
    (and server-buffer-clients (server-done)))
  (add-hook 'kill-buffer-hook 'fp-kill-server-with-buffer-routine))
```

You can also set the value of this editor via the commmand-line option '-editor' or in your ipythonrc file. This is useful if you wish to use specifically for IPython an editor different from your typical default (and for Windows users who tend to use fewer environment variables).

5.1.3 Color

The default IPython configuration has most bells and whistles turned on (they're pretty safe). But there's one that may cause problems on some systems: the use of color on screen for displaying information. This is very useful, since IPython can show prompts and exception tracebacks with various colors, display syntax-highlighted source code, and in general make it easier to visually parse information.

The following terminals seem to handle the color sequences fine:

- Linux main text console, KDE Konsole, Gnome Terminal, E-term, rxvt, xterm.
- CDE terminal (tested under Solaris). This one boldfaces light colors.
- (X)Emacs buffers. See the emacs section for more details on using IPython with (X)Emacs.
- A Windows (XP/2k) command prompt with pyreadline.
- A Windows (XP/2k) CygWin shell. Although some users have reported problems; it is not clear whether there is an issue for everyone or only under specific configurations. If you have full color support under cygwin, please post to the IPython mailing list so this issue can be resolved for all users.

These have shown problems:

- Windows command prompt in WinXP/2k logged into a Linux machine via telnet or ssh.

- Windows native command prompt in WinXP/2k, without Gary Bishop's extensions. Once Gary's readline library is installed, the normal WinXP/2k command prompt works perfectly.

Currently the following color schemes are available:

- NoColor: uses no color escapes at all (all escapes are empty '' '' strings). This 'scheme' is thus fully safe to use in any terminal.
- Linux: works well in Linux console type environments: dark background with light fonts. It uses bright colors for information, so it is difficult to read if you have a light colored background.
- LightBG: the basic colors are similar to those in the Linux scheme but darker. It is easy to read in terminals with light backgrounds.

IPython uses colors for two main groups of things: prompts and tracebacks which are directly printed to the terminal, and the object introspection system which passes large sets of data through a pager.

5.1.4 Input/Output prompts and exception tracebacks

You can test whether the colored prompts and tracebacks work on your system interactively by typing '%colors Linux' at the prompt (use '%colors LightBG' if your terminal has a light background). If the input prompt shows garbage like:

```
[0;32mIn [[1;32m1[0;32m]: [0;00m
```

instead of (in color) something like:

```
In [1]:
```

this means that your terminal doesn't properly handle color escape sequences. You can go to a 'no color' mode by typing '%colors NoColor'.

You can try using a different terminal emulator program (Emacs users, see below). To permanently set your color preferences, edit the file $HOME/.ipython/ipythonrc and set the colors option to the desired value.

5.1.5 Object details (types, docstrings, source code, etc.)

IPython has a set of special functions for studying the objects you are working with, discussed in detail *here*. But this system relies on passing information which is longer than your screen through a data pager, such as the common Unix less and more programs. In order to be able to see this information in color, your pager needs to be properly configured. I strongly recommend using less instead of more, as it seems that more simply can not understand colored text correctly.

In order to configure less as your default pager, do the following:

1. Set the environment PAGER variable to less.
2. Set the environment LESS variable to -r (plus any other options you always want to pass to less by default). This tells less to properly interpret control sequences, which is how color information is given to your terminal.

For the bash shell, add to your ~/.bashrc file the lines:

```
export PAGER=less
export LESS=-r
```

For the csh or tcsh shells, add to your ~/.cshrc file the lines:

```
setenv PAGER less
setenv LESS -r
```

There is similar syntax for other Unix shells, look at your system documentation for details.

If you are on a system which lacks proper data pagers (such as Windows), IPython will use a very limited builtin pager.

5.1.6 (X)Emacs configuration

Thanks to the work of Alexander Schmolck and Prabhu Ramachandran, currently (X)Emacs and IPython get along very well.

Important note: You will need to use a recent enough version of python-mode.el, along with the file ipython.el. You can check that the version you have of python-mode.el is new enough by either looking at the revision number in the file itself, or asking for it in (X)Emacs via M-x py-version. Versions 4.68 and newer contain the necessary fixes for proper IPython support.

The file ipython.el is included with the IPython distribution, in the documentation directory (where this manual resides in PDF and HTML formats).

Once you put these files in your Emacs path, all you need in your .emacs file is:

```
(require 'ipython)
```

This should give you full support for executing code snippets via IPython, opening IPython as your Python shell via `C-c !`, etc.

You can customize the arguments passed to the IPython instance at startup by setting the `py-python-command-args` variable. For example, to start always in `pylab` mode with hardcoded light-background colors, you can use:

```
(setq py-python-command-args '("-pylab" "-colors" "LightBG"))
```

If you happen to get garbage instead of colored prompts as described in the previous section, you may need to set also in your .emacs file:

```
(setq ansi-color-for-comint-mode t)
```

Notes:

- There is one caveat you should be aware of: you must start the IPython shell before attempting to execute any code regions via `C-c |`. Simply type C-c ! to start IPython before passing any code regions to the interpreter, and you shouldn't experience any problems. This is due to a bug in Python itself, which has been fixed for Python 2.3, but exists as of Python 2.2.2 (reported as SF bug [737947]).

- The (X)Emacs support is maintained by Alexander Schmolck, so all comments/requests should be directed to him through the IPython mailing lists.
- This code is still somewhat experimental so it's a bit rough around the edges (although in practice, it works quite well).
- Be aware that if you customize py-python-command previously, this value will override what ipython.el does (because loading the customization variables comes later).

5.2 Customization of IPython

There are 2 ways to configure IPython - the old way of using ipythonrc files (an INI-file like format), and the new way that involves editing your ipy_user_conf.py. Both configuration systems work at the same time, so you can set your options in both, but if you are hesitating about which alternative to choose, we recommend the ipy_user_conf.py approach, as it will give you more power and control in the long run. However, there are few options such as pylab_import_all that can only be specified in ipythonrc file or command line - the reason for this is that they are needed before IPython has been started up, and the IPApi object used in ipy_user_conf.py is not yet available at that time. A hybrid approach of specifying a few options in ipythonrc and doing the more advanced configuration in ipy_user_conf.py is also possible.

5.2.1 The ipythonrc approach

As we've already mentioned, IPython reads a configuration file which can be specified at the command line (-rcfile) or which by default is assumed to be called ipythonrc. Such a file is looked for in the current directory where IPython is started and then in your IPYTHONDIR, which allows you to have local configuration files for specific projects. In this section we will call these types of configuration files simply rcfiles (short for resource configuration file).

The syntax of an rcfile is one of key-value pairs separated by whitespace, one per line. Lines beginning with a # are ignored as comments, but comments can not be put on lines with data (the parser is fairly primitive). Note that these are not python files, and this is deliberate, because it allows us to do some things which would be quite tricky to implement if they were normal python files.

First, an rcfile can contain permanent default values for almost all command line options (except things like -help or -Version). *This section* contains a description of all command-line options. However, values you explicitly specify at the command line override the values defined in the rcfile.

Besides command line option values, the rcfile can specify values for certain extra special options which are not available at the command line. These options are briefly described below.

Each of these options may appear as many times as you need it in the file.

- include <file1> <file2> ...: you can name other rcfiles you want to recursively load up to 15 levels (don't use the <> brackets in your names!). This feature allows you to define a 'base' rcfile with general options and special-purpose files which can be loaded only when needed with particular configuration options. To make this more convenient, IPython accepts the -profile <name> option (abbreviates to -p <name>) which tells it to look for an rcfile named ipythonrc-<name>.
- import_mod <mod1> <mod2> ...: import modules with 'import <mod1>,<mod2>,...'

- import_some <mod> <f1> <f2> ...: import functions with 'from <mod> import <f1>,<f2>,...'
- import_all <mod1> <mod2> ...: for each module listed import functions with `from <mod> import *`.
- execute <python code>: give any single-line python code to be executed.
- execfile <filename>: execute the python file given with an 'execfile(filename)' command. Username expansion is performed on the given names. So if you need any amount of extra fancy customization that won't fit in any of the above 'canned' options, you can just put it in a separate python file and execute it.
- alias <alias_def>: this is equivalent to calling '%alias <alias_def>' at the IPython command line. This way, from within IPython you can do common system tasks without having to exit it or use the ! escape. IPython isn't meant to be a shell replacement, but it is often very useful to be able to do things with files while testing code. This gives you the flexibility to have within IPython any aliases you may be used to under your normal system shell.

5.2.2 ipy_user_conf.py

There should be a simple template ipy_user_conf.py file in your ~/.ipython directory. It is a plain python module that is imported during IPython startup, so you can do pretty much what you want there - import modules, configure extensions, change options, define magic commands, put variables and functions in the IPython namespace, etc. You use the IPython extension api object, acquired by IPython.ipapi.get() and documented in the "IPython extension API" chapter, to interact with IPython. A sample ipy_user_conf.py is listed below for reference:

```
# Most of your config files and extensions will probably start
# with this import

import IPython.ipapi
ip = IPython.ipapi.get()

# You probably want to uncomment this if you did %upgrade -nolegacy
# import ipy_defaults

import os

def main():

    #ip.dbg.debugmode = True
    ip.dbg.debug_stack()

    # uncomment if you want to get ipython -p sh behaviour
    # without having to use command line switches
    import ipy_profile_sh
    import jobctrl

    # Configure your favourite editor?
    # Good idea e.g. for %edit os.path.isfile

    #import ipy_editors
```

```
    # Choose one of these:

    #ipy_editors.scite()
    #ipy_editors.scite('c:/opt/scite/scite.exe')
    #ipy_editors.komodo()
    #ipy_editors.idle()
    # ... or many others, try 'ipy_editors??' after import to see them

    # Or roll your own:
    #ipy_editors.install_editor("c:/opt/jed +$line $file")


    o = ip.options
    # An example on how to set options
    #o.autocall = 1
    o.system_verbose = 0

    #import_all("os sys")
    #execf('~/_ipython/ns.py')


    # -- prompt
    # A different, more compact set of prompts from the default ones, that
    # always show your current location in the filesystem:

    #o.prompt_in1 = r'\C_LightBlue[\C_LightCyan\Y2\C_LightBlue]\C_Normal\n\C_Green|\#>'
    #o.prompt_in2 = r'.\D: '
    #o.prompt_out = r'[\#] '

    # Try one of these color settings if you can't read the text easily
    # autoexec is a list of IPython commands to execute on startup
    #o.autoexec.append('%colors LightBG')
    #o.autoexec.append('%colors NoColor')
    o.autoexec.append('%colors Linux')


# some config helper functions you can use
def import_all(modules):
    """ Usage: import_all("os sys") """
    for m in modules.split():
        ip.ex("from %s import *" % m)

def execf(fname):
    """ Execute a file in user namespace """
    ip.ex('execfile("%s")' % os.path.expanduser(fname))

main()
```

5.2.3 Fine-tuning your prompt

IPython's prompts can be customized using a syntax similar to that of the bash shell. Many of bash's escapes are supported, as well as a few additional ones. We list them below:

```
\#
    the prompt/history count number. This escape is automatically
    wrapped in the coloring codes for the currently active color scheme.
\N
    the 'naked' prompt/history count number: this is just the number
    itself, without any coloring applied to it. This lets you produce
    numbered prompts with your own colors.
\D
    the prompt/history count, with the actual digits replaced by dots.
    Used mainly in continuation prompts (prompt_in2)
\w
    the current working directory
\W
    the basename of current working directory
\Xn
    where $n=0\ldots5.$ The current working directory, with $HOME
    replaced by ~, and filtered out to contain only $n$ path elements
\Yn
    Similar to \Xn, but with the $n+1$ element included if it is ~ (this
    is similar to the behavior of the %cn escapes in tcsh)
\u
    the username of the current user
\$
    if the effective UID is 0, a #, otherwise a $
\h
    the hostname up to the first '.'
\H
    the hostname
\n
    a newline
\r
    a carriage return
\v
    IPython version string
```

In addition to these, ANSI color escapes can be insterted into the prompts, as C_ColorName. The list of valid color names is: Black, Blue, Brown, Cyan, DarkGray, Green, LightBlue, LightCyan, LightGray, LightGreen, LightPurple, LightRed, NoColor, Normal, Purple, Red, White, Yellow.

Finally, IPython supports the evaluation of arbitrary expressions in your prompt string. The prompt strings are evaluated through the syntax of PEP 215, but basically you can use $x.y to expand the value of x.y, and for more complicated expressions you can use braces: ${foo()+x} will call function foo and add to it the value of x, before putting the result into your prompt. For example, using prompt_in1 '${commands.getoutput("uptime")}nIn [#]: ' will print the result of the uptime command on each prompt (assuming the commands module has been imported in your ipythonrc file).

Prompt examples

The following options in an ipythonrc file will give you IPython's default prompts:

```
prompt_in1 'In [\#]:'
prompt_in2 '   .\D.:'
prompt_out 'Out[\#]:'
```

which look like this:

```
In [1]: 1+2
Out[1]: 3

In [2]: for i in (1,2,3):
   ...:    print i,
   ...:
1 2 3
```

These will give you a very colorful prompt with path information:

```
#prompt_in1 '\C_Red\u\C_Blue[\C_Cyan\Y1\C_Blue]\C_LightGreen\#>'
prompt_in2 ' ..\D>'
prompt_out '<\#>'
```

which look like this:

```
fperez[~/ipython]1> 1+2
                 <1> 3
fperez[~/ipython]2> for i in (1,2,3):
                ...>     print i,
                ...>
1 2 3
```

5.2.4 IPython profiles

As we already mentioned, IPython supports the -profile command-line option (see *here*). A profile is nothing more than a particular configuration file like your basic ipythonrc one, but with particular customizations for a specific purpose. When you start IPython with 'ipython -profile <name>', it assumes that in your IPYTHONDIR there is a file called ipythonrc-<name> or ipy_profile_<name>.py, and loads it instead of the normal ipythonrc.

This system allows you to maintain multiple configurations which load modules, set options, define functions, etc. suitable for different tasks and activate them in a very simple manner. In order to avoid having to repeat all of your basic options (common things that don't change such as your color preferences, for example), any profile can include another configuration file. The most common way to use profiles is then to have each one include your basic ipythonrc file as a starting point, and then add further customizations.

5.3 New configuration system

IPython has a configuration system. When running IPython for the first time, reasonable defaults are used for the configuration. The configuration of IPython can be changed in two ways:

- Configuration files

- Commands line options (which override the configuration files)

IPython has a separate configuration file for each subpackage. Thus, the main configuration files are (in your `~/.ipython` directory):

- `ipython1.core.ini`
- `ipython1.kernel.ini`
- `ipython1.notebook.ini`

To create these files for the first time, do the following:

```
from IPython.kernel.config import config_manager as kernel_config
kernel_config.write_default_config_file()
```

But, you should only need to do this if you need to modify the defaults. If needed repeat this process with the `notebook` and `core` configuration as well. If you are running into problems with IPython, you might try deleting these configuration files.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

6.1 General questions

6.2 Questions about parallel computing with IPython

6.2.1 Will IPython speed my Python code up?

Yes and no. When converting a serial code to run in parallel, there often many difficulty questions that need to be answered, such as:

- How should data be decomposed onto the set of processors?
- What are the data movement patterns?
- Can the algorithm be structured to minimize data movement?
- Is dynamic load balancing important?

We can't answer such questions for you. This is the hard (but fun) work of parallel computing. But, once you understand these things IPython will make it easier for you to implement a good solution quickly. Most importantly, you will be able to use the resulting parallel code interactively.

With that said, if your problem is trivial to parallelize, IPython has a number of different interfaces that will enable you to parallelize things is almost no time at all. A good place to start is the `map` method of our `MultiEngineClient`.

6.2.2 What is the best way to use MPI from Python?

6.2.3 What about all the other parallel computing packages in Python?

Some of the unique characteristic of IPython are:

- IPython is the only architecture that abstracts out the notion of a parallel computation in such a way that new models of parallel computing can be explored quickly and easily. If you don't like the models we provide, you can simply create your own using the capabilities we provide.
- IPython is asynchronous from the ground up (we use Twisted).

- IPython's architecture is designed to avoid subtle problems that emerge because of Python's global interpreter lock (GIL).
- While IPython's architecture is designed to support a wide range of novel parallel computing models, it is fully interoperable with traditional MPI applications.
- IPython has been used and tested extensively on modern supercomputers.
- IPython's networking layers are completely modular. Thus, is straightforward to replace our existing network protocols with high performance alternatives (ones based upon Myranet/Infiniband).
- IPython is designed from the ground up to support collaborative parallel computing. This enables multiple users to actively develop and run the *same* parallel computation.
- Interactivity is a central goal for us. While IPython does not have to be used interactivly, it can be.

6.2.4 Why The IPython controller a bottleneck in my parallel calculation?

A golden rule in parallel computing is that you should only move data around if you absolutely need to. The main reason that the controller becomes a bottleneck is that too much data is being pushed and pulled to and from the engines. If your algorithm is structured in this way, you really should think about alternative ways of handling the data movement. Here are some ideas:

1. Have the engines write data to files on the locals disks of the engines.
2. Have the engines write data to files on a file system that is shared by the engines.
3. Have the engines write data to a database that is shared by the engines.
4. Simply keep data in the persistent memory of the engines and move the computation to the data (rather than the data to the computation).
5. See if you can pass data directly between engines using MPI.

6.2.5 Isn't Python slow to be used for high-performance parallel computing?

CHAPTER

SEVEN

HISTORY

7.1 Origins

IPython was started in 2001 by Fernando Perez. IPython as we know it today grew out of the following three projects:

- ipython by Fernando Pérez. I was working on adding Mathematica-type prompts and a flexible configuration system (something better than $PYTHONSTARTUP) to the standard Python interactive interpreter.
- IPP by Janko Hauser. Very well organized, great usability. Had an old help system. IPP was used as the 'container' code into which I added the functionality from ipython and LazyPython.
- LazyPython by Nathan Gray. Simple but very powerful. The quick syntax (auto parens, auto quotes) and verbose/colored tracebacks were all taken from here.

Here is how Fernando describes it:

> When I found out about IPP and LazyPython I tried to join all three into a unified system. I thought this could provide a very nice working environment, both for regular programming and scientific computing: shell-like features, IDL/Matlab numerics, Mathematica-type prompt history and great object introspection and help facilities. I think it worked reasonably well, though it was a lot more work than I had initially planned.

7.2 Today and how we got here

This needs to be filled in.

CHAPTER

EIGHT

WHAT'S NEW

8.1 Release 0.10.1

IPython 0.10.1 was released October 11, 2010, over a year after version 0.10. This is mostly a bugfix release, since after version 0.10 was released, the development team's energy has been focused on the 0.11 series. We have nonetheless tried to backport what fixes we could into 0.10.1, as it remains the stable series that many users have in production systems they rely on.

Since the 0.11 series changes many APIs in backwards-incompatible ways, we are willing to continue maintaining the 0.10.x series. We don't really have time to actively write new code for 0.10.x, but we are happy to accept patches and pull requests on the IPython github site. If sufficient contributions are made that improve 0.10.1, we will roll them into future releases. For this purpose, we will have a branch called 0.10.2 on github, on which you can base your contributions.

For this release, we applied approximately 60 commits totaling a diff of over 7000 lines:

```
(0.10.1)amirbar[dist]> git diff --oneline rel-0.10.. | wc -l
7296
```

Highlights of this release:

- The only significant new feature is that IPython's parallel computing machinery now supports natively the Sun Grid Engine and LSF schedulers. This work was a joint contribution from Justin Riley, Satra Ghosh and Matthieu Brucher, who put a lot of work into it. We also improved traceback handling in remote tasks, as well as providing better control for remote task IDs.
- New IPython Sphinx directive. You can use this directive to mark blocks in reSructuredText documents as containig IPython syntax (including figures) and the will be executed during the build:

  ```
  .. ipython::
  ```

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

  @savefig psimple.png width=4in In [3]: plt.plot([1,2,3]) Out[3]: [<matplotlib.lines.Line2D object at 0x9b74d8c>]
- Various fixes to the standalone ipython-wx application.
- We now ship internally the excellent argparse library, graciously licensed under BSD terms by Steven Bethard. Now (2010) that argparse has become part of Python 2.7 this will be less of an issue, but

Steven's relicensing allowed us to start updating IPython to using argparse well before Python 2.7. Many thanks!

- Robustness improvements so that IPython doesn't crash if the readline library is absent (though obviously a lot of functionality that requires readline will not be available).
- Improvements to tab completion in Emacs with Python 2.6.
- Logging now supports timestamps (see `%logstart?` for full details).
- A long-standing and quite annoying bug where parentheses would be added to `print` statements, under Python 2.5 and 2.6, was finally fixed.
- Improved handling of libreadline on Apple OSX.
- Fix `reload` method of IPython demos, which was broken.
- Fixes for the ipipe/ibrowse system on OSX.
- Fixes for Zope profile.
- Fix %timeit reporting when the time is longer than 1000s.
- Avoid lockups with ? or ?? in SunOS, due to a bug in termios.
- The usual assortment of miscellaneous bug fixes and small improvements.

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

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

8.2 Release 0.10

This release brings months of slow but steady development, and will be the last before a major restructuring and cleanup of IPython's internals that is already under way. For this reason, we hope that 0.10 will be a stable and robust release so that while users adapt to some of the API changes that will come with the refactoring that will become IPython 0.11, they can safely use 0.10 in all existing projects with minimal changes (if any).

IPython 0.10 is now a medium-sized project, with roughly (as reported by David Wheeler's **sloccount** utility) 40750 lines of Python code, and a diff between 0.9.1 and this release that contains almost 28000 lines of code and documentation. Our documentation, in PDF format, is a 495-page long PDF document (also available in HTML format, both generated from the same sources).

Many users and developers contributed code, features, bug reports and ideas to this release. Please do not hesitate in contacting us if we've failed to acknowledge your contribution here. In particular, for this release we have contribution from the following people, a mix of new and regular names (in alphabetical order by first name):

- Alexander Clausen: fix #341726.
- Brian Granger: lots of work everywhere (features, bug fixes, etc).
- Daniel Ashbrook: bug report on MemoryError during compilation, now fixed.
- Darren Dale: improvements to documentation build system, feedback, design ideas.
- Fernando Perez: various places.
- Gaël Varoquaux: core code, ipythonx GUI, design discussions, etc. Lots...
- John Hunter: suggestions, bug fixes, feedback.
- Jorgen Stenarson: work on many fronts, tests, fixes, win32 support, etc.
- Laurent Dufréchou: many improvements to ipython-wx standalone app.
- Lukasz Pankowski: prefilter, *%edit*, demo improvements.
- Matt Foster: TextMate support in *%edit*.
- Nathaniel Smith: fix #237073.
- Pauli Virtanen: fixes and improvements to extensions, documentation.
- Prabhu Ramachandran: improvements to *%timeit*.
- Robert Kern: several extensions.
- Sameer D'Costa: help on critical bug #269966.
- Stephan Peijnik: feedback on Debian compliance and many man pages.
- Steven Bethard: we are now shipping his `argparse` module.
- Tom Fetherston: many improvements to `IPython.demo` module.
- Ville Vainio: lots of work everywhere (features, bug fixes, etc).
- Vishal Vasta: ssh support in ipcluster.

- Walter Doerwald: work on the `IPython.ipipe` system.

Below we give an overview of new features, bug fixes and backwards-incompatible changes. For a detailed account of every change made, feel free to view the project log with **bzr log**.

8.2.1 New features

- New *%paste* magic automatically extracts current contents of clipboard and pastes it directly, while correctly handling code that is indented or prepended with >>> or ... python prompt markers. A very useful new feature contributed by Robert Kern.
- IPython 'demos', created with the `IPython.demo` module, can now be created from files on disk or strings in memory. Other fixes and improvements to the demo system, by Tom Fetherston.
- Added `find_cmd()` function to `IPython.platutils` module, to find commands in a cross-platform manner.
- Many improvements and fixes to Gaël Varoquaux's **ipythonx**, a WX-based lightweight IPython instance that can be easily embedded in other WX applications. These improvements have made it possible to now have an embedded IPython in Mayavi and other tools.
- `MultiengineClient` objects now have a `benchmark()` method.
- The manual now includes a full set of auto-generated API documents from the code sources, using Sphinx and some of our own support code. We are now using the Numpy Documentation Standard for all docstrings, and we have tried to update as many existing ones as possible to this format.
- The new `IPython.Extensions.ipy_pretty` extension by Robert Kern provides configurable pretty-printing.
- Many improvements to the **ipython-wx** standalone WX-based IPython application by Laurent Dufréchou. It can optionally run in a thread, and this can be toggled at runtime (allowing the loading of Matplotlib in a running session without ill effects).
- IPython includes a copy of Steven Bethard's argparse in the `IPython.external` package, so we can use it internally and it is also available to any IPython user. By installing it in this manner, we ensure zero conflicts with any system-wide installation you may already have while minimizing external dependencies for new users. In IPython 0.10, We ship argparse version 1.0.
- An improved and much more robust test suite, that runs groups of tests in separate subprocesses using either Nose or Twisted's **trial** runner to ensure proper management of Twisted-using code. The test suite degrades gracefully if optional dependencies are not available, so that the **iptest** command can be run with only Nose installed and nothing else. We also have more and cleaner test decorators to better select tests depending on runtime conditions, do setup/teardown, etc.
- The new ipcluster now has a fully working ssh mode that should work on Linux, Unix and OS X. Thanks to Vishal Vatsa for implementing this!
- The wonderful TextMate editor can now be used with %edit on OS X. Thanks to Matt Foster for this patch.
- The documentation regarding parallel uses of IPython, including MPI and PBS, has been significantly updated and improved.

- The developer guidelines in the documentation have been updated to explain our workflow using **bzr** and Launchpad.
- Fully refactored **ipcluster** command line program for starting IPython clusters. This new version is a complete rewrite and 1) is fully cross platform (we now use Twisted's process management), 2) has much improved performance, 3) uses subcommands for different types of clusters, 4) uses argparse for parsing command line options, 5) has better support for starting clusters using **mpirun**, 6) has experimental support for starting engines using PBS. It can also reuse FURL files, by appropriately passing options to its subcommands. However, this new version of ipcluster should be considered a technology preview. We plan on changing the API in significant ways before it is final.
- Full description of the security model added to the docs.
- cd completer: show bookmarks if no other completions are available.
- sh profile: easy way to give 'title' to prompt: assign to variable '_prompt_title'. It looks like this:

```
[~]|1> _prompt_title = 'sudo!'
sudo![~]|2>
```

- %edit: If you do '%edit pasted_block', pasted_block variable gets updated with new data (so repeated editing makes sense)

8.2.2 Bug fixes

- Fix #368719, removed top-level debian/ directory to make the job of Debian packagers easier.
- Fix #291143 by including man pages contributed by Stephan Peijnik from the Debian project.
- Fix #358202, effectively a race condition, by properly synchronizing file creation at cluster startup time.
- *%timeit* now handles correctly functions that take a long time to execute even the first time, by not repeating them.
- Fix #239054, releasing of references after exiting.
- Fix #341726, thanks to Alexander Clausen.
- Fix #269966. This long-standing and very difficult bug (which is actually a problem in Python itself) meant long-running sessions would inevitably grow in memory size, often with catastrophic consequences if users had large objects in their scripts. Now, using *%run* repeatedly should not cause any memory leaks. Special thanks to John Hunter and Sameer D'Costa for their help with this bug.
- Fix #295371, bug in *%history*.
- Improved support for py2exe.
- Fix #270856: IPython hangs with PyGTK
- Fix #270998: A magic with no docstring breaks the '%magic magic'
- fix #271684: -c startup commands screw up raw vs. native history
- Numerous bugs on Windows with the new ipcluster have been fixed.

- The ipengine and ipcontroller scripts now handle missing furl files more gracefully by giving better error messages.
- %rehashx: Aliases no longer contain dots. python3.0 binary will create alias python30. Fixes: #259716 "commands with dots in them don't work"
- %cpaste: %cpaste -r repeats the last pasted block. The block is assigned to pasted_block even if code raises exception.
- Bug #274067 'The code in get_home_dir is broken for py2exe' was fixed.
- Many other small bug fixes not listed here by number (see the bzr log for more info).

8.2.3 Backwards incompatible changes

- *ipykit* and related files were unmaintained and have been removed.
- The `IPython.genutils.doctest_reload()` does not actually call *reload(doctest)* anymore, as this was causing many problems with the test suite. It still resets *doctest.master* to None.
- While we have not deliberately broken Python 2.4 compatibility, only minor testing was done with Python 2.4, while 2.5 and 2.6 were fully tested. But if you encounter problems with 2.4, please do report them as bugs.
- The **ipcluster** now requires a mode argument; for example to start a cluster on the local machine with 4 engines, you must now type:

```
$ ipcluster local -n 4
```

- The controller now has a `-r` flag that needs to be used if you want to reuse existing furl files. Otherwise they are deleted (the default).
- Remove ipy_leo.py. You can use **easy_install ipython-extension** to get it. (done to decouple it from ipython release cycle)

8.3 Release 0.9.1

This release was quickly made to restore compatibility with Python 2.4, which version 0.9 accidentally broke. No new features were introduced, other than some additional testing support for internal use.

8.4 Release 0.9

8.4.1 New features

- All furl files and security certificates are now put in a read-only directory named ~./ipython/security.
- A single function `get_ipython_dir()`, in `IPython.genutils` that determines the user's IPython directory in a robust manner.

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

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

8.4.2 Bug fixes

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

8.4.3 Backwards incompatible changes

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

- IPython has a larger set of dependencies if you want all of its capabilities. See the *setup.py* script for details.
- The constructors for `IPython.kernel.client.MultiEngineClient` and `IPython.kernel.client.TaskClient` no longer take the (ip,port) tuple. Instead they take the filename of a file that contains the FURL for that client. If the FURL file is in your IPYTHONDIR, it will be found automatically and the constructor can be left empty.
- The asynchronous clients in `IPython.kernel.asyncclient` are now created using the factory functions `get_multiengine_client()` and `get_task_client()`. These return a *Deferred* to the actual client.
- The command line options to *ipcontroller* and *ipengine* have changed to reflect the new Foolscap network protocol and the FURL files. Please see the help for these scripts for details.
- The configuration files for the kernel have changed because of the Foolscap stuff. If you were using custom config files before, you should delete them and regenerate new ones.

8.4.4 Changes merged in from IPython1

New features

- Much improved `setup.py` and `setupegg.py` scripts. Because Twisted and zope.interface are now easy installable, we can declare them as dependencies in our setupegg.py script.
- IPython is now compatible with Twisted 2.5.0 and 8.x.
- Added a new example of how to use `ipython1.kernel.asynclient`.
- Initial draft of a process daemon in `ipython1.daemon`. This has not been merged into IPython and is still in *ipython1-dev*.
- The `TaskController` now has methods for getting the queue status.
- The `TaskResult` objects not have information about how long the task took to run.
- We are attaching additional attributes to exceptions `(_ipython_*)` that we use to carry additional info around.
- New top-level module `asyncclient` that has asynchronous versions (that return deferreds) of the client classes. This is designed to users who want to run their own Twisted reactor.
- All the clients in `client` are now based on Twisted. This is done by running the Twisted reactor in a separate thread and using the `blockingCallFromThread()` function that is in recent versions of Twisted.
- Functions can now be pushed/pulled to/from engines using `MultiEngineClient.push_function()` and `MultiEngineClient.pull_function()`.
- Gather/scatter are now implemented in the client to reduce the work load of the controller and improve performance.

- Complete rewrite of the IPython docuementation. All of the documentation from the IPython website has been moved into docs/source as restructured text documents. PDF and HTML documentation are being generated using Sphinx.
- New developer oriented documentation: development guidelines and roadmap.
- Traditional `ChangeLog` has been changed to a more useful `changes.txt` file that is organized by release and is meant to provide something more relevant for users.

Bug fixes

- Created a proper `MANIFEST.in` file to create source distributions.
- Fixed a bug in the `MultiEngine` interface. Previously, multi-engine actions were being collected with a `DeferredList` with `fireononeerrback=1`. This meant that methods were returning before all engines had given their results. This was causing extremely odd bugs in certain cases. To fix this problem, we have 1) set `fireononeerrback=0` to make sure all results (or exceptions) are in before returning and 2) introduced a `CompositeError` exception that wraps all of the engine exceptions. This is a huge change as it means that users will have to catch `CompositeError` rather than the actual exception.

Backwards incompatible changes

- All names have been renamed to conform to the lowercase_with_underscore convention. This will require users to change references to all names like `queueStatus` to `queue_status`.
- Previously, methods like `MultiEngineClient.push()` and `MultiEngineClient.push()` used `*args` and `**kwargs`. This was becoming a problem as we weren't able to introduce new keyword arguments into the API. Now these methods simple take a dict or sequence. This has also allowed us to get rid of the `*All` methods like `pushAll()` and `pullAll()`. These things are now handled with the `targets` keyword argument that defaults to `'all'`.
- The `MultiEngineClient.magicTargets` has been renamed to `MultiEngineClient.targets`.
- All methods in the MultiEngine interface now accept the optional keyword argument `block`.
- Renamed `RemoteController` to `MultiEngineClient` and `TaskController` to `TaskClient`.
- Renamed the top-level module from `api` to `client`.
- Most methods in the multiengine interface now raise a `CompositeError` exception that wraps the user's exceptions, rather than just raising the raw user's exception.
- Changed the `setupNS` and `resultNames` in the `Task` class to `push` and `pull`.

8.5 Release 0.8.4

This was a quick release to fix an unfortunate bug that slipped into the 0.8.3 release. The `--twisted` option was disabled, as it turned out to be broken across several platforms.

8.6 Release 0.8.3

- pydb is now disabled by default (due to %run -d problems). You can enable it by passing -pydb command line argument to IPython. Note that setting it in config file won't work.

8.7 Release 0.8.2

- %pushd/%popd behave differently; now "pushd /foo" pushes CURRENT directory and jumps to /foo. The current behaviour is closer to the documented behaviour, and should not trip anyone.

8.8 Older releases

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

CHAPTER

NINE

IPYTHON DEVELOPER'S GUIDE

9.1 IPython development guidelines

9.1.1 Overview

This document describes IPython from the perspective of developers. Most importantly, it gives information for people who want to contribute to the development of IPython. So if you want to help out, read on!

9.1.2 How to contribute to IPython

IPython development is done using Bazaar [Bazaar] and Launchpad [Launchpad]. This makes it easy for people to contribute to the development of IPython. There are several ways in which you can join in.

If you have a small change that you want to send to the team, you can edit your bazaar checkout of IPython (see below) in-place, and ask bazaar for the differences:

```
$ cd /path/to/your/copy/of/ipython
$ bzr diff > my_fixes.diff
```

This produces a patch file with your fixes, which we can apply to the source tree. This file should then be attached to a ticket in our bug tracker, indicating what it does.

This model of creating small, self-contained patches works very well and there are open source projects that do their entire development this way. However, in IPython we have found that for tracking larger changes, making use of bazaar's full capabilities in conjunction with Launchpad's code hosting services makes for a much better experience.

Making your own branch of IPython allows you to refine your changes over time, track the development of the main team, and propose your own full version of the code for others to use and review, with a minimum amount of fuss. The next parts of this document will explain how to do this.

Install Bazaar and create a Launchpad account

First make sure you have installed Bazaar (see their website). To see that Bazaar is installed and knows about you, try the following:

```
$ bzr whoami
Joe Coder <jcoder@gmail.com>
```

This should display your name and email. Next, you will want to create an account on the Launchpad website and setup your ssh keys. For more information of setting up your ssh keys, see this link.

Get the main IPython branch from Launchpad

Now, you can get a copy of the main IPython development branch (we call this the "trunk"):

```
$ bzr branch lp:ipython
```

Create a working branch

When working on IPython, you won't actually make edits directly to the `lp:ipython` branch. Instead, you will create a separate branch for your changes. For now, let's assume you want to do your work in a branch named "ipython-mybranch". Create this branch by doing:

```
$ bzr branch ipython ipython-mybranch
```

When you actually create a branch, you will want to give it a name that reflects the nature of the work that you will be doing in it, like "install-docs-update".

Make edits in your working branch

Now you are ready to actually make edits in your `ipython-mybranch` branch. Before doing this, it is helpful to install this branch so you can test your changes as you work. This is easiest if you have setuptools installed. Then, just do:

```
$ cd ipython-mybranch
$ python setupegg.py develop
```

Now, make some changes. After a while, you will want to commit your changes. This let's Bazaar know that you like the changes you have made and gives you an opportunity to keep a nice record of what you have done. This looks like this:

```
$ ...do work in ipython-mybranch...
$ bzr commit -m "the commit message goes here"
```

Please note that since we now don't use an old-style linear ChangeLog (that tends to cause problems with distributed version control systems), you should ensure that your log messages are reasonably detailed. Use a docstring-like approach in the commit messages (including the second line being left *blank*):

```
Single line summary of  changes being committed.

* more details when warranted ...
* including crediting outside contributors if they sent the
  code/bug/idea!
```

As you work, you will repeat this edit/commit cycle many times. If you work on your branch for a long time, you will also want to get the latest changes from the `lp:ipython` branch. This can be done with the following sequence of commands:

```
$ ls
ipython
ipython-mybranch

$ cd ipython
$ bzr pull
$ cd ../ipython-mybranch
$ bzr merge ../ipython
$ bzr commit -m "Merging changes from trunk"
```

Along the way, you should also run the IPython test suite. You can do this using the **iptest** command (which is basically a customized version of **nosetests**):

```
$ cd
$ iptest
```

The **iptest** command will also pick up and run any tests you have written. See *_devel_testing* for further details on the testing system.

Post your branch and request a code review

Once you are done with your edits, you should post your branch on Launchpad so that other IPython developers can review the changes and help you merge your changes into the main development branch. To post your branch on Launchpad, do:

```
$ cd ipython-mybranch
$ bzr push lp:~yourusername/ipython/ipython-mybranch
```

Then, go to the IPython Launchpad site, and you should see your branch under the "Code" tab. If you click on your branch, you can provide a short description of the branch as well as mark its status. Most importantly, you should click the link that reads "Propose for merging into another branch". What does this do?

This let's the other IPython developers know that your branch is ready to be reviewed and merged into the main development branch. During this review process, other developers will give you feedback and help you get your code ready to be merged. What types of things will we be looking for:

- All code is documented.
- All code has tests.
- The entire IPython test suite passes.

Once your changes have been reviewed and approved, someone will merge them into the main development branch.

9.1.3 Some notes for core developers when merging third-party contributions

Core developers, who ultimately merge any approved branch (from themselves, another developer, or any third-party contribution) will typically use **bzr merge** to merge the branch into the trunk and push it to the main Launcphad site. This is a short list of things to keep in mind when doing this process, so that the project history is easy to understand in the long run, and that generating release notes is as painless and accurate as possible.

- When you merge any non-trivial functionality (from one small bug fix to a big feature branch), please remember to always edit the **changes_** file accordingly. This file has one main section for each release, and if you edit it as you go, noting what new features, bug fixes or API changes you have made, the release notes will be almost finished when they are needed later. This is much easier if done when you merge the work, rather than weeks or months later by re-reading a massive Bazaar log.
- When big merges are done, the practice of putting a summary commit message in the merge is *extremely* useful. It makes this kind of job much nicer, because that summary log message can be almost copy/pasted without changes, if it was well written, rather than dissecting the next-level messages from the individual commits.
- It's important that we remember to always credit who gave us something if it's not the committer. In general, we have been fairly good on this front, this is just a reminder to keep things up. As a note, if you are ever committing something that is completely (or almost so) a third-party contribution, do the commit as:

  ```
  $ bzr commit --author="Someone Else"
  ```

  This way it will show that name separately in the log, which makes it even easier to spot. Obviously we often rework third party contributions extensively, but this is still good to keep in mind for cases when we don't touch the code too much.

9.1.4 Documentation

Standalone documentation

All standalone documentation should be written in plain text (`.txt`) files using reStructuredText [reStructuredText] for markup and formatting. All such documentation should be placed in directory `docs/source` of the IPython source tree. The documentation in this location will serve as the main source for IPython documentation and all existing documentation should be converted to this format.

To build the final documentation, we use Sphinx [Sphinx]. Once you have Sphinx installed, you can build the html docs yourself by doing:

```
$ cd ipython-mybranch/docs
$ make html
```

Docstring format

Good docstrings are very important. All new code should have docstrings that are formatted using reStructuredText for markup and formatting, since it is understood by a wide variety of tools. Details about using

reStructuredText for docstrings can be found here.

Additional PEPs of interest regarding documentation of code:

- Docstring Conventions
- Docstring Processing System Framework
- Docutils Design Specification

9.1.5 Coding conventions

General

In general, we'll try to follow the standard Python style conventions as described here:

- Style Guide for Python Code

Other comments:

- In a large file, top level classes and functions should be separated by 2-3 lines to make it easier to separate them visually.
- Use 4 spaces for indentation.
- Keep the ordering of methods the same in classes that have the same methods. This is particularly true for classes that implement an interface.

Naming conventions

In terms of naming conventions, we'll follow the guidelines from the Style Guide for Python Code.

For all new IPython code (and much existing code is being refactored), we'll use:

- All `lowercase` module names.
- `CamelCase` for class names.
- `lowercase_with_underscores` for methods, functions, variables and attributes.

There are, however, some important exceptions to these rules. In some cases, IPython code will interface with packages (Twisted, Wx, Qt) that use other conventions. At some level this makes it impossible to adhere to our own standards at all times. In particular, when subclassing classes that use other naming conventions, you must follow their naming conventions. To deal with cases like this, we propose the following policy:

- If you are subclassing a class that uses different conventions, use its naming conventions throughout your subclass. Thus, if you are creating a Twisted Protocol class, used Twisted's `namingSchemeForMethodsAndAttributes.`
- All IPython's official interfaces should use our conventions. In some cases this will mean that you need to provide shadow names (first implement `fooBar` and then `foo_bar = fooBar`). We want to avoid this at all costs, but it will probably be necessary at times. But, please use this sparingly!

Implementation-specific *private* methods will use `_single_underscore_prefix`. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).

Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like `runlines()` where `runsource()` and `runcode()` had established precedent).

The old IPython codebase has a big mix of classes and modules prefixed with an explicit `IP`. In Python this is mostly unnecessary, redundant and frowned upon, as namespaces offer cleaner prefixing. The only case where this approach is justified is for classes which are expected to be imported into external namespaces and a very generic name (like Shell) is too likely to clash with something else. We'll need to revisit this issue as we clean up and refactor the code, but in general we should remove as many unnecessary `IP`/`ip` prefixes as possible. However, if a prefix seems absolutely necessary the more specific `IPY` or `ipy` are preferred.

9.1.6 Testing system

It is extremely important that all code contributed to IPython has tests. Tests should be written as unittests, doctests or as entities that the Nose [Nose] testing package will find. Regardless of how the tests are written, we will use Nose for discovering and running the tests. Nose will be required to run the IPython test suite, but will not be required to simply use IPython.

Tests of Twisted using code need to follow two additional guidelines:

1. Twisted using tests should be written by subclassing the `TestCase` class that comes with `twisted.trial.unittest`.
2. All `Deferred` instances that are created in the test must be properly chained and the final one *must* be the return value of the test method.

When these two things are done, Nose will be able to run the tests and the twisted reactor will be handled correctly.

Each subpackage in IPython should have its own `tests` directory that contains all of the tests for that subpackage. This allows each subpackage to be self-contained. A good convention to follow is to have a file named `test_foo.py` for each module `foo.py` in the package. This makes it easy to organize the tests, though like most conventions, it's OK to break it if logic and common sense dictate otherwise.

If a subpackage has any dependencies beyond the Python standard library, the tests for that subpackage should be skipped if the dependencies are not found. This is very important so users don't get tests failing simply because they don't have dependencies. We ship a set of decorators in the `IPython.testing` package to tag tests that may be platform-specific or otherwise may have restrictions; if the existing ones don't fit your needs, add a new decorator in that location so other tests can reuse it.

To run the IPython test suite, use the **iptest** command that is installed with IPython (if you are using IPython in-place, without installing it, you can find this script in the `scripts` directory):

```
$ iptest
```

This command colects all IPython tests into separate groups, and then calls either Nose with the proper options and extensions, or Twisted's **trial**. This ensures that tests that need the Twisted reactor management

facilities execute separate of Nose. If any individual test group fails, **iptest** will print what you need to type so you can rerun that particular test group alone for debugging.

By default, **iptest** runs the entire IPython test suite (skipping tests that may be platform-specific or which depend on tools you may not have). But you can also use it to run only one specific test file, or a specific test function. For example, this will run only the `test_magic` file from the test suite:

```
$ iptest IPython.tests.test_magic
----------------------------------------------------------------------
Ran 10 tests in 0.348s

OK (SKIP=3)
Deleting object: second_pass
```

while the `path:function` syntax allows you to select a specific function in that file to run:

```
$ iptest IPython.tests.test_magic:test_obj_del
----------------------------------------------------------------------
Ran 1 test in 0.204s

OK
```

Since **iptest** is based on nosetests, you can pass it any regular nosetests option. For example, you can use `--pdb` or `--pdb-failures` to automatically activate the interactive Pdb debugger on errors or failures. See the nosetests documentation for further details.

A few tips for writing tests

You can write tests either as normal test files, using all the conventions that Nose recognizes, or as doctests. Note that *all* IPython functions should have at least one example that serves as a doctest, whenever technically feasible. However, example doctests should only be in the main docstring if they are *a good example*, i.e. if they convey useful information about the function. If you simply would like to write a test as a doctest, put it in a separate test file and write a no-op function whose only purpose is its docstring.

Note, however, that in a file named `test_X`, functions whose only test is their docstring (as a doctest) and which have no test functionality of their own, should be called *doctest_foo* instead of *test_foo*, otherwise they get double-counted (the empty function call is counted as a test, which just inflates tests numbers artificially). This restriction does not apply to functions in files with other names, due to how Nose discovers tests.

You can use IPython examples in your docstrings. Those can make full use of IPython functionality (magics, variable substitution, etc), but be careful to keep them generic enough that they run identically on all Operating Systems.

The prompts in your doctests can be either of the plain Python `>>>` variety or `In [1]:` IPython style. Since this is the IPython system, after all, we encourage you to use IPython prompts throughout, unless you are illustrating a specific aspect of the normal prompts (such as the `%doctest_mode` magic).

If a test isn't safe to run inside the main nose process (e.g. because it loads a GUI toolkit), consider running it in a subprocess and capturing its output for evaluation and test decision later. Here is an example of how to do it, by relying on the builtin `_ip` object that contains the public IPython api as defined in `IPython.ipapi`:

```
def test_obj_del():
    """Test that object's __del__ methods are called on exit."""
    test_dir = os.path.dirname(__file__)
    del_file = os.path.join(test_dir,'obj_del.py')
    out = _ip.IP.getoutput('ipython %s' % del_file)
    nt.assert_equals(out,'object A deleted')
```

If a doctest contains input whose output you don't want to verify identically via doctest (random output, an object id, etc), you can mark a docstring with `#random`. All of these test will have their code executed but no output checking will be done:

```
>>> 1+3
junk goes here...   # random

>>> 1+2
again,  anything goes #random
if multiline, the random mark is only needed once.

>>> 1+2
You can also put the random marker at the end:
# random

>>> 1+2
# random
.. or at the beginning.
```

In a case where you want an *entire* docstring to be executed but not verified (this only serves to check that the code runs without crashing, so it should be used very sparingly), you can put `# all-random` in the docstring.

9.1.7 Release checklist

Most of the release process is automated by the `release` script in the `tools` directory. This is just a handy reminder for the release manager.

1. First, run `build_release`, which does all the file checking and building that the real release script will do. This will let you do test installations, check that the build procedure runs OK, etc. You may want to disable a few things like multi-version RPM building while testing, because otherwise the build takes really long.
2. Run the release script, which makes the tar.gz, eggs and Win32 .exe installer. It posts them to the site and registers the release with PyPI.
3. Updating the website with announcements and links to the updated changes.txt in html form. Remember to put a short note both on the news page of the site and on Launcphad.
4. Drafting a short release announcement with i) highlights and ii) a link to the html changes.txt.
5. Make sure that the released version of the docs is live on the site.
6. Celebrate!

9.1.8 Porting to 3.0

There are no definite plans for porting of IPython to python 3. The major issue is the dependency on twisted framework for the networking/threading stuff. It is possible that it the traditional IPython interactive console could be ported more easily since it has no such dependency. Here are a few things that will need to be considered when doing such a port especially if we want to have a codebase that works directly on both 2.x and 3.x.

1. The syntax for exceptions changed (PEP 3110). The old *except exc, var* changed to *except exc as var*. At last count there was 78 occurences of this usage in the codebase. This is a particularly problematic issue, because it's not easy to implement it in a 2.5-compatible way.

Because it is quite difficult to support simultaneously Python 2.5 and 3.x, we will likely at some point put out a release that requires strictly 2.6 and abandons 2.5 compatibility. This will then allow us to port the code to using `print()` as a function, *except exc as var* syntax, etc. But as of version 0.11 at least, we will retain Python 2.5 compatibility.

9.2 Coding guide

9.2.1 Coding conventions

In general, we'll try to follow the standard Python style conventions as described in Python's PEP 8, the official Python Style Guide.

Other comments:

- In a large file, top level classes and functions should be separated by 2-3 lines to make it easier to separate them visually.
- Use 4 spaces for indentation, *never* use hard tabs.
- Keep the ordering of methods the same in classes that have the same methods. This is particularly true for classes that implement similar interfaces and for interfaces that are similar.

Naming conventions

In terms of naming conventions, we'll follow the guidelines of PEP 8. Some of the existing code doesn't honor this perfectly, but for all new IPython code (and much existing code is being refactored), we'll use:

- All `lowercase` module names.
- `CamelCase` for class names.
- `lowercase_with_underscores` for methods, functions, variables and attributes.

This may be confusing as some of the existing codebase uses a different convention (`lowerCamelCase` for methods and attributes). Slowly, we will move IPython over to the new convention, providing shadow names for backward compatibility in public interfaces.

There are, however, some important exceptions to these rules. In some cases, IPython code will interface with packages (Twisted, Wx, Qt) that use other conventions. At some level this makes it impossible to adhere

to our own standards at all times. In particular, when subclassing classes that use other naming conventions, you must follow their naming conventions. To deal with cases like this, we propose the following policy:

- If you are subclassing a class that uses different conventions, use its naming conventions throughout your subclass. Thus, if you are creating a Twisted Protocol class, used Twisted's `namingSchemeForMethodsAndAttributes.`
- All IPython's official interfaces should use our conventions. In some cases this will mean that you need to provide shadow names (first implement `fooBar` and then `foo_bar = fooBar`). We want to avoid this at all costs, but it will probably be necessary at times. But, please use this sparingly!

Implementation-specific *private* methods will use `_single_underscore_prefix`. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).

Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like `runlines()` where `runsource()` and `runcode()` had established precedent).

The old IPython codebase has a big mix of classes and modules prefixed with an explicit `IP`. In Python this is mostly unnecessary, redundant and frowned upon, as namespaces offer cleaner prefixing. The only case where this approach is justified is for classes which are expected to be imported into external namespaces and a very generic name (like Shell) is too likely to clash with something else. We'll need to revisit this issue as we clean up and refactor the code, but in general we should remove as many unnecessary `IP`/`ip` prefixes as possible. However, if a prefix seems absolutely necessary the more specific `IPY` or `ipy` are preferred.

9.2.2 Testing system

It is extremely important that all code contributed to IPython has tests. Tests should be written as unittests, doctests or as entities that the Nose testing package will find. Regardless of how the tests are written, we will use Nose for discovering and running the tests. Nose will be required to run the IPython test suite, but will not be required to simply use IPython.

Tests of Twisted using code should be written by subclassing the `TestCase` class that comes with `twisted.trial.unittest`. When this is done, Nose will be able to run the tests and the twisted reactor will be handled correctly.

Each subpackage in IPython should have its own `tests` directory that contains all of the tests for that subpackage. This allows each subpackage to be self-contained. If a subpackage has any dependencies beyond the Python standard library, the tests for that subpackage should be skipped if the dependencies are not found. This is very important so users don't get tests failing simply because they don't have dependencies.

We also need to look into use Noses ability to tag tests to allow a more modular approach of running tests.

9.2.3 Configuration system

IPython uses .ini files for configuration purposes. This represents a huge improvement over the configuration system used in IPython. IPython works with these files using the ConfigObj package, which IPython includes as `ipython1/external/configobj.py`.

Currently, we are using raw ConfigObj objects themselves. Each subpackage of IPython should contain a `config` subdirectory that contains all of the configuration information for the subpackage. To see how configuration information is defined (along with defaults) see at the examples in `ipython1/kernel/config` and `ipython1/core/config`. Likewise, to see how the configuration information is used, see examples in `ipython1/kernel/scripts/ipengine.py`.

Eventually, we will add a new layer on top of the raw ConfigObj objects. We are calling this new layer, `tconfig`, as it will use a Traits-like validation model. We won't actually use Traits, but will implement something similar in pure Python. But, even in this new system, we will still use ConfigObj and .ini files underneath the hood. Talk to Fernando if you are interested in working on this part of IPython. The current prototype of `tconfig` is located in the IPython sandbox.

9.3 Documenting IPython

9.3.1 Standalone documentation

All standalone documentation should be written in plain text (`.txt`) files using reStructuredText for markup and formatting. All such documentation should be placed in the top level directory `docs` of the IPython source tree. Or, when appropriate, a suitably named subdirectory should be used. The documentation in this location will serve as the main source for IPython documentation and all existing documentation should be converted to this format.

The actual HTML and PDF docs are built using the Sphinx documentation generation tool. Sphinx has been adopted as the default documentation tool for Python itself as of version 2.6, as well as by a number of projects that IPython is related with, such as numpy, scipy, matplotlib, sage and nipy.

The rest of this document is mostly taken from the matploblib documentation; we are using a number of Sphinx tools and extensions written by the matplotlib team and will mostly follow their conventions, which are nicely spelled out in their guide. What follows is thus a lightly adapted version of the matplotlib documentation guide, taken with permission from the MPL team.

A bit of Python code:

```
for i in range(10):
    print i,
print "A big number:",2**34
```

An interactive Python session:

```
>>> from IPython import genutils
>>> genutils.get_ipython_dir()
'/home/fperez/.ipython'
```

An IPython session:

```
In [7]: import IPython

In [8]: print "This IPython is version:",IPython.__version__
This IPython is version: 0.9.1
```

```
In [9]: 2+4
Out[9]: 6
```

A bit of shell code:

```
cd /tmp
echo "My home directory is: $HOME"
ls
```

9.3.2 Docstring format

Good docstrings are very important. Unfortunately, Python itself only provides a rather loose standard for docstrings (PEP 257), and there is no universally accepted convention for all the different parts of a complete docstring. However, the NumPy project has established a very reasonable standard, and has developed some tools to support the smooth inclusion of such docstrings in Sphinx-generated manuals. Rather than inventing yet another pseudo-standard, IPython will be henceforth documented using the NumPy conventions; we carry copies of some of the NumPy support tools to remain self-contained, but share back upstream with NumPy any improvements or fixes we may make to the tools.

The NumPy documentation guidelines contain detailed information on this standard, and for a quick overview, the NumPy example docstring is a useful read.

As in the past IPython used epydoc, currently many docstrings still use epydoc conventions. We will update them as we go, but all new code should be fully documented using the NumPy standard.

Additional PEPs of interest regarding documentation of code. While both of these were rejected, the ideas therein form much of the basis of docutils (the machinery to process reStructuredText):

- Docstring Processing System Framework
- Docutils Design Specification

9.4 Development roadmap

IPython is an ambitious project that is still under heavy development. However, we want IPython to become useful to as many people as possible, as quickly as possible. To help us accomplish this, we are laying out a roadmap of where we are headed and what needs to happen to get there. Hopefully, this will help the IPython developers figure out the best things to work on for each upcoming release.

9.4.1 Work targeted to particular releases

Release 0.10

- Initial refactor of **ipcluster**.
- Better TextMate integration.
- Merge in the daemon branch.

Release 0.11

- Refactor the configuration system and command line options for **ipengine** and **ipcontroller**. This will include the creation of cluster directories that encapsulate all the configuration files, log files and security related files for a particular cluster.
- Refactor **ipcluster** to support the new configuration system.
- Refactor the daemon stuff to support the new configuration system.
- Merge back in the core of the notebook.

Release 0.12

- Fully integrate process startup with the daemons for full process management.
- Make the capabilites of **ipcluster** available from simple Python classes.

9.4.2 Major areas of work

Refactoring the main IPython core

Process management for `IPython.kernel`

Configuration system

Performance problems

Currently, we have a number of performance issues that are waiting to bite users:

- The controller stores a large amount of state in Python dictionaries. Under heavy usage, these dicts with get very large, causing memory usage problems. We need to develop more scalable solutions to this problem, such as using a sqlite database to store this state. This will also help the controller to be more fault tolerant.
- We currently don't have a good way of handling large objects in the controller. The biggest problem is that because we don't have any way of streaming objects, we get lots of temporary copies in the low-level buffers. We need to implement a better serialization approach and true streaming support.
- The controller currently unpickles and repickles objects. We need to use the [push|pull]_serialized methods instead.
- Currently the controller is a bottleneck. The best approach for this is to separate the controller itself into multiple processes, one for the core controller and one each for the controller interfaces.

9.5 IPython.kernel.core.notification blueprint

9.5.1 Overview

The `IPython.kernel.core.notification` module will provide a simple implementation of a notification center and support for the observer pattern within the `IPython.kernel.core`. The main intended use case is to provide notification of Interpreter events to an observing frontend during the execution of a single block of code.

9.5.2 Functional Requirements

The notification center must:

- Provide synchronous notification of events to all registered observers.
- Provide typed or labeled notification types.
- Allow observers to register callbacks for individual or all notification types.
- Allow observers to register callbacks for events from individual or all notifying objects.
- Notification to the observer consists of the notification type, notifying object and user-supplied extra information [implementation: as keyword parameters to the registered callback].
- Perform as O(1) in the case of no registered observers.
- Permit out-of-process or cross-network extension.

9.5.3 What's not included

As written, the `IPython.kernel.core.notificaiton` module does not:

- Provide out-of-process or network notifications (these should be handled by a separate, Twisted aware module in `IPython.kernel`).
- Provide zope.interface-style interfaces for the notification system (these should also be provided by the `IPython.kernel` module).

9.5.4 Use Cases

The following use cases describe the main intended uses of the notificaiton module and illustrate the main success scenario for each use case:

1. Dwight Schroot is writing a frontend for the IPython project. His frontend is stuck in the stone age and must communicate synchronously with an IPython.kernel.core.Interpreter instance. Because code is executed in blocks by the Interpreter, Dwight's UI freezes every time he executes a long block of code. To keep track of the progress of his long running block, Dwight adds the following code to his frontend's set-up code:

```
from IPython.kernel.core.notification import NotificationCenter
center = NotificationCenter.sharedNotificationCenter
center.registerObserver(self, type=IPython.kernel.core.Interpreter.STDOUT_N
```

and elsewhere in his front end:

```
def stdout_notification(self, type, notifying_object, out_string=None):
    self.writeStdOut(out_string)
```

If everything works, the Interpreter will (according to its published API) fire a notification via the `IPython.kernel.core.notification.sharedCenter` of type `STD_OUT_NOTIFICATION_TYPE` before writing anything to stdout [it's up to the Intereter implementation to figure out when to do this]. The notificaiton center will then call the registered callbacks for that event type (in this case, Dwight's frontend's stdout_notification method). Again, according to its API, the Interpreter provides an additional keyword argument when firing the notificaiton of out_string, a copy of the string it will write to stdout.

Like magic, Dwight's frontend is able to provide output, even during long-running calculations. Now if Jim could just convince Dwight to use Twisted...

2. Boss Hog is writing a frontend for the IPython project. Because Boss Hog is stuck in the stone age, his frontend will be written in a new Fortran-like dialect of python and will run only from the command line. Because he doesn't need any fancy notification system and is used to worrying about every cycle on his rat-wheel powered mini, Boss Hog is adamant that the new notification system not produce any performance penalty. As they say in Hazard county, there's no such thing as a free lunch. If he wanted zero overhead, he should have kept using IPython 0.8. Instead, those tricky Duke boys slide in a suped-up bridge-out jumpin' awkwardly confederate-lovin' notification module that imparts only a constant (and small) performance penalty when the Interpreter (or any other object) fires an event for which there are no registered observers. Of course, the same notificaiton-enabled Interpreter can then be used in frontends that require notifications, thus saving the IPython project from a nasty civil war.

3. Barry is wrting a frontend for the IPython project. Because Barry's front end is the *new hotness*, it uses an asynchronous event model to communicate with a Twisted `engineservice` that communicates with the IPython `Interpreter`. Using the `IPython.kernel.notification` module, an asynchronous wrapper on the `IPython.kernel.core.notification` module, Barry's frontend can register for notifications from the interpreter that are delivered asynchronously. Even if Barry's frontend is running on a separate process or even host from the Interpreter, the notifications are delivered, as if by dark and twisted magic. Just like Dwight's frontend, Barry's frontend can now recieve notifications of e.g. writing to stdout/stderr, opening/closing an external file, an exception in the executing code, etc.

9.6 Notes on the IPython configuration system

This document has some random notes on the configuration system.

To start, an IPython process needs:

- Configuration files
- Command line options

- Additional files (FURL files, extra scripts, etc.)

It feeds these things into the core logic of the process, and as output, produces:

- Log files
- Security files

There are a number of things that complicate this:

- A process may need to be started on a different host that doesn't have any of the config files or additional files. Those files need to be moved over and put in a staging area. The process then needs to be told about them.
- The location of the output files should somehow be set by config files or command line options.
- Our config files are very hierarchical, but command line options are flat, making it difficult to relate command line options to config files.
- Some processes (like ipcluster and the daemons) have to manage the input and output files for multiple different subprocesses, each possibly on a different host. Ahhhh!
- Our configurations are not singletons. A given user will likely have many different configurations for different clusters.

CHAPTER

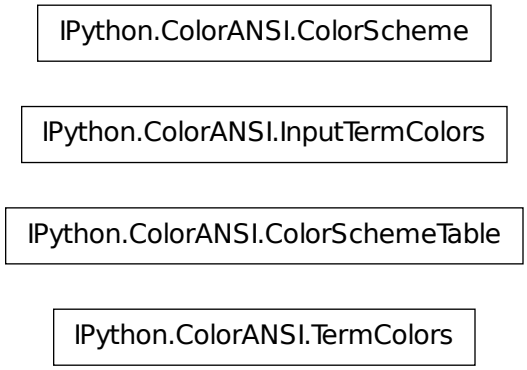
TEN

THE IPYTHON API

10.1 ColorANSI

10.1.1 Module: ColorANSI

Inheritance diagram for IPython.ColorANSI:



Tools for coloring text in ANSI terminals.

10.1.2 Classes

ColorScheme

class IPython.ColorANSI.**ColorScheme**(*_ColorScheme__scheme_name_*, *colordict=None*, ***colormap*)

Generic color scheme class. Just a name and a Struct.

__init__()

copy()
Return a full copy of the object, optionally renaming it.

ColorSchemeTable

class IPython.ColorANSI.**ColorSchemeTable**(*scheme_list=None*, *default_scheme=''*)
Bases: dict

General class to handle tables of color schemes.

It's basically a dict of color schemes with a couple of shorthand attributes and some convenient methods.

active_scheme_name -> obvious active_colors -> actual color table of the active scheme

__init__()
Create a table of color schemes.

The table can be created empty and manually filled or it can be created with a list of valid color schemes AND the specification for the default active scheme.

add_scheme()
Add a new color scheme to the table.

copy()
Return full copy of object

set_active_scheme()
Set the currently active scheme.

Names are by default compared in a case-insensitive way, but this can be changed by setting the parameter case_sensitive to true.

InputTermColors

class IPython.ColorANSI.**InputTermColors**
Color escape sequences for input prompts.

This class is similar to TermColors, but the escapes are wrapped in " and " so that readline can properly know the length of each line and can wrap lines accordingly. Use this class for any colored text which needs to be used in input prompts, such as in calls to raw_input().

This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a NoColor escape which is just the null string, suitable for defining 'dummy' color schemes in terminals which get confused by color escapes.

This class should be used as a mixin for building color schemes.

TermColors

class IPython.ColorANSI.**TermColors**
Color escape sequences.

This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a NoColor escape which is just the null string, suitable for defining 'dummy' color schemes in terminals which get confused by color escapes.

This class should be used as a mixin for building color schemes.

10.1.3 Function

`IPython.ColorANSI.`**`make_color_table`**`()`

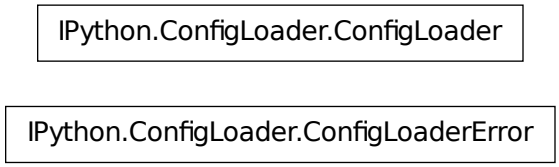
Build a set of color attributes in a class.

Helper function for building the *TermColors classes.

10.2 ConfigLoader

10.2.1 Module: `ConfigLoader`

Inheritance diagram for `IPython.ConfigLoader`:



Configuration loader

10.2.2 Classes

`ConfigLoader`

class `IPython.ConfigLoader.`**`ConfigLoader`**(*conflict=None*, *field_sep=None*, *reclimit=15*)

Configuration file loader capable of handling recursive inclusions and with parametrized conflict resolution for multiply found keys.

`__init__``()`

The reclimit parameter controls the number of recursive configuration file inclusions. This way we can stop early on (before python's own recursion limit is hit) if there is a circular inclusion.

•conflict: dictionary for conflict resolutions (see Struct.merge())

load()
Load a configuration file, return the resulting Struct.

Call: load_config(fname,convert=None,conflict=None,recurse_key='')

- fname: file to load from.
- convert: dictionary of type conversions (see read_dict())
- recurse_key: keyword in dictionary to trigger recursive file inclusions.

reset()

ConfigLoaderError

class IPython.ConfigLoader.**ConfigLoaderError**(*args=None*)
Bases: exceptions.Exception

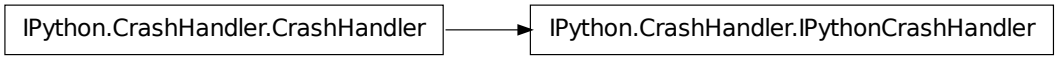
Exception for ConfigLoader class.

__init__()

10.3 CrashHandler

10.3.1 Module: CrashHandler

Inheritance diagram for IPython.CrashHandler:



sys.excepthook for IPython itself, leaves a detailed report on disk.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

10.3.2 Classes

CrashHandler

class IPython.CrashHandler.**CrashHandler**(*IP*, *app_name*, *contact_name*, *contact_email*, *bug_tracker*, *crash_report_fname*, *show_crash_traceback=True*)

Customizable crash handlers for IPython-based systems.

Instances of this class provide a __call__ method which can be used as a sys.excepthook, i.e., the __call__ signature is:

def __call__(self,etype, evalue, etb)

__init__()

New crash handler.

Inputs:

•IP: a running IPython instance, which will be queried at crash time

for internal information.

•app_name: a string containing the name of your application.

•contact_name: a string with the name of the person to contact.

•contact_email: a string with the email address of the contact.

•bug_tracker: a string with the URL for your project's bug tracker.

•crash_report_fname: a string with the filename for the crash report

to be saved in. These reports are left in the ipython user directory as determined by the running IPython instance.

Optional inputs:

•show_crash_traceback(True): if false, don't print the crash

traceback on stderr, only generate the on-disk report

Non-argument instance attributes:

These instances contain some non-argument attributes which allow for further customization of the crash handler's behavior. Please see the source for further details.

make_report()

Return a string containing a crash report.

IPythonCrashHandler

class IPython.CrashHandler.**IPythonCrashHandler**(*IP*)

Bases: IPython.CrashHandler.CrashHandler

sys.excepthook for IPython itself, leaves a detailed report on disk.

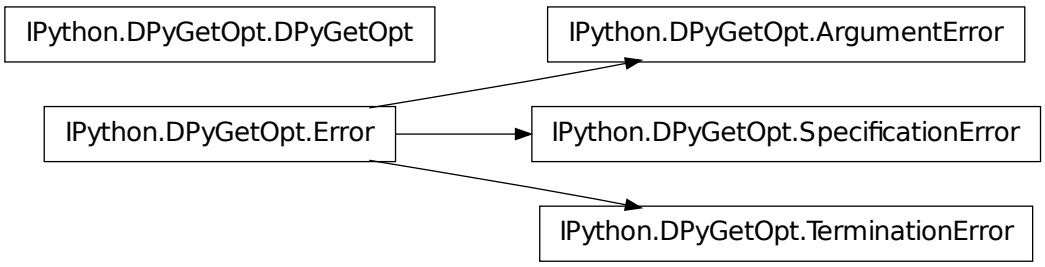
__init__()

make_report()
Return a string containing a crash report.

10.4 DPyGetOpt

10.4.1 Module: DPyGetOpt

Inheritance diagram for IPython.DPyGetOpt:



DPyGetOpt – Demiurge Python GetOptions Module

This module is modeled after perl's Getopt::Long module– which is, in turn, modeled after GNU's extended getopt() function.

Upon instantiation, the option specification should be a sequence (list) of option definitions.

Options that take no arguments should simply contain the name of the option. If a ! is post-pended, the option can be negated by prepending 'no'; ie 'debug!' specifies that -debug and -nodebug should be accepted.

Mandatory arguments to options are specified using a postpended '=' + a type specifier. '=s' specifies a mandatory string argument, '=i' specifies a mandatory integer argument, and '=f' specifies a mandatory real number. In all cases, the '=' can be substituted with ':' to specify that the argument is optional.

Dashes '-' in option names are allowed.

If an option has the character '@' postpended (after the argumentation specification), it can appear multiple times within each argument list that is processed. The results will be stored in a list.

The option name can actually be a list of names separated by '|' characters; ie– 'foo|bar|baz=f@' specifies that all -foo, -bar, and -baz options that appear on within the parsed argument list must have a real number argument and that the accumulated list of values will be available under the name 'foo'

10.4.2 Classes

ArgumentError

class IPython.DPyGetOpt.**ArgumentError**
Bases: IPython.DPyGetOpt.Error

Exception indicating an error in the arguments passed to DPyGetOpt.processArguments.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

DPyGetOpt

class IPython.DPyGetOpt.**DPyGetOpt**(*spec=None*, *terminators=[*, '–' *]*)

__init__()
Declare and intialize instance variables

Yes, declaration is not necessary... but one of the things I sorely miss from C/Obj-C is the concept of having an interface definition that clearly declares all instance variables and methods without providing any implementation

details. it is a useful reference!

all instance variables are initialized to 0/Null/None of the appropriate type– not even the default value...

addOptionConfigurationTuple()

addOptionConfigurationTuples()

addTerminator()
Adds newTerm as terminator of option processing.

Whenever the option processor encounters one of the terminators during option processing, the processing of options terminates immediately, all remaining options are stored in the termValues instance variable and the full name of the terminator is stored in the terminator instance variable.

ignoreCase()
Returns 1 if the option processor will ignore case when processing options.

isPosixCompliant()
Returns the value of the posix compliance flag.

parseConfiguration()

processArguments()
Processes args, a list of arguments (including options).

If args is the same as sys.argv, automatically trims the first argument (the executable name/path).

If an exception is not raised, the argument list was parsed correctly.

Upon successful completion, the freeValues instance variable will contain all the arguments that were not associated with an option in the order they were encountered. optionValues is a dictionary containing the value of each option– the method valueForOption() can be used to query this dictionary. terminator will contain the argument encountered that terminated option processing (or None, if a terminator was never encountered) and termValues will contain all of the options that appeared after the Terminator (or an empty list).

setAllowAbbreviations()
Enables and disables the expansion of abbreviations during option processing.

setIgnoreCase()
Enables and disables ignoring case during option processing.

setPosixCompliance()
Enables and disables posix compliance.

When enabled, '+' can be used as an option prefix and free values can be mixed with options.

valueForOption()
Return the value associated with optionName. If optionName was not encountered during parsing of the arguments, returns the defaultValue (which defaults to None).

willAllowAbbreviations()
Returns 1 if abbreviated options will be automatically expanded to the non-abbreviated form (instead of causing an unrecognized option error).

Error

class IPython.DPyGetOpt.**Error**
Bases: exceptions.Exception

Base class for exceptions in the DPyGetOpt module.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

SpecificationError

class IPython.DPyGetOpt.**SpecificationError**
Bases: IPython.DPyGetOpt.Error

Exception indicating an error with an option specification.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

TerminationError

class IPython.DPyGetOpt.**TerminationError**
Bases: IPython.DPyGetOpt.Error

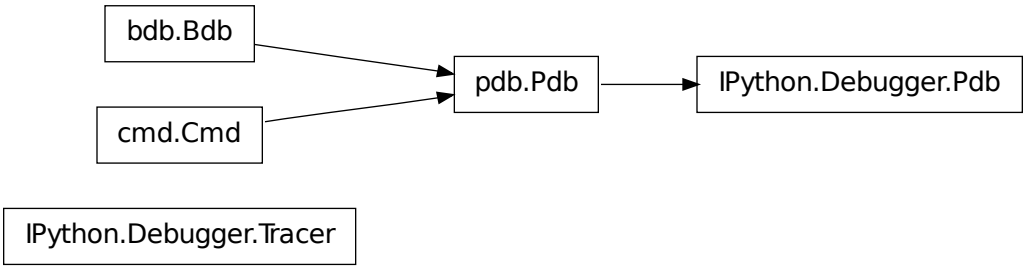
Exception indicating an error with an option processing terminator.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.5 Debugger

10.5.1 Module: Debugger

Inheritance diagram for IPython.Debugger:



Pdb debugger class.

Modified from the standard pdb.Pdb class to avoid including readline, so that the command line completion of other programs which include this isn't damaged.

In the future, this class will be expanded with improvements over the standard pdb.

The code in this file is mainly lifted out of cmd.py in Python 2.2, with minor changes. Licensing should therefore be under the standard Python terms. For details on the PSF (Python Software Foundation) standard license, see:

http://www.python.org/2.2.3/license.html

10.5.2 Classes

Pdb

class IPython.Debugger.**Pdb**(*color_scheme='NoColor'*, *completekey=None*, *stdin=None*, *stdout=None*)
Bases: pdb.Pdb

Modified Pdb class, does not load readline.

__init__()

checkline()
Check whether specified line seems to be executable.

Return *lineno* if it is, 0 if not (e.g. a docstring, comment, blank line or EOF). Warning: testing is not comprehensive.

do_d()

do_down()

do_l()

do_list()

do_pdef()
The debugger interface to magic_pdef

do_pdoc()
The debugger interface to magic_pdoc

do_pinfo()
The debugger equivalant of ?obj

do_q()

do_quit()

do_u()

do_up()

format_stack_entry()

interaction()

list_command_pydb()
List command to use if we have a newer pydb installed

new_do_down()

new_do_frame()

new_do_quit()

new_do_restart()
Restart command. In the context of ipython this is exactly the same thing as 'quit'.

new_do_up()

postloop()

print_list_lines()
The printing (as opposed to the parsing part of a 'list' command.

print_stack_entry()

print_stack_trace()

set_colors()
Shorthand access to the color table scheme selector method.

Tracer

class IPython.Debugger.**Tracer**(*colors=None*)

Bases: object

Class for local debugging, similar to pdb.set_trace.

Instances of this class, when called, behave like pdb.set_trace, but providing IPython's enhanced capabilities.

This is implemented as a class which must be initialized in your own code and not as a standalone function because we need to detect at runtime whether IPython is already active or not. That detection is done in the constructor, ensuring that this code plays nicely with a running IPython, while functioning acceptably (though with limitations) if outside of it.

__init__()

Create a local debugger instance.

Parameters

- *colors* (None): a string containing the name of the color scheme to

use, it must be one of IPython's valid color schemes. If not given, the function will default to the current IPython scheme when running inside IPython, and to 'NoColor' otherwise.

Usage example:

from IPython.Debugger import Tracer; debug_here = Tracer()

... later in your code debug_here() # -> will open up the debugger at that point.

Once the debugger activates, you can use all of its regular commands to step through code, set breakpoints, etc. See the pdb documentation from the Python standard library for usage details.

10.5.3 Functions

IPython.Debugger.**BdbQuit_IPython_excepthook**()

IPython.Debugger.**BdbQuit_excepthook**()

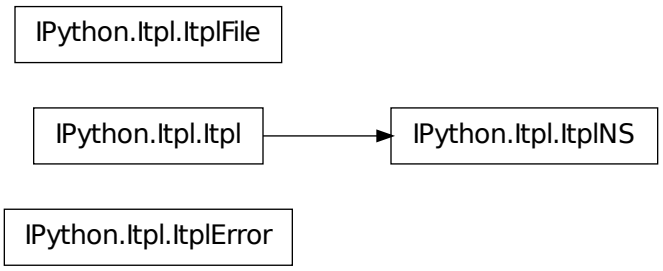
IPython.Debugger.**decorate_fn_with_doc**()

Make new_fn have old_fn's doc string. This is particularly useful for the **do_**... commands that hook into the help system. Adapted from from a comp.lang.python posting by Duncan Booth.

10.6 Itpl

10.6.1 Module: Itpl

Inheritance diagram for IPython.Itpl:



String interpolation for Python (by Ka-Ping Yee, 14 Feb 2000).

This module lets you quickly and conveniently interpolate values into strings (in the flavour of Perl or Tcl, but with less extraneous punctuation). You get a bit more power than in the other languages, because this module allows subscripting, slicing, function calls, attribute lookup, or arbitrary expressions. Variables and expressions are evaluated in the namespace of the caller.

The itpl() function returns the result of interpolating a string, and printpl() prints out an interpolated string. Here are some examples:

> from Itpl import printpl printpl("Here is a $string.") printpl("Here is a $module.member.") printpl("Here is an $object.member.") printpl("Here is a $functioncall(with, arguments).") printpl("Here is an ${arbitrary + expression}.") printpl("Here is an $array[3] member.") printpl("Here is a $dictionary['member'].")

The filter() function filters a file object so that output through it is interpolated. This lets you produce the illusion that Python knows how to do interpolation:

> import Itpl sys.stdout = Itpl.filter() f = "fancy" print "Is this not $f?" print "Standard output has been replaced with a $sys.stdout object." sys.stdout = Itpl.unfilter() print "Okay, back $to $normal."

Under the hood, the Itpl class represents a string that knows how to interpolate values. An instance of the class parses the string once upon initialization; the evaluation and substitution can then be done each time the instance is evaluated with str(instance). For example:

> from Itpl import Itpl s = Itpl("Here is $foo.") foo = 5 print str(s) foo = "bar" print str(s)

10.6.2 Classes

Itpl

class IPython.Itpl.**Itpl**(*format*, *codec='UTF-8'*, *encoding_errors='backslashreplace'*)

Class representing a string with interpolation abilities.

Upon creation, an instance works out what parts of the format string are literal and what parts need to be evaluated. The evaluation and substitution happens in the namespace of the caller when

str(instance) is called.

__init__()

The single mandatory argument to this constructor is a format string.

The format string is parsed according to the following rules:

1. **A dollar sign and a name, possibly followed by any of:**
   - an open-paren, and anything up to the matching paren
   - an open-bracket, and anything up to the matching bracket
   - a period and a name

   any number of times, is evaluated as a Python expression.
2. A dollar sign immediately followed by an open-brace, and anything up to the matching close-brace, is evaluated as a Python expression.
3. Outside of the expressions described in the above two rules, two dollar signs in a row give you one literal dollar sign.

Optional arguments:

•codec('utf_8'): a string containing the name of a valid Python

codec.

•encoding_errors('backslashreplace'): a string with a valid error handling

policy. See the codecs module documentation for details.

These are used to encode the format string if a call to str() fails on the expanded result.

ItplError

class IPython.Itpl.**ItplError**(*text*, *pos*)

Bases: exceptions.ValueError

__init__()

ItplFile

class IPython.Itpl.**ItplFile**(*file*)

A file object that filters each write() through an interpolator.

__init__()

write()

ItplNS

class IPython.Itpl.**ItplNS**(*format*, *globals*, *locals=None*, *codec='utf_8'*, *encoding_errors='backslashreplace'*)

Bases: IPython.Itpl.Itpl

Class representing a string with interpolation abilities.

This inherits from Itpl, but at creation time a namespace is provided where the evaluation will occur. The interpolation becomes a bit more efficient, as no traceback needs to be extracte. It also allows the caller to supply a different namespace for the interpolation to occur than its own.

__init__()

ItplNS(format,globals[,locals]) -> interpolating string instance.

This constructor, besides a format string, takes a globals dictionary and optionally a locals (which defaults to globals if not provided).

For further details, see the Itpl constructor.

10.6.3 Functions

IPython.Itpl.**filter**()

Return an ItplFile that filters writes to the given file object.

'file = filter(file)' replaces 'file' with a filtered object that has a write() method. When called with no argument, this creates a filter to sys.stdout.

IPython.Itpl.**itpl**()

IPython.Itpl.**itplns**()

IPython.Itpl.**matchorfail**()

IPython.Itpl.**printpl**()

IPython.Itpl.**printplns**()

IPython.Itpl.**unfilter**()

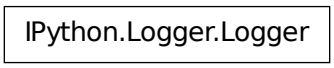
Return the original file that corresponds to the given ItplFile.

'file = unfilter(file)' undoes the effect of 'file = filter(file)'. 'sys.stdout = unfilter()' undoes the effect of 'sys.stdout = filter()'.

10.7 Logger

10.7.1 Module: Logger

Inheritance diagram for IPython.Logger:



Logger class for IPython's logging facilities.

10.7.2 Logger

class IPython.Logger.**Logger**(*shell*, *logfname='Logger.log'*, *loghead=''*, *logmode='over'*)
Bases: object

A Logfile class with different policies for file creation

__init__()

close_log()
Fully stop logging and close log file.

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

log()
Write the line to a log and create input cache variables _i*.

Inputs:

•line_ori: unmodified input line from the user. This is not

necessarily valid Python.

•line_mod: possibly modified input, such as the transformations made

by input prefilters or input handlers of various kinds. This should always be valid Python.

•continuation: if True, indicates this is part of multi-line input.

log_write()
Write data to the log file, if active

logmode

logstart()
Generate a new log-file with a default header.

Raises RuntimeError if the log has already been started

logstate()
Print a status message about the logger.

logstop()
Fully stop logging and close log file.

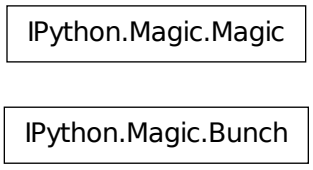
In order to start logging again, a new logstart() call needs to be made, possibly (though not necessarily) with a new filename, mode and other options.

switch_log()
Switch logging on/off. val should be ONLY a boolean.

10.8 Magic

10.8.1 Module: Magic

Inheritance diagram for IPython.Magic:



Magic functions for InteractiveShell.

10.8.2 Classes

Bunch

class IPython.Magic.**Bunch**

Magic

class IPython.Magic.**Magic**(*shell*)
Magic functions for InteractiveShell.

Shell functions which can be reached as %function_name. All magic functions should accept a string, which they can parse for their own needs. This can make some functions easier to type, eg *%cd ../* vs. *%cd("../")*

ALL definitions MUST begin with the prefix **magic_**. The user won't need it at the command line, but it is is needed in the definition.

__init__()

arg_err()
Print docstring if incorrect arguments were passed

default_option()
Make an entry in the options_table for fn, with value optstr

extract_input_slices()
Return as a string a set of input history slices.

Inputs:

•slices: the set of slices is given as a list of strings (like

['1','4:8','9'], since this function is for use by magic functions which get their arguments as strings.

Optional inputs:

•raw(False): by default, the processed input is used. If this is

true, the raw input history is used instead.

Note that slices can be called with two notations:

N:M -> standard python form, means including items N...(M-1).

N-M -> include items N..M (closed endpoint).

format_latex()
Format a string for latex inclusion.

format_screen()
Format a string for screen printing.

This removes some latex-type format codes.

lsmagic()
Return a list of currently available magic functions.

Gives a list of the bare names after mangling (['ls','cd', ...], not ['magic_ls','magic_cd',...]

magic_Exit()
Exit IPython without confirmation.

magic_Pprint()
Toggle pretty printing on/off.

magic_alias()
Define an alias for a system command.

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

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

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

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

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

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

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

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

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

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

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

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

magic_autocall()

Make functions callable without having to type parentheses.

Usage:

%autocall [mode]

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

In more detail, these values mean:

0 -> fully disabled

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

In this mode, you get:

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

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

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

In [2]: float ——> float() Out[2]: 0.0

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

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

all-random (note for auto-testing)

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

magic_automagic()
Make magic functions callable without having to type the initial %.

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

•on,1,True: to activate

•off,0,False: to deactivate.

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

magic_bg()
Run a job in the background, in a separate thread.

For example,

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

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

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

to assign this result to variable 'myvar'.

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

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

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

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

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

Jobs = __builtins__.jobs

magic_bookmark()
Manage IPython's bookmark system.

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

You can later on access a bookmarked folder with: %cd -b <name>

or simply '%cd <name>' if there is no directory called <name> AND there is such a bookmark defined.

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

magic_cd()

Change the current working directory.

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

Usage:

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

cd -: changes to the last visited directory.

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

cd –foo: change to directory that matches 'foo' in history

cd -b <bookmark_name>: jump to a bookmark set by %bookmark

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

Options:

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

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

magic_color_info()

Toggle color_info.

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

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

magic_colors()

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

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

magic_cpaste()
Allows you to paste & execute a pre-formatted code block from clipboard.

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

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

You can also pass a variable name as an argument, e.g. '%cpaste foo'. This assigns the pasted block to variable 'foo' as string, without dedenting or executing it (preceding >>> and + is still stripped)

'%cpaste -r' re-executes the block previously entered by cpaste.

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

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

See Also:

paste automatically pull code from clipboard.

magic_debug()
Activate the interactive debugger in post-mortem mode.

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

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

magic_dhist()
Print your history of visited directories.

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

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

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

magic_dirs()
Return the current directory stack.

magic_doctest_mode()
Toggle doctest mode on and off.

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

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

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

magic_ed()
Alias to %edit.

magic_edit()
Bring up an editor and execute the resulting code.

Usage: %edit [options] [args]

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

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

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

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

Options:

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

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

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

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

Arguments:

If arguments are given, the following possibilites exist:

•The arguments are numbers or pairs of colon-separated numbers (like

1 4:8 9). These are interpreted as lines of previous input to be loaded into the editor. The syntax is the same of the %macro command.

•If the argument doesn't start with a number, it is evaluated as a

variable and its contents loaded into the editor. You can thus edit any string which contains python code (including the result of previous edits).

•If the argument is the name of an object (other than a string),

IPython will try to locate the file where it was defined and open the editor at the point where it is defined. You can use *%edit function* to load an editor exactly at the point where 'function' is defined, edit it and have the file be executed automatically.

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

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

•If the argument is not found as a variable, IPython will look for a

file with that name (adding .py if necessary) and load it into the editor. It will execute its contents with execfile() when you exit, loading any code in the file into your interactive namespace.

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

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

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

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

We can then call the function foo():

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

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

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

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

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

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

In [5]: ed Editing... done. Executing edited code... hello Out[5]: "print 'hello'n"

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

In [6]: ed _ Editing... done. Executing edited code... hello world Out[6]: "print 'hello world'n"

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

In [7]: ed _8 Editing... done. Executing edited code... hello again Out[7]: "print 'hello again'n"

Changing the default editor hook:

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

magic_env()
List environment variables.

magic_exit()
Exit IPython, confirming if configured to do so.

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

magic_logoff()
Temporarily stop logging.

You must have previously started logging.

magic_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.

magic_logstart()
Start logging anywhere in a session.

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

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

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

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

append: well, that says it.backup: rename (if exists) to name~ and start name.global: single logfile in your home dir, appended to.over : overwrite existing log.rotate: create rotating logs name.1~, name.2~, etc.

Options:

-o: log also IPython's output. In this mode, all commands which generate an Out[NN] prompt are recorded to the logfile, right after their corresponding input line. The output

lines are always prepended with a '#[Out]# ' marker, so that the log remains valid Python code.

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

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

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

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

magic_logstate()
Print the status of the logging system.

magic_logstop()
Fully stop logging and close log file.

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

magic_lsmagic()
List currently available magic functions.

magic_macro()
Define a set of input lines as a macro for future re-execution.

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

Options:

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

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

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

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

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

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

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

In [55]: %macro my_macro 44-47 49

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

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

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

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

'print macro_name'.

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

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

magic_magic()
Print information about the magic function system.

Supported formats: -latex, -brief, -rest

magic_page()
Pretty print the object and display it through a pager.

%page [options] OBJECT

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

Options:

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

magic_paste()
Allows you to paste & execute a pre-formatted code block from clipboard.

The text is pulled directly from the clipboard without user intervention.

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

You can also pass a variable name as an argument, e.g. '%paste foo'. This assigns the pasted block to variable 'foo' as string, without dedenting or executing it (preceding >>> and + is still stripped)

'%paste -r' re-executes the block previously entered by cpaste.

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

See Also:

cpaste manually paste code into terminal until you mark its end.

magic_pdb()
Control the automatic calling of the pdb interactive debugger.

Call as '%pdb on', '%pdb 1', '%pdb off' or '%pdb 0'. If called without argument it works as a toggle.

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

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

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

magic_pdef()
Print the definition header for any callable object.

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

magic_pdoc()
Print the docstring for an object.

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

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

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

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

magic_pinfo()
Provide detailed information about an object.

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

magic_popd()
Change to directory popped off the top of the stack.

magic_profile()
Print your currently active IPyhton profile.

magic_prun()
Run a statement through the python code profiler.

Usage: %prun [options] statement

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

Options:

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

•A string: only information for function names containing this string

is printed.

•An integer: only these many lines are printed.

•A float (between 0 and 1): this fraction of the report is printed

(for example, use a limit of 0.4 to see the topmost 40% only).

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

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

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

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

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

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

Valid Arg Meaning "calls" call count "cumulative" cumulative time "file" file name "module" file name "pcalls" primitive call count "line" line number "name" function name "nfl" name/file/line "stdname" standard name "time" internal time

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

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

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

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

You can read the complete documentation for the profile module with:

```
In [1]: import profile; profile.help()
```

magic_psearch()

Search for object in namespaces by wildcard.

%psearch [options] PATTERN [OBJECT TYPE]

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

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

Arguments:

PATTERN

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

[OBJECT TYPE]

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

Options:

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

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

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

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

Examples:

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

Case sensitve search:

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

Show objects beginning with a single _:

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

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

magic_pushd()
Place the current dir on stack and change directory.

Usage: %pushd ['dirname']

magic_pwd()
Return the current working directory path.

magic_pycat()
Show a syntax-highlighted file through a pager.

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

magic_quickref()
Show a quick reference sheet

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

magic_r()
Repeat previous input.

Note: Consider using the more powerfull %rep instead!

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

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

magic_rehashx()
Update the alias table with all executable files in $PATH.

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

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

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

magic_reset()

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

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

Parameters **-y** : force reset without asking for confirmation.

Examples

In [6]: a = 1

In [7]: a Out[7]: 1

In [8]: 'a' in _ip.user_ns Out[8]: True

In [9]: %reset -f

In [10]: 'a' in _ip.user_ns Out[10]: False

magic_run()

Run the named file inside IPython as a program.

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

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

This is similar to running at a system prompt: $ python file args

but with the advantage of giving you IPython's tracebacks, and of loading all variables into your interactive namespace for further use (unless -p is used, see below).

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

Options:

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

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

-e: ignore sys.exit() calls or SystemExit exceptions in the script being run. This is particularly useful if IPython is being used to run unittests, which always exit with a sys.exit() call. In such cases you are interested in the output of the test results, not in seeing a traceback of the unittest module.

-t: print timing information at the end of the run. IPython will give you an estimated CPU time consumption for your script, which under Unix uses the resource module to avoid the

wraparound problems of time.clock(). Under Unix, an estimate of time spent on system tasks is also given (for Windows platforms this is reported as 0.0).

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

For example (testing the script uniq_stable.py):

In [1]: run -t uniq_stable

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

In [2]: run -t -N5 uniq_stable

IPython CPU timings (estimated):Total runs performed: 5

Times : Total Per runUser : 0.910862 s, 0.1821724 s.System: 0.0 s, 0.0 s.

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

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

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

%run -d -b40 myscript

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

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

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

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

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

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

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

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

magic_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.

magic_save()

Save a set of lines to a given filename.

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

Options:

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

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

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

magic_sc()

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

DEPRECATED. Suboptimal, retained for backwards compatibility.

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

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

"myfiles = !ls ~"

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

– %sc [options] varname=command

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

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

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

Options:

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

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

In most cases you should not need to split as a list, because the returned value is a special type of string which can automatically provide its contents either as a list (split on newlines) or as a

space-separated string. These are convenient, respectively, either for sequential processing or to be passed to a shell command.

For example:

all-random

Capture into variable a In [1]: sc a=ls *py

a is a string with embedded newlines In [2]: a Out[2]: 'setup.pynwin32_manual_post_install.py'

which can be seen as a list: In [3]: a.l Out[3]: ['setup.py', 'win32_manual_post_install.py']

or as a whitespace-separated string: In [4]: a.s Out[4]: 'setup.py win32_manual_post_install.py'

a.s is useful to pass as a single command line: In [5]: !wc -l $a.s

146 setup.py 130 win32_manual_post_install.py 276 total

while the list form is useful to loop over: In [6]: for f in a.l:

...: !wc -l $f ...:

146 setup.py 130 win32_manual_post_install.py

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

In [7]: sc -l b=ls *py

In [8]: b Out[8]: ['setup.py', 'win32_manual_post_install.py']

In [9]: b.s Out[9]: 'setup.py win32_manual_post_install.py'

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

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

magic_sx()

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

%sx command

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

Notes:

1) If an input line begins with '!!', then %sx is automatically invoked. That is, while:

!ls

causes ipython to simply issue system('ls'), typing !!ls

is a shorthand equivalent to: %sx ls

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

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

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

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

magic_system_verbose()
Set verbose printing of system calls.

If called without an argument, act as a toggle

magic_time()
Time execution of a Python statement or expression.

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

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

Some examples:

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

In [2]: n = 1000000

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

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

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

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

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

magic_timeit()
Time execution of a Python statement or expression

Usage: %timeit [-n<N> -r<R> [-tl-c]] statement

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

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

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

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

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

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

Examples:

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

In [2]: u = None

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

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

In [5]: import time

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

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

magic_unalias()
Remove an alias

magic_upgrade()
Upgrade your IPython installation

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

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

magic_who()
Print all interactive variables, with some minimal formatting.

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

%who function str

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

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

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

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

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

magic_who_ls()
Return a sorted list of all interactive variables.

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

magic_whos()
Like %who, but gives some extra information about each variable.

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

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

•For {},[],(): their length.

•For numpy and Numeric arrays, a summary with shape, number of

elements, typecode and size in memory.

•Everything else: a string representation, snipping their middle if

too long.

magic_xmode()
Switch modes for the exception handlers.

Valid modes: Plain, Context and Verbose.

If called without arguments, acts as a toggle.

parse_options()
Parse options passed to an argument string.

The interface is similar to that of getopt(), but it returns back a Struct with the options as keys and the stripped argument string still as a string.

arg_str is quoted as a true sys.argv vector by using shlex.split. This allows us to easily expand variables, glob files, quote arguments, etc.

Options: -mode: default 'string'. If given as 'list', the argument string is returned as a list (split on whitespace) instead of a string.

-list_all: put all option values in lists. Normally only options appearing more than once are put in a list.

-posix (True): whether to split the input line in POSIX mode or not, as per the conventions outlined in the shlex module from the standard library.

profile_missing_notice()

10.8.3 Functions

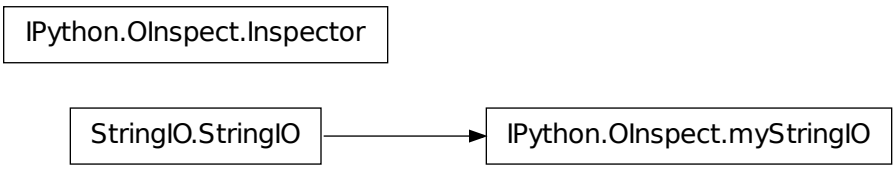
IPython.Magic.**compress_dhist**()

IPython.Magic.**on_off**()
Return an ON/OFF string for a 1/0 input. Simple utility function.

10.9 OInspect

10.9.1 Module: OInspect

Inheritance diagram for IPython.OInspect:



Tools for inspecting Python objects.

Uses syntax highlighting for presenting the various information elements.

Similar in spirit to the inspect module, but all calls take a name argument to reference the name under which an object is being read.

10.9.2 Classes

Inspector

class IPython.OInspect.**Inspector**(*color_table*, *code_color_table*, *scheme*, *str_detail_level=0*)

__init__()

noinfo()
Generic message when no information is found.

pdef()
Print the definition header for any callable object.

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

pdoc()
Print the docstring for any object.

Optional: -formatter: a function to run the docstring through for specially formatted docstrings.

pfile()
Show the whole file where an object was defined.

pinfo()
Show detailed information about an object.

Optional arguments:

•oname: name of the variable pointing to the object.

•formatter: special formatter for docstrings (see pdoc)

•info: a structure with some information fields which may have been

precomputed already.

•detail_level: if set to 1, more information is given.

psearch()
Search namespaces with wildcards for objects.

Arguments:

•pattern: string containing shell-like wildcards to use in namespace

searches and optionally a type specification to narrow the search to objects of that type.

•ns_table: dict of name->namespaces for search.

Optional arguments:

•ns_search: list of namespace names to include in search.

•ignore_case(False): make the search case-insensitive.

•show_all(False): show all names, including those starting with

underscores.

psource()
Print the source code for an object.

set_active_scheme()

myStringIO

class IPython.OInspect.**myStringIO**(*buf*='')
Bases: StringIO.StringIO

Adds a writeln method to normal StringIO.

__init__()

writeln()
Does a write() and then a write(‘ ‘)

10.9.3 Functions

IPython.OInspect.**getargspec**()
Get the names and default values of a function’s arguments.

A tuple of four things is returned: (args, varargs, varkw, defaults). ‘args’ is a list of the argument names (it may contain nested lists). ‘varargs’ and ‘varkw’ are the names of the * and ** arguments or None. ‘defaults’ is an n-tuple of the default values of the last n arguments.

Modified version of inspect.getargspec from the Python Standard Library.

IPython.OInspect.**getdoc**()
Stable wrapper around inspect.getdoc.

This can’t crash because of attribute problems.

It also attempts to call a getdoc() method on the given object. This allows objects which provide their docstrings via non-standard mechanisms (like Pyro proxies) to still be inspected by ipython’s ? system.

IPython.OInspect.**getsource**()
Wrapper around inspect.getsource.

This can be modified by other projects to provide customized source extraction.

Inputs:

•obj: an object whose source code we will attempt to extract.

Optional inputs:

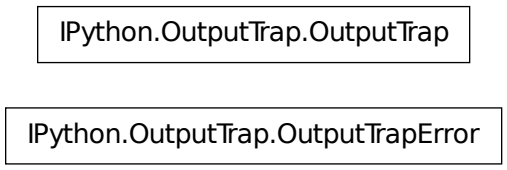
•is_binary: whether the object is known to come from a binary source.

This implementation will skip returning any output for binary objects, but custom extractors may know how to meaningfully process them.

10.10 OutputTrap

10.10.1 Module: OutputTrap

Inheritance diagram for IPython.OutputTrap:



Class to trap stdout and stderr and log them separately.

10.10.2 Classes

OutputTrap

class IPython.OutputTrap.**OutputTrap**(*name='Generic Output Trap'*, *out_head='Standard Output. '*, *err_head='Standard Error. '*, *sum_sep='n'*, *debug=0*, *trap_out=0*, *trap_err=0*, *quiet_out=0*, *quiet_err=0*)

Class to trap standard output and standard error. They get logged in StringIO objects which are available as <instance>.out and <instance>.err. The class also offers summary methods which format this data a bit.

A word of caution: because it blocks messages, using this class can make debugging very tricky. If you are having bizarre problems silently, try turning your output traps off for a while. You can call the constructor with the parameter debug=1 for these cases. This turns actual trapping off, but you can keep the rest of your code unchanged (this has already been a life saver).

Example:

config: trapper with a line of dots as log separator (final 'n' needed) config = OutputTrap('Config','Out ','Err ','.'*80+'n')

start trapping output config.trap_all()

now all output is logged ... # do stuff...

output back to normal: config.release_all()

print all that got logged: print config.summary()

print individual raw data: print config.out.getvalue() print config.err.getvalue()

__init__()

flush()
Flush stdout and stderr

flush_all()
Flush stdout and stderr

flush_err()
Flush the stdout log. All data held in the log is lost.

flush_out()
Flush the stdout log. All data held in the log is lost.

release()
Release both stdout and stderr.

Cacthes and discards OutputTrapError exceptions raised.

release_all()
Release both stdout and stderr.

Cacthes and discards OutputTrapError exceptions raised.

release_err()
Release stderr.

release_out()
Release stdout.

summary()
Return as a string the log from stdout and stderr, prepending a separator to each (defined in __init__ as sum_sep).

summary_all()
Return as a string the log from stdout and stderr, prepending a separator to each (defined in __init__ as sum_sep).

summary_err()
Return as a string the log from stderr.

summary_out()
Return as a string the log from stdout.

trap()
Trap and log both stdout and stderr.

Cacthes and discards OutputTrapError exceptions raised.

trap_all()
Trap and log both stdout and stderr.

Cacthes and discards OutputTrapError exceptions raised.

trap_err()
Trap and log stderr.

trap_out()
Trap and log stdout.

OutputTrapError

class IPython.OutputTrap.**OutputTrapError**(*args=None*)
Bases: exceptions.Exception

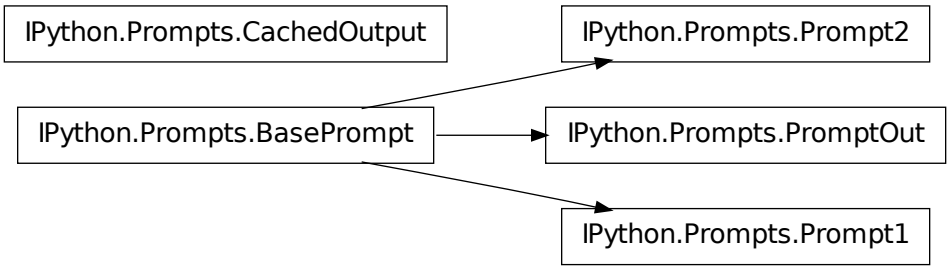
Exception for OutputTrap class.

__init__()

10.11 Prompts

10.11.1 Module: Prompts

Inheritance diagram for IPython.Prompts:



Classes for handling input/output prompts.

10.11.2 Classes

BasePrompt

class IPython.Prompts.**BasePrompt**(*cache*, *sep*, *prompt*, *pad_left=False*)
Bases: object

Interactive prompt similar to Mathematica's.

__init__()

cwd_filt()
Return the last depth elements of the current working directory.

$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2()
Return the last depth elements of the current working directory.

$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template
Template for prompt string creation

set_p_str()
Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write()

CachedOutput

class IPython.Prompts.**CachedOutput**(*shell*, *cache_size*, *Pprint*, *colors='NoColor'*, *input_sep='n'*, *output_sep='n'*, *output_sep2=''*, *ps1=None*, *ps2=None*, *ps_out=None*, *pad_left=True*)
Class for printing output from calculations while keeping a cache of reults. It dynamically creates global variables prefixed with _ which contain these results.

Meant to be used as a sys.displayhook replacement, providing numbered prompts and cache services.

Initialize with initial and final values for cache counter (this defines the maximum size of the cache.

__init__()

display()
Default printer method, uses pprint.

Do ip.set_hook("result_display", my_displayhook) for custom result display, e.g. when your own objects need special formatting.

flush()

set_colors()
Set the active color scheme and configure colors for the three prompt subsystems.

update()

Prompt1

class IPython.Prompts.**Prompt1**(*cache*, *sep='n'*, *prompt='In[\#]: '*, *pad_left=True*)
Bases: IPython.Prompts.BasePrompt

Input interactive prompt similar to Mathematica's.

__init__()

auto_rewrite()
Print a string of the form '—>' which lines up with the previous input string. Useful for systems which re-write the user input when handling automatically special syntaxes.

set_colors()

Prompt2

class IPython.Prompts.**Prompt2**(*cache*, *prompt='.\D.: '*, *pad_left=True*)
Bases: IPython.Prompts.BasePrompt

Interactive continuation prompt.

__init__()

set_colors()

set_p_str()

PromptOut

class IPython.Prompts.**PromptOut**(*cache*, *sep=''*, *prompt='Out[\#]: '*, *pad_left=True*)
Bases: IPython.Prompts.BasePrompt

Output interactive prompt similar to Mathematica's.

__init__()

set_colors()

10.11.3 Functions

IPython.Prompts.**multiple_replace**()
Replace in 'text' all occurences of any key in the given dictionary by its corresponding value. Returns the new string.

IPython.Prompts.**str_safe**()
Convert to a string, without ever raising an exception.

If str(arg) fails, <ERROR: ... > is returned, where ... is the exception error message.

10.12 PyColorize

10.12.1 Module: PyColorize

Inheritance diagram for IPython.PyColorize:

IPython.PyColorize.Parser

Class and program to colorize python source code for ANSI terminals.

Based on an HTML code highlighter by Jurgen Hermann found at: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52298

Modifications by Fernando Perez (fperez@colorado.edu).

Information on the original HTML highlighter follows:

MoinMoin - Python Source Parser

Title: Colorize Python source using the built-in tokenizer

Submitter: Jurgen Hermann Last Updated:2001/04/06

Version no:1.2

Description:

This code is part of MoinMoin (http://moin.sourceforge.net/) and converts Python source code to HTML markup, rendering comments, keywords, operators, numeric and string literals in different colors.

It shows how to use the built-in keyword, token and tokenize modules to scan Python source code and re-emit it with no changes to its original formatting (which is the hard part).

10.12.2 Parser

class IPython.PyColorize.**Parser**(*color_table=None*, *out=<open file '<stdout>', mode 'w' at 0x403b2078>*)

Format colored Python source.

__init__()

Create a parser with a specified color table and output channel.

Call format() to process code.

format()

format2()

Parse and send the colored source.

If out and scheme are not specified, the defaults (given to constructor) are used.

out should be a file-type object. Optionally, out can be given as the string 'str' and the parser will automatically return the output in a string.

IPython.PyColorize.**main**()

Run as a command-line script: colorize a python file or stdin using ANSI color escapes and print to stdout.

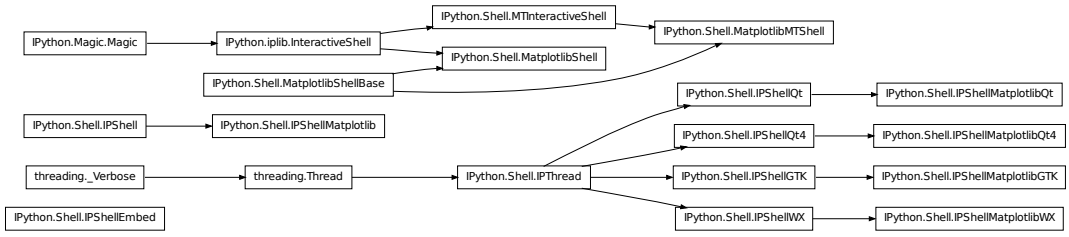
Inputs:

•argv(None): a list of strings like sys.argv[1:] giving the command-line arguments. If None, use sys.argv[1:].

10.13 Shell

10.13.1 Module: Shell

Inheritance diagram for IPython.Shell:



IPython Shell classes.

All the matplotlib support code was co-developed with John Hunter, matplotlib's author.

10.13.2 Classes

IPShell

class IPython.Shell.**IPShell**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*, *shell_class=<class 'IPython.iplib.InteractiveShell'>*)

Create an IPython instance.

__init__()

mainloop()

IPShellEmbed

class IPython.Shell.**IPShellEmbed**(*argv=None*, *banner=''*, *exit_msg=None*, *rc_override=None*, *user_ns=None*)

Allow embedding an IPython shell into a running program.

Instances of this class are callable, with the __call__ method being an alias to the embed() method of an InteractiveShell instance.

Usage (see also the example-embed.py file for a running example):

ipshell = IPShellEmbed([argv,banner,exit_msg,rc_override])

•argv: list containing valid command-line options for IPython, as they

would appear in sys.argv[1:].

For example, the following command-line options:

$ ipython -prompt_in1 'Input <#>' -colors LightBG

would be passed in the argv list as:

['-prompt_in1','Input <#>','-colors','LightBG']

•banner: string which gets printed every time the interpreter starts.

•exit_msg: string which gets printed every time the interpreter exits.

•rc_override: a dict or Struct of configuration options such as those

used by IPython. These options are read from your ~/.ipython/ipythonrc file when the Shell object is created. Passing an explicit rc_override dict with any options you want allows you to override those values at creation time without having to modify the file. This way you can create embeddable instances configured in any way you want without editing any global files (thus keeping your interactive IPython configuration unchanged).

Then the ipshell instance can be called anywhere inside your code:

ipshell(header='') -> Opens up an IPython shell.

•header: string printed by the IPython shell upon startup. This can let

you know where in your code you are when dropping into the shell. Note that 'banner' gets prepended to all calls, so header is used for location-specific information.

For more details, see the __call__ method below.

When the IPython shell is exited with Ctrl-D, normal program execution resumes.

This functionality was inspired by a posting on comp.lang.python by cmkl <cmkleffner@gmx.de> on Dec. 06/01 concerning similar uses of pyrepl, and by the IDL stop/continue commands.

__init__()
Note that argv here is a string, NOT a list.

get_dummy_mode()
Return the current value of the dummy mode parameter.

restore_system_completer()
Restores the readline completer which was in place.

This allows embedded IPython within IPython not to disrupt the parent's completion.

set_banner()
Sets the global banner.

This banner gets prepended to every header printed when the shell instance is called.

set_dummy_mode()
Sets the embeddable shell's dummy mode parameter.

set_dummy_mode(dummy): dummy = 0 or 1.

This parameter is persistent and makes calls to the embeddable shell silently return without performing any action. This allows you to globally activate or deactivate a shell you're using with a single call.

If you need to manually

set_exit_msg()
Sets the global exit_msg.

This exit message gets printed upon exiting every time the embedded shell is called. It is None by default.

IPShellGTK

class IPython.Shell.**IPShellGTK**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*, *shell_class=<class 'IPython.Shell.MTInteractiveShell'>*)
Bases: IPython.Shell.IPThread

Run a gtk mainloop() in a separate thread.

Python commands can be passed to the thread where they will be executed. This is implemented by periodically checking for passed code using a GTK timeout callback.

__init__()

mainloop()

on_timer()
Called when GTK is idle.

Must return True always, otherwise GTK stops calling it

IPShellMatplotlib

class IPython.Shell.**IPShellMatplotlib**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*)
Bases: IPython.Shell.IPShell

Subclass IPShell with MatplotlibShell as the internal shell.

Single-threaded class, meant for the Tk* and FLTK* backends.

Having this on a separate class simplifies the external driver code.

__init__()

IPShellMatplotlibGTK

class IPython.Shell.**IPShellMatplotlibGTK**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*)

Bases: IPython.Shell.IPShellGTK

Subclass IPShellGTK with MatplotlibMTShell as the internal shell.

Multi-threaded class, meant for the GTK* backends.

__init__()

IPShellMatplotlibQt

class IPython.Shell.**IPShellMatplotlibQt**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*)

Bases: IPython.Shell.IPShellQt

Subclass IPShellQt with MatplotlibMTShell as the internal shell.

Multi-threaded class, meant for the Qt* backends.

__init__()

IPShellMatplotlibQt4

class IPython.Shell.**IPShellMatplotlibQt4**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*)

Bases: IPython.Shell.IPShellQt4

Subclass IPShellQt4 with MatplotlibMTShell as the internal shell.

Multi-threaded class, meant for the Qt4* backends.

__init__()

IPShellMatplotlibWX

class IPython.Shell.**IPShellMatplotlibWX**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*)

Bases: IPython.Shell.IPShellWX

Subclass IPShellWX with MatplotlibMTShell as the internal shell.

Multi-threaded class, meant for the WX* backends.

__init__()

IPShellQt

class IPython.Shell.**IPShellQt**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=0*, *shell_class=<class 'IPython.Shell.MTInteractiveShell'>*)

Bases: IPython.Shell.IPThread

Run a Qt event loop in a separate thread.

Python commands can be passed to the thread where they will be executed. This is implemented by periodically checking for passed code using a Qt timer / slot.

__init__()

mainloop()

on_timer()

IPShellQt4

class IPython.Shell.**IPShellQt4**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=0*, *shell_class=<class 'IPython.Shell.MTInteractiveShell'>*)

Bases: IPython.Shell.IPThread

Run a Qt event loop in a separate thread.

Python commands can be passed to the thread where they will be executed. This is implemented by periodically checking for passed code using a Qt timer / slot.

__init__()

mainloop()

on_timer()

IPShellWX

class IPython.Shell.**IPShellWX**(*argv=None*, *user_ns=None*, *user_global_ns=None*, *debug=1*, *shell_class=<class 'IPython.Shell.MTInteractiveShell'>*)

Bases: IPython.Shell.IPThread

Run a wx mainloop() in a separate thread.

Python commands can be passed to the thread where they will be executed. This is implemented by periodically checking for passed code using a GTK timeout callback.

__init__()

mainloop()

wxexit()

IPThread

class IPython.Shell.**IPThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *verbose=None*)

Bases: threading.Thread

__init__()

run()

MTInteractiveShell

class IPython.Shell.**MTInteractiveShell**(*name*, *usage=None*, *rc=Struct({'__allownew': True, 'args': None, 'opts': None})*, *user_ns=None*, *user_global_ns=None*, *banner2=''*, *gui_timeout=10*, ***kw*)

Bases: IPython.iplib.InteractiveShell

Simple multi-threaded shell.

__init__()

Similar to the normal InteractiveShell, but with threading control

kill()

Kill the thread, returning when it has been shut down.

runcode()

Execute a code object.

Multithreaded wrapper around IPython's runcode().

runsource()

Compile and run some source in the interpreter.

Modified version of code.py's runsource(), to handle threading issues. See the original for full docstring details.

MatplotlibMTShell

class IPython.Shell.**MatplotlibMTShell**(*name*, *usage=None*, *rc=Struct({'__allownew': True, 'args': None, 'opts': None})*, *user_ns=None*, *user_global_ns=None*, ***kw*)

Bases: IPython.Shell.MatplotlibShellBase, IPython.Shell.MTInteractiveShell

Multi-threaded shell with matplotlib support.

__init__()

MatplotlibShell

class IPython.Shell.**MatplotlibShell**(*name*, *usage=None*, *rc=Struct({'__allownew': True, 'args': None, 'opts': None})*, *user_ns=None*, *user_global_ns=None*, ***kw*)

Bases: IPython.Shell.MatplotlibShellBase, IPython.iplib.InteractiveShell

Single-threaded shell with matplotlib support.

__init__()

MatplotlibShellBase

class IPython.Shell.**MatplotlibShellBase**

Mixin class to provide the necessary modifications to regular IPython shell classes for matplotlib support.

Given Python's MRO, this should be used as the FIRST class in the inheritance hierarchy, so that it overrides the relevant methods.

magic_run()

Run the named file inside IPython as a program.

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

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

This is similar to running at a system prompt: $ python file args

but with the advantage of giving you IPython's tracebacks, and of loading all variables into your interactive namespace for further use (unless -p is used, see below).

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

Options:

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

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

-e: ignore sys.exit() calls or SystemExit exceptions in the script being run. This is particularly useful if IPython is being used to run unittests, which always exit with a sys.exit() call. In such cases you are interested in the output of the test results, not in seeing a traceback of the unittest module.

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

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

For example (testing the script uniq_stable.py):

In [1]: run -t uniq_stable

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

In [2]: run -t -N5 uniq_stable

IPython CPU timings (estimated):Total runs performed: 5

Times : Total Per runUser : 0.910862 s, 0.1821724 s.System: 0.0 s, 0.0 s.

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

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

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

%run -d -b40 myscript

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

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

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

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

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

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

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

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

*** Modified %run for Matplotlib, with proper interactive handling ***

mplot_exec()
Execute a matplotlib script.

This is a call to execfile(), but wrapped in safeties to properly handle interactive rendering and backend switching.

10.13.3 Functions

IPython.Shell.**check_gtk**()

IPython.Shell.**get_tk**()
Tries to import Tkinter and returns a withdrawn Tkinter root window. If Tkinter is already imported or not available, this returns None. This function calls *hijack_tk* underneath.

IPython.Shell.**hijack_gtk**()
Modifies pyGTK's mainloop with a dummy so user code does not block IPython. This function returns the original *gtk.mainloop* function that has been hijacked.

IPython.Shell.**hijack_qt**()
Modifies PyQt's mainloop with a dummy so user code does not block IPython. This function returns the original *qt.qApp.exec_loop* function that has been hijacked.

IPython.Shell.**hijack_qt4**()
Modifies PyQt4's mainloop with a dummy so user code does not block IPython. This function returns the original *QtGui.qApp.exec_* function that has been hijacked.

IPython.Shell.**hijack_tk**()
Modifies Tkinter's mainloop with a dummy so when a module calls mainloop, it does not block.

IPython.Shell.**hijack_wx**()
Modifies wxPython's MainLoop with a dummy so user code does not block IPython. The hijacked mainloop function is returned.

IPython.Shell.**kill_embedded**()
%kill_embedded : deactivate for good the current embedded IPython.

This function (after asking for confirmation) sets an internal flag so that an embedded IPython will never activate again. This is useful to permanently disable a shell that is being called inside a loop: once you've figured out what you needed from it, you may then kill it and the program will then continue to run without the interactive shell interfering again.

IPython.Shell.**start**()
Return a running shell instance, dealing with threading options.

This is a factory function which will instantiate the proper IPython shell based on the user's threading choice. Such a selector is needed because different GUI toolkits require different thread handling details.

IPython.Shell.**update_tk**()
Updates the Tkinter event loop. This is typically called from the respective WX or GTK mainloops.

10.14 UserConfig.ipy_user_conf

10.14.1 Module: UserConfig.ipy_user_conf

10.14.2 Functions

10.15 background_jobs

10.15.1 Module: background_jobs

Inheritance diagram for IPython.background_jobs:

Manage background (threaded) jobs conveniently from an interactive shell.

This module provides a BackgroundJobManager class. This is the main class meant for public usage, it implements an object which can create and manage new background jobs.

It also provides the actual job classes managed by these BackgroundJobManager objects, see their docstrings below.

This system was inspired by discussions with B. Granger and the BackgroundCommand class described in the book Python Scripting for Computational Science, by H. P. Langtangen:

http://folk.uio.no/hpl/scripting

(although ultimately no code from this text was used, as IPython's system is a separate implementation).

10.15.2 Classes

BackgroundJobBase

class IPython.background_jobs.**BackgroundJobBase**

Bases: threading.Thread

Base class to build BackgroundJob classes.

The derived classes must implement:

•Their own __init__, since the one here raises NotImplementedError. The

derived constructor must call self._init() at the end, to provide common initialization.

•A strform attribute used in calls to __str__.

•A call() method, which will make the actual execution call and must

return a value to be held in the 'result' field of the job object.

__init__()

run()

traceback()

BackgroundJobExpr

class IPython.background_jobs.**BackgroundJobExpr**(*expression*, *glob=None*, *loc=None*)

Bases: IPython.background_jobs.BackgroundJobBase

Evaluate an expression as a background job (uses a separate thread).

__init__()

Create a new job from a string which can be fed to eval().

global/locals dicts can be provided, which will be passed to the eval call.

call()

BackgroundJobFunc

class IPython.background_jobs.**BackgroundJobFunc**(*func*, **args*, ***kwargs*)

Bases: IPython.background_jobs.BackgroundJobBase

Run a function call as a background job (uses a separate thread).

__init__()

Create a new job from a callable object.

Any positional arguments and keyword args given to this constructor after the initial callable are passed directly to it.

call()

BackgroundJobManager

class IPython.background_jobs.**BackgroundJobManager**

Class to manage a pool of backgrounded threaded jobs.

Below, we assume that 'jobs' is a BackgroundJobManager instance.

Usage summary (see the method docstrings for details):

jobs.new(...) -> start a new job

jobs() or jobs.status() -> print status summary of all jobs

jobs[N] -> returns job number N.

foo = jobs[N].result -> assign to variable foo the result of job N

jobs[N].traceback() -> print the traceback of dead job N

jobs.remove(N) -> remove (finished) job N

jobs.flush_finished() -> remove all finished jobs

As a convenience feature, BackgroundJobManager instances provide the utility result and traceback methods which retrieve the corresponding information from the jobs list:

jobs.result(N) <–> jobs[N].result jobs.traceback(N) <–> jobs[N].traceback()

While this appears minor, it allows you to use tab completion interactively on the job manager instance.

In interactive mode, IPython provides the magic fuction %bg for quick creation of backgrounded expression-based jobs. Type bg? for details.

__init__()

flush_finished()

Flush all jobs finished (completed and dead) from lists.

Running jobs are never flushed.

It first calls _status_new(), to update info. If any jobs have completed since the last _status_new() call, the flush operation aborts.

new()

Add a new background job and start it in a separate thread.

There are two types of jobs which can be created:

1. Jobs based on expressions which can be passed to an eval() call. The expression must be given as a string. For example:

job_manager.new('myfunc(x,y,z=1)'[,glob[,loc]])

The given expression is passed to eval(), along with the optional global/local dicts provided. If no dicts are given, they are extracted automatically from the caller's frame.

A Python statement is NOT a valid eval() expression. Basically, you can only use as an eval() argument something which can go on the right of an '=' sign and be assigned to a variable.

For example,"print 'hello'" is not valid, but '2+3' is.

2. Jobs given a function object, optionally passing additional positional arguments:

job_manager.new(myfunc,x,y)

The function is called with the given arguments.

If you need to pass keyword arguments to your function, you must supply them as a dict named kw:

job_manager.new(myfunc,x,y,kw=dict(z=1))

The reason for this assymmetry is that the new() method needs to maintain access to its own keywords, and this prevents name collisions between arguments to new() and arguments to your own functions.

In both cases, the result is stored in the job.result field of the background job object.

Notes and caveats:

1. All threads running share the same standard output. Thus, if your background jobs generate output, it will come out on top of whatever you are currently writing. For this reason, background jobs are best used with silent functions which simply return their output.

2. Threads also all work within the same global namespace, and this system does not lock interactive variables. So if you send job to the background which operates on a mutable object for a long time, and start modifying that same mutable object interactively (or in another backgrounded job), all sorts of bizarre behaviour will occur.

3. If a background job is spending a lot of time inside a C extension module which does not release the Python Global Interpreter Lock (GIL), this will block the IPython prompt. This is simply because the Python interpreter can only switch between threads at Python bytecodes. While the execution is inside C code, the interpreter must simply wait unless the extension module releases the GIL.

4. There is no way, due to limitations in the Python threads library, to kill a thread once it has started.

remove()
Remove a finished (completed or dead) job.

result()
result(N) -> return the result of job N.

status()
Print a status of all jobs currently being managed.

traceback()

10.16 clipboard

10.16.1 Module: clipboard

Utilities for accessing the platform's clipboard.

10.16.2 Functions

IPython.clipboard.**osx_clipboard_get**()
Get the clipboard's text on OS X.

IPython.clipboard.**tkinter_clipboard_get**()
Get the clipboard's text using Tkinter.

This is the default on systems that are not Windows or OS X. It may interfere with other UI toolkits and should be replaced with an implementation that uses that toolkit.

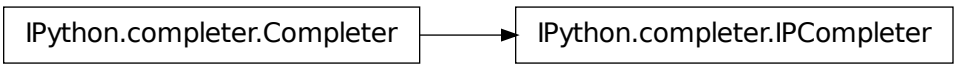
IPython.clipboard.**win32_clipboard_get**()
Get the current clipboard's text on Windows.

Requires Mark Hammond's pywin32 extensions.

10.17 completer

10.17.1 Module: completer

Inheritance diagram for IPython.completer:



Word completion for IPython.

This module is a fork of the rlcompleter module in the Python standard library. The original enhancements made to rlcompleter have been sent upstream and were accepted as of Python 2.3, but we need a lot more functionality specific to IPython, so this module will continue to live as an IPython-specific utility.

Original rlcompleter documentation:

This requires the latest extension to the readline module (the completes keywords, built-ins and globals in __main__; when completing NAME.NAME..., it evaluates (!) the expression up to the last dot and completes its attributes.

It's very cool to do "import string" type "string.", hit the completion key (twice), and see the list of names defined by the string module!

Tip: to use the tab key as the completion key, call

readline.parse_and_bind("tab: complete")

Notes:

- Exceptions raised by the completer function are *ignored* (and

generally cause the completion to fail). This is a feature – since readline sets the tty device in raw (or cbreak) mode, printing a traceback wouldn't work well without some complicated hoopla to save, reset and restore the tty state.

- The evaluation of the NAME.NAME... form may cause arbitrary

application defined code to be executed if an object with a __getattr__ hook is found. Since it is the responsibility of the application (or the user) to enable this feature, I consider this an acceptable risk. More complicated expressions (e.g. function calls or indexing operations) are *not* evaluated.

- GNU readline is also used by the built-in functions input() and

raw_input(), and thus these also benefit/suffer from the completer features. Clearly an interactive application can benefit by specifying its own completer function and using raw_input() for all its input.

- When the original stdin is not a tty device, GNU readline is never

used, and this module (and the readline module) are silently inactive.

10.17.2 Classes

Completer

class IPython.completer.**Completer**(*namespace=None*, *global_namespace=None*)

__init__()
Create a new completer for the command line.

Completer([namespace,global_namespace]) -> completer instance.

If unspecified, the default namespace where completions are performed is __main__ (technically, __main__.__dict__). Namespaces should be given as dictionaries.

An optional second namespace can be given. This allows the completer to handle cases where both the local and global scopes need to be distinguished.

Completer instances should be used as the completion mechanism of readline via the set_completer() call:

readline.set_completer(Completer(my_namespace).complete)

attr_matches()
Compute matches when text contains a dot.

Assuming the text is of the form NAME.NAME....[NAME], and is evaluatable in self.namespace or self.global_namespace, it will be evaluated and its attributes (as revealed by dir()) are used as possible completions. (For class instances, class members are are also considered.)

WARNING: this can still invoke arbitrary C code, if an object with a __getattr__ hook is evaluated.

complete()
Return the next possible completion for 'text'.

This is called successively with state == 0, 1, 2, ... until it returns None. The completion should begin with 'text'.

global_matches()
Compute matches when text is a simple name.

Return a list of all keywords, built-in functions and names currently defined in self.namespace or self.global_namespace that match.

IPCompleter

class IPython.completer.**IPCompleter**(*shell*, *namespace=None*, *global_namespace=None*, *omit__names=0*, *alias_table=None*)

Bases: IPython.completer.Completer

Extension of the completer class with IPython-specific features

__init__()

IPCompleter() -> completer

Return a completer object suitable for use by the readline library via readline.set_completer().

Inputs:

•shell: a pointer to the ipython shell itself. This is needed

because this completer knows about magic functions, and those can only be accessed via the ipython instance.

•namespace: an optional dict where completions are performed.

•global_namespace: secondary optional dict for completions, to

handle cases (such as IPython embedded inside functions) where both Python scopes are visible.

•The optional omit__names parameter sets the completer to omit the

'magic' names (__magicname__) for python objects unless the text to be completed explicitly starts with one or more underscores.

•If alias_table is supplied, it should be a dictionary of aliases

to complete.

alias_matches()

Match internal system aliases

all_completions()

Return all possible completions for the benefit of emacs.

complete()

Return the next possible completion for 'text'.

This is called successively with state == 0, 1, 2, ... until it returns None. The completion should begin with 'text'.

Keywords

•line_buffer: string

If not given, the completer attempts to obtain the current line buffer via readline. This keyword allows clients which are requesting for text completions in non-readline contexts to inform the completer of the entire text.

dispatch_custom_completer()

file_matches()
Match filenames, expanding ~USER type strings.

Most of the seemingly convoluted logic in this completer is an attempt to handle filenames with spaces in them. And yet it's not quite perfect, because Python's readline doesn't expose all of the GNU readline details needed for this to be done correctly.

For a filename with a space in it, the printed completions will be only the parts after what's already been typed (instead of the full completions, as is normally done). I don't think with the current (as of Python 2.3) Python readline it's possible to do better.

python_func_kw_matches()
Match named parameters (kwargs) of the last open function

python_matches()
Match attributes or global python names

10.18 config.api

10.18.1 Module: config.api

Inheritance diagram for IPython.config.api:

This is the official entry point to IPython's configuration system.

10.18.2 ConfigObjManager

class IPython.config.api.**ConfigObjManager**(*configObj*, *filename*)
Bases: object

__init__()

get_config_obj()

resolve_file_path()
Resolve filenames into absolute paths.

This function looks in the following directories in order:

1. In the current working directory or by absolute path with ~ expanded
2. In ipythondir if that is set
3. In the IPYTHONDIR environment variable if it exists
4. In the ~/.ipython directory

Note: The IPYTHONDIR is also used by the trunk version of IPython so changing it will also affect it was well.

update_config_obj()

update_config_obj_from_default_file()

update_config_obj_from_file()

write_config_obj_to_file()

write_default_config_file()

10.19 config.cutils

10.19.1 Module: config.cutils

Configuration-related utilities for all IPython.

IPython.config.cutils.**import_item**()
Import and return bar given the string foo.bar.

10.20 deep_reload

10.20.1 Module: deep_reload

A module to change reload() so that it acts recursively. To enable it type:

```
>>> import __builtin__, deep_reload
>>> __builtin__.reload = deep_reload.reload
```

You can then disable it with:

```
>>> __builtin__.reload = deep_reload.original_reload
```

Alternatively, you can add a dreload builtin alongside normal reload with:

```
>>> __builtin__.dreload = deep_reload.reload
```

This code is almost entirely based on knee.py from the standard library.

10.20.2 Functions

IPython.deep_reload.**deep_import_hook**()

IPython.deep_reload.**deep_reload_hook**()

IPython.deep_reload.**determine_parent**()

IPython.deep_reload.**ensure_fromlist**()

IPython.deep_reload.**find_head_package**()

IPython.deep_reload.**import_module**()

IPython.deep_reload.**load_tail**()

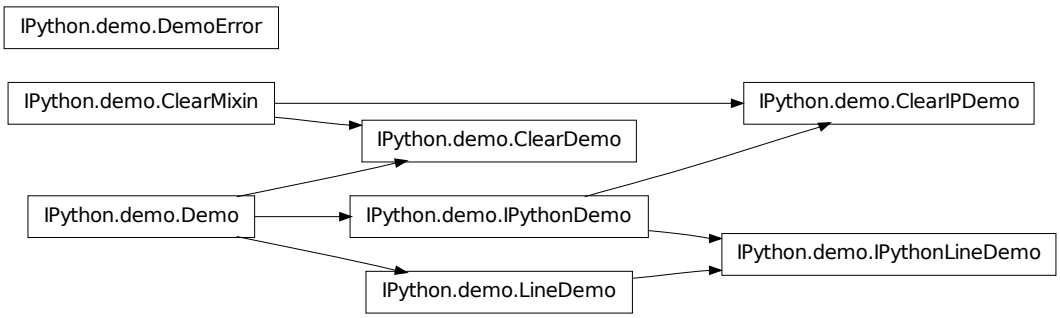
IPython.deep_reload.**reload**()

Recursively reload all modules used in the given module. Optionally takes a list of modules to exclude from reloading. The default exclude list contains sys, __main__, and __builtin__, to prevent, e.g., resetting display, exception, and io hooks.

10.21 demo

10.21.1 Module: demo

Inheritance diagram for IPython.demo:



Module for interactive demos using IPython.

This module implements a few classes for running Python scripts interactively in IPython for demonstrations. With very simple markup (a few tags in comments), you can control points where the script stops executing and returns control to IPython.

Provided classes

The classes are (see their docstrings for further details):

- Demo: pure python demos
- IPythonDemo: demos with input to be processed by IPython as if it had been

typed interactively (so magics work, as well as any other special syntax you may have added via input prefilters).

- LineDemo: single-line version of the Demo class. These demos are executed

one line at a time, and require no markup.

- IPythonLineDemo: IPython version of the LineDemo class (the demo is

executed a line at a time, but processed via IPython).

- ClearMixin: mixin to make Demo classes with less visual clutter. It declares an empty marquee and a pre_cmd that clears the screen before each block (see Subclassing below).
- ClearDemo, ClearIPDemo: mixin-enabled versions of the Demo and IPythonDemo classes.

Subclassing

The classes here all include a few methods meant to make customization by subclassing more convenient. Their docstrings below have some more details:

- marquee(): generates a marquee to provide visible on-screen markers at each block start and end.
- pre_cmd(): run right before the execution of each block.
- post_cmd(): run right after the execution of each block. If the block raises an exception, this is NOT called.

Operation

The file is run in its own empty namespace (though you can pass it a string of arguments as if in a command line environment, and it will see those as sys.argv). But at each stop, the global IPython namespace is updated with the current internal demo namespace, so you can work interactively with the data accumulated so far.

By default, each block of code is printed (with syntax highlighting) before executing it and you have to confirm execution. This is intended to show the code to an audience first so you can discuss it, and only proceed with execution once you agree. There are a few tags which allow you to modify this behavior.

The supported tags are:

<demo> stop

Defines block boundaries, the points where IPython stops execution of the file and returns to the interactive prompt.

You can optionally mark the stop tag with extra dashes before and after the word 'stop', to help visually distinguish the blocks in a text editor:

<demo> — stop —

<demo> silent

Make a block execute silently (and hence automatically). Typically used in cases where you have some boilerplate or initialization code which you need executed but do not want to be seen in the demo.

<demo> auto

Make a block execute automatically, but still being printed. Useful for simple code which does not warrant discussion, since it avoids the extra manual confirmation.

<demo> auto_all

This tag can _only_ be in the first block, and if given it overrides the individual auto tags to make the whole demo fully automatic (no block asks for confirmation). It can also be given at creation time (or the attribute set later) to override what's in the file.

While _any_ python file can be run as a Demo instance, if there are no stop tags the whole file will run in a single block (no different that calling first %pycat and then %run). The minimal markup to make this useful is to place a set of stop tags; the other tags are only there to let you fine-tune the execution.

This is probably best explained with the simple example file below. You can copy this into a file named ex_demo.py, and try running it via:

from IPython.demo import Demo d = Demo('ex_demo.py') d() <— Call the d object (omit the parens if you have autocall set to 2).

Each time you call the demo object, it runs the next block. The demo object has a few useful methods for navigation, like again(), edit(), jump(), seek() and back(). It can be reset for a new run via reset() or reloaded from disk (in case you've edited the source) via reload(). See their docstrings below.

Note: To make this simpler to explore, a file called "demo-exercizer.py" has been added to the "docs/examples/core" directory. Just cd to this directory in an IPython session, and type:

```
%run demo-exercizer.py
```

and then follow the directions.

Example

The following is a very simple example of a valid demo file.

#################### EXAMPLE DEMO <ex_demo.py> ############################### ''' A simple interactive demo to illustrate the use of IPython's Demo class.'''

print 'Hello, welcome to an interactive IPython demo.'

The mark below defines a block boundary, which is a point where IPython will # stop execution and return to the interactive prompt. The dashes are actually # optional and used only as a visual aid to clearly separate blocks while # editing the demo code. # <demo> stop

x = 1 y = 2

<demo> stop

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

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

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

print ‘z=’,x

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

print ‘bye!’ ################### END EXAMPLE DEMO <ex_demo.py> ############################

10.21.2 Classes

ClearDemo

class IPython.demo.**ClearDemo**(*src*, *title=’‘*, *arg_str=’‘*, *auto_all=None*)

Bases: IPython.demo.ClearMixin, IPython.demo.Demo

__init__()

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(‘’): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

ClearIPDemo

class IPython.demo.**ClearIPDemo**(*src*, *title=’‘*, *arg_str=’‘*, *auto_all=None*)

Bases: IPython.demo.ClearMixin, IPython.demo.IPythonDemo

__init__()

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(''): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

ClearMixin

class IPython.demo.**ClearMixin**
Bases: object

Use this mixin to make Demo classes with less visual clutter.

Demos using this mixin will clear the screen before every block and use blank marquees.

Note that in order for the methods defined here to actually override those of the classes it's mixed with, it must go /first/ in the inheritance tree. For example:

class ClearIPDemo(ClearMixin,IPythonDemo): pass

will provide an IPythonDemo class with the mixin's features.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

marquee()
Blank marquee that returns '' no matter what the input.

pre_cmd()
Method called before executing each block.

This one simply clears the screen.

Demo

class IPython.demo.**Demo**(*src*, *title=''*, *arg_str=''*, *auto_all=None*)
Bases: object

__init__()
Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(''): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again()
Move the seek pointer back one block and re-execute.

back()
Move the seek pointer back num blocks (default is 1).

edit()
Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use reload() when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload()
Load file object.

jump()
Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee()
Return the input string centered in a 'marquee'.

post_cmd()
Method called after executing each block.

pre_cmd()
Method called before executing each block.

reload()
Reload source from disk and initialize state.

reset()
Reset the namespace and seek pointer to restart the demo

runlines()
Execute a string with one or more lines of code

seek()
Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show()
Show a single block on screen

show_all()
Show entire demo on screen, block by block

DemoError

class IPython.demo.**DemoError**
Bases: exceptions.Exception

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

IPythonDemo

class IPython.demo.**IPythonDemo**(*src*, *title=''*, *arg_str=''*, *auto_all=None*)
Bases: IPython.demo.Demo

Class for interactive demos with IPython's input processing applied.

This subclasses Demo, but instead of executing each block by the Python interpreter (via exec), it actually calls IPython on it, so that any input filters which may be in place are applied to the input block.

If you have an interactive environment which exposes special input processing, you can use this class instead to write demo scripts which operate exactly as if you had typed them interactively. The default Demo class requires the input to be valid, pure Python code.

__init__()
Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(''): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

runlines()

Execute a string with one or more lines of code

IPythonLineDemo

class IPython.demo.**IPythonLineDemo**(*src*, *title=''*, *arg_str=''*, *auto_all=None*)

Bases: IPython.demo.IPythonDemo, IPython.demo.LineDemo

Variant of the LineDemo class whose input is processed by IPython.

__init__()

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(''): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

LineDemo

class IPython.demo.**LineDemo**(*src*, *title=''*, *arg_str=''*, *auto_all=None*)

Bases: IPython.demo.Demo

Demo where each line is executed as a separate block.

The input script should be valid Python code.

This class doesn't require any markup at all, and it's meant for simple scripts (with no nesting or any kind of indentation) which consist of multiple lines of input to be executed, one at a time, as if they had been typed in the interactive prompt.

Note: the input can not have *any* indentation, which means that only single-lines of input are accepted, not even function definitions are valid.

__init__()

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

•**src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

•title: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

•arg_str(''): a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

•auto_all(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

reload()

Reload source from disk and initialize state.

10.21.3 Function

IPython.demo.**re_mark**()

10.22 dtutils

10.22.1 Module: dtutils

Doctest-related utilities for IPython.

For most common uses, all you should need to run is:

```
from IPython.dtutils import idoctest
```

See the idoctest docstring below for usage details.

10.22.2 Functions

IPython.dtutils.**idoctest**()

Interactively prompt for input and run it as a doctest.

To finish entering input, enter two blank lines or Ctrl-D (EOF). If you use Ctrl-C, the example is aborted and all input discarded.

Keywords

ns [dict (None)] Namespace where the code should be executed. If not given, the IPython interactive namespace is used.

eraise [bool (False)] If true, immediately raise any exceptions instead of reporting them at the end. This allows you to then do interactive debugging via IPython's facilities (use %debug after the fact, or with %pdb for automatic activation).

end_mark [str ('–')] String to explicitly indicate the end of input.

IPython.dtutils.**rundoctest**()

Run a the input source as a doctest, in the caller's namespace.

Parameters

text [str] Source to execute.

Keywords

ns [dict (None)] Namespace where the code should be executed. If not given, the caller's locals and globals are used.

eraise [bool (False)] If true, immediately raise any exceptions instead of reporting them at the end. This allows you to then do interactive debugging via IPython's facilities (use %debug after the fact, or with %pdb for automatic activation).

10.23 excolors

10.23.1 Module: excolors

Color schemes for exception handling code in IPython.

IPython.excolors.**exception_colors**()

Return a color table with fields for exception reporting.

The table is an instance of ColorSchemeTable with schemes added for 'Linux', 'LightBG' and 'NoColor' and fields for exception handling filled in.

Examples:

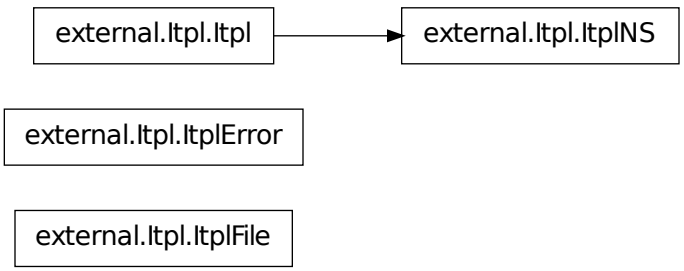
```
>>> ec = exception_colors()
>>> ec.active_scheme_name
''
>>> print ec.active_colors
None
```

Now we activate a color scheme: >>> ec.set_active_scheme('NoColor') >>> ec.active_scheme_name 'NoColor' >>> ec.active_colors.keys() ['em', 'caret', '__allownew', 'name', 'val', 'vName', 'Normal', 'normalEm', 'filename', 'linenoEm', 'excName', 'lineno', 'valEm', 'filenameEm', 'nameEm', 'line', 'topline']

10.24 external.Itpl

10.24.1 Module: external.Itpl

Inheritance diagram for IPython.external.Itpl:



String interpolation for Python (by Ka-Ping Yee, 14 Feb 2000).

This module lets you quickly and conveniently interpolate values into strings (in the flavour of Perl or Tcl, but with less extraneous punctuation). You get a bit more power than in the other languages, because this module allows subscripting, slicing, function calls, attribute lookup, or arbitrary expressions. Variables and expressions are evaluated in the namespace of the caller.

The itpl() function returns the result of interpolating a string, and printpl() prints out an interpolated string. Here are some examples:

from Itpl import printpl printpl("Here is a $string.") printpl("Here is a $module.member.") printpl("Here is an $object.member.") printpl("Here is a $functioncall(with, arguments).")

printpl("Here is an ${arbitrary + expression}.") printpl("Here is an $array[3] member.") printpl("Here is a $dictionary['member'].")

The filter() function filters a file object so that output through it is interpolated. This lets you produce the illusion that Python knows how to do interpolation:

import Itpl sys.stdout = Itpl.filter() f = "fancy" print "Is this not $f?" print "Standard output has been replaced with a $sys.stdout object." sys.stdout = Itpl.unfilter() print "Okay, back $to $normal."

Under the hood, the Itpl class represents a string that knows how to interpolate values. An instance of the class parses the string once upon initialization; the evaluation and substitution can then be done each time the instance is evaluated with str(instance). For example:

from Itpl import Itpl s = Itpl("Here is $foo.") foo = 5 print str(s) foo = "bar" print str(s)

10.24.2 Classes

Itpl

class IPython.external.Itpl.**Itpl**(*format*, *codec='utf_8'*, *encoding_errors='backslashreplace'*)

Class representing a string with interpolation abilities.

Upon creation, an instance works out what parts of the format string are literal and what parts need to be evaluated. The evaluation and substitution happens in the namespace of the caller when str(instance) is called.

__init__()

The single mandatory argument to this constructor is a format string.

The format string is parsed according to the following rules:

1. **A dollar sign and a name, possibly followed by any of:**
   - an open-paren, and anything up to the matching paren
   - an open-bracket, and anything up to the matching bracket
   - a period and a name

   any number of times, is evaluated as a Python expression.
2. A dollar sign immediately followed by an open-brace, and anything up to the matching close-brace, is evaluated as a Python expression.
3. Outside of the expressions described in the above two rules, two dollar signs in a row give you one literal dollar sign.

Optional arguments:

•codec('utf_8'): a string containing the name of a valid Python

codec.

•encoding_errors('backslashreplace'): a string with a valid error handling

policy. See the codecs module documentation for details.

These are used to encode the format string if a call to str() fails on the expanded result.

ItplError

class IPython.external.Itpl.**ItplError**(*text*, *pos*)
Bases: exceptions.ValueError

__init__()

ItplFile

class IPython.external.Itpl.**ItplFile**(*file*)
A file object that filters each write() through an interpolator.

__init__()

write()

ItplNS

class IPython.external.Itpl.**ItplNS**(*format*, *globals*, *locals=None*, *codec='utf_8'*, *encoding_errors='backslashreplace'*)
Bases: IPython.external.Itpl.Itpl

Class representing a string with interpolation abilities.

This inherits from Itpl, but at creation time a namespace is provided where the evaluation will occur. The interpolation becomes a bit more efficient, as no traceback needs to be extracte. It also allows the caller to supply a different namespace for the interpolation to occur than its own.

__init__()
ItplNS(format,globals[,locals]) -> interpolating string instance.

This constructor, besides a format string, takes a globals dictionary and optionally a locals (which defaults to globals if not provided).

For further details, see the Itpl constructor.

10.24.3 Functions

IPython.external.Itpl.**filter**()
Return an ItplFile that filters writes to the given file object.

'file = filter(file)' replaces 'file' with a filtered object that has a write() method. When called with no argument, this creates a filter to sys.stdout.

IPython.external.Itpl.**itpl**()

IPython.external.Itpl.**itplns**()

IPython.external.Itpl.**matchorfail**()

IPython.external.Itpl.**printpl**()

IPython.external.Itpl.**printplns**()

IPython.external.Itpl.**unfilter**()

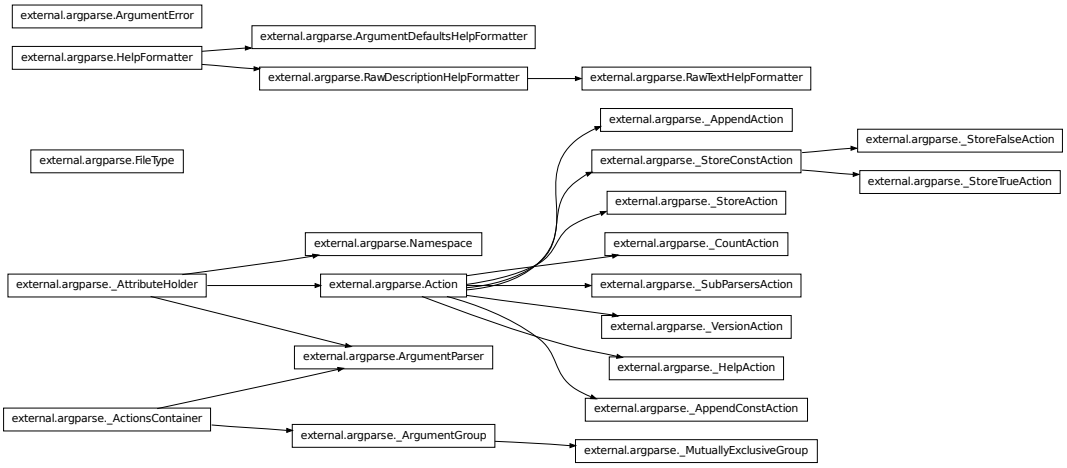
Return the original file that corresponds to the given ItplFile.

'file = unfilter(file)' undoes the effect of 'file = filter(file)'. 'sys.stdout = unfilter()' undoes the effect of 'sys.stdout = filter()'.

10.25 external.argparse

10.25.1 Module: external.argparse

Inheritance diagram for IPython.external.argparse:



Command-line parsing library

This module is an optparse-inspired command-line parsing library that:

- handles both optional and positional arguments
- produces highly informative usage messages
- supports parsers that dispatch to sub-parsers

The following is a simple usage example that sums integers from the command-line and writes the result to a file:

```
parser = argparse.ArgumentParser(
    description='sum the integers at the command line')
```

```
parser.add_argument(
    'integers', metavar='int', nargs='+', type=int,
    help='an integer to be summed')
parser.add_argument(
    '--log', default=sys.stdout, type=argparse.FileType('w'),
    help='the file where the sum should be written')
args = parser.parse_args()
args.log.write('%s' % sum(args.integers))
args.log.close()
```

The module contains the following public classes:

- **ArgumentParser – The main entry point for command-line parsing. As the** example above shows, the add_argument() method is used to populate the parser with actions for optional and positional arguments. Then the parse_args() method is invoked to convert the args at the command-line into an object with attributes.
- **ArgumentError – The exception raised by ArgumentParser objects when** there are errors with the parser's actions. Errors raised while parsing the command-line are caught by ArgumentParser and emitted as command-line messages.
- **FileType – A factory for defining types of files to be created. As the** example above shows, instances of FileType are typically passed as the type= argument of add_argument() calls.
- **Action – The base class for parser actions. Typically actions are** selected by passing strings like 'store_true' or 'append_const' to the action= argument of add_argument(). However, for greater customization of ArgumentParser actions, subclasses of Action may be defined and passed as the action= argument.
- **HelpFormatter, RawDescriptionHelpFormatter, RawTextHelpFormatter,** ArgumentDefaultsHelpFormatter – Formatter classes which may be passed as the formatter_class= argument to the ArgumentParser constructor. HelpFormatter is the default, RawDescriptionHelpFormatter and RawTextHelpFormatter tell the parser not to change the formatting for help text, and ArgumentDefaultsHelpFormatter adds information about argument defaults to the help.

All other classes in this module are considered implementation details. (Also note that HelpFormatter and RawDescriptionHelpFormatter are only considered public as object names – the API of the formatter objects is still considered an implementation detail.)

10.25.2 Classes

Action

class IPython.external.argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: IPython.external.argparse._AttributeHolder

Information about how to convert command line strings to Python objects.

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The keyword arguments to the Action constructor are also all attributes of Action instances.

Keyword Arguments:

- **option_strings – A list of command-line option strings which** should be associated with this action.
- dest – The name of the attribute to hold the created object(s)
- **nargs – The number of command-line arguments that should be** consumed. By default, one argument will be consumed and a single value will be produced. Other values include:
  - N (an integer) consumes N arguments (and produces a list)
  - '?' consumes zero or one arguments
  - '*' consumes zero or more arguments (and produces a list)
  - '+' consumes one or more arguments (and produces a list)

  Note that the difference between the default and nargs=1 is that with the default, a single value will be produced, while with nargs=1, a list containing a single value will be produced.
- **const – The value to be produced if the option is specified and the** option uses an action that takes no values.
- default – The value to be produced if the option is not specified.
- **type – The type which the command-line arguments should be converted** to, should be one of 'string', 'int', 'float', 'complex' or a callable object that accepts a single string argument. If None, 'string' is assumed.
- **choices – A container of values that should be allowed. If not None,** after a command-line argument has been converted to the appropriate type, an exception will be raised if it is not a member of this collection.
- **required – True if the action must always be specified at the** command line. This is only meaningful for optional command-line arguments.
- help – The help string describing the argument.
- **metavar – The name to be used for the option's argument with the** help string. If None, the 'dest' value will be used as the name.

__init__()

ArgumentDefaultsHelpFormatter

class IPython.external.argparse.**ArgumentDefaultsHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: IPython.external.argparse.HelpFormatter

Help message formatter which adds default values to argument help.

Only the name of this class is considered a public API. All the methods provided by the class are considered an implementation detail.

__init__()

ArgumentError

class IPython.external.argparse.**ArgumentError**(*argument*, *message*)
Bases: exceptions.Exception

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

__init__()

ArgumentParser

class IPython.external.argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'IPython.external.argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)
Bases: IPython.external.argparse._AttributeHolder, IPython.external.argparse._ActionsContainer

Object for parsing command line strings into Python objects.

Keyword Arguments:

- prog – The name of the program (default: sys.argv[0])
- usage – A usage message (default: auto-generated from arguments)
- description – A description of what the program does
- epilog – Text following the argument descriptions
- version – Add a -v/–version option with the given version string
- parents – Parsers whose arguments should be copied into this one
- formatter_class – HelpFormatter class for printing help messages
- prefix_chars – Characters that prefix optional arguments
- **fromfile_prefix_chars – Characters that prefix files containing** additional arguments

- argument_default – The default value for all arguments
- conflict_handler – String indicating how to handle conflicts
- add_help – Add a -h/-help option

__init__()

add_subparsers()

error()
Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

exit()

format_help()

format_usage()

format_version()

parse_args()

parse_known_args()

print_help()

print_usage()

print_version()

FileType

class IPython.external.argparse.**FileType**(*mode='r'*, *bufsize=None*)
Bases: object

Factory for creating file object types

Instances of FileType are typically passed as type= arguments to the ArgumentParser add_argument() method.

Keyword Arguments:

- **mode – A string indicating how the file is to be opened. Accepts the** same values as the builtin open() function.
- **bufsize – The file's desired buffer size. Accepts the same values as** the builtin open() function.

__init__()

HelpFormatter

class IPython.external.argparse.**HelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: object

Formatter for generating usage messages and argument help strings.

Only the name of this class is considered a public API. All the methods provided by the class are considered an implementation detail.

__init__()

add_argument()

add_arguments()

add_text()

add_usage()

end_section()

format_help()

start_section()

Namespace

class IPython.external.argparse.**Namespace**(***kwargs*)

Bases: IPython.external.argparse._AttributeHolder

Simple object for storing attributes.

Implements equality by attribute names and values, and provides a simple string representation.

__init__()

RawDescriptionHelpFormatter

class IPython.external.argparse.**RawDescriptionHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: IPython.external.argparse.HelpFormatter

Help message formatter which retains any formatting in descriptions.

Only the name of this class is considered a public API. All the methods provided by the class are considered an implementation detail.

__init__()

RawTextHelpFormatter

class IPython.external.argparse.**RawTextHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: IPython.external.argparse.RawDescriptionHelpFormatter

Help message formatter which retains formatting of all help text.

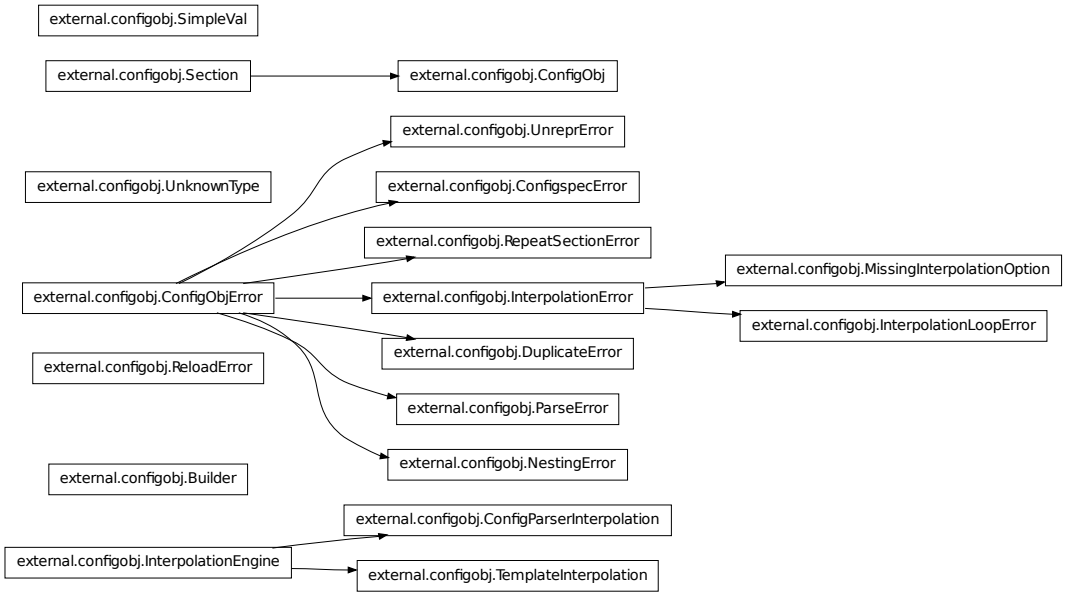
Only the name of this class is considered a public API. All the methods provided by the class are considered an implementation detail.

__init__()

10.26 external.configobj

10.26.1 Module: external.configobj

Inheritance diagram for IPython.external.configobj:



10.26.2 Classes

Builder

class IPython.external.configobj.**Builder**
Bases: object

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

build()

build_Add()

build_Const()

build_Dict()

build_Getattr()

build_List()

build_Name()

build_Tuple()

build_UnaryAdd()

build_UnarySub()

ConfigObj

class IPython.external.configobj.**ConfigObj**(*infile=None*, *options=None*, ***kwargs*)
Bases: IPython.external.configobj.Section

An object to read, create, and write config files.

__init__()
Parse a config file or create a config file object.

```
ConfigObj(infile=None, options=None, **kwargs)
```

reload()
Reload a ConfigObj from file.

This method raises a ReloadError if the ConfigObj doesn't have a filename attribute pointing to a file.

reset()
Clear ConfigObj instance and restore to 'freshly created' state.

validate()
Test the ConfigObj against a configspec.

It uses the validator object from *validate.py*.

To run validate on the current ConfigObj, call:

```
test = config.validate(validator)
```

(Normally having previously passed in the configspec when the ConfigObj was created - you can dynamically assign a dictionary of checks to the `configspec` attribute of a section though).

It returns `True` if everything passes, or a dictionary of pass/fails (True/False). If every member of a subsection passes, it will just have the value `True`. (It also returns `False` if all members fail).

In addition, it converts the values from strings to their native types if their checks pass (and `stringify` is set).

If `preserve_errors` is `True` (`False` is default) then instead of a marking a fail with a `False`, it will preserve the actual exception object. This can contain info about the reason for failure. For example the `VdtValueTooSmallError` indicates that the value supplied was too small. If a value (or section) is missing it will still be marked as `False`.

You must have the validate module to use `preserve_errors=True`.

You can then use the `flatten_errors` function to turn your nested results dictionary into a flattened list of failures - useful for displaying meaningful error messages.

write()

Write the current ConfigObj as a file

tekNico: FIXME: use StringIO instead of real files

```
>>> filename = a.filename
>>> a.filename = 'test.ini'
>>> a.write()
>>> a.filename = filename
>>> a == ConfigObj('test.ini', raise_errors=True)
1
```

ConfigObjError

class IPython.external.configobj.**ConfigObjError**(*message=''*, *line_number=None*, *line=''*)

Bases: exceptions.SyntaxError

This is the base class for all errors that ConfigObj raises. It is a subclass of SyntaxError.

__init__()

ConfigParserInterpolation

class IPython.external.configobj.**ConfigParserInterpolation**(*section*)

Bases: IPython.external.configobj.InterpolationEngine

Behaves like ConfigParser.

__init__()

ConfigspecError

class IPython.external.configobj.**ConfigspecError**(*message=''*, *line_number=None*, *line=''*)

Bases: IPython.external.configobj.ConfigObjError

An error occured whilst parsing a configspec.

__init__()

DuplicateError

class IPython.external.configobj.**DuplicateError**(*message=''*, *line_number=None*, *line=''*)

Bases: IPython.external.configobj.ConfigObjError

The keyword or section specified already exists.

__init__()

InterpolationEngine

class IPython.external.configobj.**InterpolationEngine**(*section*)

Bases: object

A helper class to help perform string interpolation.

This class is an abstract base class; its descendants perform the actual work.

__init__()

interpolate()

InterpolationError

class IPython.external.configobj.**InterpolationError**(*message=''*, *line_number=None*, *line=''*)

Bases: IPython.external.configobj.ConfigObjError

Base class for the two interpolation errors.

__init__()

InterpolationLoopError

class IPython.external.configobj.**InterpolationLoopError**(*option*)

Bases: IPython.external.configobj.InterpolationError

Maximum interpolation depth exceeded in string interpolation.

__init__()

MissingInterpolationOption

class IPython.external.configobj.**MissingInterpolationOption**(*option*)
Bases: IPython.external.configobj.InterpolationError

A value specified for interpolation was missing.

__init__()

NestingError

class IPython.external.configobj.**NestingError**(*message=''*, *line_number=None*, *line=''*)
Bases: IPython.external.configobj.ConfigObjError

This error indicates a level of nesting that doesn't match.

__init__()

ParseError

class IPython.external.configobj.**ParseError**(*message=''*, *line_number=None*, *line=''*)
Bases: IPython.external.configobj.ConfigObjError

This error indicates that a line is badly written. It is neither a valid `key = value` line, nor a valid section marker line.

__init__()

ReloadError

class IPython.external.configobj.**ReloadError**
Bases: exceptions.IOError

A 'reload' operation failed. This exception is a subclass of `IOError`.

__init__()

RepeatSectionError

class IPython.external.configobj.**RepeatSectionError**(*message=''*, *line_number=None*, *line=''*)
Bases: IPython.external.configobj.ConfigObjError

This error indicates additional sections in a section with a `__many__` (repeated) section.

__init__()

Section

class IPython.external.configobj.**Section**(*parent*, *depth*, *main*, *indict=None*, *name=None*)

Bases: dict

A dictionary-like object that represents a section in a config file.

It does string interpolation if the 'interpolation' attribute of the 'main' object is set to True.

Interpolation is tried first from this object, then from the 'DEFAULT' section of this object, next from the parent and its 'DEFAULT' section, and so on until the main object is reached.

A Section will behave like an ordered dictionary - following the order of the `scalars` and `sections` attributes. You can use this to change the order of members.

Iteration follows the order: scalars, then sections.

__init__()

- parent is the section above
- depth is the depth level of this section
- main is the main ConfigObj
- indict is a dictionary to initialise the section with

as_bool()

Accepts a key as input. The corresponding value must be a string or the objects (`True` or 1) or (`False` or 0). We allow 0 and 1 to retain compatibility with Python 2.2.

If the string is one of `True`, `On`, `Yes`, or `1` it returns `True`.

If the string is one of `False`, `Off`, `No`, or `0` it returns `False`.

`as_bool` is not case sensitive.

Any other input will raise a `ValueError`.

```
>>> a = ConfigObj()
>>> a['a'] = 'fish'
>>> a.as_bool('a')
Traceback (most recent call last):
ValueError: Value "fish" is neither True nor False
>>> a['b'] = 'True'
>>> a.as_bool('b')
1
>>> a['b'] = 'off'
>>> a.as_bool('b')
0
```

as_float()

A convenience method which coerces the specified value to a float.

If the value is an invalid literal for `float`, a `ValueError` will be raised.

```
>>> a = ConfigObj()
>>> a['a'] = 'fish'
>>> a.as_float('a')
Traceback (most recent call last):
ValueError: invalid literal for float(): fish
>>> a['b'] = '1'
>>> a.as_float('b')
1.0
>>> a['b'] = '3.2'
>>> a.as_float('b')
3.2000000000000002
```

as_int()

A convenience method which coerces the specified value to an integer.

If the value is an invalid literal for `int`, a `ValueError` will be raised.

```
>>> a = ConfigObj()
>>> a['a'] = 'fish'
>>> a.as_int('a')
Traceback (most recent call last):
ValueError: invalid literal for int(): fish
>>> a['b'] = '1'
>>> a.as_int('b')
1
>>> a['b'] = '3.2'
>>> a.as_int('b')
Traceback (most recent call last):
ValueError: invalid literal for int(): 3.2
```

clear()

A version of clear that also affects scalars/sections Also clears comments and configspec.

Leaves other attributes alone : depth/main/parent are not affected

decode()

Decode all strings and values to unicode, using the specified encoding.

Works with subsections and list values.

Uses the `walk` method.

Testing `encode` and `decode`. >>> m = ConfigObj(a) >>> m.decode('ascii') >>> def testuni(val): ... for entry in val: ... if not isinstance(entry, unicode): ... print >> sys.stderr, type(entry) ... raise AssertionError, 'decode failed.' ... if isinstance(val[entry], dict): ... testuni(val[entry]) ... elif not isinstance(val[entry], unicode): ... raise AssertionError, 'decode failed.' >>> testuni(m) >>> m.encode('ascii') >>> a == m 1

dict()

Return a deepcopy of self as a dictionary.

All members that are `Section` instances are recursively turned to ordinary dictionaries - by calling their `dict` method.

```
>>> n = a.dict()
>>> n == a
1
>>> n is a
0
```

encode()
Encode all strings and values from unicode, using the specified encoding.

Works with subsections and list values. Uses the `walk` method.

get()
A version of `get` that doesn't bypass string interpolation.

istrue()
A deprecated version of `as_bool`.

items()
D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems()
D.iteritems() -> an iterator over the (key, value) items of D

iterkeys()
D.iterkeys() -> an iterator over the keys of D

itervalues()
D.itervalues() -> an iterator over the values of D

keys()
D.keys() -> list of D's keys

merge()
A recursive update - useful for merging config files.

```
>>> a = '''[section1]
...     option1 = True
...     [[subsection]]
...     more_options = False
...     # end of file'''.splitlines()
>>> b = '''# File is user.ini
...     [section1]
...     option1 = False
...     # end of file'''.splitlines()
>>> c1 = ConfigObj(b)
>>> c2 = ConfigObj(a)
>>> c2.merge(c1)
>>> c2
{'section1': {'option1': 'False', 'subsection': {'more_options': 'False'}}}
```

pop()
'D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised'

popitem()
Pops the first (key,val)

rename()
Change a keyname to another, without changing position in sequence.

Implemented so that transformations can be made on keys, as well as on values. (used by encode and decode)

Also renames comments.

restore_default()
Restore (and return) default value for the specified key.

This method will only work for a ConfigObj that was created with a configspec and has been validated.

If there is no default value for this key, `KeyError` is raised.

restore_defaults()
Recursively restore default values to all members that have them.

This method will only work for a ConfigObj that was created with a configspec and has been validated.

It doesn't delete or modify entries without default values.

setdefault()
A version of setdefault that sets sequence if appropriate.

update()
A version of update that uses our `__setitem__`.

values()
D.values() -> list of D's values

walk()
Walk every member and call a function on the keyword and value.

Return a dictionary of the return values

If the function raises an exception, raise the errror unless `raise_errors=False`, in which case set the return value to `False`.

Any unrecognised keyword arguments you pass to walk, will be pased on to the function you pass in.

Note: if `call_on_sections` is `True` then - on encountering a subsection, *first* the function is called for the *whole* subsection, and then recurses into it's members. This means your function must be able to handle strings, dictionaries and lists. This allows you to change the key of subsections as well as for ordinary members. The return value when called on the whole subsection has to be discarded.

See the encode and decode methods for examples, including functions.

Caution: You can use `walk` to transform the names of members of a section but you mustn't add or delete members.

```
>>> config = '''[XXXXsection]
... XXXXkey = XXXXvalue'''.splitlines()
>>> cfg = ConfigObj(config)
>>> cfg
{'XXXXsection': {'XXXXkey': 'XXXXvalue'}}
>>> def transform(section, key):
...     val = section[key]
...     newkey = key.replace('XXXX', 'CLIENT1')
...     section.rename(key, newkey)
...     if isinstance(val, (tuple, list, dict)):
...         pass
...     else:
...         val = val.replace('XXXX', 'CLIENT1')
...         section[newkey] = val
>>> cfg.walk(transform, call_on_sections=True)
{'CLIENT1section': {'CLIENT1key': None}}
>>> cfg
{'CLIENT1section': {'CLIENT1key': 'CLIENT1value'}}
```

SimpleVal

class IPython.external.configobj.**SimpleVal**
Bases: object

A simple validator. Can be used to check that all members expected are present.

To use it, provide a configspec with all your members in (the value given will be ignored). Pass an instance of `SimpleVal` to the `validate` method of your `ConfigObj`. `validate` will return `True` if all members are present, or a dictionary with True/False meaning present/missing. (Whole missing sections will be replaced with `False`)

__init__()

check()
A dummy check method, always returns the value unchanged.

TemplateInterpolation

class IPython.external.configobj.**TemplateInterpolation**(*section*)
Bases: IPython.external.configobj.InterpolationEngine

Behaves like string.Template.

__init__()

UnknownType

class IPython.external.configobj.**UnknownType**
Bases: exceptions.Exception

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

UnreprError

class IPython.external.configobj.**UnreprError**(*message=''*, *line_number=None*, *line=''*)
Bases: IPython.external.configobj.ConfigObjError

An error parsing in unrepr mode.

__init__()

10.26.3 Functions

IPython.external.configobj.**flatten_errors**()
An example function that will turn a nested dictionary of results (as returned by ConfigObj.validate) into a flat list.

cfg is the ConfigObj instance being checked, res is the results dictionary returned by validate.

(This is a recursive function, so you shouldn't use the levels or results arguments - they are used by the function.

Returns a list of keys that failed. Each member of the list is a tuple :

```
([list of sections...], key, result)
```

If validate was called with preserve_errors=False (the default) then result will always be False.

list of sections is a flattened list of sections that the key was found in.

If the section was missing then key will be None.

If the value (or section) was missing then result will be False.

If validate was called with preserve_errors=True and a value was present, but failed the check, then result will be the exception object returned. You can use this as a string that describes the failure.

For example *The value "3" is of the wrong type*.

```
>>> import validate
>>> vtor = validate.Validator()
>>> my_ini = '''
...     option1 = True
...     [section1]
...     option1 = True
```

```
...     [section2]
...     another_option = Probably
...     [section3]
...     another_option = True
...     [[section3b]]
...     value = 3
...     value2 = a
...     value3 = 11
...     '''
>>> my_cfg = '''
...     option1 = boolean()
...     option2 = boolean()
...     option3 = boolean(default=Bad_value)
...     [section1]
...     option1 = boolean()
...     option2 = boolean()
...     option3 = boolean(default=Bad_value)
...     [section2]
...     another_option = boolean()
...     [section3]
...     another_option = boolean()
...     [[section3b]]
...     value = integer
...     value2 = integer
...     value3 = integer(0, 10)
...         [[[section3b-sub]]]
...         value = string
...     [section4]
...     another_option = boolean()
...     '''
>>> cs = my_cfg.split('\n')
>>> ini = my_ini.split('\n')
>>> cfg = ConfigObj(ini, configspec=cs)
>>> res = cfg.validate(vtor, preserve_errors=True)
>>> errors = []
>>> for entry in flatten_errors(cfg, res):
...     section_list, key, error = entry
...     section_list.insert(0, '[root]')
...     if key is not None:
...        section_list.append(key)
...     else:
...         section_list.append('[missing]')
...     section_string = ', '.join(section_list)
...     errors.append((section_string, ' = ', error))
>>> errors.sort()
>>> for entry in errors:
...     print entry[0], entry[1], (entry[2] or 0)
[root], option2  =  0
[root], option3  =  the value "Bad_value" is of the wrong type.
[root], section1, option2  =  0
[root], section1, option3  =  the value "Bad_value" is of the wrong type.
[root], section2, another_option  =  the value "Probably" is of the wrong type.
[root], section3, section3b, section3b-sub, [missing]  =  0
```

```
        [root], section3, section3b, value2  =  the value "a" is of the wrong type.
        [root], section3, section3b, value3  =  the value "11" is too big.
        [root], section4, [missing]  =  0
```

IPython.external.configobj.**getObj**()

IPython.external.configobj.**match_utf8**()

IPython.external.configobj.**unrepr**()

10.27 external.guid

10.27.1 Module: external.guid

10.27.2 Functions

IPython.external.guid.**extract_counter**()
Extracts the counter from the guid (returns the bits in decimal)

IPython.external.guid.**extract_ip**()
Extracts the ip portion out of the guid and returns it as a string like 10.10.10.10

IPython.external.guid.**extract_time**()
Extracts the time portion out of the guid and returns the number of seconds since the epoch as a float

IPython.external.guid.**generate**()
Generates a new guid. A guid is unique in space and time because it combines the machine IP with the current time in milliseconds. Be careful about sending in a specified IP address because the ip makes it unique in space. You could send in the same IP address that is created on another machine.

10.28 external.mglob

10.28.1 Module: external.mglob

mglob - enhanced file list expansion module

Use as stand-alone utility (for xargs, *backticks* etc.), or a globbing library for own python programs. Globbing the sys.argv is something that almost every Windows script has to perform manually, and this module is here to help with that task. Also Unix users will benefit from enhanced modes such as recursion, exclusion, directory omission...

Unlike glob.glob, directories are not included in the glob unless specified with 'dir:'

'expand' is the function to use in python programs. Typical use to expand argv (esp. in windows):

```
try:
    import mglob
    files = mglob.expand(sys.argv[1:])
except ImportError:
```

```
    print "mglob not found; try 'easy_install mglob' for extra features"
    files = sys.argv[1:]
```

Note that for unix, shell expands *normal* wildcards (*.cpp, etc.) in argv. Therefore, you might want to use quotes with normal wildcards to prevent this expansion, in order for mglob to see the wildcards and get the wanted behaviour. Not quoting the wildcards is harmless and typically has equivalent results, though.

Author: Ville Vainio <vivainio@gmail.com> License: MIT Open Source license

10.28.2 Functions

`IPython.external.mglob.`**`expand`**`()`
Expand the glob(s) in flist.

flist may be either a whitespace-separated list of globs/files or an array of globs/files.

if exp_dirs is true, directory names in glob are expanded to the files contained in them - otherwise, directory names are returned as is.

`IPython.external.mglob.`**`init_ipython`**`()`
register %mglob for IPython

`IPython.external.mglob.`**`main`**`()`

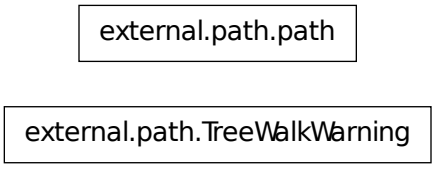
`IPython.external.mglob.`**`mglob_f`**`()`

`IPython.external.mglob.`**`test`**`()`

10.29 external.path

10.29.1 Module: `external.path`

Inheritance diagram for `IPython.external.path`:



path.py - An object representing a path to a file or directory.

Example:

from IPython.external.path import path d = path('/home/guido/bin') for f in d.files('*.py'):

f.chmod(0755)

This module requires Python 2.2 or later.

URL: http://www.jorendorff.com/articles/python/path Author: Jason Orendorff <jason.orendorff@gmail.com> (and others - see the url!) Date: 9 Mar 2007

10.29.2 Classes

TreeWalkWarning

class IPython.external.path.**TreeWalkWarning**
Bases: exceptions.Warning

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

path

class IPython.external.path.**path**
Bases: str

Represents a filesystem path.

For documentation on individual methods, consult their counterparts in os.path.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

abspath()

access()
Return true if current user has access to this path.

mode - One of the constants os.F_OK, os.R_OK, os.W_OK, os.X_OK

atime
Last access time of the file.

basename()
Returns the final component of a pathname

bytes()
Open this file, read all bytes, return them as a string.

chmod()

chown()

chroot()

copy()
Copy data and mode bits ("cp src dst").

The destination may be a directory.

copy2()
Copy data and all stat info ("cp -p src dst").

The destination may be a directory.

copyfile()
Copy data from src to dst

copymode()
Copy mode bits from src to dst

copystat()
Copy all stat info (mode bits, atime, mtime, flags) from src to dst

copytree()
Recursively copy a directory tree using copy2().

The destination directory must not already exist. If exception(s) occur, an Error is raised with a list of reasons.

If the optional symlinks flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied.

The optional ignore argument is a callable. If given, it is called with the *src* parameter, which is the directory being visited by copytree(), and *names* which is the list of *src* contents, as returned by os.listdir():

callable(src, names) -> ignored_names

Since copytree() is called recursively, the callable will be called once for each directory that is copied. It returns a list of names relative to the *src* directory that should not be copied.

XXX Consider this example code rather than the ultimate tool.

ctime
Creation time of the file.

dirname()

dirs()
D.dirs() -> List of this directory's subdirectories.

The elements of the list are path objects. This does not walk recursively into subdirectories (but see path.walkdirs).

With the optional 'pattern' argument, this only lists directories whose names match the given pattern. For example, d.dirs('build-*').

drive
The drive specifier, for example 'C:'. This is always empty on systems that don't use drive specifiers.

exists()
Test whether a path exists. Returns False for broken symbolic links

expand()
Clean up a filename by calling expandvars(), expanduser(), and normpath() on it.

This is commonly everything needed to clean up a filename read from a configuration file, for example.

expanduser()

expandvars()

ext
The file extension, for example '.py'.

files()
D.files() -> List of the files in this directory.

The elements of the list are path objects. This does not walk into subdirectories (see path.walkfiles).

With the optional 'pattern' argument, this only lists files whose names match the given pattern. For example, d.files('*.pyc').

fnmatch()
Return True if self.name matches the given pattern.

pattern - A filename pattern with wildcards, for example '*.py'.

get_owner()
Return the name of the owner of this file or directory.

This follows symbolic links.

On Windows, this returns a name of the form ur'DOMAINUser Name'. On Windows, a group can own a file or directory.

getatime()
Return the last access time of a file, reported by os.stat().

getctime()
Return the metadata change time of a file, reported by os.stat().

classmethod getcwd()
Return the current working directory as a path object.

getmtime()
Return the last modification time of a file, reported by os.stat().

getsize()
Return the size of a file, reported by os.stat().

glob()
Return a list of path objects that match the pattern.

pattern - a path relative to this directory, with wildcards.

For example, path('/users').glob('*/bin/*') returns a list of all the files users have in their bin directories.

isabs()
Test whether a path is absolute

isdir()
Return true if the pathname refers to an existing directory.

isfile()
Test whether a path is a regular file

islink()
Test whether a path is a symbolic link

ismount()
Test whether a path is a mount point

joinpath()
Join two or more path components, adding a separator character (os.sep) if needed. Returns a new path object.

lines()
Open this file, read all lines, return them in a list.

Optional arguments:

encoding - The Unicode encoding (or character set) of the file. The default is None, meaning the content of the file is read as 8-bit characters and returned as a list of (non-Unicode) str objects.

errors - How to handle Unicode errors; see help(str.decode) for the options. Default is 'strict'

retain - If true, retain newline characters; but all newline character combinations ('r', 'n', 'rn') are translated to 'n'. If false, newline characters are stripped off. Default is True.

This uses 'U' mode in Python 2.3 and later.

link()
Create a hard link at 'newpath', pointing to this file.

listdir()
D.listdir() -> List of items in this directory.

Use D.files() or D.dirs() instead if you want a listing of just files or just subdirectories.

The elements of the list are path objects.

With the optional 'pattern' argument, this only lists items whose names match the given pattern.

lstat()
Like path.stat(), but do not follow symbolic links.

makedirs()

mkdir()

move()
Recursively move a file or directory to another location. This is similar to the Unix "mv" command.

If the destination is a directory or a symlink to a directory, the source is moved inside the directory. The destination path must not already exist.

If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on our current filesystem, then rename() is used. Otherwise, src is copied to the destination and then removed. A lot more could be done here... A look at a mv.c shows a lot of the issues this implementation glosses over.

mtime
Last-modified time of the file.

name
The name of this file or directory without the full path.

For example, path('/usr/local/lib/libpython.so').name == 'libpython.so'

namebase
The same as path.name, but with one file extension stripped off.

For example, path('/home/guido/python.tar.gz').name == 'python.tar.gz', but path('/home/guido/python.tar.gz').namebase == 'python.tar'

normcase()

normpath()

open()
Open this file. Return a file object.

owner
Name of the owner of this file or directory.

parent
This path's parent directory, as a new path object.

For example, path('/usr/local/lib/libpython.so').parent == path('/usr/local/lib')

pathconf()

read_md5()
Calculate the md5 hash for this file.

This reads through the entire file.

readlink()
Return the path to which this symbolic link points.

The result may be an absolute or a relative path.

readlinkabs()
Return the path to which this symbolic link points.

The result is always an absolute path.

realpath()

relpath()
Return this path as a relative path, based from the current working directory.

relpathto()
Return a relative path from self to dest.

If there is no relative path from self to dest, for example if they reside on different drives in Windows, then this returns dest.abspath().

remove()

removedirs()

rename()

renames()

rmdir()

rmtree()
Recursively delete a directory tree.

If ignore_errors is set, errors are ignored; otherwise, if onerror is set, it is called to handle the error with arguments (func, path, exc_info) where func is os.listdir, os.remove, or os.rmdir; path is the argument to that function that caused it to fail; and exc_info is a tuple returned by sys.exc_info(). If ignore_errors is false and onerror is None, an exception is raised.

samefile()
Test whether two pathnames reference the same actual file

size
Size of the file, in bytes.

splitall()
Return a list of the path components in this path.

The first item in the list will be a path. Its value will be either os.curdir, os.pardir, empty, or the root directory of this path (for example, '/' or 'C:\'). The other items in the list will be strings.

path.path.joinpath(*result) will yield the original path.

splitdrive()
p.splitdrive() -> Return (p.drive, <the rest of p>).

Split the drive specifier from this path. If there is no drive specifier, p.drive is empty, so the return value is simply (path(''), p). This is always the case on Unix.

splitext()
p.splitext() -> Return (p.stripext(), p.ext).

Split the filename extension from this path and return the two parts. Either part may be empty.

The extension is everything from '.' to the end of the last path segment. This has the property that if (a, b) == p.splitext(), then a + b == p.

splitpath()
p.splitpath() -> Return (p.parent, p.name).

stat()
Perform a stat() system call on this path.

statvfs()
Perform a statvfs() system call on this path.

stripext()
p.stripext() -> Remove one file extension from the path.

For example, path('/home/guido/python.tar.gz').stripext() returns path('/home/guido/python.tar').

symlink()
Create a symbolic link at 'newlink', pointing here.

text()
Open this file, read it in, return the content as a string.

This uses 'U' mode in Python 2.3 and later, so 'rn' and 'r' are automatically translated to 'n'.

Optional arguments:

encoding - The Unicode encoding (or character set) of the file. If present, the content of the file is decoded and returned as a unicode object; otherwise it is returned as an 8-bit str.

errors - How to handle Unicode errors; see help(str.decode) for the options. Default is 'strict'.

touch()
Set the access/modified times of this file to the current time. Create the file if it does not exist.

unlink()

utime()
Set the access and modified times of this file.

walk()
D.walk() -> iterator over files and subdirs, recursively.

The iterator yields path objects naming each child item of this directory and its descendants. This requires that D.isdir().

This performs a depth-first traversal of the directory tree. Each directory is returned just before all its children.

The errors= keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. The other allowed values are 'warn', which reports the error via warnings.warn(), and 'ignore'.

walkdirs()
D.walkdirs() -> iterator over subdirs, recursively.

With the optional 'pattern' argument, this yields only directories whose names match the given pattern. For example, mydir.walkdirs('*test') yields only directories with names ending in 'test'.

The errors= keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. The other allowed values are 'warn', which reports the error via warnings.warn(), and 'ignore'.

walkfiles()
D.walkfiles() -> iterator over files in D, recursively.

The optional argument, pattern, limits the results to files with names that match the pattern. For example, mydir.walkfiles('*.tmp') yields only files with the .tmp extension.

write_bytes()
Open this file and write the given bytes to it.

Default behavior is to overwrite any existing file. Call p.write_bytes(bytes, append=True) to append instead.

write_lines()
Write the given lines of text to this file.

By default this overwrites any existing file at this path.

This puts a platform-specific newline sequence on every line. See 'linesep' below.

lines - A list of strings.

encoding - A Unicode encoding to use. This applies only if 'lines' contains any Unicode strings.

errors - How to handle errors in Unicode encoding. This also applies only to Unicode strings.

linesep - The desired line-ending. This line-ending is applied to every line. If a line already has any standard line ending ('r', 'n', 'rn', u'x85', u'rx85', u'u2028'), that will be stripped off and this will be used instead. The default is os.linesep, which is platform-dependent ('rn' on Windows, 'n' on Unix, etc.) Specify None to write the lines as-is, like file.writelines().

Use the keyword argument append=True to append lines to the file. The default is to overwrite the file. Warning: When you use this with Unicode data, if the encoding of the existing data in the file is different from the encoding you specify with the encoding= parameter, the result is mixed-encoding data, which can really confuse someone trying to read the file later.

write_text()
Write the given text to this file.

The default behavior is to overwrite any existing file; to append instead, use the 'append=True' keyword argument.

There are two differences between path.write_text() and path.write_bytes(): newline handling and Unicode handling. See below.

Parameters:

- text - str/unicode - The text to be written.
- encoding - str - The Unicode encoding that will be used. This is ignored if 'text' isn't a Unicode string.

- errors - str - How to handle Unicode encoding errors. Default is 'strict'. See help(unicode.encode) for the options. This is ignored if 'text' isn't a Unicode string.
- linesep - keyword argument - str/unicode - The sequence of characters to be used to mark end-of-line. The default is os.linesep. You can also specify None; this means to leave all newlines as they are in 'text'.
- append - keyword argument - bool - Specifies what to do if the file already exists (True: append to the end of it; False: overwrite it.) The default is False.

— Newline handling.

write_text() converts all standard end-of-line sequences ('n', 'r', and 'rn') to your platform's default end-of-line sequence (see os.linesep; on Windows, for example, the end-of-line marker is 'rn').

If you don't like your platform's default, you can override it using the 'linesep=' keyword argument. If you specifically want write_text() to preserve the newlines as-is, use 'linesep=None'.

This applies to Unicode text the same as to 8-bit text, except there are three additional standard Unicode end-of-line sequences: u'x85', u'rx85', and u'u2028'.

(This is slightly different from when you open a file for writing with fopen(filename, "w") in C or file(filename, 'w') in Python.)

— Unicode

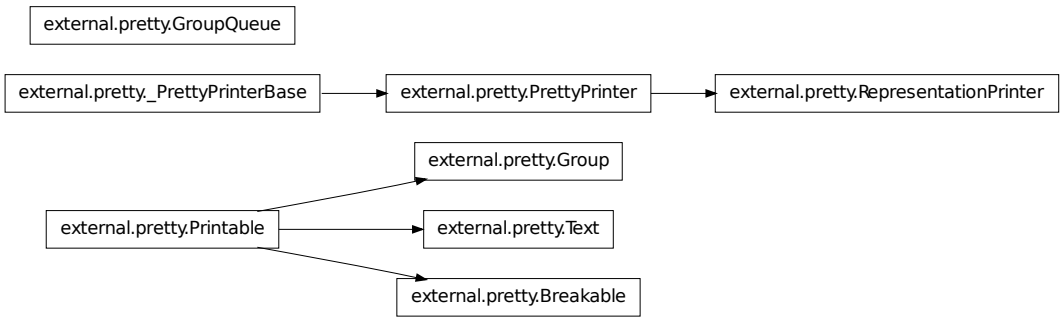
If 'text' isn't Unicode, then apart from newline handling, the bytes are written verbatim to the file. The 'encoding' and 'errors' arguments are not used and must be omitted.

If 'text' is Unicode, it is first converted to bytes using the specified 'encoding' (or the default encoding if 'encoding' isn't specified). The 'errors' argument applies only to this conversion.

10.30 external.pretty

10.30.1 Module: `external.pretty`

Inheritance diagram for `IPython.external.pretty`:



pretty ~~

Python advanced pretty printer. This pretty printer is intended to replace the old *pprint* python module which does not allow developers to provide their own pretty print callbacks.

This module is based on ruby's *prettyprint.rb* library by *Tanaka Akira*.

Example Usage

To directly print the representation of an object use *pprint*:

```
from pretty import pprint
pprint(complex_object)
```

To get a string of the output use *pretty*:

```
from pretty import pretty
string = pretty(complex_object)
```

Extending

The pretty library allows developers to add pretty printing rules for their own objects. This process is straightforward. All you have to do is to add a *__pretty__* method to your object and call the methods on the pretty printer passed:

```
class MyObject(object):

    def __pretty__(self, p, cycle):
        ...
```

Depending on the python version you want to support you have two possibilities. The following list shows the python 2.5 version and the compatibility one.

Here the example implementation of a *__pretty__* method for a list subclass for python 2.5 and higher (python 2.5 requires the with statement __future__ import):

```
class MyList(list):

    def __pretty__(self, p, cycle):
        if cycle:
            p.text('MyList(...)')
        else:
            with p.group(8, 'MyList([', '])'):
                for idx, item in enumerate(self):
                    if idx:
                        p.text(',')
                        p.breakable()
                    p.pretty(item)
```

The *cycle* parameter is *True* if pretty detected a cycle. You *have* to react to that or the result is an infinite loop. *p.text()* just adds non breaking text to the output, *p.breakable()* either adds a whitespace or breaks here. If you pass it an argument it's used instead of the default space. *p.pretty* prettyprints another object using the pretty print method.

The first parameter to the *group* function specifies the extra indentation of the next line. In this example the next item will either be not breaked (if the items are short enough) or aligned with the right edge of the opening bracked of *MyList*.

If you want to support python 2.4 and lower you can use this code:

```
class MyList(list):

    def __pretty__(self, p, cycle):
        if cycle:
            p.text('MyList(...)')
        else:
            p.begin_group(8, 'MyList([')
            for idx, item in enumerate(self):
                if idx:
                    p.text(',')
                    p.breakable()
                p.pretty(item)
            p.end_group(8, '])')
```

If you just want to indent something you can use the group function without open / close parameters. Under python 2.5 you can also use this code:

```
with p.indent(2):
    ...
```

Or under python2.4 you might want to modify `p.indentation` by hand but this is rather ugly.

10.30.2 Classes

Breakable

class IPython.external.pretty.**Breakable**(*seq*, *width*, *pretty*)
Bases: IPython.external.pretty.Printable

__init__()

output()

Group

class IPython.external.pretty.**Group**(*depth*)
Bases: IPython.external.pretty.Printable

__init__()

GroupQueue

class IPython.external.pretty.**GroupQueue**(**groups*)
Bases: object

__init__()

deq()

enq()

remove()

PrettyPrinter

class IPython.external.pretty.**PrettyPrinter**(*output*, *max_width=79*, *newline='n'*)
Bases: IPython.external.pretty._PrettyPrinterBase

Baseclass for the *RepresentationPrinter* prettyprinter that is used to generate pretty reprs of objects. Contrary to the *RepresentationPrinter* this printer knows nothing about the default pprinters or the *__pretty__* callback method.

__init__()

begin_group()
Begin a group. If you want support for python < 2.5 which doesn't has the with statement this is the preferred way:

p.begin_group(1, '{') ... p.end_group(1, '}')

The python 2.5 expression would be this:

with p.group(1, '{', '}'): ...

The first parameter specifies the indentation for the next line (usually the width of the opening text), the second the opening text. All parameters are optional.

breakable()
Add a breakable separator to the output. This does not mean that it will automatically break here. If no breaking on this position takes place the *sep* is inserted which default to one space.

end_group()
End a group. See *begin_group* for more details.

flush()
Flush data that is left in the buffer.

text()
Add literal text to the output.

Printable

class IPython.external.pretty.**Printable**
Bases: object

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

output()

RepresentationPrinter

class IPython.external.pretty.**RepresentationPrinter**(*output*, *verbose=False*, *max_width=79*, *newline='n'*)
Bases: IPython.external.pretty.PrettyPrinter

Special pretty printer that has a *pretty* method that calls the pretty printer for a python object.

This class stores processing data on *self* so you must *never* use this class in a threaded environment. Always lock it or reinstanciate it.

Instances also have a verbose flag callbacks can access to control their output. For example the default instance repr prints all attributes and methods that are not prefixed by an underscore if the printer is in verbose mode.

__init__()

pretty()
Pretty print the given object.

Text

class IPython.external.pretty.**Text**
Bases: IPython.external.pretty.Printable

__init__()

add()

output()

10.30.3 Functions

IPython.external.pretty.**for_type**()
Add a pretty printer for a given type.

IPython.external.pretty.**for_type_by_name**()
Add a pretty printer for a type specified by the module and name of a type rather than the type object itself.

IPython.external.pretty.**pprint**()
Like *pretty* but print to stdout.

IPython.external.pretty.**pretty**()
Pretty print the object's representation.

10.31 external.simplegeneric

10.31.1 Module: external.simplegeneric

10.31.2 Functions

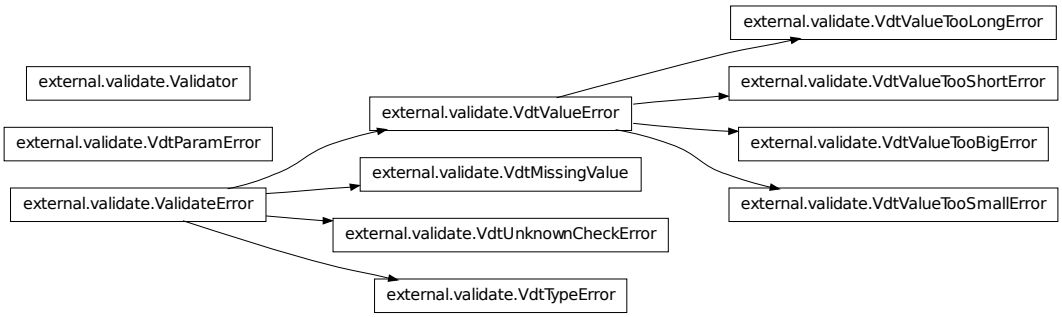
IPython.external.simplegeneric.**generic**()
Create a simple generic function

IPython.external.simplegeneric.**test_suite**()

10.32 external.validate

10.32.1 Module: external.validate

Inheritance diagram for IPython.external.validate:



The Validator object is used to check that supplied values conform to a specification.

The value can be supplied as a string - e.g. from a config file. In this case the check will also *convert* the value to the required type. This allows you to add validation as a transparent layer to access data stored as strings. The validation checks that the data is correct *and* converts it to the expected type.

Some standard checks are provided for basic data types. Additional checks are easy to write. They can be provided when the `Validator` is instantiated or added afterwards.

The standard functions work with the following basic data types :

- integers
- floats
- booleans
- strings
- ip_addr

plus lists of these datatypes

Adding additional checks is done through coding simple functions.

The full set of standard checks are :

- **'integer': matches integer values (including negative)** Takes optional 'min' and 'max' arguments :

```
integer()
integer(3, 9)   # any value from 3 to 9
integer(min=0)  # any positive value
integer(max=9)
```

- **'float': matches float values** Has the same parameters as the integer check.
- **'boolean': matches boolean values - `True` or `False`**

  Acceptable string values for True are : true, on, yes, 1

  Acceptable string values for False are : false, off, no, 0

Any other value raises an error.

- **'ip_addr': matches an Internet Protocol address, v.4, represented** by a dotted-quad string, i.e. '1.2.3.4'.
- **'string': matches any string.** Takes optional keyword args 'min' and 'max' to specify min and max lengths of the string.
- **'list': matches any list.** Takes optional keyword args 'min', and 'max' to specify min and max sizes of the list. (Always returns a list.)
- **'tuple': matches any tuple.** Takes optional keyword args 'min', and 'max' to specify min and max sizes of the tuple. (Always returns a tuple.)
- **'int_list': Matches a list of integers.** Takes the same arguments as list.
- **'float_list': Matches a list of floats.** Takes the same arguments as list.
- **'bool_list': Matches a list of boolean values.** Takes the same arguments as list.
- **'ip_addr_list': Matches a list of IP addresses.** Takes the same arguments as list.
- **'string_list': Matches a list of strings.** Takes the same arguments as list.
- **'mixed_list': Matches a list with different types in** specific positions. List size must match the number of arguments.

  Each position can be one of : 'integer', 'float', 'ip_addr', 'string', 'boolean'

  So to specify a list with two strings followed by two integers, you write the check as :

  ```
  mixed_list('string', 'string', 'integer', 'integer')
  ```

- **'pass': This check matches everything ! It never fails** and the value is unchanged.

  It is also the default if no check is specified.

- **'option': This check matches any from a list of options.** You specify this check with :

  ```
  option('option 1', 'option 2', 'option 3')
  ```

You can supply a default value (returned if no value is supplied) using the default keyword argument.

You specify a list argument for default using a list constructor syntax in the check :

```
checkname(arg1, arg2, default=list('val 1', 'val 2', 'val 3'))
```

A badly formatted set of arguments will raise a `VdtParamError`.

10.32.2 Classes

ValidateError

class `IPython.external.validate.`**ValidateError**
Bases: `exceptions.Exception`

This error indicates that the check failed. It can be the base class for more specific errors.

Any check function that fails ought to raise this error. (or a subclass)

```
>>> raise ValidateError
Traceback (most recent call last):
ValidateError
```

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Validator

class IPython.external.validate.**Validator**(*functions=None*)
Bases: object

Validator is an object that allows you to register a set of 'checks'. These checks take input and test that it conforms to the check.

This can also involve converting the value from a string into the correct datatype.

The check method takes an input string which configures which check is to be used and applies that check to a supplied value.

An example input string would be: 'int_range(param1, param2)'

You would then provide something like:

```
>>> def int_range_check(value, min, max):
...     # turn min and max from strings to integers
...     min = int(min)
...     max = int(max)
...     # check that value is of the correct type.
...     # possible valid inputs are integers or strings
...     # that represent integers
...     if not isinstance(value, (int, long, StringTypes)):
...         raise VdtTypeError(value)
...     elif isinstance(value, StringTypes):
...         # if we are given a string
...         # attempt to convert to an integer
...         try:
...             value = int(value)
...         except ValueError:
...             raise VdtValueError(value)
...     # check the value is between our constraints
...     if not min <= value:
...          raise VdtValueTooSmallError(value)
...     if not value <= max:
...          raise VdtValueTooBigError(value)
...     return value

>>> fdict = {'int_range': int_range_check}
>>> vtr1 = Validator(fdict)
>>> vtr1.check('int_range(20, 40)', '30')
30
>>> vtr1.check('int_range(20, 40)', '60')
```

```
Traceback (most recent call last):
VdtValueTooBigError: the value "60" is too big.
```

New functions can be added with :

```
>>> vtr2 = Validator()
>>> vtr2.functions['int_range'] = int_range_check
```

Or by passing in a dictionary of functions when Validator is instantiated.

Your functions *can* use keyword arguments, but the first argument should always be 'value'.

If the function doesn't take additional arguments, the parentheses are optional in the check. It can be written with either of :

```
keyword = function_name
keyword = function_name()
```

The first program to utilise Validator() was Michael Foord's ConfigObj, an alternative to ConfigParser which supports lists and can validate a config file using a config schema. For more details on using Validator with ConfigObj see: http://www.voidspace.org.uk/python/configobj.html

__init__()

```
>>> vtri = Validator()
```

check()

Usage: check(check, value)

Arguments: check: string representing check to apply (including arguments) value: object to be checked

Returns value, converted to correct type if necessary

If the check fails, raises a `ValidateError` subclass.

```
>>> vtor.check('yoda', '')
Traceback (most recent call last):
VdtUnknownCheckError: the check "yoda" is unknown.
>>> vtor.check('yoda()', '')
Traceback (most recent call last):
VdtUnknownCheckError: the check "yoda" is unknown.

>>> vtor.check('string(default="")', '', missing=True)
''
```

get_default_value()

Given a check, return the default value for the check (converted to the right type).

If the check doesn't specify a default value then a `KeyError` will be raised.

VdtMissingValue

class IPython.external.validate.**VdtMissingValue**
Bases: IPython.external.validate.ValidateError

No value was supplied to a check that needed one.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

VdtParamError

class IPython.external.validate.**VdtParamError**(*name*, *value*)
Bases: exceptions.SyntaxError

An incorrect parameter was passed

__init__()

```
>>> raise VdtParamError('yoda', 'jedi')
Traceback (most recent call last):
VdtParamError: passed an incorrect value "jedi" for parameter "yoda".
```

VdtTypeError

class IPython.external.validate.**VdtTypeError**(*value*)
Bases: IPython.external.validate.ValidateError

The value supplied was of the wrong type

__init__()

```
>>> raise VdtTypeError('jedi')
Traceback (most recent call last):
VdtTypeError: the value "jedi" is of the wrong type.
```

VdtUnknownCheckError

class IPython.external.validate.**VdtUnknownCheckError**(*value*)
Bases: IPython.external.validate.ValidateError

An unknown check function was requested

__init__()

```
>>> raise VdtUnknownCheckError('yoda')
Traceback (most recent call last):
VdtUnknownCheckError: the check "yoda" is unknown.
```

VdtValueError

class IPython.external.validate.**VdtValueError**(*value*)
Bases: IPython.external.validate.ValidateError

The value supplied was of the correct type, but was not an allowed value.

__init__()

```
>>> raise VdtValueError('jedi')
Traceback (most recent call last):
VdtValueError: the value "jedi" is unacceptable.
```

VdtValueTooBigError

class IPython.external.validate.**VdtValueTooBigError**(*value*)
Bases: IPython.external.validate.VdtValueError

The value supplied was of the correct type, but was too big.

__init__()

```
>>> raise VdtValueTooBigError('1')
Traceback (most recent call last):
VdtValueTooBigError: the value "1" is too big.
```

VdtValueTooLongError

class IPython.external.validate.**VdtValueTooLongError**(*value*)
Bases: IPython.external.validate.VdtValueError

The value supplied was of the correct type, but was too long.

__init__()

```
>>> raise VdtValueTooLongError('jedie')
Traceback (most recent call last):
VdtValueTooLongError: the value "jedie" is too long.
```

VdtValueTooShortError

class IPython.external.validate.**VdtValueTooShortError**(*value*)
Bases: IPython.external.validate.VdtValueError

The value supplied was of the correct type, but was too short.

__init__()

```
>>> raise VdtValueTooShortError('jed')
Traceback (most recent call last):
VdtValueTooShortError: the value "jed" is too short.
```

VdtValueTooSmallError

class IPython.external.validate.**VdtValueTooSmallError**(*value*)
Bases: IPython.external.validate.VdtValueError

The value supplied was of the correct type, but was too small.

__init__()

```
>>> raise VdtValueTooSmallError('0')
Traceback (most recent call last):
VdtValueTooSmallError: the value "0" is too small.
```

10.32.3 Functions

IPython.external.validate.**dottedQuadToNum**()
Convert decimal dotted quad string to long integer

```
>>> dottedQuadToNum('1 ')
1L
>>> dottedQuadToNum(' 1.2')
16777218L
>>> dottedQuadToNum(' 1.2.3 ')
16908291L
>>> dottedQuadToNum('1.2.3.4')
16909060L
>>> dottedQuadToNum('1.2.3. 4')
Traceback (most recent call last):
ValueError: Not a good dotted-quad IP: 1.2.3. 4
>>> dottedQuadToNum('255.255.255.255')
4294967295L
>>> dottedQuadToNum('255.255.255.256')
Traceback (most recent call last):
ValueError: Not a good dotted-quad IP: 255.255.255.256
```

IPython.external.validate.**is_bool_list**()
Check that the value is a list of booleans.

You can optionally specify the minimum and maximum number of members.

Each list member is checked that it is a boolean.

```
>>> vtor.check('bool_list', ())
[]
>>> vtor.check('bool_list', [])
[]
```

```
>>> check_res = vtor.check('bool_list', (True, False))
>>> check_res == [True, False]
1
>>> check_res = vtor.check('bool_list', [True, False])
>>> check_res == [True, False]
1
>>> vtor.check('bool_list', [True, 'a'])
Traceback (most recent call last):
VdtTypeError: the value "a" is of the wrong type.
```

IPython.external.validate.**is_boolean**()

Check if the value represents a boolean.

```
>>> vtor.check('boolean', 0)
0
>>> vtor.check('boolean', False)
0
>>> vtor.check('boolean', '0')
0
>>> vtor.check('boolean', 'off')
0
>>> vtor.check('boolean', 'false')
0
>>> vtor.check('boolean', 'no')
0
>>> vtor.check('boolean', 'nO')
0
>>> vtor.check('boolean', 'NO')
0
>>> vtor.check('boolean', 1)
1
>>> vtor.check('boolean', True)
1
>>> vtor.check('boolean', '1')
1
>>> vtor.check('boolean', 'on')
1
>>> vtor.check('boolean', 'true')
1
>>> vtor.check('boolean', 'yes')
1
>>> vtor.check('boolean', 'Yes')
1
>>> vtor.check('boolean', 'YES')
1
>>> vtor.check('boolean', '')
Traceback (most recent call last):
VdtTypeError: the value "" is of the wrong type.
>>> vtor.check('boolean', 'up')
Traceback (most recent call last):
VdtTypeError: the value "up" is of the wrong type.
```

IPython.external.validate.**is_float**()

A check that tests that a given value is a float (an integer will be accepted), and optionally - that it is between bounds.

If the value is a string, then the conversion is done - if possible. Otherwise a VdtError is raised.

This can accept negative values.

```
>>> vtor.check('float', '2')
2.0
```

From now on we multiply the value to avoid comparing decimals

```
>>> vtor.check('float', '-6.8') * 10
-68.0
>>> vtor.check('float', '12.2') * 10
122.0
>>> vtor.check('float', 8.4) * 10
84.0
>>> vtor.check('float', 'a')
Traceback (most recent call last):
VdtTypeError: the value "a" is of the wrong type.
>>> vtor.check('float(10.1)', '10.2') * 10
102.0
>>> vtor.check('float(max=20.2)', '15.1') * 10
151.0
>>> vtor.check('float(10.0)', '9.0')
Traceback (most recent call last):
VdtValueTooSmallError: the value "9.0" is too small.
>>> vtor.check('float(max=20.0)', '35.0')
Traceback (most recent call last):
VdtValueTooBigError: the value "35.0" is too big.
```

IPython.external.validate.**is_float_list**()

Check that the value is a list of floats.

You can optionally specify the minimum and maximum number of members.

Each list member is checked that it is a float.

```
>>> vtor.check('float_list', ())
[]
>>> vtor.check('float_list', [])
[]
>>> vtor.check('float_list', (1, 2.0))
[1.0, 2.0]
>>> vtor.check('float_list', [1, 2.0])
[1.0, 2.0]
>>> vtor.check('float_list', [1, 'a'])
Traceback (most recent call last):
VdtTypeError: the value "a" is of the wrong type.
```

IPython.external.validate.**is_int_list**()

Check that the value is a list of integers.

You can optionally specify the minimum and maximum number of members.

Each list member is checked that it is an integer.

```
>>> vtor.check('int_list', ())
[]
>>> vtor.check('int_list', [])
[]
>>> vtor.check('int_list', (1, 2))
[1, 2]
>>> vtor.check('int_list', [1, 2])
[1, 2]
>>> vtor.check('int_list', [1, 'a'])
Traceback (most recent call last):
VdtTypeError: the value "a" is of the wrong type.
```

IPython.external.validate.**is_integer**()

A check that tests that a given value is an integer (int, or long) and optionally, between bounds. A negative value is accepted, while a float will fail.

If the value is a string, then the conversion is done - if possible. Otherwise a VdtError is raised.

```
>>> vtor.check('integer', '-1')
-1
>>> vtor.check('integer', '0')
0
>>> vtor.check('integer', 9)
9
>>> vtor.check('integer', 'a')
Traceback (most recent call last):
VdtTypeError: the value "a" is of the wrong type.
>>> vtor.check('integer', '2.2')
Traceback (most recent call last):
VdtTypeError: the value "2.2" is of the wrong type.
>>> vtor.check('integer(10)', '20')
20
>>> vtor.check('integer(max=20)', '15')
15
>>> vtor.check('integer(10)', '9')
Traceback (most recent call last):
VdtValueTooSmallError: the value "9" is too small.
>>> vtor.check('integer(10)', 9)
Traceback (most recent call last):
VdtValueTooSmallError: the value "9" is too small.
>>> vtor.check('integer(max=20)', '35')
Traceback (most recent call last):
VdtValueTooBigError: the value "35" is too big.
>>> vtor.check('integer(max=20)', 35)
Traceback (most recent call last):
VdtValueTooBigError: the value "35" is too big.
>>> vtor.check('integer(0, 9)', False)
0
```

IPython.external.validate.**is_ip_addr**()

Check that the supplied value is an Internet Protocol address, v.4, represented by a dotted-quad string, i.e. '1.2.3.4'.

```
>>> vtor.check('ip_addr', '1 ')
'1'
>>> vtor.check('ip_addr', ' 1.2')
'1.2'
>>> vtor.check('ip_addr', ' 1.2.3 ')
'1.2.3'
>>> vtor.check('ip_addr', '1.2.3.4')
'1.2.3.4'
>>> vtor.check('ip_addr', '0.0.0.0')
'0.0.0.0'
>>> vtor.check('ip_addr', '255.255.255.255')
'255.255.255.255'
>>> vtor.check('ip_addr', '255.255.255.256')
Traceback (most recent call last):
VdtValueError: the value "255.255.255.256" is unacceptable.
>>> vtor.check('ip_addr', '1.2.3.4.5')
Traceback (most recent call last):
VdtValueError: the value "1.2.3.4.5" is unacceptable.
>>> vtor.check('ip_addr', '1.2.3. 4')
Traceback (most recent call last):
VdtValueError: the value "1.2.3. 4" is unacceptable.
>>> vtor.check('ip_addr', 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
```

IPython.external.validate.**is_ip_addr_list**()

Check that the value is a list of IP addresses.

You can optionally specify the minimum and maximum number of members.

Each list member is checked that it is an IP address.

```
>>> vtor.check('ip_addr_list', ())
[]
>>> vtor.check('ip_addr_list', [])
[]
>>> vtor.check('ip_addr_list', ('1.2.3.4', '5.6.7.8'))
['1.2.3.4', '5.6.7.8']
>>> vtor.check('ip_addr_list', ['a'])
Traceback (most recent call last):
VdtValueError: the value "a" is unacceptable.
```

IPython.external.validate.**is_list**()

Check that the value is a list of values.

You can optionally specify the minimum and maximum number of members.

It does no check on list members.

```
>>> vtor.check('list', ())
[]
>>> vtor.check('list', [])
[]
>>> vtor.check('list', (1, 2))
[1, 2]
```

```
>>> vtor.check('list', [1, 2])
[1, 2]
>>> vtor.check('list(3)', (1, 2))
Traceback (most recent call last):
VdtValueTooShortError: the value "(1, 2)" is too short.
>>> vtor.check('list(max=5)', (1, 2, 3, 4, 5, 6))
Traceback (most recent call last):
VdtValueTooLongError: the value "(1, 2, 3, 4, 5, 6)" is too long.
>>> vtor.check('list(min=3, max=5)', (1, 2, 3, 4))
[1, 2, 3, 4]
>>> vtor.check('list', 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
>>> vtor.check('list', '12')
Traceback (most recent call last):
VdtTypeError: the value "12" is of the wrong type.
```

IPython.external.validate.**is_mixed_list**()

Check that the value is a list. Allow specifying the type of each member. Work on lists of specific lengths.

You specify each member as a positional argument specifying type

Each type should be one of the following strings : 'integer', 'float', 'ip_addr', 'string', 'boolean'

So you can specify a list of two strings, followed by two integers as :

mixed_list('string', 'string', 'integer', 'integer')

The length of the list must match the number of positional arguments you supply.

```
>>> mix_str = "mixed_list('integer', 'float', 'ip_addr', 'string', 'boolean')"
>>> check_res = vtor.check(mix_str, (1, 2.0, '1.2.3.4', 'a', True))
>>> check_res == [1, 2.0, '1.2.3.4', 'a', True]
1
>>> check_res = vtor.check(mix_str, ('1', '2.0', '1.2.3.4', 'a', 'True'))
>>> check_res == [1, 2.0, '1.2.3.4', 'a', True]
1
>>> vtor.check(mix_str, ('b', 2.0, '1.2.3.4', 'a', True))
Traceback (most recent call last):
VdtTypeError: the value "b" is of the wrong type.
>>> vtor.check(mix_str, (1, 2.0, '1.2.3.4', 'a'))
Traceback (most recent call last):
VdtValueTooShortError: the value "(1, 2.0, '1.2.3.4', 'a')" is too short.
>>> vtor.check(mix_str, (1, 2.0, '1.2.3.4', 'a', 1, 'b'))
Traceback (most recent call last):
VdtValueTooLongError: the value "(1, 2.0, '1.2.3.4', 'a', 1, 'b')" is too long.
>>> vtor.check(mix_str, 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
```

This test requires an elaborate setup, because of a change in error string output from the interpreter between Python 2.2 and 2.3 .

```
>>> res_seq = (
...     'passed an incorrect value "',
...     'yoda',
...     '" for parameter "mixed_list".',
... )
>>> if INTP_VER == (2, 2):
...     res_str = "".join(res_seq)
... else:
...     res_str = "'".join(res_seq)
>>> try:
...     vtor.check('mixed_list("yoda")', ('a'))
... except VdtParamError, err:
...     str(err) == res_str
1
```

IPython.external.validate.**is_option**()

This check matches the value to any of a set of options.

```
>>> vtor.check('option("yoda", "jedi")', 'yoda')
'yoda'
>>> vtor.check('option("yoda", "jedi")', 'jed')
Traceback (most recent call last):
VdtValueError: the value "jed" is unacceptable.
>>> vtor.check('option("yoda", "jedi")', 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
```

IPython.external.validate.**is_string**()

Check that the supplied value is a string.

You can optionally specify the minimum and maximum number of members.

```
>>> vtor.check('string', '0')
'0'
>>> vtor.check('string', 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
>>> vtor.check('string(2)', '12')
'12'
>>> vtor.check('string(2)', '1')
Traceback (most recent call last):
VdtValueTooShortError: the value "1" is too short.
>>> vtor.check('string(min=2, max=3)', '123')
'123'
>>> vtor.check('string(min=2, max=3)', '1234')
Traceback (most recent call last):
VdtValueTooLongError: the value "1234" is too long.
```

IPython.external.validate.**is_string_list**()

Check that the value is a list of strings.

You can optionally specify the minimum and maximum number of members.

Each list member is checked that it is a string.

```
>>> vtor.check('string_list', ())
[]
>>> vtor.check('string_list', [])
[]
>>> vtor.check('string_list', ('a', 'b'))
['a', 'b']
>>> vtor.check('string_list', ['a', 1])
Traceback (most recent call last):
VdtTypeError: the value "1" is of the wrong type.
>>> vtor.check('string_list', 'hello')
Traceback (most recent call last):
VdtTypeError: the value "hello" is of the wrong type.
```

IPython.external.validate.**is_tuple**()

Check that the value is a tuple of values.

You can optionally specify the minimum and maximum number of members.

It does no check on members.

```
>>> vtor.check('tuple', ())
()
>>> vtor.check('tuple', [])
()
>>> vtor.check('tuple', (1, 2))
(1, 2)
>>> vtor.check('tuple', [1, 2])
(1, 2)
>>> vtor.check('tuple(3)', (1, 2))
Traceback (most recent call last):
VdtValueTooShortError: the value "(1, 2)" is too short.
>>> vtor.check('tuple(max=5)', (1, 2, 3, 4, 5, 6))
Traceback (most recent call last):
VdtValueTooLongError: the value "(1, 2, 3, 4, 5, 6)" is too long.
>>> vtor.check('tuple(min=3, max=5)', (1, 2, 3, 4))
(1, 2, 3, 4)
>>> vtor.check('tuple', 0)
Traceback (most recent call last):
VdtTypeError: the value "0" is of the wrong type.
>>> vtor.check('tuple', '12')
Traceback (most recent call last):
VdtTypeError: the value "12" is of the wrong type.
```

IPython.external.validate.**numToDottedQuad**()

Convert long int to dotted quad string

```
>>> numToDottedQuad(-1L)
Traceback (most recent call last):
ValueError: Not a good numeric IP: -1
>>> numToDottedQuad(1L)
'0.0.0.1'
>>> numToDottedQuad(16777218L)
'1.0.0.2'
>>> numToDottedQuad(16908291L)
```

```
'1.2.0.3'
>>> numToDottedQuad(16909060L)
'1.2.3.4'
>>> numToDottedQuad(4294967295L)
'255.255.255.255'
>>> numToDottedQuad(4294967296L)
Traceback (most recent call last):
ValueError: Not a good numeric IP: 4294967296
```

10.33 frontend.asyncfrontendbase

10.33.1 Module: frontend.asyncfrontendbase

Inheritance diagram for IPython.frontend.asyncfrontendbase:

Base front end class for all async frontends.

10.33.2 AsyncFrontEndBase

class IPython.frontend.asyncfrontendbase.**AsyncFrontEndBase**(*engine=None*, *history=None*)

Bases: IPython.frontend.frontendbase.FrontEndBase

Overrides FrontEndBase to wrap execute in a deferred result. All callbacks are made as callbacks on the deferred result.

__init__()

execute()

Execute the block and return the deferred result.

Parameters: block : {str, AST} blockID : any

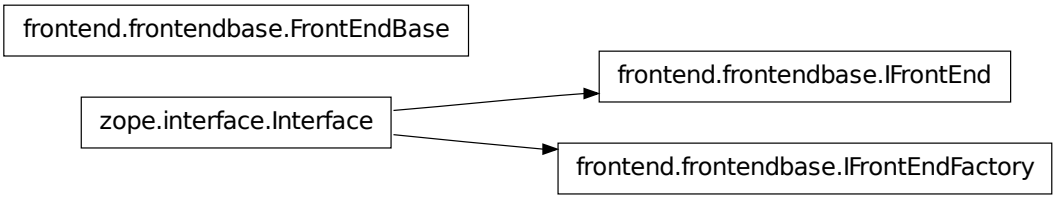
Caller may provide an ID to identify this block. result['blockID'] := blockID

Result: Deferred result of self.interpreter.execute

10.34 frontend.frontendbase

10.34.1 Module: `frontend.frontendbase`

Inheritance diagram for `IPython.frontend.frontendbase`:



frontendbase provides an interface and base class for GUI frontends for IPython.kernel/IPython.kernel.core.

Frontend implementations will likely want to subclass FrontEndBase.

Author: Barry Wark

10.34.2 Classes

FrontEndBase

class `IPython.frontend.frontendbase.`**FrontEndBase**(*shell=None*, *history=None*)

Bases: `object`

FrontEndBase manages the state tasks for a CLI frontend:

- Input and output history management
- Input/continuation and output prompt generation

Some issues (due to possibly unavailable engine):

- How do we get the current cell number for the engine?
- How do we handle completions?

__init__()

continuation_prompt()

Returns the current continuation prompt

execute()

Execute the block and return the result.

Parameters: block : {str, AST} blockID : any

Caller may provide an ID to identify this block. result['blockID'] := blockID

Result: Deferred result of self.interpreter.execute

get_history_next()
Returns next history string and increment history cursor.

get_history_previous()
Returns previous history string and decrement history cursor.

input_prompt()
Returns the current input prompt

It would be great to use ipython1.core.prompts.Prompt1 here

is_complete()
Determine if block is complete.

Parameters block : string

Result True if block can be sent to the engine without compile errors. False otherwise.

output_prompt()
Returns the output prompt for result

render_error()
Subclasses must override to render the failure.

In asynchronous frontends, this method will be called as a twisted.internet.defer.Deferred's callback. Implementations should thus return result when finished.

render_result()
Subclasses must override to render result.

In asynchronous frontends, this method will be called as a twisted.internet.defer.Deferred's callback. Implementations should thus return result when finished.

update_cell_prompt()
Subclass may override to update the input prompt for a block.

This method only really makes sens in asyncrhonous frontend. Since this method will be called as a twisted.internet.defer.Deferred's callback, implementations should return result when finished.

IFrontEnd

class IPython.frontend.frontendbase.**IFrontEnd**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Interface for frontends. All methods return t.i.d.Deferred

classmethod __init__()

IFrontEndFactory

class IPython.frontend.frontendbase.**IFrontEndFactory**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Factory interface for frontends.

classmethod **__init__**()

10.35 frontend.linefrontendbase

10.35.1 Module: frontend.linefrontendbase

Inheritance diagram for IPython.frontend.linefrontendbase:

Base front end class for all line-oriented frontends, rather than block-oriented.

Currently this focuses on synchronous frontends.

10.35.2 LineFrontEndBase

class IPython.frontend.linefrontendbase.**LineFrontEndBase**(*shell=None*, *history=None*, *banner=None*, **args*, ***kwargs*)

Bases: IPython.frontend.frontendbase.FrontEndBase

Concrete implementation of the FrontEndBase class. This is meant to be the base class behind all the frontend that are line-oriented, rather than block-oriented.

__init__()

after_execute()
All the operations required after an execution to put the terminal back in a shape where it is usable.

complete()
Complete line in engine's user_ns

Parameters **line** : string

Returns **The replacement for the line and the list of possible completions.** :

complete_current_input()
Do code completion on current line.

continuation_prompt()
Returns the current continuation prompt.

execute()
Stores the raw_string in the history, and sends the python string to the interpreter.

execute_command()
Execute a command, not only in the model, but also in the view, if any.

get_line_width()
Return the width of the line in characters.

is_complete()
Check if a string forms a complete, executable set of commands.

For the line-oriented frontend, multi-line code is not executed as soon as it is complete: the users has to enter two line returns.

new_prompt()
Prints a prompt and starts a new editing buffer.

Subclasses should use this method to make sure that the terminal is put in a state favorable for a new line input.

prefilter_input()
Prefilter the input to turn it in valid python.

render_error()
Frontend-specific rendering of error.

render_result()
Frontend-specific rendering of the result of a calculation that has been sent to an engine.

start()
Put the frontend in a state where it is ready for user interaction.

write()
Write some characters to the display.

Subclass should overide this method.

The refresh keyword argument is used in frontends with an event loop, to choose whether the write should trigget an UI refresh, and thus be syncrhonous, or not.

write_completion()
Write the list of possible completions.

new_line is the completed input line that should be displayed after the completion are writen. If None, the input_buffer before the completion is used.

IPython.frontend.linefrontendbase.**common_prefix**()
Given a list of strings, return the common prefix between all these strings.

10.36 frontend.prefilterfrontend

10.36.1 Module: frontend.prefilterfrontend

Inheritance diagram for IPython.frontend.prefilterfrontend:

Frontend class that uses IPython0 to prefilter the inputs.

Using the IPython0 mechanism gives us access to the magics.

This is a transitory class, used here to do the transition between ipython0 and ipython1. This class is meant to be short-lived as more functionnality is abstracted out of ipython0 in reusable functions and is added on the interpreter. This class can be a used to guide this refactoring.

10.36.2 PrefilterFrontEnd

class IPython.frontend.prefilterfrontend.**PrefilterFrontEnd**(*ipython0=None*, *argv=None*, **args*, ***kwargs*)

Bases: IPython.frontend.linefrontendbase.LineFrontEndBase

Class that uses ipython0 to do prefilter the input, do the completion and the magics.

The core trick is to use an ipython0 instance to prefilter the input, and share the namespace between the interpreter instance used to execute the statements and the ipython0 used for code completion...

__init__()

Parameters **ipython0: an optional ipython0 instance to use for command** :

prefiltering and completion. :

argv : list, optional

Used as the instance's argv value. If not given, [] is used.

capture_output()

Capture all the output mechanisms we can think of.

complete()

do_exit()

Exit the shell, cleanup and save the history.

execute()

prefilter_input()
Using IPython0 to prefilter the commands to turn them in executable statements that are valid Python strings.

release_output()
Release all the different captures we have made.

save_output_hooks()
Store all the output hooks we can think of, to be able to restore them.

We need to do this early, as starting the ipython0 instance will screw ouput hooks.

show_traceback()
Use ipython0 to capture the last traceback and display it.

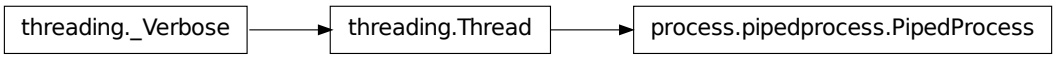
system_call()
Allows for frontend to define their own system call, to be able capture output and redirect input.

IPython.frontend.prefilterfrontend.**mk_system_call**()
given a os.system replacement, and a leading string command, returns a function that will execute the command with the given argument string.

10.37 frontend.process.pipedprocess

10.37.1 Module: frontend.process.pipedprocess

Inheritance diagram for IPython.frontend.process.pipedprocess:



Object for encapsulating process execution by using callbacks for stdout, stderr and stdin.

10.37.2 PipedProcess

class IPython.frontend.process.pipedprocess.**PipedProcess**(*command_string*, *out_callback*, *end_callback=None*)
Bases: threading.Thread

Class that encapsulates process execution by using callbacks for stdout, stderr and stdin, and providing a reliable way of killing it.

__init__()
command_string: the command line executed to start the process.

out_callback: the python callable called on stdout/stderr.

end_callback: an optional callable called when the process finishes.

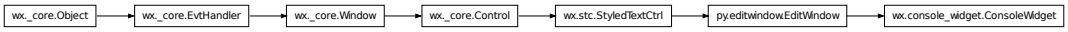
These callbacks are called from a different thread as the thread from which is started.

run()
Start the process and hook up the callbacks.

10.38 frontend.wx.console_widget

10.38.1 Module: frontend.wx.console_widget

Inheritance diagram for IPython.frontend.wx.console_widget:



A Wx widget to act as a console and input commands.

This widget deals with prompts and provides an edit buffer restricted to after the last prompt.

10.38.2 ConsoleWidget

class IPython.frontend.wx.console_widget.**ConsoleWidget**(*parent*, *id=-1*, *pos=wx.Point(-1, -1)*, *size=wx.Size(-1, -1)*, *style=262144*)

Bases: wx.py.editwindow.EditWindow

Specialized styled text control view for console-like workflow.

This widget is mainly interested in dealing with the prompt and keeping the cursor inside the editing line.

__init__()

OnUpdateUI()
Override the OnUpdateUI of the EditWindow class, to prevent syntax highlighting both for faster redraw, and for more consistent look and feel.

configure_scintilla()
Set up all the styling option of the embedded scintilla widget.

continuation_prompt()
Returns the current continuation prompt. We need to implement this method here to deal with the ascii escape sequences cleaning up.

get_line_width()
Return the width of the line in characters.

input_buffer
Returns the text in current edit buffer.

new_prompt()
Prints a prompt at start of line, and move the start of the current block there.

The prompt can be given with ascii escape sequences.

pop_completion()
Pops up an autocompletion menu. Offset is the offset in characters of the position at which the menu should appear, relativ to the cursor.

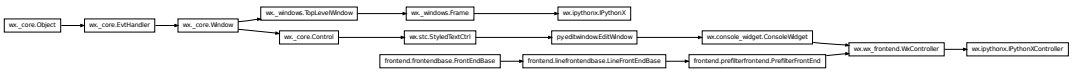
scroll_to_bottom()

write()
Write given text to buffer, while translating the ansi escape sequences.

10.39 frontend.wx.ipythonx

10.39.1 Module: frontend.wx.ipythonx

Inheritance diagram for IPython.frontend.wx.ipythonx:



Entry point for a simple application giving a graphical frontend to ipython.

10.39.2 Classes

IPythonX

class IPython.frontend.wx.ipythonx.**IPythonX**(*parent*, *id*, *title*, *debug=False*)
Bases: wx._windows.Frame

Main frame of the IPythonX app.

__init__()

on_close()
Called on closing the windows.

Stops the event loop, to close all the child windows.

IPythonXController

class IPython.frontend.wx.ipythonx.**IPythonXController**(**args*, ***kwargs*)
Bases: IPython.frontend.wx.wx_frontend.WxController

Sub class of WxController that adds some application-specific bindings.

__init__()

ask_exit()
Ask the user whether to exit.

do_exit()
Exits the interpreter, kills the windows.

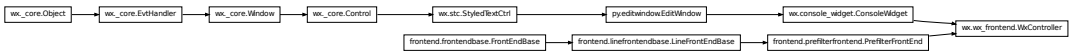
10.39.3 Function

IPython.frontend.wx.ipythonx.**main**()

10.40 frontend.wx.wx_frontend

10.40.1 Module: frontend.wx.wx_frontend

Inheritance diagram for IPython.frontend.wx.wx_frontend:



Classes to provide a Wx frontend to the IPython.kernel.core.interpreter.

This class inherits from ConsoleWidget, that provides a console-like widget to provide a text-rendering widget suitable for a terminal.

10.40.2 WxController

class IPython.frontend.wx.wx_frontend.**WxController**(*parent*, *id=-1*, *pos=wx.Point(-1, -1)*, *size=wx.Size(-1, -1)*, *style=4456448*, *styledef=None*, **args*, ***kwds*)
Bases: IPython.frontend.wx.console_widget.ConsoleWidget, IPython.frontend.prefilterfrontend.PrefilterFrontEnd

Classes to provide a Wx frontend to the IPython.kernel.core.interpreter.

This class inherits from ConsoleWidget, that provides a console-like widget to provide a text-rendering widget suitable for a terminal.

__init__()
Create Shell instance.

Parameters **styledef** : dict, optional

styledef is the dictionary of options used to define the style.

OnUpdateUI()
Override the OnUpdateUI of the EditWindow class, to prevent syntax highlighting both for faster redraw, and for more consistent look and feel.

after_execute()

buffered_write()
A write method for streams, that caches the stream in order to avoid flooding the event loop.

This can be called outside of the main loop, in separate threads.

capture_output()

clear_screen()
Empty completely the widget.

continuation_prompt()

do_calltip()
Analyse current and displays useful calltip for it.

execute()

execute_command()
Execute a command, not only in the model, but also in the view.

input_buffer
Returns the text in current edit buffer.

new_prompt()
Display a new prompt, and start a new input buffer.

raw_input()
A replacement from python's raw_input.

release_output()

render_error()

save_output_hooks()

show_traceback()

system_call()

title

write()

10.41 generics

10.41.1 Module: generics

'Generic' functions for extending IPython.

See http://cheeseshop.python.org/pypi/simplegeneric.

Here is an example from genutils.py:

> **def print_lsstring(arg):** "Prettier (non-repr-like) and more informative printer for LSString"
> print "LSString (.p, .n, .l, .s available). Value:" print arg
>
> print_lsstring = result_display.when_type(LSString)(print_lsstring)

(Yes, the nasty syntax is for python 2.3 compatibility. Your own extensions can use the niftier decorator syntax introduced in Python 2.4).

10.41.2 Functions

`IPython.generics.`**`complete_object`**`()`
Custom completer dispatching for python objects

obj is the object itself. prev_completions is the list of attributes discovered so far.

This should return the list of attributes in obj. If you only wish to add to the attributes already discovered normally, return own_attrs + prev_completions.

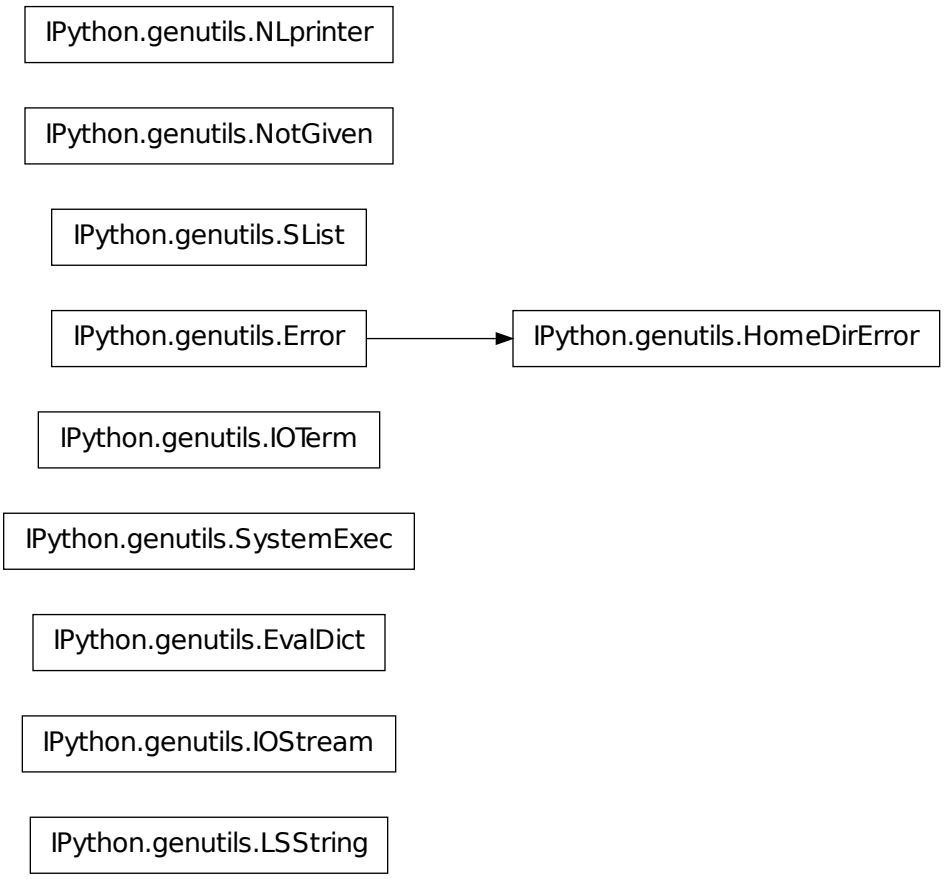
`IPython.generics.`**`inspect_object`**`()`
Called when you do obj?

`IPython.generics.`**`result_display`**`()`
print the result of computation

10.42 genutils

10.42.1 Module: genutils

Inheritance diagram for `IPython.genutils`:



General purpose utilities.

This is a grab-bag of stuff I find useful in most programs I write. Some of these things are also convenient when working at the command line.

10.42.2 Classes

Error

class IPython.genutils.**Error**
Bases: exceptions.Exception

Base class for exceptions in this module.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

EvalDict

class IPython.genutils.**EvalDict**
Emulate a dict which evaluates its contents in the caller's frame.

Usage: >>> number = 19

```
>>> text = "python"

>>> print "%(text.capitalize())s %(number/9.0).1f rules!" % EvalDict()
Python 2.1 rules!
```

HomeDirError

class IPython.genutils.**HomeDirError**
Bases: IPython.genutils.Error

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

IOStream

class IPython.genutils.**IOStream**(*stream*, *fallback*)

__init__()

close()

write()

IOTerm

class IPython.genutils.**IOTerm**(*cin=None*, *cout=None*, *cerr=None*)
Term holds the file or file-like objects for handling I/O operations.

These are normally just sys.stdin, sys.stdout and sys.stderr but for Windows they can can replaced to allow editing the strings before they are displayed.

__init__()

LSString

class IPython.genutils.**LSString**
Bases: str

String derivative with a special access attributes.

These are normal strings, but with the special attributes:

.l (or .list) : value as list (split on newlines). .n (or .nlstr): original value (the string itself).
.s (or .spstr): value as whitespace-separated string. .p (or .paths): list of path objects

Any values which require transformations are computed only once and cached.

Such strings are very useful to efficiently interact with the shell, which typically only understands whitespace-separated options for commands.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

get_list()

get_nlstr()

get_paths()

get_spstr()

l

list

n

nlstr

p

paths

s

spstr

NLprinter

class IPython.genutils.**NLprinter**
Print an arbitrarily nested list, indicating index numbers.

An instance of this class called nlprint is available and callable as a function.

nlprint(list,indent=' ',sep=': ') -> prints indenting each level by 'indent' and using 'sep' to separate the index from the value.

__init__()

NotGiven

class IPython.genutils.**NotGiven**

SList

class IPython.genutils.**SList**

Bases: list

List derivative with a special access attributes.

These are normal lists, but with the special attributes:

.l (or .list) : value as list (the list itself). .n (or .nlstr): value as a string, joined on newlines. .s (or .spstr): value as a string, joined on spaces. .p (or .paths): list of path objects

Any values which require transformations are computed only once and cached.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

fields()

Collect whitespace-separated fields from string list

Allows quick awk-like usage of string lists.

Example data (in var a, created by 'a = !ls -l')::

-rwxrwxrwx 1 ville None 18 Dec 14 2006 ChangeLog

drwxrwxrwx+ 6 ville None 0 Oct 24 18:05 IPython

a.fields(0) is ['-rwxrwxrwx', 'drwxrwxrwx+'] a.fields(1,0) is ['1 -rwxrwxrwx', '6 drwxrwxrwx+'] (note the joining by space). a.fields(-1) is ['ChangeLog', 'IPython']

IndexErrors are ignored.

Without args, fields() just split()'s the strings.

get_list()

get_nlstr()

get_paths()

get_spstr()

grep()

Return all strings matching 'pattern' (a regex or callable)

This is case-insensitive. If prune is true, return all items NOT matching the pattern.

If field is specified, the match must occur in the specified whitespace-separated field.

Examples:

```
a.grep( lambda x: x.startswith('C') )
a.grep('Cha.*log', prune=1)
a.grep('chm', field=-1)
```

l

list

n

nlstr

p

paths

s

sort()
sort by specified fields (see fields())

Example:: a.sort(1, nums = True)

Sorts a by second field, in numerical order (so that 21 > 3)

spstr

SystemExec

class IPython.genutils.**SystemExec**(*verbose=0*, *debug=0*, *header=''*, *split=0*)
Access the system and getoutput functions through a stateful interface.

Note: here we refer to the system and getoutput functions from this library, not the ones from the standard python library.

This class offers the system and getoutput functions as methods, but the verbose, debug and header parameters can be set for the instance (at creation time or later) so that they don't need to be specified on each call.

For efficiency reasons, there's no way to override the parameters on a per-call basis other than by setting instance attributes. If you need local overrides, it's best to directly call system() or getoutput().

The following names are provided as alternate options:

- xsys: alias to system
- bq: alias to getoutput

An instance can then be created as: >>> sysexec = SystemExec(verbose=1,debug=0,header='Calling: ')

__init__()
Specify the instance's values for verbose, debug and header.

bq()
Stateful interface to getoutput().

getoutput()
Stateful interface to getoutput().

getoutputerror()
Stateful interface to getoutputerror().

shell()
Stateful interface to shell(), with the same keyword parameters.

system()
Stateful interface to system(), with the same keyword parameters.

xsys()
Stateful interface to system(), with the same keyword parameters.

10.42.3 Functions

IPython.genutils.**abbrev_cwd**()
Return abbreviated version of cwd, e.g. d:mydir

IPython.genutils.**all_belong**()
Check whether a list of items ALL appear in a given list of options.

Returns a single 1 or 0 value.

IPython.genutils.**arg_split**()
Split a command line's arguments in a shell-like manner.

This is a modified version of the standard library's shlex.split() function, but with a default of posix=False for splitting, so that quotes in inputs are respected.

IPython.genutils.**ask_yes_no**()
Asks a question and returns a boolean (y/n) answer.

If default is given (one of 'y','n'), it is used if the user input is empty. Otherwise the question is repeated until an answer is given.

An EOF is treated as the default answer. If there is no default, an exception is raised to prevent infinite loops.

Valid answers are: y/yes/n/no (match is not case sensitive).

IPython.genutils.**belong**()
Check whether a list of items appear in a given list of options.

Returns a list of 1 and 0, one for each candidate given.

IPython.genutils.**chop**()
Chop a sequence into chunks of the given size.

IPython.genutils.**debugx**()
Print the value of an expression from the caller's frame.

Takes an expression, evaluates it in the caller's frame and prints both the given expression and the resulting value (as well as a debug mark indicating the name of the calling function. The input must be of a form suitable for eval().

An optional message can be passed, which will be prepended to the printed expr->value pair.

IPython.genutils.**dgrep**()
Return grep() on dir()+dir(__builtins__).

A very common use of grep() when working interactively.

IPython.genutils.**dhook_wrap**()
Wrap a function call in a sys.displayhook controller.

Returns a wrapper around func which calls func, with all its arguments and keywords unmodified, using the default sys.displayhook. Since IPython modifies sys.displayhook, it breaks the behavior of certain systems that rely on the default behavior, notably doctest.

IPython.genutils.**dir2**()
dir2(obj) -> list of strings

Extended version of the Python builtin dir(), which does a few extra checks, and supports common objects with unusual internals that confuse dir(), such as Traits and PyCrust.

This version is guaranteed to return only a list of true strings, whereas dir() returns anything that objects inject into themselves, even if they are later not really valid for attribute access (many extension libraries have such bugs).

IPython.genutils.**doctest_reload**()
Properly reload doctest to reuse it interactively.

This routine:

•imports doctest but does NOT reload it (see below).

•resets its global 'master' attribute to None, so that multiple uses of

the module interactively don't produce cumulative reports.

•Monkeypatches its core test runner method to protect it from IPython's

modified displayhook. Doctest expects the default displayhook behavior deep down, so our modification breaks it completely. For this reason, a hard monkeypatch seems like a reasonable solution rather than asking users to manually use a different doctest runner when under IPython.

Notes

This function *used to* reload doctest, but this has been disabled because reloading doctest unconditionally can cause massive breakage of other doctest-dependent modules already in memory, such as those for IPython's own testing system. The name wasn't changed to avoid breaking people's code, but the reload call isn't actually made anymore.

IPython.genutils.**error**()
Equivalent to warn(msg,level=3).

IPython.genutils.**esc_quotes**()
Return the input string with single and double quotes escaped out

IPython.genutils.**fatal**()
Equivalent to warn(msg,exit_val=exit_val,level=4).

IPython.genutils.**file_read**()
Read a file and close it. Returns the file source.

IPython.genutils.**file_readlines**()
Read a file and close it. Returns the file source using readlines().

IPython.genutils.**filefind**()
Return the given filename either in the current directory, if it exists, or in a specified list of directories.

~ expansion is done on all file and directory names.

Upon an unsuccessful search, raise an IOError exception.

IPython.genutils.**flag_calls**()
Wrap a function to detect and flag when it gets called.

This is a decorator which takes a function and wraps it in a function with a 'called' attribute. wrapper.called is initialized to False.

The wrapper.called attribute is set to False right before each call to the wrapped function, so if the call fails it remains False. After the call completes, wrapper.called is set to True and the output is returned.

Testing for truth in wrapper.called allows you to determine if a call to func() was attempted and succeeded.

IPython.genutils.**flatten**()
Flatten a list of lists (NOT recursive, only works for 2d lists).

IPython.genutils.**get_class_members**()

IPython.genutils.**get_home_dir**()
Return the closest possible equivalent to a 'home' directory.

We first try $HOME. Absent that, on NT it's $HOMEDRIVE$HOMEPATH.

Currently only Posix and NT are implemented, a HomeDirError exception is raised for all other OSes.

IPython.genutils.**get_ipython_dir**()
Get the IPython directory for this platform and user.

This uses the logic in *get_home_dir* to find the home directory and the adds either .ipython or _ipython to the end of the path.

IPython.genutils.**get_log_dir**()
Get the IPython log directory.

If the log directory does not exist, it is created.

IPython.genutils.**get_pager_cmd**()
Return a pager command.

Makes some attempts at finding an OS-correct one.

IPython.genutils.**get_pager_start**()
Return the string for paging files with an offset.

This is the '+N' argument which less and more (under Unix) accept.

IPython.genutils.**get_py_filename**()
Return a valid python filename in the current directory.

If the given name is not a file, it adds '.py' and searches again. Raises IOError with an informative message if the file isn't found.

IPython.genutils.**get_security_dir**()
Get the IPython security directory.

This directory is the default location for all security related files, including SSL/TLS certificates and FURL files.

If the directory does not exist, it is created with 0700 permissions. If it exists, permissions are set to 0700.

IPython.genutils.**get_slice**()
Get a slice of a sequence with variable step. Specify start,stop,step.

IPython.genutils.**getattr_list**()
getattr_list(obj,alist[, default]) -> attribute list.

Get a list of named attributes for an object. When a default argument is given, it is returned when the attribute doesn't exist; without it, an exception is raised in that case.

Note that alist can be given as a string, which will be automatically split into a list on whitespace. If given as a list, it must be a list of *strings* (the variable names themselves), not of variables.

IPython.genutils.**getoutput**()
Dummy substitute for perl's backquotes.

Executes a command and returns the output.

Accepts the same arguments as system(), plus:

•split(0): if true, the output is returned as a list split on newlines.

Note: a stateful version of this function is available through the SystemExec class.

This is pretty much deprecated and rarely used, genutils.getoutputerror may be what you need.

IPython.genutils.**getoutputerror**()
Return (standard output,standard error) of executing cmd in a shell.

Accepts the same arguments as system(), plus:

•split(0): if true, each of stdout/err is returned as a list split on

newlines.

Note: a stateful version of this function is available through the SystemExec class.

IPython.genutils.**grep**()
Simple minded grep-like function. grep(pat,list) returns occurrences of pat in list, None on failure.

It only does simple string matching, with no support for regexps. Use the option case=0 for case-insensitive matching.

IPython.genutils.**idgrep**()
Case-insensitive dgrep()

IPython.genutils.**igrep**()
Synonym for case-insensitive grep.

IPython.genutils.**import_fail_info**()
Inform load failure for a module.

IPython.genutils.**indent**()
Indent a string a given number of spaces or tabstops.

indent(str,nspaces=4,ntabs=0) -> indent str by ntabs+nspaces.

IPython.genutils.**info**()
Equivalent to warn(msg,level=1).

IPython.genutils.**list2dict**()
Takes a list of (key,value) pairs and turns it into a dict.

IPython.genutils.**list2dict2**()
Takes a list and turns it into a dict. Much slower than list2dict, but more versatile. This version can take lists with sublists of arbitrary length (including sclars).

IPython.genutils.**list_strings**()
Always return a list of strings, given a string or list of strings as input.

IPython.genutils.**make_quoted_expr**()
Return string s in appropriate quotes, using raw string if possible.

XXX - example removed because it caused encoding errors in documentation generation. We need a new example that doesn't contain invalid chars.

Note the use of raw string and padding at the end to allow trailing backslash.

IPython.genutils.**map_method**()
map_method(method,object_list,*args,**kw) -> list

Return a list of the results of applying the methods to the items of the argument sequence(s). If more than one sequence is given, the method is called with an argument list consisting of the corresponding item of each sequence. All sequences must be of the same length.

Keyword arguments are passed verbatim to all objects called.

This is Python code, so it's not nearly as fast as the builtin map().

IPython.genutils.**marquee**()
Return the input string centered in a 'marquee'.

IPython.genutils.**mutex_opts**()
Check for presence of mutually exclusive keys in a dict.

Call: mutex_opts(dict,[[op1a,op1b],[op2a,op2b]...]

IPython.genutils.**native_line_ends**()
Convert (in-place) a file to line-ends native to the current OS.

If the optional backup argument is given as false, no backup of the original file is left.

IPython.genutils.**num_cpus**()
Return the effective number of CPUs in the system as an integer.

This cross-platform function makes an attempt at finding the total number of available CPUs in the system, as returned by various underlying system and python calls.

If it can't find a sensible answer, it returns 1 (though an error *may* make it return a large positive number that's actually incorrect).

IPython.genutils.**optstr2types**()
Convert a string of option names to a dict of type mappings.

optstr2types(str) -> {None:'string_opts',int:'int_opts',float:'float_opts'}

This is used to get the types of all the options in a string formatted with the conventions of DPyGetOpt. The 'type' None is used for options which are strings (they need no further conversion). This function's main use is to get a typemap for use with read_dict().

IPython.genutils.**page**()
Print a string, piping through a pager after a certain length.

The screen_lines parameter specifies the number of *usable* lines of your terminal screen (total lines minus lines you need to reserve to show other information).

If you set screen_lines to a number <=0, page() will try to auto-determine your screen size and will only use up to (screen_size+screen_lines) for printing, paging after that. That is, if you want auto-detection but need to reserve the bottom 3 lines of the screen, use screen_lines = -3, and for auto-detection without any lines reserved simply use screen_lines = 0.

If a string won't fit in the allowed lines, it is sent through the specified pager command. If none given, look for PAGER in the environment, and ultimately default to less.

If no system pager works, the string is sent through a 'dumb pager' written in python, very simplistic.

IPython.genutils.**page_dumb**()
Very dumb 'pager' in Python, for when nothing else works.

Only moves forward, same interface as page(), except for pager_cmd and mode.

IPython.genutils.**page_file**()
Page a file, using an optional pager command and starting line.

IPython.genutils.**popkey**()
Return dct[key] and delete dct[key].

If default is given, return it if dct[key] doesn't exist, otherwise raise KeyError.

IPython.genutils.**print_lsstring**()
Prettier (non-repr-like) and more informative printer for LSString

IPython.genutils.**print_slist**()
Prettier (non-repr-like) and more informative printer for SList

IPython.genutils.**process_cmdline**()
Process command-line options and arguments.

Arguments:

- argv: list of arguments, typically sys.argv.
- names: list of option names. See DPyGetOpt docs for details on options

syntax.

•defaults: dict of default values.

•usage: optional usage notice to print if a wrong argument is passed.

Return a dict of options and a list of free arguments.

IPython.genutils.**qw**()

Similar to Perl's qw() operator, but with some more options.

qw(words,flat=0,sep=' ',maxsplit=-1) -> words.split(sep,maxsplit)

words can also be a list itself, and with flat=1, the output will be recursively flattened.

Examples:

```
>>> qw('1 2')
['1', '2']

>>> qw(['a b','1 2',['m n','p q']])
[['a', 'b'], ['1', '2'], [['m', 'n'], ['p', 'q']]]

>>> qw(['a b','1 2',['m n','p q']],flat=1)
['a', 'b', '1', '2', 'm', 'n', 'p', 'q']
```

IPython.genutils.**qw_lol**()

qw_lol('a b') -> [['a','b']], otherwise it's just a call to qw().

We need this to make sure the modules_some keys *always* end up as a list of lists.

IPython.genutils.**qwflat**()

Calls qw(words) in flat mode. It's just a convenient shorthand.

IPython.genutils.**raw_input_ext**()

Similar to raw_input(), but accepts extended lines if input ends with .

IPython.genutils.**raw_input_multi**()

Take multiple lines of input.

A list with each line of input as a separate element is returned when a termination string is entered (defaults to a single '.'). Input can also terminate via EOF (^D in Unix, ^Z-RET in Windows).

Lines of input which end in are joined into single entries (and a secondary continuation prompt is issued as long as the user terminates lines with). This allows entering very long strings which are still meant to be treated as single entities.

IPython.genutils.**read_dict**()

Read a dictionary of key=value pairs from an input file, optionally performing conversions on the resulting values.

read_dict(filename,type_conv,**opt) -> dict

Only one value per line is accepted, the format should be # optional comments are ignored key valuen

Args:

•type_conv: A dictionary specifying which keys need to be converted to

which types. By default all keys are read as strings. This dictionary should have as its keys valid conversion functions for strings (int,long,float,complex, or your own). The value for each key (converter) should be a whitespace separated string containing the names of all the entries in the file to be converted using that function. For keys to be left alone, use None as the conversion function (only needed with purge=1, see below).

•opt: dictionary with extra options as below (default in parens)

purge(0): if set to 1, all keys *not* listed in type_conv are purged out of the dictionary to be returned. If purge is going to be used, the set of keys to be left as strings also has to be explicitly specified using the (non-existent) conversion function None.

fs(None): field separator. This is the key/value separator to be used when parsing the file. The None default means any whitespace [behavior of string.split()].

strip(0): if 1, strip string values of leading/trailinig whitespace.

warn(1): warning level if requested keys are not found in file.

- 0: silently ignore.
- 1: inform but proceed.
- 2: raise KeyError exception.

no_empty(0): if 1, remove keys with whitespace strings as a value.

unique([]): list of keys (or space separated string) which can't be repeated. If one such key is found in the file, each new instance overwrites the previous one. For keys not listed here, the behavior is to make a list of all appearances.

Example:

If the input file test.ini contains (we put it in a string to keep the test self-contained):

```
>>> test_ini = '''\
... i 3
... x 4.5
... y 5.5
... s hi ho'''
```

Then we can use it as follows: >>> type_conv={int:'i',float:'x',None:'s'}

```
>>> d = read_dict(test_ini)

>>> sorted(d.items())
[('i', '3'), ('s', 'hi ho'), ('x', '4.5'), ('y', '5.5')]

>>> d = read_dict(test_ini,type_conv)

>>> sorted(d.items())
[('i', 3), ('s', 'hi ho'), ('x', 4.5), ('y', '5.5')]

>>> d = read_dict(test_ini,type_conv,purge=True)
```

```
>>> sorted(d.items())
[('i', 3), ('s', 'hi ho'), ('x', 4.5)]
```

IPython.genutils.**setattr_list**()
Set a list of attributes for an object taken from a namespace.

setattr_list(obj,alist,nspace) -> sets in obj all the attributes listed in alist with their values taken from nspace, which must be a dict (something like locals() will often do) If nspace isn't given, locals() of the *caller* is used, so in most cases you can omit it.

Note that alist can be given as a string, which will be automatically split into a list on whitespace. If given as a list, it must be a list of *strings* (the variable names themselves), not of variables.

IPython.genutils.**shell**()
Execute a command in the system shell, always return None.

Options:

•verbose (0): print the command to be executed.

•debug (0): only print, do not actually execute.

•header (''): Header to print on screen prior to the executed command (it

is only prepended to the command, no newlines are added).

Note: this is similar to genutils.system(), but it returns None so it can be conveniently used in interactive loops without getting the return value (typically 0) printed many times.

IPython.genutils.**snip_print**()
Print a string snipping the midsection to fit in width.

print_full: mode control:

- 0: only snip long strings
- 1: send to page() directly.
- 2: snip long strings and ask for full length viewing with page()

Return 1 if snipping was necessary, 0 otherwise.

IPython.genutils.**sort_compare**()
Sort and compare two lists.

By default it does it in place, thus modifying the lists. Use inplace = 0 to avoid that (at the cost of temporary copy creation).

IPython.genutils.**system**()
Execute a system command, return its exit status.

Options:

•verbose (0): print the command to be executed.

•debug (0): only print, do not actually execute.

•header (''): Header to print on screen prior to the executed command (it

is only prepended to the command, no newlines are added).

Note: a stateful version of this function is available through the SystemExec class.

IPython.genutils.**target_outdated**()
Determine whether a target is out of date.

target_outdated(target,deps) -> 1/0

deps: list of filenames which MUST exist. target: single filename which may or may not exist.

If target doesn't exist or is older than any file listed in deps, return true, otherwise return false.

IPython.genutils.**target_update**()
Update a target with a given command given a list of dependencies.

target_update(target,deps,cmd) -> runs cmd if target is outdated.

This is just a wrapper around target_outdated() which calls the given command if target is outdated.

IPython.genutils.**timing**()
timing(func,*args,**kw) -> t_total

Execute a function once, return the elapsed total CPU time in seconds. This is just the first value in timings_out().

IPython.genutils.**timings**()
Execute a function reps times, return a tuple with the elapsed total CPU time in seconds and the time per call. These are just the first two values in timings_out().

IPython.genutils.**timings_out**()
Execute a function reps times, return a tuple with the elapsed total CPU time in seconds, the time per call and the function's output.

Under Unix, the return value is the sum of user+system time consumed by the process, computed via the resource module. This prevents problems related to the wraparound effect which the time.clock() function has.

Under Windows the return value is in wall clock seconds. See the documentation for the time module for more details.

IPython.genutils.**uniq_stable**()
uniq_stable(elems) -> list

Return from an iterable, a list of all the unique elements in the input, but maintaining the order in which they first appear.

A naive solution to this problem which just makes a dictionary with the elements as keys fails to respect the stability condition, since dictionaries are unsorted by nature.

Note: All elements in the input must be valid dictionary keys for this routine to work, as it internally uses a dictionary for efficiency reasons.

IPython.genutils.**unquote_ends**()
Remove a single pair of quotes from the endpoints of a string.

IPython.genutils.**warn**()
Standard warning printer. Gives formatting consistency.

Output is sent to Term.cerr (sys.stderr by default).

Options:

-level(2): allows finer control: 0 -> Do nothing, dummy function. 1 -> Print message. 2 -> Print 'WARNING:' + message. (Default level). 3 -> Print 'ERROR:' + message. 4 -> Print 'FATAL ERROR:' + message and trigger a sys.exit(exit_val).

-exit_val (1): exit value returned by sys.exit() for a level 4 warning. Ignored for all other levels.

`IPython.genutils.`**`with_obj`**`()`

Set multiple attributes for an object, similar to Pascal's with.

Example: with_obj(jim,

born = 1960, haircolour = 'Brown', eyecolour = 'Green')

Credit: Greg Ewing, in http://mail.python.org/pipermail/python-list/2001-May/040703.html.

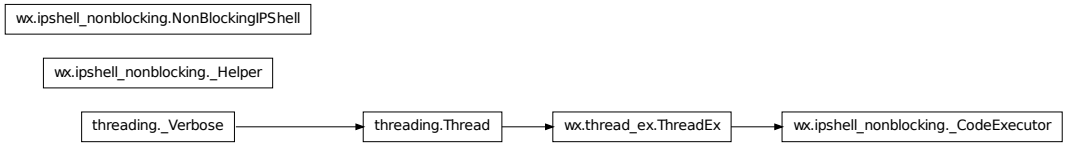
NOTE: up until IPython 0.7.2, this was called simply 'with', but 'with' has become a keyword for Python 2.5, so we had to rename it.

`IPython.genutils.`**`wrap_deprecated`**`()`

10.43 gui.wx.ipshell_nonblocking

10.43.1 Module: `gui.wx.ipshell_nonblocking`

Inheritance diagram for `IPython.gui.wx.ipshell_nonblocking`:



Provides IPython remote instance.

@author: Laurent Dufrechou laurent.dufrechou _at_ gmail.com @license: BSD

All rights reserved. This program and the accompanying materials are made available under the terms of the BSD which accompanies this distribution, and is available at U{http://www.opensource.org/licenses/bsd-license.php}

10.43.2 NonBlockingIPShell

class IPython.gui.wx.ipshell_nonblocking.**NonBlockingIPShell**(*argv=[]*, *user_ns={}*, *user_global_ns=None*, *cin=None*, *cout=None*, *cerr=None*, *ask_exit_handler=None*)

Bases: object

Create an IPython instance, running the commands in a separate, non-blocking thread. This allows embedding in any GUI without blockage.

Note: The ThreadEx class supports asynchroneous function call via raise_exc()

__init__()

@param argv: Command line options for IPython @type argv: list @param user_ns: User namespace. @type user_ns: dictionary @param user_global_ns: User global namespace. @type user_global_ns: dictionary. @param cin: Console standard input. @type cin: IO stream @param cout: Console standard output. @type cout: IO stream @param cerr: Console standard error. @type cerr: IO stream @param exit_handler: Replacement for builtin exit() function @type exit_handler: function @param time_loop: Define the sleep time between two thread's loop @type int

complete()

Returns an auto completed line and/or posibilities for completion.

@param line: Given line so far. @type line: string

@return: Line completed as for as possible, and possible further completions. @rtype: tuple

do_execute()

Tell the thread to process the 'line' command

get_banner()

Returns the IPython banner for useful info on IPython instance

@return: The banner string. @rtype: string

get_doc_text()

Returns the output of the processing that need to be paged (if any)

@return: The std output string. @rtype: string

get_help_text()

Returns the output of the processing that need to be paged via help pager(if any)

@return: The std output string. @rtype: string

get_indentation()

Returns the current indentation level Usefull to put the caret at the good start position if we want to do autoindentation.

@return: The indentation level. @rtype: int

get_prompt()
Returns current prompt inside IPython instance (Can be In [...]: ot ...:)

@return: The current prompt. @rtype: string

get_prompt_count()
Returns the prompt number. Each time a user execute a line in the IPython shell the prompt count is increased

@return: The prompt number @rtype: int

get_threading()
Returns threading status, is set to True, then each command sent to the interpreter will be executed in a separated thread allowing, for example, breaking a long running commands. Disallowing it, permits better compatibilty with instance that is embedding IPython instance.

@return: Execution method @rtype: bool

history_back()
Provides one history command back.

@return: The command string. @rtype: string

history_forward()
Provides one history command forward.

@return: The command string. @rtype: string

init_history_index()
set history to last command entered

init_ipython0()
Initialize an ipython0 instance

set_threading()
Sets threading state, if set to True, then each command sent to the interpreter will be executed in a separated thread allowing, for example, breaking a long running commands. Disallowing it, permits better compatibilty with instance that is embedding IPython instance.

@param state: Sets threading state @type bool

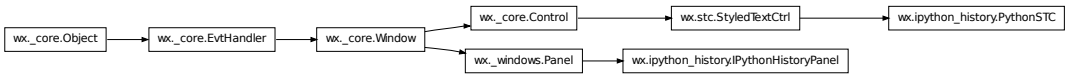
update_namespace()
Add the current dictionary to the shell namespace.

@param ns_dict: A dictionary of symbol-values. @type ns_dict: dictionary

10.44 gui.wx.ipython_history

10.44.1 Module: gui.wx.ipython_history

Inheritance diagram for IPython.gui.wx.ipython_history:



10.44.2 Classes

IPythonHistoryPanel

class IPython.gui.wx.ipython_history.**IPythonHistoryPanel**(*parent*, *flt_empty=True*, *flt_doc=True*, *flt_cmd=True*, *flt_magic=True*)

Bases: wx._windows.Panel

__init__()

evtCheckCmdFilter()

evtCheckDocFilter()

evtCheckEmptyFilter()

evtCheckMagicFilter()

getOptions()

processOptionCheckedEvt()

reloadOptions()

setOptionTrackerHook()
Define a new history tracker

updateOptionTracker()
Default history tracker (does nothing)

write()

PythonSTC

class IPython.gui.wx.ipython_history.**PythonSTC**(*parent*, *ID*, *pos=wx.Point(-1, -1)*, *size=wx.Size(-1, -1)*, *style=0*)

Bases: wx.stc.StyledTextCtrl

__init__()

Expand()

FoldAll()

OnKeyPressed()

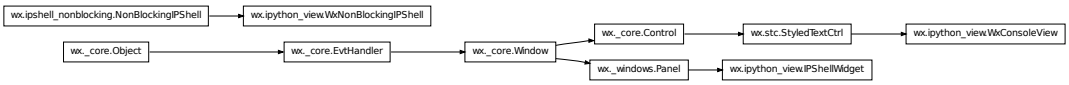
OnMarginClick()

OnUpdateUI()

10.45 gui.wx.ipython_view

10.45.1 Module: gui.wx.ipython_view

Inheritance diagram for IPython.gui.wx.ipython_view:



Provides IPython WX console widgets.

@author: Laurent Dufrechou laurent.dufrechou _at_ gmail.com This WX widget is based on the original work of Eitan Isaacson that provided the console for the GTK toolkit.

Original work from: @author: Eitan Isaacson @organization: IBM Corporation @copyright: Copyright (c) 2007 IBM Corporation @license: BSD

All rights reserved. This program and the accompanying materials are made available under the terms of the BSD which accompanies this distribution, and is available at U{http://www.opensource.org/licenses/bsd-license.php}

10.45.2 Classes

IPShellWidget

class IPython.gui.wx.ipython_view.**IPShellWidget**(*parent*, *intro=None*, *background_color='BLACK'*, *add_button_handler=None*, *wx_ip_shell=None*, *user_ns={}*, *user_global_ns=None*)

Bases: wx._windows.Panel

This is wx.Panel that embbed the IPython Thread and the wx.StyledTextControl If you want to port this to any other GUI toolkit, just replace the WxConsoleView by YOURGUIConsoleView and make YOURGUIIPythonView derivate from whatever container you want. I've choosed to derivate from a wx.Panel because it seems to be more useful Any idea to make it more 'generic' welcomed.

__init__()
Initialize. Instanciate an IPython thread. Instanciate a WxConsoleView. Redirect I/O to console.

askExitCallback()

askExitHandler()
Default exit handler

evtCheckOptionBackgroundColor()

evtCheckOptionCompletion()

evtCheckOptionThreading()

evtStateExecuteDone()

getOptions()

keyPress()
Key press callback with plenty of shell goodness, like history, autocompletions, etc.

pager()

reloadOptions()

setAskExitHandler()
Define an exit handler

setCurrentState()

setHistoryTrackerHook()
Define a new history tracker

setOptionTrackerHook()
Define a new history tracker

setStatusTrackerHook()
Define a new status tracker

stateDoExecuteLine()

stateShowPrompt()

updateHistoryTracker()
Default history tracker (does nothing)

updateOptionTracker()
Default history tracker (does nothing)

updateStatusTracker()
Default status tracker (does nothing)

WxConsoleView

class IPython.gui.wx.ipython_view.**WxConsoleView**(*parent*, *prompt*, *intro=''*, *background_color='BLACK'*, *pos=wx.Point(-1, -1)*, *ID=-1*, *size=wx.Size(-1, -1)*, *style=0*, *autocomplete_mode='IPYTHON'*)

Bases: wx.stc.StyledTextCtrl

Specialized styled text control view for console-like workflow. We use here a scintilla frontend thus it can be reused in any GUI that supports scintilla with less work.

@cvar ANSI_COLORS_BLACK: Mapping of terminal colors to X11 names. (with Black background)

@type ANSI_COLORS_BLACK: dictionary

@cvar ANSI_COLORS_WHITE: Mapping of terminal colors to X11 names. (with White background)

@type ANSI_COLORS_WHITE: dictionary

@ivar color_pat: Regex of terminal color pattern @type color_pat: _sre.SRE_Pattern

__init__()

Initialize console view.

@param parent: Parent widget @param prompt: User specified prompt @type intro: string @param intro: User specified startup introduction string @type intro: string @param background_color: Can be BLACK or WHITE @type background_color: string @param other: init param of styledTextControl (can be used as-is) @param autocomplete_mode: Can be 'IPYTHON' or 'STC'

'IPYTHON' show autocompletion the ipython way 'STC" show it scintilla text control way

OnUpdateUI()

asyncWrite()

Write given text to buffer in an asynchroneous way. It is used from another thread to be able to acces the GUI. @param text: Text to append @type text: string

buildStyles()

changeLine()

Replace currently entered command line with given text.

@param text: Text to use as replacement. @type text: string

getBackgroundColor()

getCompletionMethod()

getCurrentLine()

Get text in current command line.

@return: Text of current command line. @rtype: string

getCurrentLineEnd()

getCurrentLineStart()

getCurrentPromptStart()

getCursorPos()

getPromptLen()
Return the length of current prompt

moveCursor()

moveCursorOnNewValidKey()

removeCurrentLine()

removeFromTo()

selectFromTo()

setBackgroundColor()

setCompletionMethod()

setIndentation()

setPrompt()

setPromptCount()

showPrompt()
Prints prompt at start of line.

@param prompt: Prompt to print. @type prompt: string

write()
Write given text to buffer.

@param text: Text to append. @type text: string

writeCompletion()

writeHistory()

WxNonBlockingIPShell

class IPython.gui.wx.ipython_view.**WxNonBlockingIPShell**(*parent*, *argv=[]*, *user_ns={}*, *user_global_ns=None*, *cin=None*, *cout=None*, *cerr=None*, *ask_exit_handler=None*)

Bases: IPython.gui.wx.ipshell_nonblocking.NonBlockingIPShell

An NonBlockingIPShell Thread that is WX dependent.

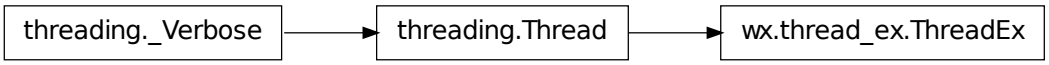
__init__()

addGUIShortcut()

10.46 gui.wx.thread_ex

10.46.1 Module: gui.wx.thread_ex

Inheritance diagram for IPython.gui.wx.thread_ex:



Thread subclass that can deal with asynchronously function calls via raise_exc.

10.46.2 ThreadEx

class IPython.gui.wx.thread_ex.**ThreadEx**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *verbose=None*)

Bases: threading.Thread

__init__()

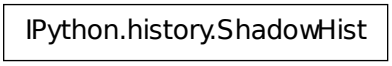
kill()
raises SystemExit in the context of the given thread, which should cause the thread to exit silently (unless caught)

raise_exc()
raises the given exception type in the context of this thread

10.47 history

10.47.1 Module: history

Inheritance diagram for IPython.history:



History related magics and functionality

10.47.2 Class

10.47.3 ShadowHist

class IPython.history.**ShadowHist**(*db*)

__init__()

add()

all()

get()

inc_idx()

10.47.4 Functions

IPython.history.**init_ipython**()

IPython.history.**magic_hist**()
Alternate name for %history.

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

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

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

Options:

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

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

-t: (default) print the 'translated' history, as IPython understands it. IPython filters your input and converts it all into valid Python source before executing it (things like magics or aliases are turned into function calls, for example). With this option, you'll see the native

history instead of the user-entered version: '%cd /' will be seen as '_ip.magic("%cd /")' instead of '%cd /'.

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

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

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

`IPython.history.`**`rep_f`**`()`

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

- %rep (no arguments):

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

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

%rep 45

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

%rep 1-4 6-7 3

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

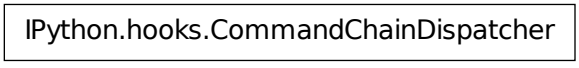
%rep foo

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

10.48 hooks

10.48.1 Module: `hooks`

Inheritance diagram for `IPython.hooks`:



hooks for IPython.

In Python, it is possible to overwrite any method of any object if you really want to. But IPython exposes a few 'hooks', methods which are _designed_ to be overwritten by users for customization purposes. This module defines the default versions of all such hooks, which get used by IPython if not overridden by the user.

hooks are simple functions, but they should be declared with 'self' as their first argument, because when activated they are registered into IPython as instance methods. The self argument will be the IPython running instance itself, so hooks have full access to the entire IPython object.

If you wish to define a new hook and activate it, you need to put the necessary code into a python file which can be either imported or execfile()'d from within your ipythonrc configuration.

For example, suppose that you have a module called 'myiphooks' in your PYTHONPATH, which contains the following definition:

import os import IPython.ipapi ip = IPython.ipapi.get()

def calljed(self,filename, linenum): "My editor hook calls the jed editor directly." print "Calling my own editor, jed ..." if os.system('jed +%d %s' % (linenum,filename)) != 0:

raise ipapi.TryNext()

ip.set_hook('editor', calljed)

You can then enable the functionality by doing 'import myiphooks' somewhere in your configuration files or ipython command line.

10.48.2 Class

10.48.3 CommandChainDispatcher

class IPython.hooks.**CommandChainDispatcher**(*commands=None*)

Dispatch calls to a chain of commands until some func can handle it

Usage: instantiate, execute "add" to add commands (with optional priority), execute normally via f() calling mechanism.

__init__()

add()

Add a func to the cmd chain with given priority

10.48.4 Functions

IPython.hooks.**clipboard_get**()

Get text from the clipboard.

IPython.hooks.**editor**()

Open the default editor at the given filename and linenumber.

This is IPython's default editor hook, you can use it as an example to write your own modified one. To set your own editor function as the new editor hook, call ip.set_hook('editor',yourfunc).

IPython.hooks.**fix_error_editor**()
Open the editor at the given filename, linenumber, column and show an error message. This is used for correcting syntax errors. The current implementation only has special support for the VIM editor, and falls back on the 'editor' hook if VIM is not used.

Call ip.set_hook('fix_error_editor',youfunc) to use your own function,

IPython.hooks.**generate_output_prompt**()

IPython.hooks.**generate_prompt**()
calculate and return a string with the prompt to display

IPython.hooks.**input_prefilter**()
Default input prefilter

This returns the line as unchanged, so that the interpreter knows that nothing was done and proceeds with "classic" prefiltering (%magics, !shell commands etc.).

Note that leading whitespace is not passed to this hook. Prefilter can't alter indentation.

IPython.hooks.**late_startup_hook**()
Executed after ipython has been constructed and configured

IPython.hooks.**pre_prompt_hook**()
Run before displaying the next prompt

Use this e.g. to display output from asynchronous operations (in order to not mess up text entry)

IPython.hooks.**pre_runcode_hook**()
Executed before running the (prefiltered) code in IPython

IPython.hooks.**result_display**()
Default display hook.

Called for displaying the result to the user.

IPython.hooks.**shell_hook**()
Run system/shell command a'la os.system()

IPython.hooks.**show_in_pager**()
Run a string through pager

IPython.hooks.**shutdown_hook**()
default shutdown hook

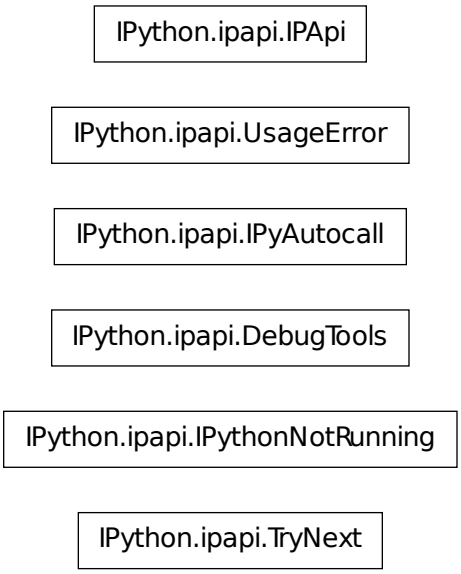
Typically, shotdown hooks should raise TryNext so all shutdown ops are done

IPython.hooks.**synchronize_with_editor**()

10.49 ipapi

10.49.1 Module: `ipapi`

Inheritance diagram for `IPython.ipapi`:



IPython customization API

Your one-stop module for configuring & extending ipython

The API will probably break when ipython 1.0 is released, but so will the other configuration method (rc files).

All names prefixed by underscores are for internal use, not part of the public api.

Below is an example that you can just put to a module and import from ipython.

A good practice is to install the config script below as e.g.

~/.ipython/my_private_conf.py

And do

import_mod my_private_conf

in ~/.ipython/ipythonrc

That way the module is imported at startup and you can have all your personal configuration (as opposed to boilerplate ipythonrc-PROFILENAME stuff) in there.

import IPython.ipapi ip = IPython.ipapi.get()

def ankka_f(self, arg): print ‘Ankka’,self,’says uppercase:’,arg.upper()

ip.expose_magic(‘ankka’,ankka_f)

ip.magic(‘alias sayhi echo “Testing, hi ok”’) ip.magic(‘alias helloworld echo “Hello world”’) ip.system(‘pwd’)

ip.ex(‘import re’) ip.ex(‘” def funcci(a,b):

print a+b

print funcci(3,4) “’) ip.ex(‘funcci(348,9)’)

def jed_editor(self,filename, linenum=None): print ‘Calling my own editor, jed ... via hook!’ import os if linenum is None: linenum = 0 os.system(‘jed +%d %s’ % (linenum, filename)) print ‘exiting jed’

ip.set_hook(‘editor’,jed_editor)

o = ip.options o.autocall = 2 # FULL autocall mode

print ‘done!’

10.49.2 Classes

DebugTools

class IPython.ipapi.**DebugTools**(*ip*)

Used for debugging mishaps in api usage

So far, tracing redefinitions is supported.

__init__()

check_hotname()

debug_stack()

hotname()

IPApi

class IPython.ipapi.**IPApi**(*ip*)

Bases: object

The actual API class for configuring IPython

You should do all of the IPython configuration by getting an IPApi object with IPython.ipapi.get() and using the attributes and methods of the returned object.

__init__()

db

A handle to persistent dict-like database (a PickleShareDB object)

defalias()
Define a new alias

_ip.defalias('bb','bldmake bldfiles')

Creates a new alias named 'bb' in ipython user namespace

defmacro()
Define a new macro

2 forms of calling:

mac = _ip.defmacro('print "hello"

print "world"')

(doesn't put the created macro on user namespace)

_ip.defmacro('build', 'bldmake bldfiles

abld build winscw udeb')

(creates a macro named 'build' in user namespace)

ev()
Evaluate python expression expr in user namespace

Returns the result of evaluation

ex()
Execute a normal python statement in user namespace

expand_alias()
Expand an alias in the command line

Returns the provided command line, possibly with the first word (command) translated according to alias expansion rules.

[ipython]|16> _ip.expand_aliases("np myfile.txt") <16> 'q:/opt/np/notepad++.exe myfile.txt'

expose_magic()
Expose own function as magic function for ipython

def foo_impl(self,parameter_s=''): 'My very own magic!. (Use docstrings, IPython reads them).' print 'Magic function. Passed parameter is between < >:' print '<%s>' % parameter_s print 'The self object is:',self

ipapi.expose_magic('foo',foo_impl)

get_db()
A handle to persistent dict-like database (a PickleShareDB object)

get_options()
All configurable variables.

itpl()
Expand Itpl format string s.

Only callable from command line (i.e. prefilter results); If you use in your scripts, you need to use a bigger depth!

load()
Load an extension.

Some modules should (or must) be 'load()':ed, rather than just imported.

Loading will do:

•run init_ipython(ip)

•run ipython_firstrun(ip)

options
All configurable variables.

runlines()
Run the specified lines in interpreter, honoring ipython directives.

This allows %magic and !shell escape notations.

Takes either all lines in one string or list of lines.

set_next_input()
Sets the 'default' input string for the next command line.

Requires readline.

Example:

[D:ipython]|1> _ip.set_next_input("Hello Word") [D:ipython]|2> Hello **Word_** # cursor is here

to_user_ns()
Inject a group of variables into the IPython user namespace.

Inputs:

•vars: string with variable names separated by whitespace, or a

dict with name/value pairs.

•interactive: if True (default), the var will be listed with

%whos et. al.

This utility routine is meant to ease interactive debugging work, where you want to easily propagate some internal variable in your code up to the interactive namespace for further exploration.

When you run code via %run, globals in your script become visible at the interactive prompt, but this doesn't happen for locals inside your own functions and methods. Yet when debugging, it is common to want to explore some internal variables further at the interactive propmt.

Examples:

To use this, you first must obtain a handle on the ipython object as indicated above, via:

import IPython.ipapi ip = IPython.ipapi.get()

Once this is done, inside a routine foo() where you want to expose variables x and y, you do the following:

def foo(): ... x = your_computation() y = something_else()

This pushes x and y to the interactive prompt immediately, even # if this routine crashes on the next line after: ip.to_user_ns('x y') ...

To expose *ALL* the local variables from the function, use: ip.to_user_ns(locals())

... # return

If you need to rename variables, the dict input makes it easy. For example, this call exposes variables 'foo' as 'x' and 'bar' as 'y' in IPython user namespace:

ip.to_user_ns(dict(x=foo,y=bar))

IPyAutocall

class IPython.ipapi.**IPyAutocall**
Instances of this class are always autocalled

This happens regardless of 'autocall' variable state. Use this to develop macro-like mechanisms.

set_ip()
Will be used to set _ip point to current ipython instance b/f call

Override this method if you don't want this to happen.

IPythonNotRunning

class IPython.ipapi.**IPythonNotRunning**(*warn=True*)
Dummy do-nothing class.

Instances of this class return a dummy attribute on all accesses, which can be called and warns. This makes it easier to write scripts which use the ipapi.get() object for informational purposes to operate both with and without ipython. Obviously code which uses the ipython object for computations will not work, but this allows a wider range of code to transparently work whether ipython is being used or not.

__init__()

TryNext

class IPython.ipapi.**TryNext**(**args*, ***kwargs*)
Bases: exceptions.Exception

Try next hook exception.

Raise this in your hook function to indicate that the next hook handler should be used to handle the operation. If you pass arguments to the constructor those arguments will be used by the next hook instead of the original ones.

__init__()

UsageError

class IPython.ipapi.**UsageError**
Bases: exceptions.Exception

Error in magic function arguments, etc.

Something that probably won't warrant a full traceback, but should nevertheless interrupt a macro / batch file.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.49.3 Functions

IPython.ipapi.**get**()
Get an IPApi object.

If allow_dummy is true, returns an instance of IPythonNotRunning instead of None if not running under IPython.

If dummy_warn is false, the dummy instance will be completely silent.

Running this should be the first thing you do when writing extensions that can be imported as normal modules. You can then direct all the configuration operations against the returned object.

IPython.ipapi.**launch_new_instance**()
Make and start a new ipython instance.

This can be called even without having an already initialized ipython session running.

This is also used as the egg entry point for the 'ipython' script.

IPython.ipapi.**make_session**()
Makes, but does not launch an IPython session.

Later on you can call obj.mainloop() on the returned object.

Inputs:

•user_ns(None): a dict to be used as the user's namespace with initial

data.

WARNING: This should *not* be run when a session exists already.

IPython.ipapi.**make_user_global_ns**()
Return a valid user global namespace.

Similar to make_user_ns(), but global namespaces are really only needed in embedded applications, where there is a distinction between the user's interactive namespace and the global one where ipython is running.

This API is currently deprecated. Use ipapi.make_user_namespaces() instead to make both the local and global namespace objects simultaneously.

Parameters

ns [dict, optional] The current user global namespace. The items in this namespace should be included in the output. If None, an appropriate blank namespace should be created.

Returns A true dict to be used as the global namespace of the interpreter.

IPython.ipapi.**make_user_namespaces**()

Return a valid local and global user interactive namespaces.

This builds a dict with the minimal information needed to operate as a valid IPython user namespace, which you can pass to the various embedding classes in ipython. The default implementation returns the same dict for both the locals and the globals to allow functions to refer to variables in the namespace. Customized implementations can return different dicts. The locals dictionary can actually be anything following the basic mapping protocol of a dict, but the globals dict must be a true dict, not even a subclass. It is recommended that any custom object for the locals namespace synchronize with the globals dict somehow.

Raises TypeError if the provided globals namespace is not a true dict.

Parameters

user_ns [dict-like, optional] The current user namespace. The items in this namespace should be included in the output. If None, an appropriate blank namespace should be created.

user_global_ns [dict, optional] The current user global namespace. The items in this namespace should be included in the output. If None, an appropriate blank namespace should be created.

Returns A tuple pair of dictionary-like object to be used as the local namespace of the interpreter and a dict to be used as the global namespace.

IPython.ipapi.**make_user_ns**()

Return a valid user interactive namespace.

This builds a dict with the minimal information needed to operate as a valid IPython user namespace, which you can pass to the various embedding classes in ipython.

This API is currently deprecated. Use ipapi.make_user_namespaces() instead to make both the local and global namespace objects simultaneously.

Parameters

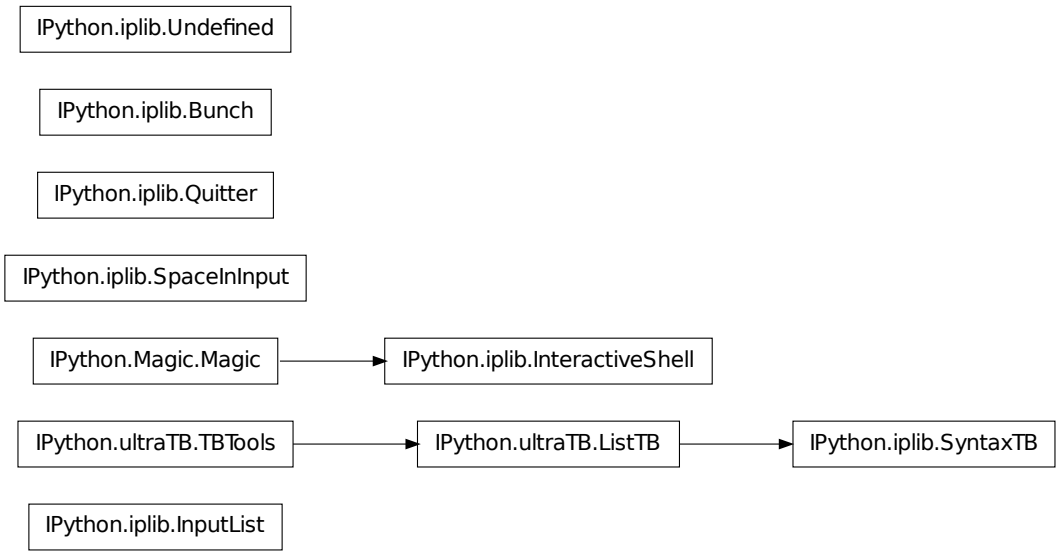
user_ns [dict-like, optional] The current user namespace. The items in this namespace should be included in the output. If None, an appropriate blank namespace should be created.

Returns A dictionary-like object to be used as the local namespace of the interpreter.

10.50 iplib

10.50.1 Module: iplib

Inheritance diagram for IPython.iplib:



IPython – An enhanced Interactive Python

Requires Python 2.4 or newer.

This file contains all the classes and helper functions specific to IPython.

10.50.2 Classes

Bunch

class IPython.iplib.**Bunch**

InputList

class IPython.iplib.**InputList**

Bases: list

Class to store user input.

It's basically a list, but slices return a string instead of a list, thus allowing things like (assuming 'In' is an instance):

exec In[4:7]

or

exec In[5:9] + In[14] + In[21:25]

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

InteractiveShell

class IPython.iplib.**InteractiveShell**(*name*, *usage=None*, *rc=Struct({'__allownew': True, 'args': None, 'opts': None})*, *user_ns=None*, *user_global_ns=None*, *banner2=''*, *custom_exceptions=((), None)*, *embedded=False*)

Bases: object, IPython.Magic.Magic

An enhanced console for Python.

__init__()

add_builtins()
Store ipython references into the builtin namespace.

Some parts of ipython operate via builtins injected here, which hold a reference to IPython itself.

alias_table_validate()
Update information about the alias table.

In particular, make sure no Python keywords/builtins are in it.

ask_exit()
Call for exiting. Can be overiden and used as a callback.

ask_yes_no()

atexit_operations()
This will be executed at the time of exit.

Saving of persistent data should be performed here.

autoindent_update()
Keep track of the indent level.

cache_main_mod()
Cache a main module's namespace.

When scripts are executed via %run, we must keep a reference to the namespace of their __main__ module (a FakeModule instance) around so that Python doesn't clear it, rendering objects defined therein useless.

This method keeps said reference in a private dict, keyed by the absolute path of the module object (which corresponds to the script path). This way, for multiple executions of the same script we only keep one copy of the namespace (the last one), thus preventing memory leaks from old references while allowing the objects from the last execution to be accessible.

Note: we can not allow the actual FakeModule instances to be deleted, because of how Python tears down modules (it hard-sets all their references to None without regard for reference counts). This method must therefore make a *copy* of the given namespace, to allow the original module's __dict__ to be cleared and reused.

Parameters **ns** : a namespace (a dict, typically)

fname [str] Filename associated with the namespace.

Examples

In [10]: import IPython

In [11]: _ip.IP.cache_main_mod(IPython.__dict__,IPython.__file__)

In [12]: IPython.__file__ in _ip.IP._main_ns_cache Out[12]: True

call_alias()
Call an alias given its name and the rest of the line.

This is only used to provide backwards compatibility for users of ipalias(), use of which is not recommended for anymore.

call_pdb
Control auto-activation of pdb at exceptions

clean_builtins()
Remove any builtins which might have been added by add_builtins, or restore overwritten ones to their previous values.

clear_main_mod_cache()
Clear the cache of main modules.

Mainly for use by utilities like %reset.

Examples

In [15]: import IPython

In [16]: _ip.IP.cache_main_mod(IPython.__dict__,IPython.__file__)

In [17]: len(_ip.IP._main_ns_cache) > 0 Out[17]: True

In [18]: _ip.IP.clear_main_mod_cache()

In [19]: len(_ip.IP._main_ns_cache) == 0 Out[19]: True

complete()
Return a sorted list of all possible completions on text.

Inputs:

- text: a string of text to be completed on.

This is a wrapper around the completion mechanism, similar to what readline does at the command line when the TAB key is hit. By exposing it as a method, it can be used by other non-readline environments (such as GUIs) for text completion.

Simple usage example:

In [7]: x = 'hello'

In [8]: x Out[8]: 'hello'

In [9]: print x hello

In [10]: _ip.IP.complete('x.l') Out[10]: ['x.ljust', 'x.lower', 'x.lstrip']

debugger()
Call the pydb/pdb debugger.

Keywords:

•force(False): by default, this routine checks the instance call_pdb

flag and does not actually invoke the debugger if the flag is false. The 'force' option forces the debugger to activate even if the flag is false.

edit_syntax_error()
The bottom half of the syntax error handler called in the main loop.

Loop until syntax error is fixed or user cancels.

embed_mainloop()
Embeds IPython into a running python program.

Input:

•header: An optional header message can be specified.

•local_ns, global_ns: working namespaces. If given as None, the

IPython-initialized one is updated with __main__.__dict__, so that program variables become visible but user-specific configuration remains possible.

•stack_depth: specifies how many levels in the stack to go to

looking for namespaces (when local_ns and global_ns are None). This allows an intermediate caller to make sure that this function gets the namespace from the intended level in the stack. By default (0) it will get its locals and globals from the immediate caller.

Warning: it's possible to use this in a program which is being run by IPython itself (via %run), but some funny things will happen (a few globals get overwritten). In the future this will be cleaned up, as there is no fundamental reason why it can't work perfectly.

excepthook()
One more defense for GUI apps that call sys.excepthook.

GUI frameworks like wxPython trap exceptions and call sys.excepthook themselves. I guess this is a feature that enables them to keep running after exceptions that would otherwise kill their

mainloop. This is a bother for IPython which excepts to catch all of the program exceptions with a try: except: statement.

Normally, IPython sets sys.excepthook to a CrashHandler instance, so if any app directly invokes sys.excepthook, it will look to the user like IPython crashed. In order to work around this, we can disable the CrashHandler and replace it with this excepthook instead, which prints a regular traceback using our InteractiveTB. In this fashion, apps which call sys.excepthook will generate a regular-looking exception from IPython, and the CrashHandler will only be triggered by real IPython crashes.

This hook should be used sparingly, only in places which are not likely to be true IPython errors.

exec_init_cmd()
Execute a command given at the command line.

This emulates Python's -c option.

exit()
Handle interactive exit.

This method calls the ask_exit callback.

expand_aliases()
Expand multiple levels of aliases:

if:

alias foo bar /tmp alias baz foo

then:

baz huhhahhei -> bar /tmp huhhahhei

getapi()
Get an IPApi object for this shell instance

Getting an IPApi object is always preferable to accessing the shell directly, but this holds true especially for extensions.

It should always be possible to implement an extension with IPApi alone. If not, contact maintainer to request an addition.

handle_alias()
Handle alias input lines.

handle_auto()
Hande lines which can be auto-executed, quoting if requested.

handle_emacs()
Handle input lines marked by python-mode.

handle_help()
Try to get some help for the object.

obj? or ?obj -> basic information. obj?? or ??obj -> more details.

handle_magic()
Execute magic functions.

handle_normal()
Handle normal input lines. Use as a template for handlers.

handle_shell_escape()
Execute the line in a shell, empty return value

history_saving_wrapper()
Wrap func for readline history saving

Convert func into callable that saves & restores history around the call

indent_current_str()
return the current level of indentation as a string

init_auto_alias()
Define some aliases automatically.

These are ALL parameter-less aliases

init_namespaces()
Initialize all user-visible namespaces to their minimum defaults.

Certain history lists are also initialized here, as they effectively act as user namespaces.

Notes

All data structures here are only filled in, they are NOT reset by this method. If they were not empty before, data will simply be added to therm.

init_readline()
Command history completion/saving/reloading.

interact()
Closely emulate the interactive Python console.

The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the real Python interpreter, followed by the current class name in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

interact_handle_input()
Handle the input line (in read-eval-print loop)

Provided for those who want to implement their own read-eval-print loop (e.g. GUIs), not used in standard IPython flow.

interact_prompt()
Print the prompt (in read-eval-print loop)

Provided for those who want to implement their own read-eval-print loop (e.g. GUIs), not used in standard IPython flow.

interact_with_readline()
Demo of using interact_handle_input, interact_prompt

This is the main read-eval-print loop. If you need to implement your own (e.g. for GUI), it should work like this.

ipalias()
Call an alias by name.

Input: a string containing the name of the alias to call and any additional arguments to be passed to the magic.

ipalias('name -opt foo bar') is equivalent to typing at the ipython prompt:

In[1]: name -opt foo bar

To call an alias without arguments, simply use ipalias('name').

This provides a proper Python function to call IPython's aliases in any valid Python code you can type at the interpreter, including loops and compound statements. It is added by IPython to the Python builtin namespace upon initialization.

ipmagic()
Call a magic function by name.

Input: a string containing the name of the magic function to call and any additional arguments to be passed to the magic.

ipmagic('name -opt foo bar') is equivalent to typing at the ipython prompt:

In[1]: %name -opt foo bar

To call a magic without arguments, simply use ipmagic('name').

This provides a proper Python function to call IPython's magics in any valid Python code you can type at the interpreter, including loops and compound statements. It is added by IPython to the Python builtin namespace upon initialization.

ipsystem()
Make a system call, using IPython.

mainloop()
Creates the local namespace and starts the mainloop.

If an optional banner argument is given, it will override the internally created default banner.

mktempfile()
Make a new tempfile and return its filename.

This makes a call to tempfile.mktemp, but it registers the created filename internally so ipython cleans it up at exit time.

Optional inputs:

•data(None): if data is given, it gets written out to the temp file

immediately, and the file is closed again.

multiline_prefilter()
Run _prefilter for each line of input

Covers cases where there are multiple lines in the user entry, which is the case when the user goes back to a multiline history entry and presses enter.

new_main_mod()
Return a new 'main' module object for user code execution.

post_config_initialization()
Post configuration init method

This is called after the configuration files have been processed to 'finalize' the initialization.

pre_config_initialization()
Pre-configuration init method

This is called before the configuration files are processed to prepare the services the config files might need.

self.rc already has reasonable default values at this point.

pre_readline()
readline hook to be used at the start of each line.

Currently it handles auto-indent only.

prefilter()
Run _prefilter for each line of input

Covers cases where there are multiple lines in the user entry, which is the case when the user goes back to a multiline history entry and presses enter.

push()
Push a line to the interpreter.

The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's runsource() method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is 1 if more input is required, 0 if the line was dealt with in some way (this is the same as runsource()).

raw_input()
Write a prompt and read a line.

The returned line does not include the trailing newline. When the user enters the EOF key sequence, EOFError is raised.

Optional inputs:

•prompt(''): a string to be printed to prompt the user.

•continue_prompt(False): whether this line is the first one or a

continuation in a sequence of inputs.

rc_set_toggle()
Set or toggle a field in IPython's rc config. structure.

If called with no arguments, it acts as a toggle.

If called with a non-existent field, the resulting AttributeError exception will propagate out.

reloadhist()
Reload the input history from disk file.

reset()
Clear all internal namespaces.

Note that this is much more aggressive than %reset, since it clears fully all namespaces, as well as all input/output lists.

resetbuffer()
Reset the input buffer.

runcode()
Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Return value: a flag indicating whether the code to be run completed successfully:

- 0: successful execution.
- 1: an error occurred.

runlines()
Run a string of one or more lines of source.

This method is capable of running a string containing multiple source lines, as if they had been entered at the IPython prompt. Since it exposes IPython's processing machinery, the given strings can contain magic calls (%magic), special shell access (!cmd), etc.

runsource()
Compile and run some source in the interpreter.

Arguments are as for compile_command().

One several things can happen:

1) The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method.

2) The input is incomplete, and more input is required; compile_command() returned None. Nothing happens.

3) The input is complete; compile_command() returned a code object. The code is executed by calling self.runcode() (which also handles run-time exceptions, except for SystemExit).

The return value is:

- True in case 2
- False in the other cases, unless an exception is raised, where

None is returned instead. This can be used by external callers to know whether to continue feeding input or not.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

safe_execfile()
A safe version of the builtin execfile().

This version will never throw an exception, and knows how to handle ipython logs as well.

Parameters

fname [string] Name of the file to be executed.

where [tuple] One or two namespaces, passed to execfile() as (globals,locals). If only one is given, it is passed as both.

Keywords islog : boolean (False)

quiet : boolean (True)

exit_ignore : boolean (False)

savehist()
Save input history to a file (via readline library).

set_autoindent()
Set the autoindent flag, checking for readline support.

If called with no arguments, it acts as a toggle.

set_completer()
reset readline's completer to be our own.

set_completer_frame()

set_crash_handler()
Set the IPython crash handler.

This must be a callable with a signature suitable for use as sys.excepthook.

set_custom_completer()
Adds a new custom completer function.

The position argument (defaults to 0) is the index in the completers list where you want the completer to be inserted.

set_custom_exc()
Set a custom exception handler, which will be called if any of the exceptions in exc_tuple occur in the mainloop (specifically, in the runcode() method.

Inputs:

•exc_tuple: a *tuple* of valid exceptions to call the defined

handler for. It is very important that you use a tuple, and NOT A LIST here, because of the way Python's except statement works. If you only want to trap a single exception, use a singleton tuple:

exc_tuple == (MyCustomException,)

•handler: this must be defined as a function with the following

basic interface: def my_handler(self,etype,value,tb).

This will be made into an instance method (via new.instancemethod) of IPython itself, and it will be called if any of the exceptions listed in the exc_tuple are caught. If the handler is None, an internal basic one is used, which just prints basic info.

WARNING: by putting in your own exception handler into IPython's main execution loop, you run a very good chance of nasty crashes. This facility should only be used if you really know what you are doing.

set_hook()
set_hook(name,hook) -> sets an internal IPython hook.

IPython exposes some of its internal API as user-modifiable hooks. By adding your function to one of these hooks, you can modify IPython's behavior to call at runtime your own routines.

showsyntaxerror()
Display the syntax error that just occurred.

This doesn't display a stack trace because there isn't one.

If a filename is given, it is stuffed in the exception instead of what was there before (because Python's parser always uses "<string>" when reading from a string).

showtraceback()
Display the exception that just occurred.

If nothing is known about the exception, this is the method which should be used throughout the code for presenting user tracebacks, rather than directly invoking the InteractiveTB object.

A specific showsyntaxerror() also exists, but this method can take care of calling it if needed, so unless you are explicitly catching a SyntaxError exception, don't try to analyze the stack manually and simply call this method.

split_user_input()

transform_alias()
Transform alias to system command string.

user_setup()
Install the user configuration directory.

Notes

DEPRECATED: use the top-level user_setup() function instead.

var_expand()
Expand python variables in a string.

The depth argument indicates how many frames above the caller should be walked to look for the local namespace where to expand variables.

The global namespace for expansion is always the user's interactive namespace.

write()
Write a string to the default output

write_err()
Write a string to the default error output

Quitter

class IPython.iplib.**Quitter**(*shell*, *name*)
Bases: object

Simple class to handle exit, similar to Python 2.5's.

It handles exiting in an ipython-safe manner, which the one in Python 2.5 doesn't do (obviously, since it doesn't know about ipython).

__init__()

SpaceInInput

class IPython.iplib.**SpaceInInput**
Bases: exceptions.Exception

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

SyntaxTB

class IPython.iplib.**SyntaxTB**(*color_scheme='NoColor'*)
Bases: IPython.ultraTB.ListTB

Extension which holds some state: the last exception value

__init__()

clear_err_state()
Return the current error state and clear it

Undefined

class IPython.iplib.**Undefined**

10.50.3 Functions

IPython.iplib.**num_ini_spaces**()
Return the number of initial spaces in a string

IPython.iplib.**softspace**()
Copied from code.py, to remove the dependency

IPython.iplib.**user_setup**()

Install or upgrade the user configuration directory.

Can be called when running for the first time or to upgrade the user's .ipython/ directory.

Parameters **ipythondir** : path

The directory to be used for installation/upgrade. In 'install' mode, if this path already exists, the function exits immediately.

rc_suffix [str] Extension for the config files. On *nix platforms it is typically the empty string, while Windows normally uses '.ini'.

mode [str, optional] Valid modes are 'install' and 'upgrade'.

interactive [bool, optional] If False, do not wait for user input on any errors. Normally after printing its status information, this function waits for the user to hit Return before proceeding. This is because the default use case is when first installing the IPython configuration, so we want the user to acknowledge the initial message, which contains some useful information.

10.51 ipmaker

10.51.1 Module: ipmaker

IPython – An enhanced Interactive Python

Requires Python 2.1 or better.

This file contains the main make_IPython() starter function.

10.51.2 Functions

IPython.ipmaker.**force_import**()

IPython.ipmaker.**make_IPython**()

This is a dump of IPython into a single function.

Later it will have to be broken up in a sensible manner.

Arguments:

•argv: a list similar to sys.argv[1:]. It should NOT contain the desired

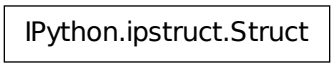
script name, b/c DPyGetOpt strips the first argument only for the real sys.argv.

•user_ns: a dict to be used as the user's namespace.

10.52 ipstruct

10.52.1 Module: ipstruct

Inheritance diagram for IPython.ipstruct:



Mimic C structs with lots of extra functionality.

10.52.2 Struct

class IPython.ipstruct.**Struct**(*dict=None*, ***kw*)

Class to mimic C structs but also provide convenient dictionary-like functionality.

Instances can be initialized with a dictionary, a list of key=value pairs or both. If both are present, the dictionary must come first.

Because Python classes provide direct assignment to their members, it's easy to overwrite normal methods (S.copy = 1 would destroy access to S.copy()). For this reason, all builtin method names are protected and can't be assigned to. An attempt to do s.copy=1 or s['copy']=1 will raise a KeyError exception. If you really want to, you can bypass this protection by directly assigning to __dict__: s.__dict__['copy']=1 will still work. Doing this will break functionality, though. As in most of Python, namespace protection is weakly enforced, so feel free to shoot yourself if you really want to.

Note that this class uses more memory and is *much* slower than a regular dictionary, so be careful in situations where memory or performance are critical. But for day to day use it should behave fine. It is particularly convenient for storing configuration data in programs.

+,+=,- and -= are implemented. +/+= do merges (non-destructive updates), -/-= remove keys from the original. See the method descripitions.

This class allows a quick access syntax: both s.key and s['key'] are valid. This syntax has a limitation: each 'key' has to be explicitly accessed by its original name. The normal s.key syntax doesn't provide access to the keys via variables whose values evaluate to the desired keys. An example should clarify this:

Define a dictionary and initialize both with dict and k=v pairs: >>> d={'a':1,'b':2} >>> s=Struct(d,hi=10,ho=20)

The return of __repr__ can be used to create a new instance: >>> s Struct({'__allownew': True, 'a': 1, 'b': 2, 'hi': 10, 'ho': 20})

Note: the special '__allownew' key is used for internal purposes.

__str__ (called by print) shows it's not quite a regular dictionary: >>> print s Struct({'__allownew': True, 'a': 1, 'b': 2, 'hi': 10, 'ho': 20})

Access by explicitly named key with dot notation: >>> s.a 1

Or like a dictionary: >>> s['a'] 1

If you want a variable to hold the key value, only dictionary access works: >>> key='hi' >>> s.key Traceback (most recent call last):

File "<stdin>", line 1, in ?

AttributeError: Struct instance has no attribute 'key'

```
>>> s[key]
10
```

Another limitation of the s.key syntax (and Struct(key=val) initialization): keys can't be numbers. But numeric keys can be used and accessed using the dictionary syntax. Again, an example:

This doesn't work (prompt changed to avoid confusing the test system): ->> s=Struct(4='hi') Traceback (most recent call last):

...

SyntaxError: keyword can't be an expression

But this does: >>> s=Struct() >>> s[4]='hi' >>> s Struct({4: 'hi', '__allownew': True}) >>> s[4] 'hi'

__init__()
Initialize with a dictionary, another Struct, or by giving explicitly the list of attributes.

Both can be used, but the dictionary must come first: Struct(dict), Struct(k1=v1,k2=v2) or Struct(dict,k1=v1,k2=v2).

allow_new_attr()
Set whether new attributes can be created inside struct

This can be used to catch typos by verifying that the attribute user tries to change already exists in this Struct.

clear()
Clear all attributes.

copy()
Return a (shallow) copy of a Struct.

dict()
Return the Struct's dictionary.

dictcopy()
Return a (shallow) copy of the Struct's dictionary.

get()
S.get(k[,d]) -> S[k] if k in S, else d. d defaults to None.

has_key()
Like has_key() dictionary method.

hasattr()
hasattr function available as a method.

Implemented like has_key, to make sure that all available keys in the internal dictionary of the Struct appear also as attributes (even numeric keys).

items()
Return the items in the Struct's dictionary, in the same format as a call to {}.items().

keys()
Return the keys in the Struct's dictionary, in the same format as a call to {}.keys().

merge()
S.merge(data,conflict,k=v1,k=v2,...) -> merge data and k=v into S.

This is similar to update(), but much more flexible. First, a dict is made from data+key=value pairs. When merging this dict with the Struct S, the optional dictionary 'conflict' is used to decide what to do.

If conflict is not given, the default behavior is to preserve any keys with their current value (the opposite of the update method's behavior).

conflict is a dictionary of binary functions which will be used to solve key conflicts. It must have the following structure:

conflict == { fn1 : [Skey1,Skey2,...], fn2 : [Skey3], etc }

Values must be lists or whitespace separated strings which are automatically converted to lists of strings by calling string.split().

Each key of conflict is a function which defines a policy for resolving conflicts when merging with the input data. Each fn must be a binary function which returns the desired outcome for a key conflict. These functions will be called as fn(old,new).

An example is probably in order. Suppose you are merging the struct S with a dict D and the following conflict policy dict:

S.merge(D,{fn1:['a','b',4], fn2:'key_c key_d'})

If the key 'a' is found in both S and D, the merge method will call:

S['a'] = fn1(S['a'],D['a'])

As a convenience, merge() provides five (the most commonly needed) pre-defined policies: preserve, update, add, add_flip and add_s. The easiest explanation is their implementation:

preserve = lambda old,new: old update = lambda old,new: new add = lambda old,new: old + new add_flip = lambda old,new: new + old # note change of order! add_s = lambda old,new: old + ' ' + new # only works for strings!

You can use those four words (as strings) as keys in conflict instead of defining them as functions, and the merge method will substitute the appropriate functions for you. That is, the call

S.merge(D,{'preserve':'a b c','add':[4,5,'d'],my_function:[6]})

will automatically substitute the functions preserve and add for the names 'preserve' and 'add' before making any function calls.

For more complicated conflict resolution policies, you still need to construct your own functions.

popitem()
S.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if S is empty.

setdefault()
S.setdefault(k[,d]) -> S.get(k,d), also set S[k]=d if k not in S

update()
Update (merge) with data from another Struct or from a dictionary. Optionally, one or more key=value pairs can be given at the end for direct update.

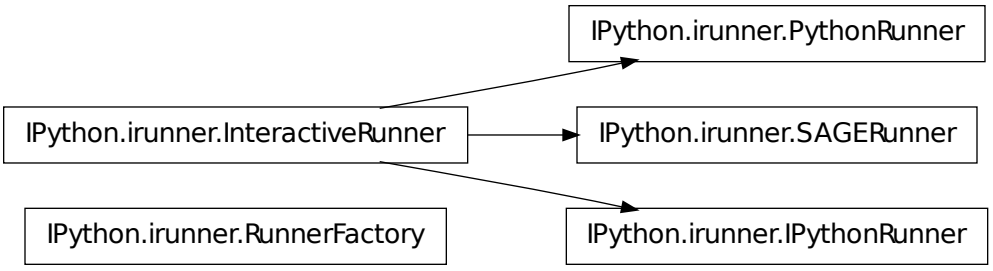
values()
Return the values in the Struct's dictionary, in the same format as a call to {}.values().

Can be called with an optional argument keys, which must be a list or tuple of keys. In this case it returns only the values corresponding to those keys (allowing a form of 'slicing' for Structs).

10.53 irunner

10.53.1 Module: irunner

Inheritance diagram for IPython.irunner:



Module for interactively running scripts.

This module implements classes for interactively running scripts written for any system with a prompt which can be matched by a regexp suitable for pexpect. It can be used to run as if they had been typed up interactively, an arbitrary series of commands for the target system.

The module includes classes ready for IPython (with the default prompts), plain Python and SAGE, but making a new one is trivial. To see how to use it, simply run the module as a script:

./irunner.py –help

This is an extension of Ken Schutte <kschutte-AT-csail.mit.edu>'s script contributed on the ipython-user list:

http://scipy.net/pipermail/ipython-user/2006-May/001705.html

NOTES:

- This module requires pexpect, available in most linux distros, or which can

be downloaded from

http://pexpect.sourceforge.net

- Because pexpect only works under Unix or Windows-Cygwin, this has the same

limitations. This means that it will NOT work under native windows Python.

10.53.2 Classes

IPythonRunner

class IPython.irunner.**IPythonRunner**(*program='ipython'*, *args=None*, *out=<open file '<stdout>'*, *mode 'w' at 0x403b2078>*, *echo=True*)

Bases: IPython.irunner.InteractiveRunner

Interactive IPython runner.

This initalizes IPython in 'nocolor' mode for simplicity. This lets us avoid having to write a regexp that matches ANSI sequences, though pexpect does support them. If anyone contributes patches for ANSI color support, they will be welcome.

It also sets the prompts manually, since the prompt regexps for pexpect need to be matched to the actual prompts, so user-customized prompts would break this.

__init__()

New runner, optionally passing the ipython command to use.

InteractiveRunner

class IPython.irunner.**InteractiveRunner**(*program*, *prompts*, *args=None*, *out=<open file '<stdout>'*, *mode 'w' at 0x403b2078>*, *echo=True*)

Bases: object

Class to run a sequence of commands through an interactive program.

__init__()

Construct a runner.

Inputs:

•program: command to execute the given program.

•prompts: a list of patterns to match as valid prompts, in the

format used by pexpect. This basically means that it can be either a string (to be compiled as a regular expression) or a list of such (it must be a true list, as pexpect does type checks).

If more than one prompt is given, the first is treated as the main program prompt and the others as 'continuation' prompts, like python's. This means that blank lines in the input source are ommitted when the first prompt is matched, but are NOT ommitted when the continuation one matches, since this is how python signals the end of multiline input interactively.

Optional inputs:

•args(None): optional list of strings to pass as arguments to the

child program.

•out(sys.stdout): if given, an output stream to be used when writing

output. The only requirement is that it must have a .write() method.

Public members not parameterized in the constructor:

•delaybeforesend(0): Newer versions of pexpect have a delay before

sending each new input. For our purposes here, it's typically best to just set this to zero, but if you encounter reliability problems or want an interactive run to pause briefly at each prompt, just increase this value (it is measured in seconds). Note that this variable is not honored at all by older versions of pexpect.

close()
close child process

main()
Run as a command-line script.

run_file()
Run the given file interactively.

Inputs:

-fname: name of the file to execute.

See the run_source docstring for the meaning of the optional arguments.

run_source()
Run the given source code interactively.

Inputs:

•source: a string of code to be executed, or an open file object we

can iterate over.

Optional inputs:

•interact(False): if true, start to interact with the running

program at the end of the script. Otherwise, just exit.

•get_output(False): if true, capture the output of the child process

(filtering the input commands out) and return it as a string.

Returns: A string containing the process output, but only if requested.

PythonRunner

class IPython.irunner.**PythonRunner**(*program='python'*, *args=None*, *out=<open file '<stdout>'*, *mode 'w' at 0x403b2078>*, *echo=True*)

Bases: IPython.irunner.InteractiveRunner

Interactive Python runner.

__init__()

New runner, optionally passing the python command to use.

RunnerFactory

class IPython.irunner.**RunnerFactory**(*out=<open file '<stdout>'*, *mode 'w' at 0x403b2078>*)

Bases: object

Code runner factory.

This class provides an IPython code runner, but enforces that only one runner is ever instantiated. The runner is created based on the extension of the first file to run, and it raises an exception if a runner is later requested for a different extension type.

This ensures that we don't generate example files for doctest with a mix of python and ipython syntax.

__init__()

Instantiate a code runner.

SAGERunner

class IPython.irunner.**SAGERunner**(*program='sage'*, *args=None*, *out=<open file '<stdout>'*, *mode 'w' at 0x403b2078>*, *echo=True*)

Bases: IPython.irunner.InteractiveRunner

Interactive SAGE runner.

WARNING: this runner only works if you manually configure your SAGE copy to use 'colors NoColor' in the ipythonrc config file, since currently the prompt matching regexp does not identify color sequences.

__init__()

New runner, optionally passing the sage command to use.

10.53.3 Functions

IPython.irunner.**main**()
Run as a command-line script.

IPython.irunner.**pexpect_monkeypatch**()
Patch pexpect to prevent unhandled exceptions at VM teardown.

Calling this function will monkeypatch the pexpect.spawn class and modify its __del__ method to make it more robust in the face of failures that can occur if it is called when the Python VM is shutting down.

Since Python may fire __del__ methods arbitrarily late, it's possible for them to execute during the teardown of the Python VM itself. At this point, various builtin modules have been reset to None. Thus, the call to self.close() will trigger an exception because it tries to call os.close(), and os is now None.

10.54 kernel.client

10.54.1 Module: kernel.client

This module contains blocking clients for the controller interfaces.

Unlike the clients in *asyncclient.py*, the clients in this module are fully blocking. This means that methods on the clients return the actual results rather than a deferred to the result. Also, we manage the Twisted reactor for you. This is done by running the reactor in a thread.

The main classes in this module are:

- MultiEngineClient
- TaskClient
- Task
- CompositeError

10.54.2 Functions

IPython.kernel.client.**get_multiengine_client**()
Get the blocking MultiEngine client.

Parameters

furl_or_file [str] A furl or a filename containing a furl. If empty, the default furl_file will be used

Returns The connected MultiEngineClient instance

IPython.kernel.client.**get_task_client**()
Get the blocking Task client.

Parameters

furl_or_file [str] A furl or a filename containing a furl. If empty, the default furl_file will be used

Returns The connected TaskClient instance

10.55 kernel.clientconnector

10.55.1 Module: `kernel.clientconnector`

Inheritance diagram for `IPython.kernel.clientconnector`:

A class for handling client connections to the controller.

10.55.2 `ClientConnector`

class `IPython.kernel.clientconnector.`**`ClientConnector`**

Bases: `object`

This class gets remote references from furls and returns the wrapped clients.

This class is also used in *client.py* and *asyncclient.py* to create a single per client-process Tub.

`__init__``()`

`get_client``()`

Get a remote reference and wrap it in a client by furl.

This method first gets a remote reference and then calls its *get_client_name* method to find the apprpriate client class that should be used to wrap the remote reference.

Parameters

furl_or_file [str] A furl or a filename containing a furl

Returns A deferred to the actual client class

`get_multiengine_client``()`

Get the multiengine controller client.

This method is a simple wrapper around *get_client* that allow *furl_or_file* to be empty, in which case, the furls is taken from the default furl file given in the configuration.

Parameters

furl_or_file [str] A furl or a filename containing a furl. If empty, the default furl_file will be used

Returns A deferred to the actual client class

get_reference()
Get a remote reference using a furl or a file containing a furl.

Remote references are cached locally so once a remote reference has been retrieved for a given furl, the cached version is returned.

Parameters

furl_or_file [str] A furl or a filename containing a furl

Returns A deferred to a remote reference

get_task_client()
Get the task controller client.

This method is a simple wrapper around *get_client* that allow *furl_or_file* to be empty, in which case, the furls is taken from the default furl file given in the configuration.

Parameters

furl_or_file [str] A furl or a filename containing a furl. If empty, the default furl_file will be used

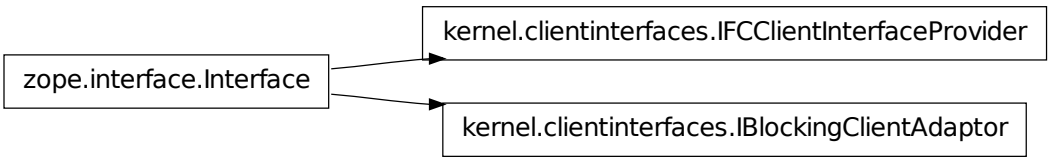
Returns A deferred to the actual client class

save_ref()
Cache a remote reference by its furl.

10.56 kernel.clientinterfaces

10.56.1 Module: kernel.clientinterfaces

Inheritance diagram for IPython.kernel.clientinterfaces:



General client interfaces.

10.56.2 Classes

IBlockingClientAdaptor

class IPython.kernel.clientinterfaces.**IBlockingClientAdaptor**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

IFCClientInterfaceProvider

class IPython.kernel.clientinterfaces.**IFCClientInterfaceProvider**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

10.57 kernel.codeutil

10.57.1 Module: kernel.codeutil

Utilities to enable code objects to be pickled.

Any process that import this module will be able to pickle code objects. This includes the func_code attribute of any function. Once unpickled, new functions can be built using new.function(code, globals()). Eventually we need to automate all of this so that functions themselves can be pickled.

Reference: A. Tremols, P Cogolo, "Python Cookbook," p 302-305

10.57.2 Functions

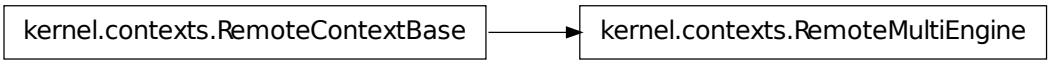
IPython.kernel.codeutil.**code_ctor**()

IPython.kernel.codeutil.**reduce_code**()

10.58 kernel.contexts

10.58.1 Module: kernel.contexts

Inheritance diagram for IPython.kernel.contexts:



Context managers for IPython.

Python 2.5 introduced the *with* statement, which is based on the context manager protocol. This module offers a few context managers for common cases, which can also be useful as templates for writing new, application-specific managers.

10.58.2 Classes

RemoteContextBase

class IPython.kernel.contexts.**RemoteContextBase**
Bases: object

__init__()

findsource()

RemoteMultiEngine

class IPython.kernel.contexts.**RemoteMultiEngine**(*mec*)
Bases: IPython.kernel.contexts.RemoteContextBase

__init__()

10.58.3 Functions

IPython.kernel.contexts.**remote**()
Raises a special exception meant to be caught by context managers.

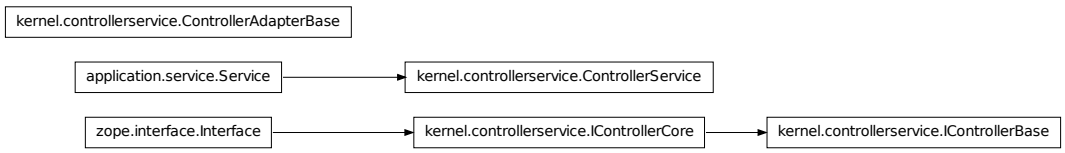
IPython.kernel.contexts.**strip_whitespace**()
strip leading whitespace from input source.

Parameters

10.59 kernel.controllerservice

10.59.1 Module: kernel.controllerservice

Inheritance diagram for IPython.kernel.controllerservice:



A Twisted Service for the IPython Controller.

The IPython Controller:

- Listens for Engines to connect and then manages access to those engines.
- Listens for clients and passes commands from client to the Engines.
- Exposes an asynchronous interfaces to the Engines which themselves can block.
- Acts as a gateway to the Engines.

The design of the controller is somewhat abstract to allow flexibility in how the controller is presented to clients. This idea is that there is a basic ControllerService class that allows engines to connect to it. But, this basic class has no client interfaces. To expose client interfaces developers provide an adapter that makes the ControllerService look like something. For example, one client interface might support task farming and another might support interactive usage. The important thing is that by using interfaces and adapters, a single controller can be accessed from multiple interfaces. Furthermore, by adapting various client interfaces to various network protocols, each client interface can be exposed to multiple network protocols. See multiengine.py for an example of how to adapt the ControllerService to a client interface.

10.59.2 Classes

ControllerAdapterBase

class IPython.kernel.controllerservice.**ControllerAdapterBase**(*controller*)

Bases: object

All Controller adapters should inherit from this class.

This class provides a wrapped version of the IControllerBase interface that can be used to easily create new custom controllers. Subclasses of this will provide a full implementation of IControllerBase.

This class doesn't implement any client notification mechanism. That is up to subclasses.

__init__()

on_n_engines_registered_do()

on_register_engine_do()

on_register_engine_do_not()

on_unregister_engine_do()

on_unregister_engine_do_not()

register_engine()

unregister_engine()

ControllerService

class IPython.kernel.controllerservice.**ControllerService**(*maxEngines=511*, *saveIDs=False*)

Bases: object, twisted.application.service.Service

A basic Controller represented as a Twisted Service.

This class doesn't implement any client notification mechanism. That is up to adapted subclasses.

__init__()

on_n_engines_registered_do()

on_register_engine_do()

on_register_engine_do_not()

on_unregister_engine_do()

on_unregister_engine_do_not()

register_engine()
Register new engine connection

unregister_engine()
Unregister engine by id.

IControllerBase

class IPython.kernel.controllerservice.**IControllerBase**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.controllerservice.IControllerCore

The basic controller interface.

classmethod __init__()

IControllerCore

class IPython.kernel.controllerservice.**IControllerCore**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Basic methods any controller must have.

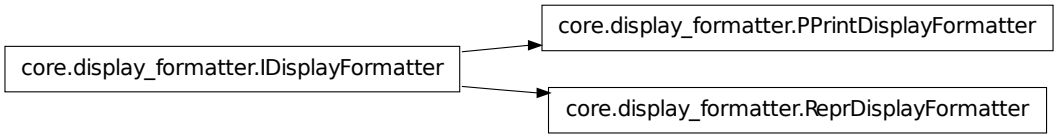
This is basically the aspect of the controller relevant to the engines and does not assume anything about how the engines will be presented to a client.

classmethod __init__()

10.60 kernel.core.display_formatter

10.60.1 Module: kernel.core.display_formatter

Inheritance diagram for IPython.kernel.core.display_formatter:



Objects for replacing sys.displayhook().

10.60.2 Classes

IDisplayFormatter

class IPython.kernel.core.display_formatter.**IDisplayFormatter**

Bases: object

Objects conforming to this interface will be responsible for formatting representations of objects that pass through sys.displayhook() during an interactive interpreter session.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

PPrintDisplayFormatter

class IPython.kernel.core.display_formatter.**PPrintDisplayFormatter**
Bases: IPython.kernel.core.display_formatter.IDisplayFormatter

Return a pretty-printed string representation of an object.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ReprDisplayFormatter

class IPython.kernel.core.display_formatter.**ReprDisplayFormatter**
Bases: IPython.kernel.core.display_formatter.IDisplayFormatter

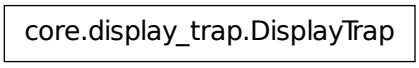
Return the repr() string representation of an object.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.61 kernel.core.display_trap

10.61.1 Module: kernel.core.display_trap

Inheritance diagram for IPython.kernel.core.display_trap:



Manager for replacing sys.displayhook().

10.61.2 DisplayTrap

class IPython.kernel.core.display_trap.**DisplayTrap**(*formatters=None*, *callbacks=None*)
Bases: object

Object to trap and format objects passing through sys.displayhook().

This trap maintains two lists of callables: formatters and callbacks. The formatters take the *last* object that has gone through since the trap was set and returns a string representation. Callbacks are executed on *every* object that passes through the displayhook and does not return anything.

__init__()

add_to_message()
Add the formatted display of the objects to the message dictionary being returned from the interpreter to its listeners.

clear()
Reset the stored object.

hook()
This method actually implements the hook.

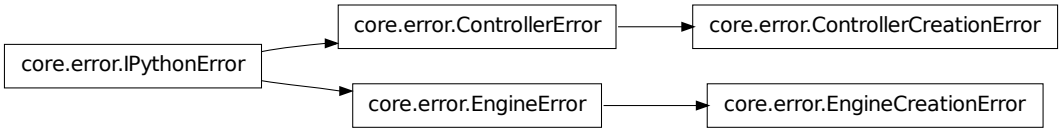
set()
Set the hook.

unset()
Unset the hook.

10.62 kernel.core.error

10.62.1 Module: kernel.core.error

Inheritance diagram for IPython.kernel.core.error:



error.py

We declare here a class hierarchy for all exceptions produced by IPython, in cases where we don't just raise one from the standard library.

10.62.2 Classes

ControllerCreationError

class IPython.kernel.core.error.**ControllerCreationError**
Bases: IPython.kernel.core.error.ControllerError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ControllerError

class IPython.kernel.core.error.**ControllerError**
Bases: IPython.kernel.core.error.IPythonError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

EngineCreationError

class IPython.kernel.core.error.**EngineCreationError**
Bases: IPython.kernel.core.error.EngineError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

EngineError

class IPython.kernel.core.error.**EngineError**
Bases: IPython.kernel.core.error.IPythonError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

IPythonError

class IPython.kernel.core.error.**IPythonError**
Bases: exceptions.Exception

Base exception that all of our exceptions inherit from.

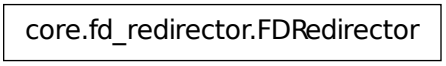
This can be raised by code that doesn't have any more specific information.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.63 kernel.core.fd_redirector

10.63.1 Module: kernel.core.fd_redirector

Inheritance diagram for IPython.kernel.core.fd_redirector:



Stdout/stderr redirector, at the OS level, using file descriptors.

This also works under windows.

10.63.2 FDRedirector

class IPython.kernel.core.fd_redirector.**FDRedirector**(*fd=1*)
Bases: object

Class to redirect output (stdout or stderr) at the OS level using file descriptors.

__init__()
fd is the file descriptor of the outpout you want to capture. It can be STDOUT or STERR.

flush()
Flush the captured output, similar to the flush method of any stream.

getvalue()
Return the output captured since the last getvalue, or the start of the redirection.

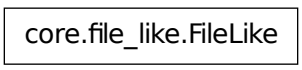
start()
Setup the redirection.

stop()
Unset the redirection and return the captured output.

10.64 kernel.core.file_like

10.64.1 Module: kernel.core.file_like

Inheritance diagram for IPython.kernel.core.file_like:



File like object that redirects its write calls to a given callback.

10.64.2 FileLike

class IPython.kernel.core.file_like.**FileLike**(*write_callback*)
Bases: object

FileLike object that redirects all write to a callback.

Only the write-related methods are implemented, as well as those required to read a StringIO.

__init__()

close()
This method is there for compatibility with other file-like objects.

flush()
This method is there for compatibility with other file-like objects.

getvalue()
This method is there for compatibility with other file-like objects.

isatty()
This method is there for compatibility with other file-like objects.

reset()
This method is there for compatibility with other file-like objects.

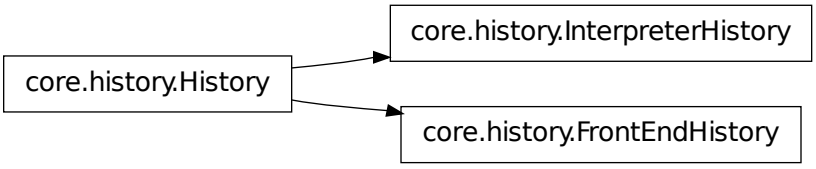
truncate()
This method is there for compatibility with other file-like objects.

writelines()

10.65 kernel.core.history

10.65.1 Module: kernel.core.history

Inheritance diagram for IPython.kernel.core.history:



Manage the input and output history of the interpreter and the frontend.

There are 2 different history objects, one that lives in the interpreter, and one that lives in the frontend. They are synced with a diff at each execution of a command, as the interpreter history is a real stack, its existing entries are not mutable.

10.65.2 Classes

FrontEndHistory

class IPython.kernel.core.history.**FrontEndHistory**(*input_cache=None*, *output_cache=None*)

Bases: IPython.kernel.core.history.History

An object managing the input and output history at the frontend. It is used as a local cache to reduce network latency problems and multiple users editing the same thing.

__init__()

add_items()

Adds the given command list to the stack of executed commands.

History

class IPython.kernel.core.history.**History**(*input_cache=None*, *output_cache=None*)

Bases: object

An object managing the input and output history.

__init__()

get_history_item()

Returns the history string at index, where index is the distance from the end (positive).

InterpreterHistory

class IPython.kernel.core.history.**InterpreterHistory**(*input_cache=None*, *output_cache=None*)

Bases: IPython.kernel.core.history.History

An object managing the input and output history at the interpreter level.

__init__()

get_history_item()

Returns the history string at index, where index is the distance from the end (positive).

get_input_after()

Returns the list of the commands entered after index.

get_input_cache()

setup_namespace()
Add the input and output caches into the interpreter's namespace with IPython-conventional names.

Parameters **namespace** : dict

update_history()
Update the history objects that this object maintains and the interpreter's namespace.

Parameters **interpreter** : Interpreter

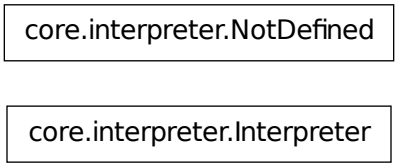
python : str

The real Python code that was translated and actually executed.

10.66 kernel.core.interpreter

10.66.1 Module: kernel.core.interpreter

Inheritance diagram for IPython.kernel.core.interpreter:



Central interpreter object for an IPython engine.

The interpreter is the object whose job is to process lines of user input and actually execute them in the user's namespace.

10.66.2 Classes

Interpreter

class IPython.kernel.core.interpreter.**Interpreter**(*user_ns=None*, *global_ns=None*, *translator=None*, *magic=None*, *display_formatters=None*, *traceback_formatters=None*, *output_trap=None*, *history=None*, *message_cache=None*, *filename='<string>'*, *config=None*)

Bases: object

An interpreter object.

fixme: needs to negotiate available formatters with frontends.

Important: the interpeter should be built so that it exposes a method for each attribute/method of its sub-object. This way it can be replaced by a network adapter.

__init__()

complete()
Complete the given text.

Parameters

text [str] Text fragment to be completed on. Typically this is

error()
Pass an error message back to the shell.

Parameters **text** : str

Notes

This should only be called when self.message is set. In other words, when code is being executed.

execute()
Execute some IPython commands.

1. Translate them into Python.
2. Run them.
3. Trap stdout/stderr.
4. Trap sys.displayhook().
5. Trap exceptions.

6.Return a message object.

Parameters **commands** : str

The raw commands that the user typed into the prompt.

Returns **message** : dict

The dictionary of responses. See the README.txt in this directory for an explanation of the format.

execute_block()

Execute a single block of code in the user namespace.

Return value: a flag indicating whether the code to be run completed successfully:

- 0: successful execution.
- 1: an error occurred.

execute_macro()

Execute the value of a macro.

Parameters **macro** : Macro

execute_python()

Actually run the Python code in the namespace.

Parameters

python [str] Pure, exec'able Python code. Special IPython commands should have already been translated into pure Python.

feed_block()

Compile some source in the interpreter.

One several things can happen:

1) The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError).

2) The input is incomplete, and more input is required; compile_command() returned None. Nothing happens.

3) The input is complete; compile_command() returned a code object. The code is executed by calling self.runcode() (which also handles run-time exceptions, except for SystemExit).

The return value is:

- True in case 2
- False in the other cases, unless an exception is raised, where

None is returned instead. This can be used by external callers to know whether to continue feeding input or not.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

formatTraceback()
Put a formatted version of the traceback into value and reraise.

When exceptions have to be sent over the network, the traceback needs to be put into the value of the exception in a nicely formatted way. The method takes the type, value and tb of an exception and puts a string representation of the tb into the value of the exception and reraises it.

Currently this method uses the ultraTb formatter from IPython trunk. Eventually it should simply use the traceback formatters in core that are loaded into self.tracback_trap.formatters.

generate_prompt()
Calculate and return a string with the prompt to display.

Parameters

is_continuation [bool] Whether the input line is continuing multiline input or not, so

that a proper continuation prompt can be computed.

getCommand()
Gets the ith message in the message_cache.

This is implemented here for compatibility with the old ipython1 shell I am not sure we need this though. I even seem to remember that we were going to get rid of it.

ipmagic()
Call a magic function by name.

ipmagic('name -opt foo bar') is equivalent to typing at the ipython prompt:

In[1]: %name -opt foo bar

To call a magic without arguments, simply use ipmagic('name').

This provides a proper Python function to call IPython's magics in any valid Python code you can type at the interpreter, including loops and compound statements. It is added by IPython to the Python builtin namespace upon initialization.

Parameters **arg_string** : str

A string containing the name of the magic function to call and any additional arguments to be passed to the magic.

Returns **something** : object

The return value of the actual object.

ipsystem()
Execute a command in a system shell while expanding variables in the current namespace.

Parameters **command** : str

pack_exception()

pull()
Get an item out of the namespace by key.

Parameters **key** : str

Returns **value** : object

Raises **TypeError if the key is not a string.** :

NameError if the object doesn't exist. :

pull_function()

push()
Put value into the namespace with name key.

Parameters ****kwds** :

push_function()

reset()
Reset the interpreter.

Currently this only resets the users variables in the namespace. In the future we might want to also reset the other stateful things like that the Interpreter has, like In, Out, etc.

set_traps()
Set all of the output, display, and traceback traps.

setup_message()
Return a message object.

This method prepares and returns a message dictionary. This dict contains the various fields that are used to transfer information about execution, results, tracebacks, etc, to clients (either in or out of process ones). Because of the need to work with possibly out of process clients, this dict MUST contain strictly pickle-safe values.

setup_namespace()
Add things to the namespace.

split_commands()
Split multiple lines of code into discrete commands that can be executed singly.

Parameters **python** : str

Pure, exec'able Python code.

Returns **commands** : list of str

Separate commands that can be exec'ed independently.

unset_traps()
Unset all of the output, display, and traceback traps.

var_expand()
Expand $variables in the current namespace using Itpl.

Parameters **template** : str

NotDefined

class IPython.kernel.core.interpreter.**NotDefined**
Bases: object

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.66.3 Functions

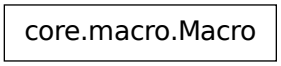
IPython.kernel.core.interpreter.**default_display_formatters**()
Return a list of default display formatters.

IPython.kernel.core.interpreter.**default_traceback_formatters**()
Return a list of default traceback formatters.

10.67 kernel.core.macro

10.67.1 Module: kernel.core.macro

Inheritance diagram for IPython.kernel.core.macro:



Support for interactive macros in IPython

10.67.2 Macro

class IPython.kernel.core.macro.**Macro**(*data*)
Simple class to store the value of macros as strings.

This allows us to later exec them by checking when something is an instance of this class.

__init__()

10.68 kernel.core.magic

10.68.1 Module: kernel.core.magic

Inheritance diagram for IPython.kernel.core.magic:

10.68.2 Magic

class IPython.kernel.core.magic.**Magic**(*interpreter*, *config=None*)
Bases: object

An object that maintains magic functions.

__init__()

has_magic()
Return True if this object provides a given magic.

Parameters **name** : str

magic_env()
List environment variables.

magic_pwd()
Return the current working directory path.

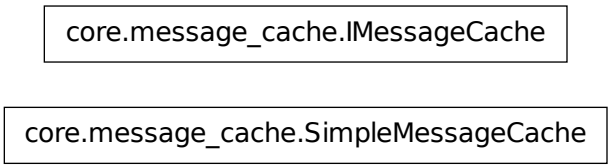
object_find()
Find an object in the available namespaces.

fixme: this should probably be moved elsewhere. The interpreter?

10.69 kernel.core.message_cache

10.69.1 Module: kernel.core.message_cache

Inheritance diagram for IPython.kernel.core.message_cache:



Storage for the responses from the interpreter.

10.69.2 Classes

IMessageCache

class IPython.kernel.core.message_cache.**IMessageCache**
Bases: object

Storage for the response from the interpreter.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

add_message()
Add a message dictionary to the cache.

Parameters **i** : int

message : dict

get_message()
Get the message from the cache.

Parameters **i** : int, optional

The number of the message. If not provided, return the highest-numbered message.

Returns **message** : dict

Raises **IndexError if the message does not exist in the cache.** :

SimpleMessageCache

class IPython.kernel.core.message_cache.**SimpleMessageCache**
Bases: object

Simple dictionary-based, in-memory storage of the responses from the interpreter.

__init__()

add_message()
Add a message dictionary to the cache.

Parameters **i** : int

message : dict

get_message()
Get the message from the cache.

Parameters **i** : int, optional

The number of the message. If not provided, return the highest-numbered message.

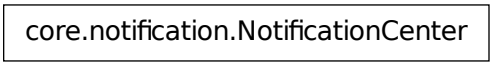
Returns **message** : dict

Raises **IndexError if the message does not exist in the cache.** :

10.70 kernel.core.notification

10.70.1 Module: kernel.core.notification

Inheritance diagram for IPython.kernel.core.notification:



The IPython Core Notification Center.

See docs/source/development/notification_blueprint.txt for an overview of the notification module.

10.70.2 NotificationCenter

class IPython.kernel.core.notification.**NotificationCenter**

Bases: object

Synchronous notification center

Examples

```
>>> import IPython.kernel.core.notification as notification
>>> def callback(theType, theSender, args={}):
...     print theType,theSender,args
...
>>> notification.sharedCenter.add_observer(callback, 'NOTIFICATION_TYPE', None)
>>> notification.sharedCenter.post_notification('NOTIFICATION_TYPE', object())
NOTIFICATION_TYPE ...
```

__init__()

add_observer()

Add an observer callback to this notification center.

The given callback will be called upon posting of notifications of the given type/sender and will receive any additional kwargs passed to post_notification.

Parameters **observerCallback** : callable

Callable. Must take at least two arguments:: observerCallback(type, sender, args={})

theType : hashable

The notification type. If None, all notifications from sender will be posted.

sender : hashable

The notification sender. If None, all notifications of theType will be posted.

post_notification()
Post notification (type,sender,**kwargs) to all registered observers.

Implementation notes:

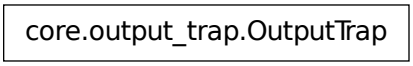
- If no registered observers, performance is O(1).
- Notificaiton order is undefined.
- Notifications are posted synchronously.

remove_all_observers()
Removes all observers from this notification center

10.71 kernel.core.output_trap

10.71.1 Module: kernel.core.output_trap

Inheritance diagram for IPython.kernel.core.output_trap:



Trap stdout/stderr.

10.71.2 OutputTrap

class IPython.kernel.core.output_trap.**OutputTrap**(*out=None*, *err=None*)
Bases: object

Object which can trap text sent to stdout and stderr.

__init__()

add_to_message()
Add the text from stdout and stderr to the message from the interpreter to its listeners.

Parameters **message** : dict

clear()
Clear out the buffers.

err_text
Return the text currently in the stderr buffer.

out_text
Return the text currently in the stdout buffer.

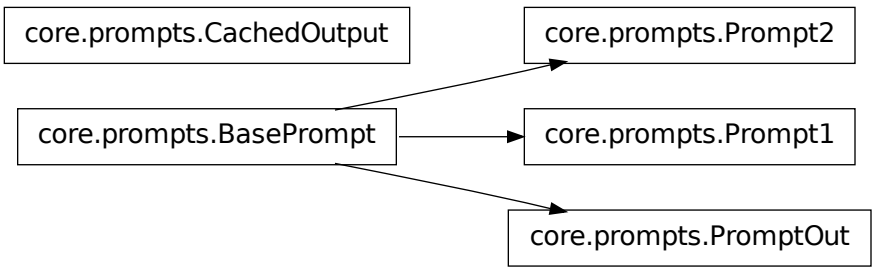
set()
Set the hooks.

unset()
Remove the hooks.

10.72 kernel.core.prompts

10.72.1 Module: kernel.core.prompts

Inheritance diagram for IPython.kernel.core.prompts:



Classes for handling input/output prompts.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

10.72.2 Classes

BasePrompt

class IPython.kernel.core.prompts.**BasePrompt**(*cache*, *sep*, *prompt*, *pad_left=False*)
Bases: object

Interactive prompt similar to Mathematica's.

__init__()

cwd_filt()
Return the last depth elements of the current working directory.

$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2()
Return the last depth elements of the current working directory.

$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template
Template for prompt string creation

set_p_str()
Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write()

CachedOutput

class IPython.kernel.core.prompts.**CachedOutput**(*shell*, *cache_size*, *Pprint*, *colors='NoColor'*, *input_sep='n'*, *output_sep='n'*, *output_sep2=''*, *ps1=None*, *ps2=None*, *ps_out=None*, *pad_left=True*)
Class for printing output from calculations while keeping a cache of reults. It dynamically creates global variables prefixed with _ which contain these results.

Meant to be used as a sys.displayhook replacement, providing numbered prompts and cache services.

Initialize with initial and final values for cache counter (this defines the maximum size of the cache.

__init__()

display()
Default printer method, uses pprint.

Do ip.set_hook("result_display", my_displayhook) for custom result display, e.g. when your own objects need special formatting.

flush()

set_colors()
Set the active color scheme and configure colors for the three prompt subsystems.

update()

Prompt1

class IPython.kernel.core.prompts.**Prompt1**(*cache*, *sep='n'*, *prompt='In[, \#], : '*, *pad_left=True*)
Bases: IPython.kernel.core.prompts.BasePrompt

Input interactive prompt similar to Mathematica's.

__init__()

auto_rewrite()
Print a string of the form '—>' which lines up with the previous input string. Useful for systems which re-write the user input when handling automatically special syntaxes.

set_colors()

Prompt2

class IPython.kernel.core.prompts.**Prompt2**(*cache*, *prompt=' .\D.: '*, *pad_left=True*)
Bases: IPython.kernel.core.prompts.BasePrompt

Interactive continuation prompt.

__init__()

set_colors()

set_p_str()

PromptOut

class IPython.kernel.core.prompts.**PromptOut**(*cache*, *sep=''*, *prompt='Out[, \#], : '*, *pad_left=True*)
Bases: IPython.kernel.core.prompts.BasePrompt

Output interactive prompt similar to Mathematica's.

__init__()

set_colors()

10.72.3 Functions

IPython.kernel.core.prompts.**multiple_replace**()
Replace in 'text' all occurences of any key in the given dictionary by its corresponding value. Returns the new string.

IPython.kernel.core.prompts.**str_safe**()
Convert to a string, without ever raising an exception.

If str(arg) fails, <ERROR: ... > is returned, where ... is the exception error message.

10.73 kernel.core.redirector_output_trap

10.73.1 Module: kernel.core.redirector_output_trap

Inheritance diagram for IPython.kernel.core.redirector_output_trap:

Trap stdout/stderr, including at the OS level. Calls a callback with the output each time Python tries to write to the stdout or stderr.

10.73.2 RedirectorOutputTrap

class IPython.kernel.core.redirector_output_trap.**RedirectorOutputTrap**(*out_callback*, *err_callback*)

Bases: IPython.kernel.core.output_trap.OutputTrap

Object which can trap text sent to stdout and stderr.

__init__()
out_callback : callable called when there is output in the stdout err_callback : callable called when there is output in the stderr

on_err_write()
Callback called when there is some Python output on stderr.

on_out_write()
Callback called when there is some Python output on stdout.

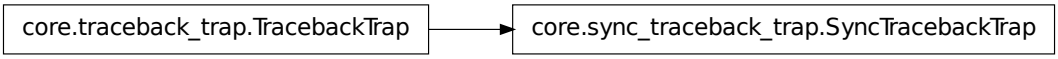
set()
Set the hooks: set the redirectors and call the base class.

unset()
Remove the hooks: call the base class and stop the redirectors.

10.74 kernel.core.sync_traceback_trap

10.74.1 Module: kernel.core.sync_traceback_trap

Inheritance diagram for IPython.kernel.core.sync_traceback_trap:



Object to manage sys.excepthook().

Synchronous version: prints errors when called.

10.74.2 SyncTracebackTrap

class IPython.kernel.core.sync_traceback_trap.**SyncTracebackTrap**(*sync_formatter=None*, *formatters=None*, *raise_exception=True*)

Bases: IPython.kernel.core.traceback_trap.TracebackTrap

TracebackTrap that displays immediatly the traceback in addition to capturing it. Useful in frontends, as without this traceback trap, some tracebacks never get displayed.

__init__()

sync_formatter: Callable to display the traceback. formatters: A list of formatters to apply.

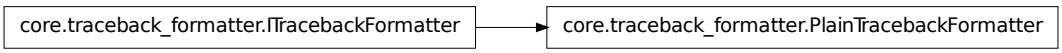
hook()

This method actually implements the hook.

10.75 kernel.core.traceback_formatter

10.75.1 Module: kernel.core.traceback_formatter

Inheritance diagram for IPython.kernel.core.traceback_formatter:



Some formatter objects to extract traceback information by replacing sys.excepthook().

10.75.2 Classes

ITracebackFormatter

class IPython.kernel.core.traceback_formatter.**ITracebackFormatter**
Bases: object

Objects conforming to this interface will format tracebacks into other objects.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

PlainTracebackFormatter

class IPython.kernel.core.traceback_formatter.**PlainTracebackFormatter**(*limit=None*)
Bases: IPython.kernel.core.traceback_formatter.ITracebackFormatter

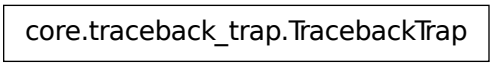
Return a string with the regular traceback information.

__init__()

10.76 kernel.core.traceback_trap

10.76.1 Module: kernel.core.traceback_trap

Inheritance diagram for IPython.kernel.core.traceback_trap:



Object to manage sys.excepthook().

10.76.2 TracebackTrap

class IPython.kernel.core.traceback_trap.**TracebackTrap**(*formatters=None*)
Bases: object

Object to trap and format tracebacks.

__init__()

add_to_message()
Add the formatted display of the traceback to the message dictionary being returned from the interpreter to its listeners.

Parameters **message** : dict

clear()
Remove the stored traceback.

hook()
This method actually implements the hook.

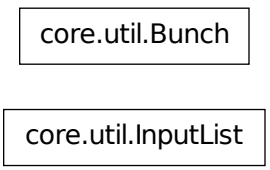
set()
Set the hook.

unset()
Unset the hook.

10.77 kernel.core.util

10.77.1 Module: kernel.core.util

Inheritance diagram for IPython.kernel.core.util:



10.77.2 Classes

Bunch

class IPython.kernel.core.util.**Bunch**(**args*, ***kwds*)
Bases: dict

A dictionary that exposes its keys as attributes.

__init__()

InputList

class IPython.kernel.core.util.**InputList**
Bases: list

Class to store user input.

It's basically a list, but slices return a string instead of a list, thus allowing things like (assuming 'In' is an instance):

exec In[4:7]

or

exec In[5:9] + In[14] + In[21:25]

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

add()
Add a command to the list with the appropriate index.

If the index is greater than the current length of the list, empty strings are added in between.

10.77.3 Functions

IPython.kernel.core.util.**esc_quotes**()
Return the input string with single and double quotes escaped out.

IPython.kernel.core.util.**getoutputerror**()
Executes a command and returns the output.

Parameters **cmd** : str

The command to execute.

verbose : bool

If True, print the command to be executed.

debug : bool

Only print, do not actually execute.

header : str

Header to print to screen prior to the executed command. No extra newlines are added.

split : bool

If True, return the output as a list split on newlines.

IPython.kernel.core.util.**make_quoted_expr**()
Return string s in appropriate quotes, using raw string if possible.

XXX - example removed because it caused encoding errors in documentation generation. We need a new example that doesn't contain invalid chars.

Note the use of raw string and padding at the end to allow trailing backslash.

IPython.kernel.core.util.**system_shell**()
Execute a command in the system shell; always return None.

This returns None so it can be conveniently used in interactive loops without getting the return value (typically 0) printed many times.

Parameters **cmd** : str

The command to execute.

verbose : bool

If True, print the command to be executed.

debug : bool

Only print, do not actually execute.

header : str

Header to print to screen prior to the executed command. No extra newlines are added.

10.78 kernel.engineconnector

10.78.1 Module: kernel.engineconnector

Inheritance diagram for IPython.kernel.engineconnector:

A class that manages the engines connection to the controller.

10.78.2 EngineConnector

class IPython.kernel.engineconnector.**EngineConnector**(*tub*)
Bases: object

Manage an engines connection to a controller.

This class takes a foolscap *Tub* and provides a *connect_to_controller* method that will use the *Tub* to connect to a controller and register the engine with the controller.

__init__()

connect_to_controller()

Make a connection to a controller specified by a furl.

This method takes an *IEngineBase* instance and a foolcap URL and uses the *tub* attribute to make a connection to the controller. The foolscap URL contains all the information needed to connect to the controller, including the ip and port as well as any encryption and authentication information needed for the connection.

After getting a reference to the controller, this method calls the *register_engine* method of the controller to actually register the engine.

Parameters

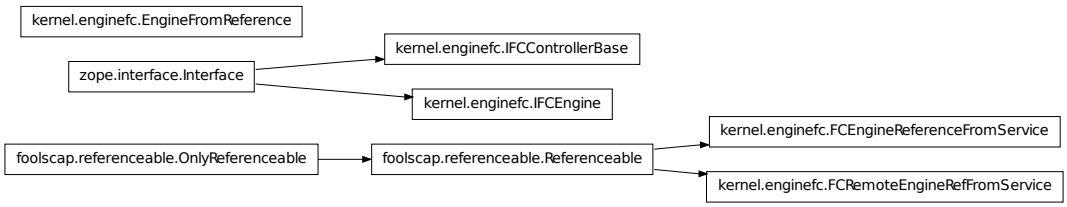
engine_service [IEngineBase] An instance of an *IEngineBase* implementer

furl_or_file [str] A furl or a filename containing a furl

10.79 kernel.enginefc

10.79.1 Module: kernel.enginefc

Inheritance diagram for IPython.kernel.enginefc:



Expose the IPython EngineService using the Foolscap network protocol.

Foolscap is a high-performance and secure network protocol.

10.79.2 Classes

EngineFromReference

class IPython.kernel.enginefc.**EngineFromReference**(*reference*)

Bases: object

Adapt a *RemoteReference* to an *IEngineBase* implementing object.

When an engine connects to a controller, it calls the *register_engine* method of the controller and passes the controller a *RemoteReference* to itself. This class is used to adapt this *RemoteReference* to an object that implements the full *IEngineBase* interface.

See the documentation of *IEngineBase* for details on the methods.

__init__()

callRemote()

checkReturnForFailure()
See if a returned value is a pickled Failure object.

To distinguish between general pickled objects and pickled Failures, the other side should prepend the string FAILURE: to any pickled Failure.

clear_properties()

del_properties()

execute()

get_id()
Return the Engines id.

get_properties()

get_result()

has_properties()

id
Return the Engines id.

keys()

kill()

killBack()

properties

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()
Older version of pushSerialize.

reset()

set_id()
Set the Engines id.

set_properties()

syncProperties()

FCEngineReferenceFromService

class IPython.kernel.enginefc.**FCEngineReferenceFromService**(*service*)
Bases: foolscap.referenceable.Referenceable, object

Adapt an *IEngineBase* to an *IFCEngine* implementer.

This exposes an *IEngineBase* to foolscap by adapting it to a *foolscap.Referenceable*.

See the documentation of the *IEngineBase* methods for more details.

__init__()

remote_clear_properties()

remote_del_properties()

remote_execute()

remote_get_id()

remote_get_properties()

remote_get_result()

remote_has_properties()

remote_keys()

remote_kill()

remote_pull()

remote_pull_function()

remote_pull_serialized()

remote_push()

remote_push_function()

remote_push_serialized()

remote_reset()

remote_set_id()

remote_set_properties()

FCRemoteEngineRefFromService

class IPython.kernel.enginefc.**FCRemoteEngineRefFromService**(*service*)
Bases: foolscap.referenceable.Referenceable

Adapt an *IControllerBase* to an *IFCControllerBase*.

__init__()

remote_register_engine()

IFCControllerBase

class IPython.kernel.enginefc.**IFCControllerBase**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Interface that tells how an Engine sees a Controller.

In our architecture, the Controller listens for Engines to connect and register. This interface defines that registration method as it is exposed over the Foolscap network protocol

classmethod __init__()

IFCEngine

class IPython.kernel.enginefc.**IFCEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

An interface that exposes an EngineService over Foolscap.

The methods in this interface are similar to those from IEngine, but their arguments and return values slightly different to reflect that FC cannot send arbitrary objects. We handle this by pickling/unpickling that the two endpoints.

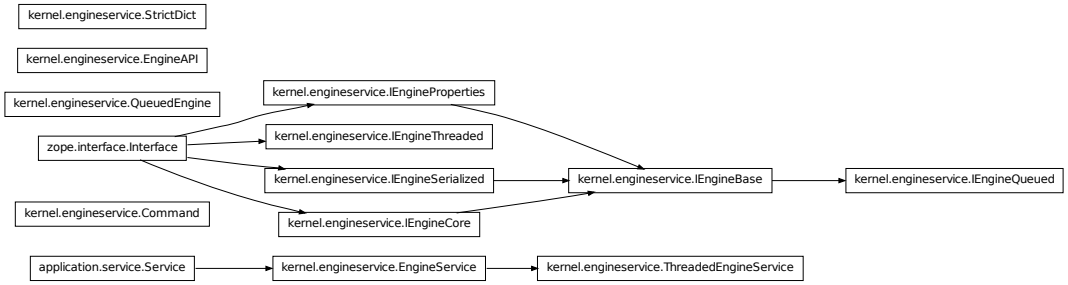
If a remote or local exception is raised, the appropriate Failure will be returned instead.

classmethod __init__()

10.80 kernel.engineservice

10.80.1 Module: kernel.engineservice

Inheritance diagram for IPython.kernel.engineservice:



A Twisted Service Representation of the IPython core.

The IPython Core exposed to the network is called the Engine. Its representation in Twisted in the EngineService. Interfaces and adapters are used to abstract out the details of the actual network protocol used. The EngineService is an Engine that knows nothing about the actual protocol used.

The EngineService is exposed with various network protocols in modules like:

enginepb.py enginevanilla.py

As of 12/12/06 the classes in this module have been simplified greatly. It was felt that we had over-engineered things. To improve the maintainability of the code we have taken out the ICompleteEngine interface and the completeEngine method that automatically added methods to engines.

10.80.2 Classes

Command

class IPython.kernel.engineservice.**Command**(*remoteMethod*, **args*, ***kwargs*)
Bases: object

A command object that encapslates queued commands.

This class basically keeps track of a command that has been queued in a QueuedEngine. It manages the deferreds and hold the method to be called and the arguments to that method.

__init__()
Build a new Command object.

handleError()
When an error has occured, relay it to self.deferred.

handleResult()
When the result is ready, relay it to self.deferred.

setDeferred()
Sets the deferred attribute of the Command.

EngineAPI

class IPython.kernel.engineservice.**EngineAPI**(*id*)
Bases: object

This is the object through which the user can edit the *properties* attribute of an Engine. The Engine Properties object copies all object in and out of itself. See the EngineProperties object for details.

__init__()

EngineService

class IPython.kernel.engineservice.**EngineService**(*shellClass=<class 'IPython.kernel.core.interpreter.Interpreter'>*, *mpi=None*)
Bases: object, twisted.application.service.Service

Adapt a IPython shell into a IEngine implementing Twisted Service.

__init__()
Create an EngineService.

shellClass: something that implements IInterpreter or core1 mpi: an mpi module that has rank and size attributes

addIDToResult()

clear_properties()

del_properties()

execute()

executeAndRaise()
Call a method of self.shell and wrap any exception.

get_properties()

get_result()

has_properties()

id

keys()
Return a list of variables names in the users top level namespace.

This used to return a dict of all the keys/repr(values) in the user's namespace. This was too much info for the ControllerService to handle so it is now just a list of keys.

kill()

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()

reset()

set_properties()

IEngineBase

class IPython.kernel.engineservice.**IEngineBase**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.engineservice.IEngineCore, IPython.kernel.engineservice.IEngineSerialized, IPython.kernel.engineservice.IEngineProperties

The basic engine interface that EngineService will implement.

This exists so it is easy to specify adapters that adapt to and from the API that the basic EngineService implements.

classmethod __init__()

IEngineCore

class IPython.kernel.engineservice.**IEngineCore**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

The minimal required interface for the IPython Engine.

This interface provides a formal specification of the IPython core. All these methods should return deferreds regardless of what side of a network connection they are on.

In general, this class simply wraps a shell class and wraps its return values as Deferred objects. If the underlying shell class method raises an exception, this class should convert it to a twisted.failure.Failure that will be propagated along the Deferred's errback chain.

In addition, Failures are aggressive. By this, we mean that if a method is performing multiple actions (like pulling multiple object) if any single one fails, the entire method will fail with that Failure. It is all or nothing.

classmethod __init__()

IEngineProperties

class IPython.kernel.engineservice.**IEngineProperties**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Methods for access to the properties object of an Engine

classmethod __init__()

IEngineQueued

class IPython.kernel.engineservice.**IEngineQueued**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.engineservice.IEngineBase

Interface for adding a queue to an IEngineBase.

This interface extends the IEngineBase interface to add methods for managing the engine's queue. The implicit details of this interface are that the execution of all methods declared in IEngineBase should appropriately be put through a queue before execution.

All methods should return deferreds.

classmethod __init__()

IEngineSerialized

class IPython.kernel.engineservice.**IEngineSerialized**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Push/Pull methods that take Serialized objects.

All methods should return deferreds.

classmethod __init__()

IEngineThreaded

class IPython.kernel.engineservice.**IEngineThreaded**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

A place holder for threaded commands.

All methods should return deferreds.

classmethod __init__()

QueuedEngine

class IPython.kernel.engineservice.**QueuedEngine**(*engine*)
Bases: object

Adapt an IEngineBase to an IEngineQueued by wrapping it.

The resulting object will implement IEngineQueued which extends IEngineCore which extends (IEngineBase, IEngineSerialized).

This seems like the best way of handling it, but I am not sure. The other option is to have the various base interfaces be used like mix-in intefaces. The problem I have with this is adpatation is more difficult and complicated because there can be can multiple original and final Interfaces.

__init__()
Create a QueuedEngine object from an engine

engine: An implementor of IEngineCore and IEngineSerialized keepUpToDate: whether to update the remote status when the

queue is empty. Defaults to False.

abortCommand()
Abort current command.

This eats the Failure but first passes it onto the Deferred that the user has.

It also clear out the queue so subsequence commands don't run.

clear_properties()

clear_queue()
Clear the queue, but doesn't cancel the currently running commmand.

del_properties()

execute()

finishCommand()
Finish currrent command.

get_properties()

get_result()

has_properties()

keys()

kill()

properties

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()

queue_status()

register_failure_observer()

reset()

runCurrentCommand()
Run current command.

saveResult()
Put the result in the history.

set_properties()

submitCommand()
Submit command to queue.

unregister_failure_observer()

StrictDict

class IPython.kernel.engineservice.**StrictDict**(**args*, ***kwargs*)
Bases: dict

This is a strict copying dictionary for use as the interface to the properties of an Engine.

Important This object copies the values you set to it, and returns copies to you when you request them. The only way to change properties os explicitly through the setitem and getitem of the dictionary interface.

Example:

```
>>> e = get_engine(id)
>>> L = [1,2,3]
>>> e.properties['L'] = L
>>> L == e.properties['L']
True
>>> L.append(99)
>>> L == e.properties['L']
False
```

Note that getitem copies, so calls to methods of objects do not affect the properties, as seen here:

```
>>> e.properties[1] = range(2)
>>> print e.properties[1]
[0, 1]
```

```
>>> e.properties[1].append(2)
>>> print e.properties[1]
[0, 1]
```

__init__()

clear()

pop()

popitem()

subDict()

update()

ThreadedEngineService

class IPython.kernel.engineservice.**ThreadedEngineService**(*shellClass=<class 'IPython.kernel.core.interpreter.Interprete*, *mpi=None*)

Bases: IPython.kernel.engineservice.EngineService

An EngineService subclass that defers execute commands to a separate thread.

ThreadedEngineService uses twisted.internet.threads.deferToThread to defer execute requests to a separate thread. GUI frontends may want to use ThreadedEngineService as the engine in an IPython.frontend.frontendbase.FrontEndBase subclass to prevent block execution from blocking the GUI thread.

__init__()

execute()

wrapped_execute()
Wrap self.shell.execute to add extra information to tracebacks

10.80.3 Functions

IPython.kernel.engineservice.**drop_engine**()
remove an engine

IPython.kernel.engineservice.**get_engine**()
Get the Engine API object, whcih currently just provides the properties object, by ID

IPython.kernel.engineservice.**queue**()

10.81 kernel.error

10.81.1 Module: `kernel.error`

Inheritance diagram for `IPython.kernel.error`:

Classes and functions for kernel related errors and exceptions.

10.81.2 Classes

AbortedPendingDeferredError

class IPython.kernel.error.**AbortedPendingDeferredError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ClientError

class IPython.kernel.error.**ClientError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

CompositeError

class IPython.kernel.error.**CompositeError**(*message*, *elist*)
Bases: IPython.kernel.error.KernelError

__init__()

print_tracebacks()

raise_exception()

ConnectionError

class IPython.kernel.error.**ConnectionError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

FileTimeoutError

class IPython.kernel.error.**FileTimeoutError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

IdInUse

class IPython.kernel.error.**IdInUse**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

InvalidClientID

class IPython.kernel.error.**InvalidClientID**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

InvalidDeferredID

class IPython.kernel.error.**InvalidDeferredID**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

InvalidEngineID

class IPython.kernel.error.**InvalidEngineID**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

InvalidProperty

class IPython.kernel.error.**InvalidProperty**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

KernelError

class IPython.kernel.error.**KernelError**
Bases: IPython.kernel.core.error.IPythonError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

MessageSizeError

class IPython.kernel.error.**MessageSizeError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

MissingBlockArgument

class IPython.kernel.error.**MissingBlockArgument**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

NoEnginesRegistered

class IPython.kernel.error.**NoEnginesRegistered**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

NotAPendingResult

class IPython.kernel.error.**NotAPendingResult**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

NotDefined

class IPython.kernel.error.**NotDefined**(*name*)
Bases: IPython.kernel.error.KernelError

__init__()

PBMessageSizeError

class IPython.kernel.error.**PBMessageSizeError**
Bases: IPython.kernel.error.MessageSizeError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ProtocolError

class IPython.kernel.error.**ProtocolError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

QueueCleared

class IPython.kernel.error.**QueueCleared**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ResultAlreadyRetrieved

class IPython.kernel.error.**ResultAlreadyRetrieved**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ResultNotCompleted

class IPython.kernel.error.**ResultNotCompleted**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

SecurityError

class IPython.kernel.error.**SecurityError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

SerializationError

class IPython.kernel.error.**SerializationError**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

StopLocalExecution

class IPython.kernel.error.**StopLocalExecution**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

TaskAborted

class IPython.kernel.error.**TaskAborted**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

TaskRejectError

class IPython.kernel.error.**TaskRejectError**
Bases: IPython.kernel.error.KernelError

Exception to raise when a task should be rejected by an engine.

This exception can be used to allow a task running on an engine to test if the engine (or the user's namespace on the engine) has the needed task dependencies. If not, the task should raise this exception. For the task to be retried on another engine, the task should be created with the *retries* argument > 1.

The advantage of this approach over our older properties system is that tasks have full access to the user's namespace on the engines and the properties don't have to be managed or tested by the controller.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

TaskTimeout

class IPython.kernel.error.**TaskTimeout**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

UnpickleableException

class IPython.kernel.error.**UnpickleableException**
Bases: IPython.kernel.error.KernelError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.81.3 Function

IPython.kernel.error.**collect_exceptions**()

10.82 kernel.fcutil

10.82.1 Module: kernel.fcutil

Foolscap related utilities.

10.82.2 Functions

IPython.kernel.fcutil.**check_furl_file_security**()
Remove the old furl_file if changing security modes.

IPython.kernel.fcutil.**find_furl**()

IPython.kernel.fcutil.**is_secure**()

IPython.kernel.fcutil.**is_valid**()

10.83 kernel.magic

10.83.1 Module: kernel.magic

Magic command interface for interactive parallel work.

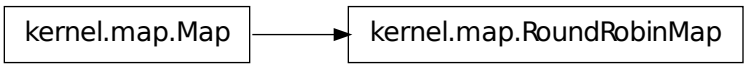
10.83.2 Functions

IPython.kernel.magic.**pxrunsource**()

10.84 kernel.map

10.84.1 Module: kernel.map

Inheritance diagram for IPython.kernel.map:



Classes used in scattering and gathering sequences.

Scattering consists of partitioning a sequence and sending the various pieces to individual nodes in a cluster.

10.84.2 Classes

Map

class IPython.kernel.map.**Map**
A class for partitioning a sequence using a map.

concatenate()

getPartition()
Returns the pth partition of q partitions of seq.

joinPartitions()

RoundRobinMap

class IPython.kernel.map.**RoundRobinMap**
Bases: IPython.kernel.map.Map

Partitions a sequence in a roun robin fashion.

This currently does not work!

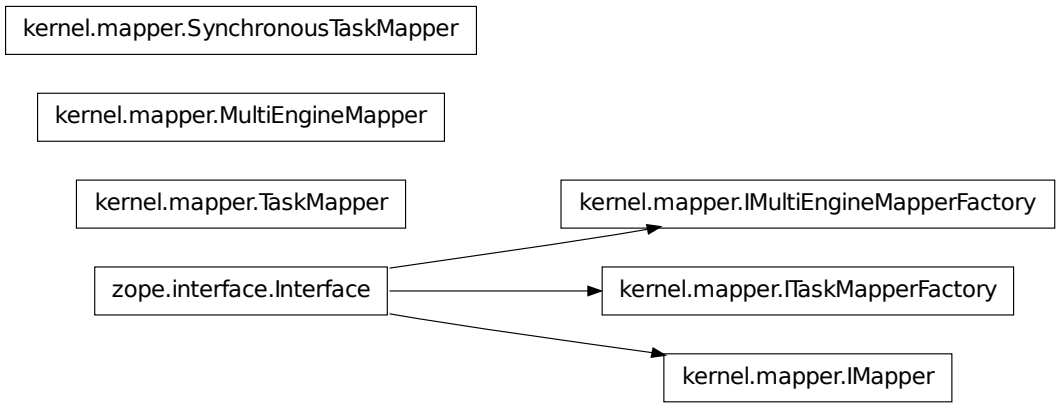
getPartition()

joinPartitions()

10.85 kernel.mapper

10.85.1 Module: kernel.mapper

Inheritance diagram for IPython.kernel.mapper:



A parallelized version of Python's builtin map.

10.85.2 Classes

IMapper

class IPython.kernel.mapper.**IMapper**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

The basic interface for a Mapper.

This defines a generic interface for mapping. The idea of this is similar to that of Python's builtin *map* function, which applies a function elementwise to a sequence.

classmethod __init__()

IMultiEngineMapperFactory

class IPython.kernel.mapper.**IMultiEngineMapperFactory**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

An interface for something that creates *IMapper* instances.

classmethod __init__()

ITaskMapperFactory

class IPython.kernel.mapper.**ITaskMapperFactory**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

An interface for something that creates *IMapper* instances.

classmethod __init__()

MultiEngineMapper

class IPython.kernel.mapper.**MultiEngineMapper**(*multiengine*, *dist='b'*, *targets='all'*, *block=True*)

Bases: object

A Mapper for *IMultiEngine* implementers.

__init__()

Create a Mapper for a multiengine.

The value of all arguments are used for all calls to *map*. This class allows these arguemnts to be set for a series of map calls.

Parameters

multiengine [*IMultiEngine* implementer] The multiengine to use for running the map commands

dist [str] The type of decomposition to use. Only block ('b') is supported currently

targets [(str, int, tuple of ints)] The engines to use in the map

block [boolean] Whether to block when the map is applied

map()

Apply func to *sequences elementwise. Like Python's builtin map.

This version is not load balanced.

SynchronousTaskMapper

class IPython.kernel.mapper.**SynchronousTaskMapper**(*task_controller*, *clear_before=False*, *clear_after=False*, *retries=0*, *recovery_task=None*, *depend=None*, *block=True*)

Bases: object

Make an *IBlockingTaskClient* look like an *IMapper*.

This class provides a load balanced version of *map*.

__init__()

Create a *IMapper* given a *IBlockingTaskClient* and arguments.

The additional arguments are those that are common to all types of tasks and are described in the documentation for *IPython.kernel.task.BaseTask*.

Parameters

task_controller [an *IBlockingTaskClient* implementer] The *TaskController* to use for calls to *map*

map()

Apply func to *sequences elementwise. Like Python's builtin map.

This version is load balanced.

TaskMapper

class IPython.kernel.mapper.**TaskMapper**(*task_controller*, *clear_before=False*, *clear_after=False*, *retries=0*, *recovery_task=None*, *depend=None*, *block=True*)

Bases: object

Make an *ITaskController* look like an *IMapper*.

This class provides a load balanced version of *map*.

__init__()

Create a *IMapper* given a *TaskController* and arguments.

The additional arguments are those that are common to all types of tasks and are described in the documentation for *IPython.kernel.task.BaseTask*.

Parameters

task_controller [an *IBlockingTaskClient* implementer] The *TaskController* to use for calls to *map*

map()

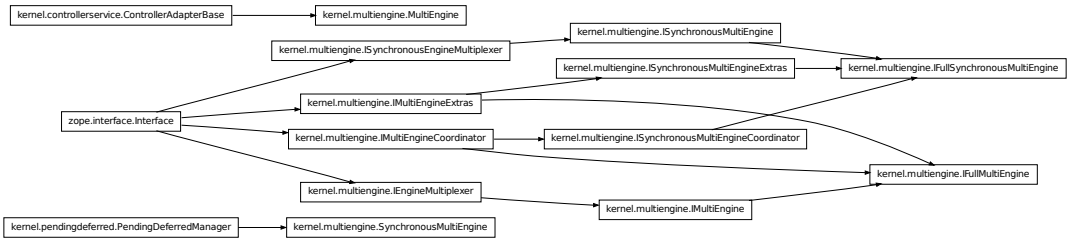
Apply func to *sequences elementwise. Like Python's builtin map.

This version is load balanced.

10.86 kernel.multiengine

10.86.1 Module: kernel.multiengine

Inheritance diagram for IPython.kernel.multiengine:



Adapt the IPython ControllerServer to IMultiEngine.

This module provides classes that adapt a ControllerService to the IMultiEngine interface. This interface is a basic interactive interface for working with a set of engines where it is desired to have explicit access to each registered engine.

The classes here are exposed to the network in files like:

- multienginevanilla.py
- multienginepb.py

10.86.2 Classes

IEngineMultiplexer

class IPython.kernel.multiengine.**IEngineMultiplexer**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Interface to multiple engines implementing IEngineCore/Serialized/Queued.

This class simply acts as a multiplexer of methods that are in the various IEngines* interfaces. Thus the methods here are jut like those in the IEngine* interfaces, but with an extra first argument, targets. The targets argument can have the following forms:

•targets = 10 # Engines are indexed by ints

•targets = [0,1,2,3] # A list of ints

•targets = 'all' # A string to indicate all targets

If targets is bad in any way, an InvalidEngineID will be raised. This includes engines not being registered.

All IEngineMultiplexer multiplexer methods must return a Deferred to a list with length equal to the number of targets. The elements of the list will correspond to the return of the corresponding IEngine method.

Failures are aggressive, meaning that if an action fails for any target, the overall action will fail immediately with that Failure.

Parameters

targets [int, list of ints, or 'all'] Engine ids the action will apply to.

Returns Deferred to a list of results for each engine.

Exception

InvalidEngineID If the targets argument is bad or engines aren't registered.

NoEnginesRegistered If there are no engines registered and targets='all'

classmethod __init__()

IFullMultiEngine

class IPython.kernel.multiengine.**IFullMultiEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.IMultiEngine, IPython.kernel.multiengine.IMultiEngineCoordinator, IPython.kernel.multiengine.IMultiEngineExtras

classmethod __init__()

IFullSynchronousMultiEngine

class IPython.kernel.multiengine.**IFullSynchronousMultiEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.ISynchronousMultiEngine, IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator, IPython.kernel.multiengine.ISynchronousMultiEngineExtras

classmethod __init__()

IMultiEngine

class IPython.kernel.multiengine.**IMultiEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.IEngineMultiplexer

A controller that exposes an explicit interface to all of its engines.

This is the primary inteface for interactive usage.

classmethod __init__()

IMultiEngineCoordinator

class IPython.kernel.multiengine.**IMultiEngineCoordinator**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Methods that work on multiple engines explicitly.

classmethod __init__()

IMultiEngineExtras

class IPython.kernel.multiengine.**IMultiEngineExtras**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

ISynchronousEngineMultiplexer

class IPython.kernel.multiengine.**ISynchronousEngineMultiplexer**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

ISynchronousMultiEngine

class IPython.kernel.multiengine.**ISynchronousMultiEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.ISynchronousEngineMultiplexer

Synchronous, two-phase version of IMultiEngine.

Methods in this interface are identical to those of IMultiEngine, but they take one additional argument:

execute(lines, targets='all') -> execute(lines, targets='all, block=True)

Parameters

block [boolean] Should the method return a deferred to a deferredID or the actual result. If block=False a deferred to a deferredID is returned and the user must call *get_pending_deferred* at a later point. If block=True, a deferred to the actual result comes back.

classmethod __init__()

ISynchronousMultiEngineCoordinator

class IPython.kernel.multiengine.**ISynchronousMultiEngineCoordinator**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.IMultiEngineCoordinator

Methods that work on multiple engines explicitly.

classmethod __init__()

ISynchronousMultiEngineExtras

class IPython.kernel.multiengine.**ISynchronousMultiEngineExtras**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.multiengine.IMultiEngineExtras

classmethod __init__()

MultiEngine

class IPython.kernel.multiengine.**MultiEngine**(*controller*)

Bases: IPython.kernel.controllerservice.ControllerAdapterBase

The representation of a ControllerService as a IMultiEngine.

Although it is not implemented currently, this class would be where a client/notification API is implemented. It could inherit from something like results.NotifierParent and then use the notify method to send notifications.

__init__()

clear_properties()

clear_queue()

del_properties()

engineList()
Parse the targets argument into a list of valid engine objects.

Parameters

targets [int, list of ints or 'all'] The targets argument to be parsed.

Returns List of engine objects.

Exception

InvalidEngineID If targets is not valid or if an engine is not registered.

execute()

get_ids()

get_properties()

get_result()

has_properties()

keys()

kill()

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()

queue_status()

reset()

set_properties()

SynchronousMultiEngine

class IPython.kernel.multiengine.**SynchronousMultiEngine**(*multiengine*)
Bases: IPython.kernel.pendingdeferred.PendingDeferredManager

Adapt an *IMultiEngine* -> *ISynchronousMultiEngine*

Warning, this class uses a decorator that currently uses **kwargs. Because of this block must be passed as a kwarg, not positionally.

__init__()

clear_properties()

clear_queue()

del_properties()

execute()

get_ids()
Return a list of registered engine ids.

Never use the two phase block/non-block stuff for this.

get_properties()

get_result()

has_properties()

keys()

kill()

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()

queue_status()

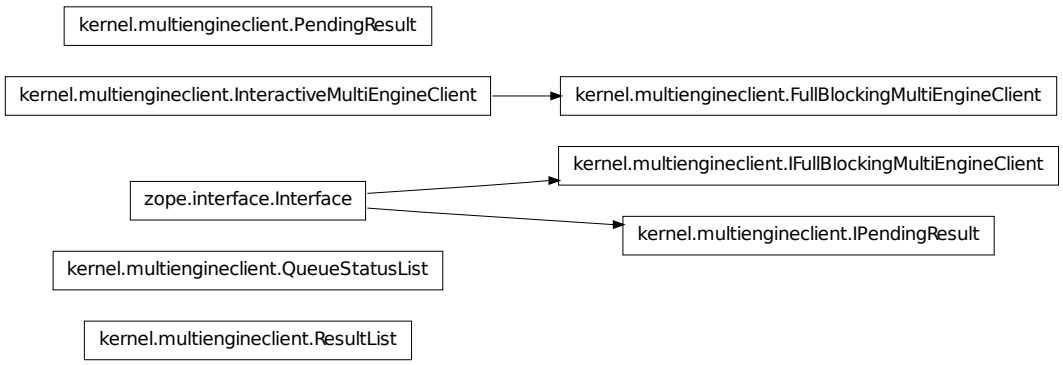
reset()

set_properties()

10.87 kernel.multiengineclient

10.87.1 Module: kernel.multiengineclient

Inheritance diagram for IPython.kernel.multiengineclient:



General Classes for IMultiEngine clients.

10.87.2 Classes

FullBlockingMultiEngineClient

class IPython.kernel.multiengineclient.**FullBlockingMultiEngineClient**(*smultiengine*)
Bases: IPython.kernel.multiengineclient.InteractiveMultiEngineClient

A blocking client to the *IMultiEngine* controller interface.

This class allows users to use a set of engines for a parallel computation through the *IMultiEngine* interface. In this interface, each engine has a specific id (an int) that is used to refer to the engine, run code on it, etc.

__init__()

barrier()
Synchronize a set of *PendingResults*.

This method is a synchronization primitive that waits for a set of *PendingResult* objects to complete. More specifically, barier does the following.

- The ‘PendingResult‘s are sorted by result_id.
- The *get_result* method is called for each *PendingResult* sequentially with block=True.
- If a *PendingResult* gets a result that is an exception, it is trapped and can be re-raised later by calling *get_result* again.
- The ‘PendingResult‘s are flushed from the controller.

After barrier has been called on a *PendingResult*, its results can be retrieved by calling *get_result* again or accesing the *r* attribute of the instance.

benchmark()
Run performance benchmarks for the current IPython cluster.

This method tests both the latency of sending command and data to the engines as well as the throughput of sending large objects to the engines using push. The latency is measured by having one or more engines execute the command 'pass'. The throughput is measure by sending an NumPy array of size *push_size* to one or more engines.

These benchmarks will vary widely on different hardware and networks and thus can be used to get an idea of the performance characteristics of a particular configuration of an IPython controller and engines.

This function is not testable within our current testing framework.

clear_pending_results()
Clear all pending deferreds/results from the controller.

For each *PendingResult* that is created by this client, the controller holds on to the result for that *PendingResult*. This can be a problem if there are a large number of *PendingResult* objects that are created.

Once the result of the *PendingResult* has been retrieved, the result is removed from the controller, but if a user doesn't get a result (they just ignore the *PendingResult*) the result is kept forever on the controller. This method allows the user to clear out all un-retrieved results on the controller.

clear_properties()

clear_queue()
Clear out the controller's queue for an engine.

The controller maintains a queue for each engine. This clear it out.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

del_properties()

execute()
Execute code on a set of engines.

Parameters

lines [str] The Python code to execute as a string

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

flush()
Clear all pending deferreds/results from the controller.

For each *PendingResult* that is created by this client, the controller holds on to the result for that *PendingResult*. This can be a problem if there are a large number of *PendingResult* objects that are created.

Once the result of the *PendingResult* has been retrieved, the result is removed from the controller, but if a user doesn't get a result (they just ignore the *PendingResult*) the result is kept forever on the controller. This method allows the user to clear out all un-retrieved results on the controller.

gather()
Gather a partitioned sequence on a set of engines as a single local seq.

get_ids()
Returns the ids of currently registered engines.

get_pending_deferred()

get_properties()

get_result()
Get a previous result.

When code is executed in an engine, a dict is created and returned. This method retrieves that dict for previous commands.

Parameters

i [int] The number of the result to get

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

has_properties()

keys()
Get a list of all the variables in an engine's namespace.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

kill()
Kill the engines and controller.

This method is used to stop the engine and controller by calling *reactor.stop*.

Parameters

controller [boolean] If True, kill the engines and controller. If False, just the engines

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

map()
A parallel version of Python's builtin *map* function.

This method applies a function to sequences of arguments. It follows the same syntax as the builtin *map*.

This method creates a mapper objects by calling *self.mapper* with no arguments and then uses that mapper to do the mapping. See the documentation of *mapper* for more details.

mapper()
Create a mapper object that has a *map* method.

This method returns an object that implements the *IMapper* interface. This method is a factory that is used to control how the map happens.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

parallel()
A decorator that turns a function into a parallel function.

This can be used as:

@parallel() def f(x, y)

...

f(range(10), range(10))

This causes f(0,0), f(1,1), ... to be called in parallel.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

pull()
Pull Python objects by key out of engines namespaces.

Parameters

keys [str or list of str] The names of the variables to be pulled

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

pull_function()
Pull a Python function from an engine.

This method is used to pull a Python function from an engine. Closures are not supported.

Parameters

keys [str or list of str] The names of the functions to be pulled

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

pull_serialized()

push()

Push a dictionary of keys and values to engines namespace.

Each engine has a persistent namespace. This method is used to push Python objects into that namespace.

The objects in the namespace must be pickleable.

Parameters

namespace [dict] A dict that contains Python objects to be injected into the engine persistent namespace.

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

push_function()

Push a Python function to an engine.

This method is used to push a Python function to an engine. This method can then be used in code on the engines. Closures are not supported.

Parameters

namespace [dict] A dict whose values are the functions to be pushed. The keys give that names that the function will appear as in the engines namespace.

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

push_serialized()

queue_status()

Get the status of an engines queue.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

raw_map()

A parallelized version of Python's builtin map.

This has a slightly different syntax than the builtin *map*. This is needed because we need to have keyword arguments and thus can't use *args to capture all the sequences. Instead, they must be passed in a list or tuple.

raw_map(func, seqs) -> map(func, seqs[0], seqs[1], ...)

Most users will want to use parallel functions or the *mapper* and *map* methods for an API that follows that of the builtin *map*.

reset()
Reset an engine.

This method clears out the namespace of an engine.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

run()
Run a Python code in a file on the engines.

Parameters

filename [str] The name of the local file to run

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

scatter()
Partition a Python sequence and send the partitions to a set of engines.

set_properties()

zip_pull()

IFullBlockingMultiEngineClient

class IPython.kernel.multiengineclient.**IFullBlockingMultiEngineClient**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

IPendingResult

class IPython.kernel.multienginecient.**IPendingResult**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

A representation of a result that is pending.

This class is similar to Twisted's *Deferred* object, but is designed to be used in a synchronous context.

classmethod __init__()

InteractiveMultiEngineClient

class IPython.kernel.multienginecient.**InteractiveMultiEngineClient**

Bases: object

A mixin class that add a few methods to a multiengine client.

The methods in this mixin class are designed for interactive usage.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

activate()

Make this *MultiEngineClient* active for parallel magic commands.

IPython has a magic command syntax to work with *MultiEngineClient* objects. In a given IPython session there is a single active one. While there can be many *MultiEngineClient* created and used by the user, there is only one active one. The active *MultiEngineClient* is used whenever the magic commands %px and %autopx are used.

The activate() method is called on a given *MultiEngineClient* to make it active. Once this has been done, the magic commands can be used.

findsource_file()

findsource_ipython()

PendingResult

class IPython.kernel.multienginecient.**PendingResult**(*client*, *result_id*)

Bases: object

A representation of a result that is not yet ready.

A user should not create a *PendingResult* instance by hand.

Methods:

- *get_result*

- *add_callback*

Properties:

- *r*

__init__()
Create a PendingResult with a result_id and a client instance.

The client should implement *_getPendingResult(result_id, block)*.

add_callback()
Add a callback that is called with the result.

If the original result is result, adding a callback will cause f(result, *args, **kwargs) to be returned instead. If multiple callbacks are registered, they are chained together: the result of one is passed to the next and so on.

Unlike Twisted's Deferred object, there is no errback chain. Thus any exception raised will not be caught and handled. User must catch these by hand when calling *get_result*.

get_result()
Get a result that is pending.

This method will connect to an IMultiEngine adapted controller and see if the result is ready. If the action triggers an exception raise it and record it. This method records the result/exception once it is retrieved. Calling *get_result* again will get this cached result or will re-raise the exception. The .r attribute is a property that calls *get_result* with block=True.

Parameters

default The value to return if the result is not ready.

block [boolean] Should I block for the result.

Returns The actual result or the default value.

r
This property is a shortcut to a *get_result(block=True)*.

QueueStatusList

class IPython.kernel.multiengineclient.**QueueStatusList**
Bases: list

A subclass of list that pretty prints the output of *queue_status*.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

ResultList

class IPython.kernel.multiengineclient.**ResultList**
Bases: list

A subclass of list that pretty prints the output of *execute/get_result*.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.87.3 Functions

IPython.kernel.multienginеclient.**remote**()

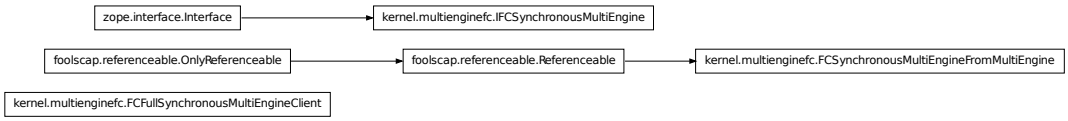
IPython.kernel.multienginеclient.**strip_whitespace**()

IPython.kernel.multienginеclient.**wrapResultList**()
A function that wraps the output of *execute/get_result -> ResultList*.

10.88 kernel.multienginefc

10.88.1 Module: kernel.multienginefc

Inheritance diagram for IPython.kernel.multienginefc:



Expose the multiengine controller over the Foolscap network protocol.

10.88.2 Classes

FCFullSynchronousMultiEngineClient

class IPython.kernel.multienginefc.**FCFullSynchronousMultiEngineClient**(*remote_reference*)
Bases: object

__init__()

adapt_to_blocking_client()

clear_pending_deferreds()

clear_properties()

clear_queue()

del_properties()

execute()

gather()

get_ids()

get_pending_deferred()

get_properties()

get_result()

has_properties()

keys()

kill()

map()
A parallel version of Python's builtin *map* function.

This method applies a function to sequences of arguments. It follows the same syntax as the builtin *map*.

This method creates a mapper objects by calling *self.mapper* with no arguments and then uses that mapper to do the mapping. See the documentation of *mapper* for more details.

mapper()
Create a mapper object that has a *map* method.

This method returns an object that implements the *IMapper* interface. This method is a factory that is used to control how the map happens.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

parallel()
A decorator that turns a function into a parallel function.

This can be used as:

@parallel() def f(x, y)

...

f(range(10), range(10))

This causes f(0,0), f(1,1), ... to be called in parallel.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

pull()

pull_function()

pull_serialized()

push()

push_function()

push_serialized()

queue_status()

raw_map()
A parallelized version of Python's builtin map.

This has a slightly different syntax than the builtin *map*. This is needed because we need to have keyword arguments and thus can't use *args to capture all the sequences. Instead, they must be passed in a list or tuple.

raw_map(func, seqs) -> map(func, seqs[0], seqs[1], ...)

Most users will want to use parallel functions or the *mapper* and *map* methods for an API that follows that of the builtin *map*.

reset()

run()

scatter()

set_properties()

unpackage()

zip_pull()

FCSynchronousMultiEngineFromMultiEngine

class IPython.kernel.multienginefc.**FCSynchronousMultiEngineFromMultiEngine**(*multiengine*)
Bases: foolscap.referenceable.Referenceable

Adapt *IMultiEngine* -> *ISynchronousMultiEngine* -> *IFCSynchronousMultiEngine*.

__init__()

packageFailure()

packageSuccess()

remote_clear_pending_deferreds()

remote_clear_properties()

remote_clear_queue()

remote_del_properties()

remote_execute()

remote_get_client_name()

remote_get_ids()
Get the ids of the registered engines.

This method always blocks.

remote_get_pending_deferred()

remote_get_properties()

remote_get_result()

remote_has_properties()

remote_keys()

remote_kill()

remote_pull()

remote_pull_function()

remote_pull_serialized()

remote_push()

remote_push_function()

remote_push_serialized()

remote_queue_status()

remote_reset()

remote_set_properties()

IFCSynchronousMultiEngine

class IPython.kernel.multienginefc.**IFCSynchronousMultiEngine**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

Foolscap interface to *ISynchronousMultiEngine*.

The methods in this interface are similar to those of *ISynchronousMultiEngine*, but their arguments and return values are pickled if they are not already simple Python types that can be send over XML-RPC.

See the documentation of *ISynchronousMultiEngine* and *IMultiEngine* for documentation about the methods.

Most methods in this interface act like the *ISynchronousMultiEngine* versions and can be called in blocking or non-blocking mode.

classmethod **__init__**()

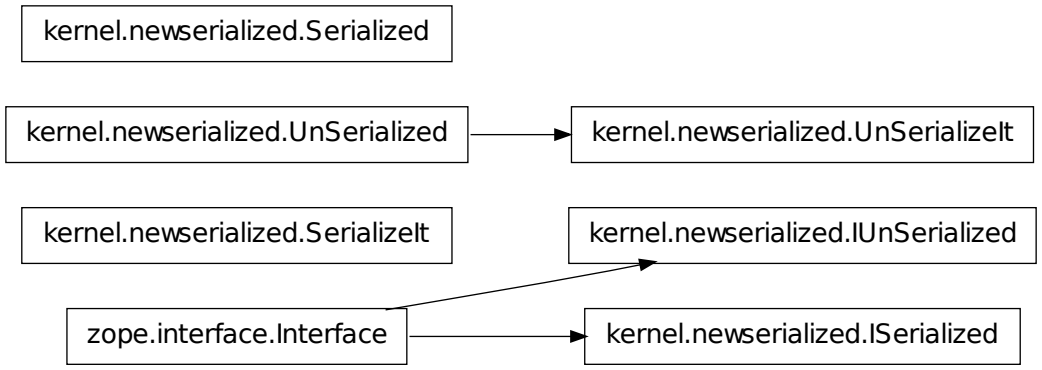
10.88.3 Function

IPython.kernel.multienginefc.**packageResult**()

10.89 kernel.newserialized

10.89.1 Module: kernel.newserialized

Inheritance diagram for IPython.kernel.newserialized:



Refactored serialization classes and interfaces.

10.89.2 Classes

ISerialized

class IPython.kernel.newserialized.**ISerialized**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod **__init__**()

IUnSerialized

class IPython.kernel.newserialized.**IUnSerialized**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

SerializeIt

class IPython.kernel.newserialized.**SerializeIt**(*unSerialized*)

Bases: object

__init__()

getData()

getDataSize()

getMetadata()

getTypeDescriptor()

Serialized

class IPython.kernel.newserialized.**Serialized**(*data*, *typeDescriptor*, *metadata={}*)

Bases: object

__init__()

getData()

getDataSize()

getMetadata()

getTypeDescriptor()

UnSerializeIt

class IPython.kernel.newserialized.**UnSerializeIt**(*serialized*)

Bases: IPython.kernel.newserialized.UnSerialized

__init__()

getObject()

UnSerialized

class IPython.kernel.newserialized.**UnSerialized**(*obj*)

Bases: object

__init__()

getObject()

10.89.3 Functions

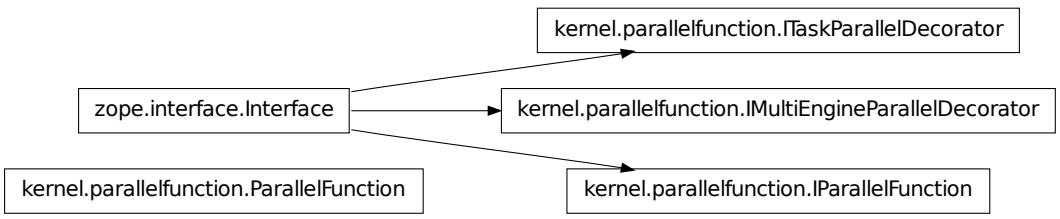
IPython.kernel.newserialized.**serialize**()

IPython.kernel.newserialized.**unserialize**()

10.90 kernel.parallelfunction

10.90.1 Module: kernel.parallelfunction

Inheritance diagram for IPython.kernel.parallelfunction:



A parallelized function that does scatter/execute/gather.

10.90.2 Classes

IMultiEngineParallelDecorator

class IPython.kernel.parallelfunction.**IMultiEngineParallelDecorator**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

A decorator that creates a parallel function.

classmethod __init__()

IParallelFunction

class IPython.kernel.parallelfunction.**IParallelFunction**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

classmethod __init__()

ITaskParallelDecorator

class IPython.kernel.parallelfunction.**ITaskParallelDecorator**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

A decorator that creates a parallel function.

classmethod __init__()

ParallelFunction

class IPython.kernel.parallelfunction.**ParallelFunction**(*mapper*)

Bases: object

The implementation of a parallel function.

A parallel function is similar to Python's map function:

map(func, *sequences) -> pfunc(*sequences)

Parallel functions should be created by using the @parallel decorator.

__init__()

Create a parallel function from an *IMapper*.

Parameters

mapper [an *IMapper* implementer.] The mapper to use for the parallel function

10.91 kernel.pbutil

10.91.1 Module: kernel.pbutil

Utilities for PB using modules.

10.91.2 Functions

IPython.kernel.pbutil.**checkMessageSize**()
Check string m to see if it violates banana.SIZE_LIMIT.

This should be used on the client side of things for push, scatter and push_serialized and on the other end for pull, gather and pull_serialized.

Parameters

m [string] Message whose size will be checked.

info [string] String describing what object the message refers to.

Exceptions

- *PBMessageSizeError*: Raised in the message is > banana.SIZE_LIMIT

Returns The original message or a Failure wrapping a PBMessageSizeError

IPython.kernel.pbutil.**packageFailure**()
Clean and pickle a failure preappending the string FAILURE:

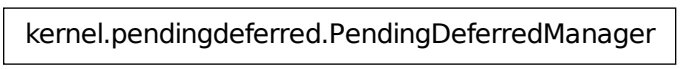
IPython.kernel.pbutil.**unpackageFailure**()
See if a returned value is a pickled Failure object.

To distinguish between general pickled objects and pickled Failures, the other side should prepend the string FAILURE: to any pickled Failure.

10.92 kernel.pendingdeferred

10.92.1 Module: kernel.pendingdeferred

Inheritance diagram for IPython.kernel.pendingdeferred:



Classes to manage pending Deferreds.

A pending deferred is a deferred that may or may not have fired. This module is useful for taking a class whose methods return deferreds and wrapping it to provide API that keeps track of those deferreds for later retrieval. See the tests for examples of its usage.

10.92.2 PendingDeferredManager

class IPython.kernel.pendingdeferred.**PendingDeferredManager**
Bases: object

A class to track pending deferreds.

To track a pending deferred, the user of this class must first get a deferredID by calling *get_next_deferred_id*. Then the user calls *save_pending_deferred* passing that id and the deferred to be tracked. To later retrieve it, the user calls *get_pending_deferred* passing the id.

__init__()
Manage pending deferreds.

clear_pending_deferreds()
Remove all the deferreds I am tracking.

delete_pending_deferred()
Remove a deferred I am tracking and add a null Errback.

Parameters

deferredID [str] The id of a deferred that I am tracking.

get_deferred_id()

get_pending_deferred()

quick_has_id()

save_pending_deferred()
Save the result of a deferred for later retrieval.

This works even if the deferred has not fired.

Only callbacks and errbacks applied to d before this method is called will be called no the final result.

IPython.kernel.pendingdeferred.**two_phase**()
Wrap methods that return a deferred into a two phase process.

This transforms:

```
foo(arg1, arg2, ...) -> foo(arg1, arg2,...,block=True).
```

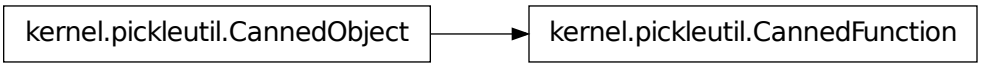
The wrapped method will then return a deferred to a deferred id. This will only work on method of classes that inherit from *PendingDeferredManager*, as that class provides an API for

block is a boolean to determine if we should use the two phase process or just simply call the wrapped method. At this point block does not have a default and it probably won't.

10.93 kernel.pickleutil

10.93.1 Module: kernel.pickleutil

Inheritance diagram for IPython.kernel.pickleutil:



Pickle related utilities.

10.93.2 Classes

CannedFunction

class IPython.kernel.pickleutil.**CannedFunction**(*f*)

Bases: IPython.kernel.pickleutil.CannedObject

__init__()

getFunction()

CannedObject

class IPython.kernel.pickleutil.**CannedObject**

Bases: object

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.93.3 Functions

IPython.kernel.pickleutil.**can**()

IPython.kernel.pickleutil.**canDict**()

IPython.kernel.pickleutil.**canSequence**()

IPython.kernel.pickleutil.**rebindFunctionGlobals**()

IPython.kernel.pickleutil.**uncan**()

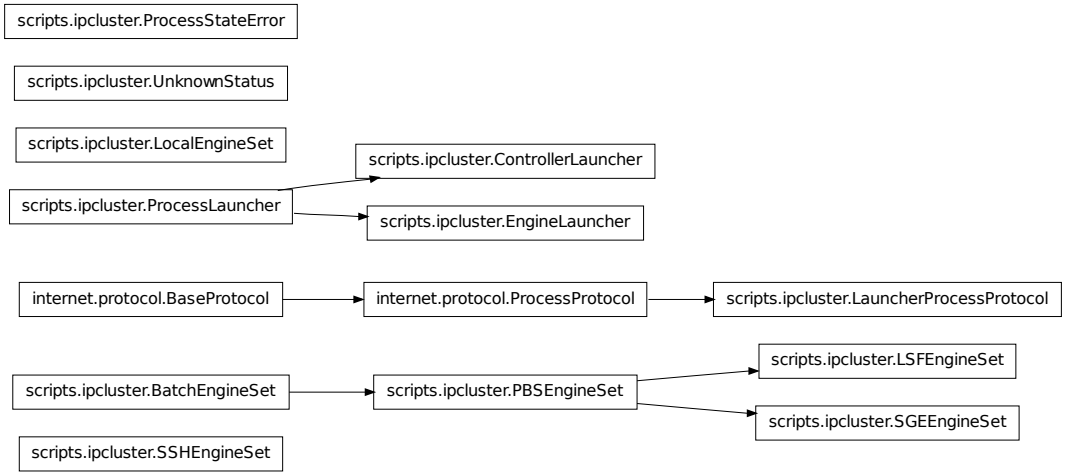
IPython.kernel.pickleutil.**uncanDict**()

IPython.kernel.pickleutil.**uncanSequence**()

10.94 kernel.scripts.ipcluster

10.94.1 Module: kernel.scripts.ipcluster

Inheritance diagram for IPython.kernel.scripts.ipcluster:



Start an IPython cluster = (controller + engines).

10.94.2 Classes

BatchEngineSet

class IPython.kernel.scripts.ipcluster.**BatchEngineSet**(*template_file*, *queue*, ***kwargs*)

Bases: object

__init__()

handle_error()

kill()

parse_job_id()

start()

ControllerLauncher

class IPython.kernel.scripts.ipcluster.**ControllerLauncher**(*extra_args=None*)

Bases: IPython.kernel.scripts.ipcluster.ProcessLauncher

__init__()

EngineLauncher

class IPython.kernel.scripts.ipcluster.**EngineLauncher**(*extra_args=None*)
Bases: IPython.kernel.scripts.ipcluster.ProcessLauncher

__init__()

LSFEngineSet

class IPython.kernel.scripts.ipcluster.**LSFEngineSet**(*template_file*, *queue*, ***kwargs*)
Bases: IPython.kernel.scripts.ipcluster.PBSEngineSet

__init__()

LauncherProcessProtocol

class IPython.kernel.scripts.ipcluster.**LauncherProcessProtocol**(*process_launcher*)
Bases: twisted.internet.protocol.ProcessProtocol

A ProcessProtocol to go with the ProcessLauncher.

__init__()

connectionMade()

errReceived()

outReceived()

processEnded()

LocalEngineSet

class IPython.kernel.scripts.ipcluster.**LocalEngineSet**(*extra_args=None*)
Bases: object

__init__()

interrupt_then_kill()

signal()

start()

PBSEngineSet

class IPython.kernel.scripts.ipcluster.**PBSEngineSet**(*template_file*, *queue*, ***kwargs*)

Bases: IPython.kernel.scripts.ipcluster.BatchEngineSet

__init__()

ProcessLauncher

class IPython.kernel.scripts.ipcluster.**ProcessLauncher**(*cmd_and_args*)

Bases: object

Start and stop an external process in an asynchronous manner.

Currently this uses deferreds to notify other parties of process state changes. This is an awkward design and should be moved to using a formal NotificationCenter.

__init__()

fire_start_deferred()

fire_stop_deferred()

get_stop_deferred()

interrupt_then_kill()

running

signal()

Send a signal to the process.

The argument sig can be ('KILL','INT', etc.) or any signal number.

start()

ProcessStateError

class IPython.kernel.scripts.ipcluster.**ProcessStateError**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

SGEEngineSet

class IPython.kernel.scripts.ipcluster.**SGEEngineSet**(*template_file*, *queue*, ***kwargs*)

Bases: IPython.kernel.scripts.ipcluster.PBSEngineSet

__init__()

SSHEngineSet

class IPython.kernel.scripts.ipcluster.**SSHEngineSet**(*engine_hosts*, *sshx=None*, *copyenvs=None*, *ipengine='ipengine'*)

Bases: object

__init__()

Start a controller on localhost and engines using ssh.

The engine_hosts argument is a dict with hostnames as keys and the number of engine (int) as values. sshx is the name of a local file that will be used to run remote commands. This file is used to setup the environment properly.

kill()

start()

UnknownStatus

class IPython.kernel.scripts.ipcluster.**UnknownStatus**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.94.3 Functions

IPython.kernel.scripts.ipcluster.**check_reuse**()

Check to see if we should try to resuse FURL files.

IPython.kernel.scripts.ipcluster.**check_security**()

Check to see if we should run with SSL support.

IPython.kernel.scripts.ipcluster.**escape_strings**()

IPython.kernel.scripts.ipcluster.**get_args**()

IPython.kernel.scripts.ipcluster.**main**()

IPython.kernel.scripts.ipcluster.**main_local**()

IPython.kernel.scripts.ipcluster.**main_lsf**()

IPython.kernel.scripts.ipcluster.**main_mpi**()

IPython.kernel.scripts.ipcluster.**main_pbs**()

IPython.kernel.scripts.ipcluster.**main_sge**()

IPython.kernel.scripts.ipcluster.**main_ssh**()

Start a controller on localhost and engines using ssh.

Your clusterfile should look like:

```
send_furl = False # True, if you want
engines = {
    'engine_host1' : engine_count,
    'engine_host2' : engine_count2
}
```

10.95 kernel.scripts.ipcontroller

10.95.1 Module: kernel.scripts.ipcontroller

The IPython controller.

10.95.2 Functions

IPython.kernel.scripts.ipcontroller.**get_temp_furlfile**()

IPython.kernel.scripts.ipcontroller.**init_config**()
Initialize the configuration using default and command line options.

IPython.kernel.scripts.ipcontroller.**main**()
After creating the configuration information, start the controller.

IPython.kernel.scripts.ipcontroller.**make_client_service**()
Create a service that will listen for clients.

This service is simply a *foolscap.Tub* instance that has a set of Referenceables registered with it.

IPython.kernel.scripts.ipcontroller.**make_engine_service**()
Create a service that will listen for engines.

This service is simply a *foolscap.Tub* instance that has a set of Referenceables registered with it.

IPython.kernel.scripts.ipcontroller.**make_tub**()
Create a listening tub given an ip, port, and cert_file location.

Parameters

ip [str] The ip address that the tub should listen on. Empty means all

port [int] The port that the tub should listen on. A value of 0 means pick a random port

secure: boolean Will the connection be secure (in the foolscap sense)

cert_file: A filename of a file to be used for theSSL certificate

IPython.kernel.scripts.ipcontroller.**start_controller**()
Start the controller by creating the service hierarchy and starting the reactor.

This method does the following:

•It starts the controller logging

- In execute an import statement for the controller
- It creates 2 *foolscap.Tub* instances for the client and the engines and registers *foolscap.Referenceables* with the tubs to expose the controller to engines and clients.

10.96 kernel.scripts.ipengine

10.96.1 Module: `kernel.scripts.ipengine`

Start the IPython Engine.

10.96.2 Functions

`IPython.kernel.scripts.ipengine.`**`init_config`**`()`
Initialize the configuration using default and command line options.

`IPython.kernel.scripts.ipengine.`**`main`**`()`
After creating the configuration information, start the engine.

`IPython.kernel.scripts.ipengine.`**`start_engine`**`()`
Start the engine, by creating it and starting the Twisted reactor.

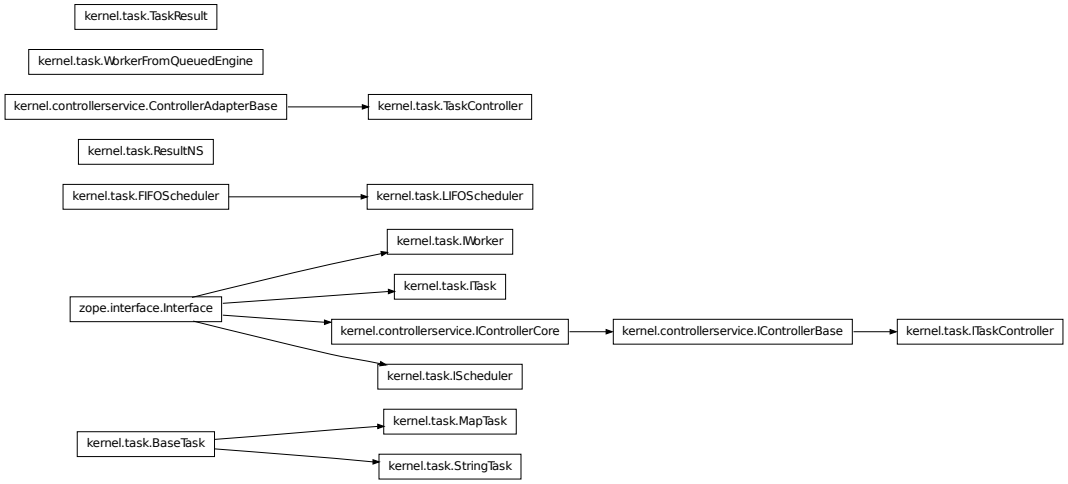
This method does:

- If it exists, runs the *mpi_import_statement* to call *MPI_Init*
- Starts the engine logging
- Creates an IPython shell and wraps it in an *EngineService*
- Creates a *foolscap.Tub* to use in connecting to a controller.
- Uses the tub and the *EngineService* along with a Foolscap URL (or FURL) to connect to the controller and register the engine with the controller

10.97 kernel.task

10.97.1 Module: `kernel.task`

Inheritance diagram for `IPython.kernel.task`:



Task farming representation of the ControllerService.

10.97.2 Classes

BaseTask

class IPython.kernel.task.**BaseTask**(*clear_before=False*, *clear_after=False*, *retries=0*, *recovery_task=None*, *depend=None*)

Bases: object

Common fuctionality for all objects implementing *ITask*.

__init__()

Make a generic task.

Parameters

clear_before [boolean] Should the engines namespace be cleared before the task is run

clear_after [boolean] Should the engines namespace be clear after the task is run

retries [int] The number of times a task should be retries upon failure

recovery_task [any task object] If a task fails and it has a recovery_task, that is run upon a retry

depend [FunctionType] A function that is called to test for properties. This function must take one argument, the properties dict and return a boolean

can_task()

check_depend()
Calls self.depend(properties) to see if a task should be run.

post_task()
Clear the engine after running the task if clear_after is set.

pre_task()
Clear the engine before running the task if clear_before is set.

process_result()
Process a task result.

This is the default *process_result* that just returns the raw result or a *Failure*.

start_time()
Start the basic timers.

stop_time()
Stop the basic timers.

submit_task()

uncan_task()

FIFOScheduler

class IPython.kernel.task.**FIFOScheduler**
Bases: object

A basic First-In-First-Out (Queue) Scheduler.

This is the default Scheduler for the *TaskController*. See the docstrings for *IScheduler* for interface details.

__init__()

add_task()

add_worker()

ntasks

nworkers

pop_task()

pop_worker()

schedule()

taskids

workerids

IScheduler

class IPython.kernel.task.**IScheduler**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

The interface for a Scheduler.

classmethod **__init__**()

ITask

class IPython.kernel.task.**ITask**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

This interface provides a generic definition of what constitutes a task.

There are two sides to a task. First a task needs to take input from a user to determine what work is performed by the task. Second, the task needs to have the logic that knows how to turn that information info specific calls to a worker, through the *IQueuedEngine* interface.

Many method in this class get two things passed to them: a Deferred and an IQueuedEngine implementer. Such methods should register callbacks on the Deferred that use the IQueuedEngine to accomplish something. See the existing task objects for examples.

classmethod **__init__**()

ITaskController

class IPython.kernel.task.**ITaskController**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: IPython.kernel.controllerservice.IControllerBase

The Task based interface to a *ControllerService* object

This adapts a *ControllerService* to the ITaskController interface.

classmethod **__init__**()

IWorker

class IPython.kernel.task.**IWorker**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

The Basic Worker Interface.

A worked is a representation of an Engine that is ready to run tasks.

classmethod **__init__**()

LIFOScheduler

class IPython.kernel.task.**LIFOScheduler**
Bases: IPython.kernel.task.FIFOScheduler

A Last-In-First-Out (Stack) Scheduler.

This scheduler should naively reward fast engines by giving them more jobs. This risks starvation, but only in cases with low load, where starvation does not really matter.

__init__()

add_task()

add_worker()

MapTask

class IPython.kernel.task.**MapTask**(*function*, *args=None*, *kwargs=None*, *clear_before=False*, *clear_after=False*, *retries=0*, *recovery_task=None*, *depend=None*)
Bases: IPython.kernel.task.BaseTask

A task that consists of a function and arguments.

__init__()
Create a task based on a function, args and kwargs.

This is a simple type of task that consists of calling: function(*args, **kwargs) and wrapping the result in a *TaskResult*.

The return value of the function, or a *Failure* wrapping an exception is the task result for this type of task.

can_task()

submit_task()

uncan_task()

ResultNS

class IPython.kernel.task.**ResultNS**(*dikt*)
Bases: object

A dict like object for holding the results of a task.

The result namespace object for use in *TaskResult* objects as tr.ns. It builds an object from a dictionary, such that it has attributes according to the key,value pairs of the dictionary.

This works by calling setattr on ALL key,value pairs in the dict. If a user chooses to overwrite the *__repr__* or *__getattr__* attributes, they can. This can be a bad idea, as it may corrupt standard behavior of the ns object.

Examples

```
>>> ns = ResultNS({'a':17,'foo':range(3)})
>>> print ns
NS{'a': 17, 'foo': [0, 1, 2]}
>>> ns.a
17
>>> ns['foo']
[0, 1, 2]
```

__init__()

StringTask

class IPython.kernel.task.**StringTask**(*expression*, *pull=None*, *push=None*, *clear_before=False*, *clear_after=False*, *retries=0*, *recovery_task=None*, *depend=None*)

Bases: IPython.kernel.task.BaseTask

A task that consists of a string of Python code to run.

__init__()

Create a task based on a Python expression and variables

This type of task lets you push a set of variables to the engines namespace, run a Python string in that namespace and then bring back a different set of Python variables as the result.

Because this type of task can return many results (through the *pull* keyword argument) it returns a special *TaskResult* object that wraps the pulled variables, statistics about the run and any exceptions raised.

process_result()

submit_task()

TaskController

class IPython.kernel.task.**TaskController**(*controller*)

Bases: IPython.kernel.controllerservice.ControllerAdapterBase

The Task based interface to a Controller object.

If you want to use a different scheduler, just subclass this and set the *SchedulerClass* member to the *class* of your chosen scheduler.

__init__()

SchedulerClass

alias of FIFOScheduler

abort()

Remove a task from the queue if it has not been run already.

barrier()

checkIdle()

clear()
Clear all previously run tasks from the task controller.

This is needed because the task controller keep all task results in memory. This can be a problem is there are many completed tasks. Users should call this periodically to clean out these cached task results.

distributeTasks()
Distribute tasks while self.scheduler has things to do.

failIdle()

get_task_result()
Returns a *Deferred* to the task result, or None.

queue_status()

readmitWorker()
Readmit a worker to the scheduler.

This is outside *taskCompleted* because of the *failurePenalty* being implemented through *reactor.callLater*.

registerWorker()
Called by controller.register_engine.

run()
Run a task and return *Deferred* to its taskid.

spin()

taskCompleted()
This is the err/callback for a completed task.

unregisterWorker()
Called by controller.unregister_engine

TaskResult

class IPython.kernel.task.**TaskResult**(*results*, *engineid*)
Bases: object

An object for returning task results for certain types of tasks.

This object encapsulates the results of a task. On task success it will have a keys attribute that will have a list of the variables that have been pulled back. These variables are accessible as attributes of this class as well. On success the failure attribute will be None.

In task failure, keys will be empty, but failure will contain the failure object that encapsulates the remote exception. One can also simply call the *raise_exception* method of this class to re-raise any remote exception in the local session.

The *TaskResult* has a *.ns* member, which is a property for access to the results. If the Task had pull=['a', 'b'], then the Task Result will have attributes *tr.ns.a*, *tr.ns.b* for those values. Accessing *tr.ns* will raise the remote failure if the task failed.

The *engineid* attribute should have the *engineid* of the engine that ran the task. But, because engines can come and go, the *engineid* may not continue to be valid or accurate.

The *taskid* attribute simply gives the *taskid* that the task is tracked under.

__init__()

ns

raise_exception()
Re-raise any remote exceptions in the local python session.

WorkerFromQueuedEngine

class IPython.kernel.task.**WorkerFromQueuedEngine**(*qe*)
Bases: object

Adapt an *IQueuedEngine* to an *IWorker* object

__init__()

properties

run()
Run task in worker's namespace.

This takes a task and calls methods on the task that actually cause *self.queuedEngine* to do the task. See the methods of *ITask* for more information about how these methods are called.

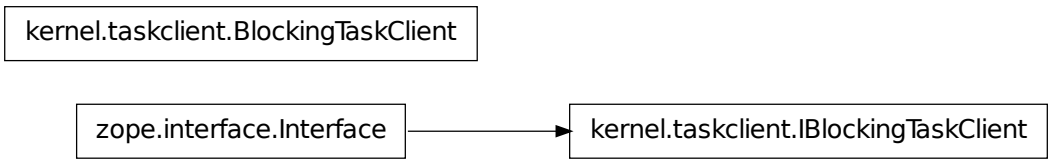
Parameters task : a *Task* object

Returns *Deferred* to a tuple of (success, result) where success if a boolean that signifies success or failure and result is the task result.

10.98 kernel.taskclient

10.98.1 Module: kernel.taskclient

Inheritance diagram for IPython.kernel.taskclient:



A blocking version of the task client.

10.98.2 Classes

BlockingTaskClient

class IPython.kernel.taskclient.**BlockingTaskClient**(*task_controller*)
Bases: object

A blocking task client that adapts a non-blocking one.

__init__()

abort()
Abort a task by taskid.

Parameters

taskid [int] The taskid of the task to be aborted.

barrier()
Block until a set of tasks are completed.

Parameters

taskids [list, tuple] A sequence of taskids to block on.

clear()
Clear all previously run tasks from the task controller.

This is needed because the task controller keep all task results in memory. This can be a problem is there are many completed tasks. Users should call this periodically to clean out these cached task results.

get_task_result()
Get a task result by taskid.

Parameters

taskid [int] The taskid of the task to be retrieved.

block [boolean] Should I block until the task is done?

Returns A *TaskResult* object that encapsulates the task result.

map()
Apply func to *sequences elementwise. Like Python's builtin map.

This version is load balanced.

mapper()
Create an *IMapper* implementer with a given set of arguments.

The *IMapper* created using a task controller is load balanced.

See the documentation for *IPython.kernel.task.BaseTask* for documentation on the arguments to this method.

parallel()

queue_status()
Get a dictionary with the current state of the task queue.

Parameters

verbose [boolean] If True, return a list of taskids. If False, simply give the number of tasks with each status.

Returns A dict with the queue status.

run()
Run a task on the *TaskController*.

See the documentation of the *MapTask* and *StringTask* classes for details on how to build a task of different types.

Parameters task : an *ITask* implementer

Returns The int taskid of the submitted task. Pass this to *get_task_result* to get the *TaskResult* object.

spin()
Touch the scheduler, to resume scheduling without submitting a task.

This method only needs to be called in unusual situations where the scheduler is idle for some reason.

IBlockingTaskClient

class IPython.kernel.taskclient.**IBlockingTaskClient**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: zope.interface.Interface

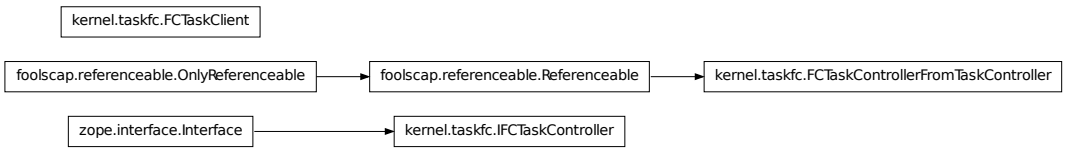
A vague interface of the blocking task client

classmethod __init__()

10.99 kernel.taskfc

10.99.1 Module: kernel.taskfc

Inheritance diagram for IPython.kernel.taskfc:



A Foolscap interface to a TaskController.

This class lets Foolscap clients talk to a TaskController.

10.99.2 Classes

FCTaskClient

class IPython.kernel.taskfc.**FCTaskClient**(*remote_reference*)

Bases: object

Client class for Foolscap exposed *TaskController*.

This class is an adapter that makes a *RemoteReference* to a *TaskController* look like an actual *ITaskController* on the client side.

This class also implements *IBlockingClientAdaptor* so that clients can automatically get a blocking version of this class.

__init__()

abort()

Abort a task by taskid.

Parameters

taskid [int] The taskid of the task to be aborted.

adapt_to_blocking_client()

Wrap self in a blocking version that implements ‘IBlockingTaskClient.

barrier()

Block until a set of tasks are completed.

Parameters

taskids [list, tuple] A sequence of taskids to block on.

clear()
Clear previously run tasks from the task controller. :Parameters:

taskids [list, tuple, None] A sequence of taskids whose results we should drop. if None: clear all results

Returns An int, the number of tasks cleared

This is needed because the task controller keep all task results in memory. This can be a problem is there are many completed tasks. Users should call this periodically to clean out these cached task results.

get_task_result()
Get a task result by taskid.

Parameters

taskid [int] The taskid of the task to be retrieved.

block [boolean] Should I block until the task is done?

Returns A *TaskResult* object that encapsulates the task result.

map()
Apply func to *sequences elementwise. Like Python's builtin map.

This version is load balanced.

mapper()
Create an *IMapper* implementer with a given set of arguments.

The *IMapper* created using a task controller is load balanced.

See the documentation for *IPython.kernel.task.BaseTask* for documentation on the arguments to this method.

parallel()

queue_status()
Get a dictionary with the current state of the task queue.

Parameters

verbose [boolean] If True, return a list of taskids. If False, simply give the number of tasks with each status.

Returns A dict with the queue status.

run()
Run a task on the *TaskController*.

See the documentation of the *MapTask* and *StringTask* classes for details on how to build a task of different types.

Parameters task : an *ITask* implementer

Returns The int taskid of the submitted task. Pass this to *get_task_result* to get the *TaskResult* object.

spin()
Touch the scheduler, to resume scheduling without submitting a task.

This method only needs to be called in unusual situations where the scheduler is idle for some reason.

unpackage()

FCTaskControllerFromTaskController

class IPython.kernel.taskfc.**FCTaskControllerFromTaskController**(*taskController*)
Bases: foolscap.referenceable.Referenceable

Adapt a *TaskController* to an *IFCTaskController*

This class is used to expose a *TaskController* over the wire using the Foolscap network protocol.

__init__()

packageFailure()

packageSuccess()

remote_abort()

remote_barrier()

remote_clear()

remote_get_client_name()

remote_get_task_result()

remote_queue_status()

remote_run()

remote_spin()

IFCTaskController

class IPython.kernel.taskfc.**IFCTaskController**(*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)
Bases: zope.interface.Interface

Foolscap interface to task controller.

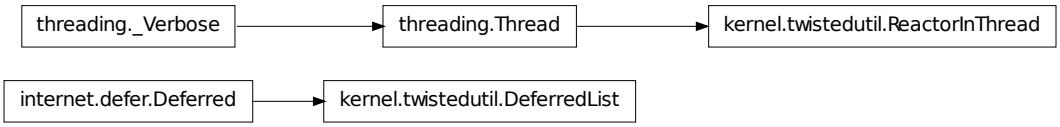
See the documentation of *ITaskController* for more information.

classmethod __init__()

10.100 kernel.twistedutil

10.100.1 Module: kernel.twistedutil

Inheritance diagram for IPython.kernel.twistedutil:



Things directly related to all of twisted.

10.100.2 Classes

DeferredList

class IPython.kernel.twistedutil.**DeferredList**(*deferredList*, *fireOnOneCallback=0*, *fireOnOneErrback=0*, *consumeErrors=0*, *logErrors=0*)

Bases: twisted.internet.defer.Deferred

I combine a group of deferreds into one callback.

I track a list of L{Deferred}s for their callbacks, and make a single callback when they have all completed, a list of (success, result) tuples, 'success' being a boolean.

Note that you can still use a L{Deferred} after putting it in a DeferredList. For example, you can suppress 'Unhandled error in Deferred' messages by adding errbacks to the Deferreds *after* putting them in the DeferredList, as a DeferredList won't swallow the errors. (Although a more convenient way to do this is simply to set the consumeErrors flag)

Note: This is a modified version of the twisted.internet.defer.DeferredList

__init__()

Initialize a DeferredList.

@type deferredList: C{list} of L{Deferred}s @param deferredList: The list of deferreds to track. @param fireOnOneCallback: (keyword param) a flag indicating that

only one callback needs to be fired for me to call my callback

@param fireOnOneErrback: (keyword param) a flag indicating that only one errback needs to be fired for me to call my errback

@param consumeErrors: (keyword param) a flag indicating that any errors raised in the original deferreds should be consumed by this DeferredList. This is useful to prevent spurious warnings being logged.

ReactorInThread

class IPython.kernel.twistedutil.**ReactorInThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *verbose=None*)

Bases: threading.Thread

Run the twisted reactor in a different thread.

For the process to be able to exit cleanly, do the following:

rit = ReactorInThread() rit.setDaemon(True) rit.start()

__init__()

run()

stop()

10.100.3 Functions

IPython.kernel.twistedutil.**gatherBoth**()
This is like gatherBoth, but sets consumeErrors=1.

IPython.kernel.twistedutil.**parseResults**()
Pull out results/Failures from a DeferredList.

IPython.kernel.twistedutil.**wait_for_file**()
Wait (poll) for a file to be created.

This method returns a Deferred that will fire when a file exists. It works by polling os.path.isfile in time intervals specified by the delay argument. If *max_tries* is reached, it will errback with a *FileTimeoutError*.

Parameters **filename** : str

The name of the file to wait for.

delay : float

The time to wait between polls.

max_tries : int

The max number of attempts before raising *FileTimeoutError*

Returns **d** : Deferred

A Deferred instance that will fire when the file exists.

10.101 kernel.util

10.101.1 Module: `kernel.util`

General utilities for kernel related things.

10.101.2 Functions

`IPython.kernel.util.`**`catcher`**`()`

`IPython.kernel.util.`**`curry`**`()`
Curry the function f with curryArgs and curryKWargs.

`IPython.kernel.util.`**`printer`**`()`

`IPython.kernel.util.`**`tarModule`**`()`
Makes a tarball (as a string) of a locally imported module.

This method looks at the __file__ attribute of an imported module and makes a tarball of the top level of the module. It then reads the tarball into a binary string.

The method returns the tarball's name and the binary string representing the tarball.

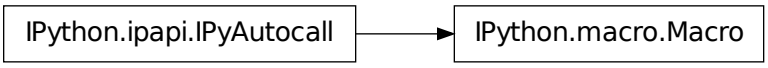
Notes:

- It will handle both single module files, as well as packages.
- The byte code files (*.pyc) are not deleted.
- It has not been tested with modules containing extension code, but it should work in most cases.
- There are cross platform issues.

10.102 macro

10.102.1 Module: `macro`

Inheritance diagram for `IPython.macro`:



Support for interactive macros in IPython

10.102.2 Macro

class IPython.macro.**Macro**(*data*)

Bases: IPython.ipapi.IPyAutocall

Simple class to store the value of macros as strings.

Macro is just a callable that executes a string of IPython input when called.

Args to macro are available in _margv list if you need them.

__init__()

10.103 numutils

10.103.1 Module: numutils

10.103.2 Functions

10.104 platutils

10.104.1 Module: platutils

Inheritance diagram for IPython.platutils:

Proxy module for accessing platform specific utility functions.

Importing this module should give you the implementations that are correct for your operation system, from platutils_PLATFORMNAME module.

10.104.2 Class

10.104.3 FindCmdError

class IPython.platutils.**FindCmdError**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.104.4 Functions

IPython.platutils.**find_cmd**()
Find full path to executable cmd in a cross platform manner.

This function tries to determine the full path to a command line program using *which* on Unix/Linux/OS X and *win32api* on Windows. Most of the time it will use the version that is first on the users *PATH*. If cmd is *python* return *sys.executable*.

Parameters **cmd** : str

The command line program to look for.

IPython.platutils.**freeze_term_title**()

IPython.platutils.**get_long_path_name**()
Expand a path into its long form.

On Windows this expands any ~ in the paths. On other platforms, it is a null operation.

IPython.platutils.**set_term_title**()
Set terminal title using the necessary platform-dependent calls.

IPython.platutils.**term_clear**()

IPython.platutils.**toggle_set_term_title**()
Control whether set_term_title is active or not.

set_term_title() allows writing to the console titlebar. In embedded widgets this can cause problems, so this call can be used to toggle it on or off as needed.

The default state of the module is for the function to be disabled.

Parameters **val** : bool

If True, set_term_title() actually writes to the terminal (using the appropriate platform-specific module). If False, it is a no-op.

10.105 platutils_dummy

10.105.1 Module: platutils_dummy

Platform specific utility functions, dummy version

This has empty implementation of the platutils functions, used for unsupported operating systems.

Authors

- Ville Vainio <vivainio@gmail.com>

10.105.2 Functions

IPython.platutils_dummy.**find_cmd**()
Find the full path to a command using which.

IPython.platutils_dummy.**get_long_path_name**()
Dummy no-op.

IPython.platutils_dummy.**set_term_title**()
Dummy no-op.

10.106 platutils_posix

10.106.1 Module: platutils_posix

Platform specific utility functions, posix version

Importing this module directly is not portable - rather, import platutils to use these functions in platform agnostic fashion.

10.106.2 Functions

IPython.platutils_posix.**find_cmd**()
Find the full path to a command using which.

IPython.platutils_posix.**get_long_path_name**()
Dummy no-op.

IPython.platutils_posix.**term_clear**()

10.107 platutils_win32

10.107.1 Module: platutils_win32

10.107.2 Functions

10.108 prefilter

10.108.1 Module: prefilter

Inheritance diagram for IPython.prefilter:

IPython.prefilter.LineInfo

Classes and functions for prefiltering (transforming) a line of user input. This module is responsible, primarily, for breaking the line up into useful pieces and triggering the appropriate handlers in iplib to do the actual transforming work.

10.108.2 Class

10.108.3 LineInfo

class IPython.prefilter.**LineInfo**(*line*, *continue_prompt*)

Bases: object

A single line of input and associated info.

Includes the following as properties:

line The original, raw line

continue_prompt Is this line a continuation in a sequence of multiline input?

pre The initial esc character or whitespace.

preChar The escape character(s) in pre or the empty string if there isn't one. Note that '!!' is a possible value for preChar. Otherwise it will always be a single character.

preWhitespace The leading whitespace from pre if it exists. If there is a preChar, this is just ''.

iFun The 'function part', which is basically the maximal initial sequence of valid python identifiers and the '.' character. This is what is checked for alias and magic transformations, used for auto-calling, etc.

theRest Everything else on the line.

__init__()

ofind()

Do a full, attribute-walking lookup of the iFun in the various namespaces for the given IPython InteractiveShell instance.

Return a dict with keys: found,obj,ospace,ismagic

Note: can cause state changes because of calling getattr, but should only be run if autocall is on and if the line hasn't matched any other, less dangerous handlers.

Does cache the results of the call, so can be called multiple times without worrying about *further* damaging state.

10.108.4 Functions

IPython.prefilter.**checkAlias**()
Check if the initital identifier on the line is an alias.

IPython.prefilter.**checkAssignment**()
Check to see if user is assigning to a var for the first time, in which case we want to avoid any sort of automagic / autocall games.

This allows users to assign to either alias or magic names true python variables (the magic/alias systems always take second seat to true python code). E.g. ls='hi', or ls,that=1,2

IPython.prefilter.**checkAutocall**()
Check if the initial word/function is callable and autocall is on.

IPython.prefilter.**checkAutomagic**()
If the iFun is magic, and automagic is on, run it. Note: normal, non-auto magic would already have been triggered via '%' in check_esc_chars. This just checks for automagic. Also, before triggering the magic handler, make sure that there is nothing in the user namespace which could shadow it.

IPython.prefilter.**checkEmacs**()
Emacs ipython-mode tags certain input lines.

IPython.prefilter.**checkEscChars**()
Check for escape character and return either a handler to handle it, or None if there is no escape char.

IPython.prefilter.**checkIPyAutocall**()
Instances of IPyAutocall in user_ns get autocalled immediately

IPython.prefilter.**checkMultiLineMagic**()
Allow ! and !! in multi-line statements if multi_line_specials is on

IPython.prefilter.**checkPythonOps**()
If the 'rest' of the line begins with a function call or pretty much any python operator, we should simply execute the line (regardless of whether or not there's a possible autocall expansion). This avoids spurious (and very confusing) geattr() accesses.

IPython.prefilter.**checkShellEscape**()

IPython.prefilter.**isShadowed**()
Is the given identifier defined in one of the namespaces which shadow the alias and magic namespaces? Note that an identifier is different than iFun, because it can not contain a '.' character.

IPython.prefilter.**prefilter**()
Call one of the passed-in InteractiveShell's handler preprocessors, depending on the form of the line. Return the results, which must be a value, even if it's a blank ('').

IPython.prefilter.**splitUserInput**()
Split user input into pre-char/whitespace, function part and rest.

Mostly internal to this module, but also used by iplib.expand_aliases, which passes in a shell pattern.

10.109 shellglobals

10.109.1 Module: shellglobals

Some globals used by the main Shell classes.

IPython.shellglobals.**run_in_frontend**()
Check if source snippet can be run in the REPL thread, as opposed to GUI mainloop (to prevent unnecessary hanging of mainloop).

10.110 strdispatch

10.110.1 Module: strdispatch

Inheritance diagram for IPython.strdispatch:

String dispatch class to match regexps and dispatch commands.

10.110.2 StrDispatch

class IPython.strdispatch.**StrDispatch**
Bases: object

Dispatch (lookup) a set of strings / regexps for match.

Example:

```
>>> dis = StrDispatch()
>>> dis.add_s('hei',34, priority = 4)
>>> dis.add_s('hei',123, priority = 2)
>>> dis.add_re('h.i', 686)
>>> print list(dis.flat_matches('hei'))
[123, 34, 686]
```

__init__()

add_re()
Adds a target regexp for dispatching

add_s()
Adds a target 'string' for dispatching

dispatch()
Get a seq of Commandchain objects that match key

flat_matches()
Yield all 'value' targets, without priority

s_matches()

10.111 testing.decorator_msim

10.111.1 Module: testing.decorator_msim

10.111.2 Functions

IPython.testing.decorator_msim.**decorator**()
General purpose decorator factory: takes a caller function as input and returns a decorator with the same attributes. A caller function is any function like this:

```
def caller(func, *args, **kw):
    # do something
    return func(*args, **kw)
```

Here is an example of usage:

```
>>> @decorator
... def chatty(f, *args, **kw):
...     print "Calling %r" % f.__name__
...     return f(*args, **kw)

>>> chatty.__name__
'chatty'

>>> @chatty
... def f(): pass
...
>>> f()
Calling 'f'
```

For sake of convenience, the decorator factory can also be called with two arguments. In this casem `decorator(caller, func)` is just a shortcut for `decorator(caller)(func)`.

IPython.testing.decorator_msim.**getinfo**()
Returns an info dictionary containing: - name (the name of the function : str) - argnames (the names of the arguments : list) - defaults (the values of the default arguments : tuple) - signature (the signature : str) - doc (the docstring : str) - module (the module name : str) - dict (the function __dict__ : str)

```
>>> def f(self, x=1, y=2, *args, **kw): pass

>>> info = getinfo(f)
```

```
>>> info["name"]
'f'
>>> info["argnames"]
['self', 'x', 'y', 'args', 'kw']

>>> info["defaults"]
(1, 2)

>>> info["signature"]
'self, x, y, *args, **kw'
```

IPython.testing.decorator_msim.**update_wrapper**()

An improvement over functools.update_wrapper. By default it works the same, but if the 'create' flag is set, generates a copy of the wrapper with the right signature and update the copy, not the original. Moreovoer, 'wrapped' can be a dictionary with keys 'name', 'doc', 'module', 'dict', 'defaults'.

10.112 testing.decorators

10.112.1 Module: testing.decorators

Decorators for labeling test objects.

Decorators that merely return a modified version of the original function object are straightforward. Decorators that return a new function object need to use nose.tools.make_decorator(original_function)(decorator) in returning the decorator, in order to preserve metadata such as function name, setup and teardown functions and so on - see nose.tools for more information.

This module provides a set of useful decorators meant to be ready to use in your own tests. See the bottom of the file for the ready-made ones, and if you find yourself writing a new one that may be of generic use, add it here.

NOTE: This file contains IPython-specific decorators and imports the numpy.testing.decorators file, which we've copied verbatim. Any of our own code will be added at the bottom if we end up extending this.

10.112.2 Functions

IPython.testing.decorators.**apply_wrapper**()

Apply a wrapper to a function for decoration.

This mixes Michele Simionato's decorator tool with nose's make_decorator, to apply a wrapper in a decorator so that all nose attributes, as well as function signature and other properties, survive the decoration cleanly. This will ensure that wrapped functions can still be well introspected via IPython, for example.

IPython.testing.decorators.**make_label_dec**()

Factory function to create a decorator that applies one or more labels.

Parameters label : string or sequence One or more labels that will be applied by the decorator to the functions

it decorates. Labels are attributes of the decorated function with their value set to True.

Keywords ds : string An optional docstring for the resulting decorator. If not given, a default docstring is auto-generated.

Returns A decorator.

Examples

A simple labeling decorator: >>> slow = make_label_dec('slow') >>> print slow.__doc__ Labels a test as 'slow'.

And one that uses multiple labels and a custom docstring: >>> rare = make_label_dec(['slow','hard'], ... "Mix labels 'slow' and 'hard' for rare tests.") >>> print rare.__doc__ Mix labels 'slow' and 'hard' for rare tests.

Now, let's test using this one: >>> @rare ... def f(): pass ... >>> >>> f.slow True >>> f.hard True

IPython.testing.decorators.**numpy_not_available**()
Can numpy be imported? Returns true if numpy does NOT import.

This is used to make a decorator to skip tests that require numpy to be available, but delay the 'import numpy' to test execution time.

IPython.testing.decorators.**skip**()
Decorator factory - mark a test function for skipping from test suite.

Parameters

msg [string] Optional message to be added.

Returns

decorator [function] Decorator, which, when applied to a function, causes SkipTest to be raised, with the optional message added.

IPython.testing.decorators.**skipif**()
Make function raise SkipTest exception if skip_condition is true

Parameters **skip_condition** : bool or callable.

Flag to determine whether to skip test. If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed. msg : string

Message to give on raising a SkipTest exception

Returns **decorator** : function

Decorator, which, when applied to a function, causes SkipTest to be raised when the skip_condition was True, and the function to be called normally otherwise.

Notes

You will see from the code that we had to further decorate the decorator with the nose.tools.make_decorator function in order to transmit function name, and various other metadata.

10.113 testing.decorators_numpy

10.113.1 Module: `testing.decorators_numpy`

Decorators for labeling test objects

Decorators that merely return a modified version of the original function object are straightforward. Decorators that return a new function object need to use nose.tools.make_decorator(original_function)(decorator) in returning the decorator, in order to preserve metadata such as function name, setup and teardown functions and so on - see nose.tools for more information.

10.113.2 Functions

`IPython.testing.decorators_numpy.`**`setastest`**`()`

Signals to nose that this function is or is not a test

Parameters **tf** : bool

If True specifies this is a test, not a test otherwise

This decorator cannot use the nose namespace, because it can be :

called from a non-test module. See also istest and nottest in :

nose.tools :

`IPython.testing.decorators_numpy.`**`skipif`**`()`

Make function raise SkipTest exception if skip_condition is true

Parameters **skip_condition** : bool or callable.

Flag to determine whether to skip test. If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

msg [string] Message to give on raising a SkipTest exception

Returns **decorator** : function

Decorator, which, when applied to a function, causes SkipTest to be raised when the skip_condition was True, and the function to be called normally otherwise.

Notes

You will see from the code that we had to further decorate the decorator with the nose.tools.make_decorator function in order to transmit function name, and various other metadata.

IPython.testing.decorators_numpy.**skipknownfailure**()
Decorator to raise SkipTest for test known to fail

IPython.testing.decorators_numpy.**slow**()
Labels a test as 'slow'.

The exact definition of a slow test is obviously both subjective and hardware-dependent, but in general any individual test that requires more than a second or two should be labeled as slow (the whole suite consits of thousands of tests, so even a second is significant).

10.114 testing.decorators_trial

10.114.1 Module: testing.decorators_trial

Testing related decorators for use with twisted.trial.

The decorators in this files are designed to follow the same API as those in the decorators module (in this same directory). But they can be used with twisted.trial

10.114.2 Functions

IPython.testing.decorators_trial.**numpy_not_available**()
Can numpy be imported? Returns true if numpy does NOT import.

This is used to make a decorator to skip tests that require numpy to be available, but delay the 'import numpy' to test execution time.

IPython.testing.decorators_trial.**skip**()
Create a decorator that marks a test function for skipping.

This is a decorator factory that returns a decorator that will cause tests to be skipped.

Parameters **msg** : str

Optional message to be added.

Returns **decorator** : function

Decorator, which, when applied to a function, sets the skip attribute of the function causing *twisted.trial* to skip it.

IPython.testing.decorators_trial.**skipif**()
Create a decorator that marks a test function for skipping.

The is a decorator factory that returns a decorator that will conditionally skip a test based on the value of skip_condition. The skip_condition argument can either be a boolean or a callable that returns a boolean.

Parameters **skip_condition** : boolean or callable

If this evaluates to True, the test is skipped.

msg : str

The message to print if the test is skipped.

Returns **decorator** : function

The decorator function that can be applied to the test function.

10.115 testing.iptest

10.115.1 Module: `testing.iptest`

Inheritance diagram for `IPython.testing.iptest`:

IPython Test Suite Runner.

This module provides a main entry point to a user script to test IPython itself from the command line. There are two ways of running this script:

1. With the syntax *iptest all*. This runs our entire test suite by calling this script (with different arguments) or trial recursively. This causes modules and package to be tested in different processes, using nose or trial where appropriate.
2. With the regular nose syntax, like *iptest -vvs IPython*. In this form the script simply calls nose, but with special command line flags and plugins loaded.

For now, this script requires that both nose and twisted are installed. This will change in the future.

10.115.2 Class

10.115.3 `IPTester`

class `IPython.testing.iptest.`**`IPTester`**(*runner='iptest'*, *params=None*)

Bases: `object`

Call that calls iptest or trial in a subprocess.

`__init__`()

run()
Run the stored commands

10.115.4 Functions

IPython.testing.iptest.**main**()

IPython.testing.iptest.**make_runners**()
Define the modules and packages that need to be tested.

IPython.testing.iptest.**run_iptest**()
Run the IPython test suite using nose.

This function is called when this script is **not** called with the form *iptest all*. It simply calls nose with appropriate command line flags and accepts all of the standard nose arguments.

IPython.testing.iptest.**run_iptestall**()
Run the entire IPython test suite by calling nose and trial.

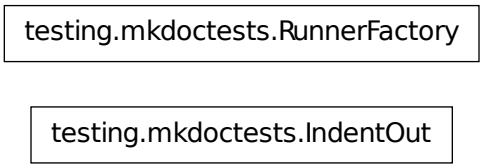
This function constructs IPTester instances for all IPython modules and package and then runs each of them. This causes the modules and packages of IPython to be tested each in their own subprocess using nose or twisted.trial appropriately.

IPython.testing.iptest.**test_for**()
Test to see if mod is importable.

10.116 testing.mkdoctests

10.116.1 Module: testing.mkdoctests

Inheritance diagram for IPython.testing.mkdoctests:



Utility for making a doctest file out of Python or IPython input.

%prog [options] input_file [output_file]

This script is a convenient generator of doctest files that uses IPython's irunner script to execute valid Python or IPython input in a separate process, capture all of the output, and write it to an output file.

It can be used in one of two ways:

1. With a plain Python or IPython input file (denoted by extensions '.py' or '.ipy'. In this case, the output is an auto-generated reST file with a basic header, and the captured Python input and output contained in an indented code block.

   If no output filename is given, the input name is used, with the extension replaced by '.txt'.

2. With an input template file. Template files are simply plain text files with special directives of the form

   %run filename

   to include the named file at that point.

   If no output filename is given and the input filename is of the form 'base.tpl.txt', the output will be automatically named 'base.txt'.

10.116.2 Classes

IndentOut

class IPython.testing.mkdoctests.**IndentOut**(*out=<open file '<stdout>', mode 'w' at 0x403b2078>*, *indent=4*)

Bases: object

A simple output stream that indents all output by a fixed amount.

Instances of this class trap output to a given stream and first reformat it to indent every input line.

__init__()

Create an indented writer.

Keywords

- *out* : stream (sys.stdout) Output stream to actually write to after indenting.
- *indent* : int Number of spaces to indent every input line by.

close()

flush()

write()

Write a string to the output stream.

RunnerFactory

class IPython.testing.mkdoctests.**RunnerFactory**(*out=<open file '<stdout>', mode 'w' at 0x403b2078>*)

Bases: object

Code runner factory.

This class provides an IPython code runner, but enforces that only one runner is every instantiated. The runner is created based on the extension of the first file to run, and it raises an exception if a runner is later requested for a different extension type.

This ensures that we don't generate example files for doctest with a mix of python and ipython syntax.

__init__()
Instantiate a code runner.

10.116.3 Function

IPython.testing.mkdoctests.**main**()
Run as a script.

10.117 testing.parametric

10.117.1 Module: testing.parametric

Parametric testing on top of twisted.trial.unittest.

10.117.2 Functions

IPython.testing.parametric.**Parametric**()
Register parametric tests with a class.

IPython.testing.parametric.**parametric**()
Mark f as a parametric test.

IPython.testing.parametric.**partial**()
Generate a partial class method.

10.118 testing.plugin.dtexample

10.118.1 Module: testing.plugin.dtexample

Simple example using doctests.

This file just contains doctests both using plain python and IPython prompts. All tests should be loaded by nose.

10.118.2 Functions

IPython.testing.plugin.dtexample.**ipfunc**()
Some ipython tests...

In [1]: import os

In [3]: 2+3 Out[3]: 5

In [26]: for i in range(3): ....: print i, ...: print i+1, ...:

0 1 1 2 2 3

Examples that access the operating system work:

In [1]: !echo hello hello

In [2]: !echo hello > /tmp/foo

In [3]: !cat /tmp/foo hello

In [4]: rm -f /tmp/foo

It's OK to use '_' for the last result, but do NOT try to use IPython's numbered history of _NN outputs, since those won't exist under the doctest environment:

In [7]: 'hi' Out[7]: 'hi'

In [8]: print repr(_) 'hi'

In [7]: 3+4 Out[7]: 7

In [8]: _+3 Out[8]: 10

In [9]: ipfunc() Out[9]: 'ipfunc'

`IPython.testing.plugin.dtexample.`**`iprand`**`()`

Some ipython tests with random output.

In [7]: 3+4 Out[7]: 7

In [8]: print 'hello' world # random

In [9]: iprand() Out[9]: 'iprand'

`IPython.testing.plugin.dtexample.`**`iprand_all`**`()`

Some ipython tests with fully random output.

all-random

In [7]: 1 Out[7]: 99

In [8]: print 'hello' world

In [9]: iprand_all() Out[9]: 'junk'

`IPython.testing.plugin.dtexample.`**`pyfunc`**`()`

Some pure python tests...

```
>>> pyfunc()
'pyfunc'

>>> import os
```

```
>>> 2+3
5

>>> for i in range(3):
...     print i,
...     print i+1,
...
0 1 1 2 2 3
```

IPython.testing.plugin.dtexample.**random_all**()

A function where we ignore the output of ALL examples.

Examples:

all-random

This mark tells the testing machinery that all subsequent examples should be treated as random (ignoring their output). They are still executed, so if a they raise an error, it will be detected as such, but their output is completely ignored.

```
>>> 1+3
junk goes here...

>>> 1+3
klasdfj;

>>> 1+2
again,  anything goes
blah...
```

IPython.testing.plugin.dtexample.**ranfunc**()

A function with some random output.

Normal examples are verified as usual: >>> 1+3 4

But if you put '# random' in the output, it is ignored: >>> 1+3 junk goes here... # random

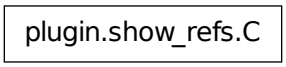
```
>>> 1+2
again,  anything goes #random
if multiline, the random mark is only needed once.

>>> 1+2
You can also put the random marker at the end:
# random

>>> 1+2
# random
.. or at the beginning.
```

More correct input is properly verified: >>> ranfunc() 'ranfunc'

10.119 testing.plugin.show_refs

10.119.1 Module: testing.plugin.show_refs

Inheritance diagram for IPython.testing.plugin.show_refs:



Simple script to show reference holding behavior.

This is used by a companion test case.

10.119.2 C

class IPython.testing.plugin.show_refs.**C**
Bases: object

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

10.120 testing.plugin.simple

10.120.1 Module: testing.plugin.simple

Simple example using doctests.

This file just contains doctests both using plain python and IPython prompts. All tests should be loaded by nose.

10.120.2 Functions

IPython.testing.plugin.simple.**ipyfunc2**()
Some pure python tests...

```
>>> 1+1
2
```

IPython.testing.plugin.simple.**pyfunc**()
Some pure python tests...

```
>>> pyfunc()
'pyfunc'

>>> import os

>>> 2+3
5

>>> for i in range(3):
...     print i,
...     print i+1,
...
0 1 1 2 2 3
```

10.121 testing.plugin.test_ipdoctest

10.121.1 Module: testing.plugin.test_ipdoctest

Tests for the ipdoctest machinery itself.

Note: in a file named test_X, functions whose only test is their docstring (as a doctest) and which have no test functionality of their own, should be called 'doctest_foo' instead of 'test_foo', otherwise they get double-counted (the empty function call is counted as a test, which just inflates tests numbers artificially).

10.121.2 Functions

IPython.testing.plugin.test_ipdoctest.**doctest_multiline1**()
The ipdoctest machinery must handle multiline examples gracefully.

In [2]: for i in range(10): ...: print i, ...:

0 1 2 3 4 5 6 7 8 9

IPython.testing.plugin.test_ipdoctest.**doctest_multiline2**()
Multiline examples that define functions and print output.

In [7]: def f(x): ...: return x+1 ...:

In [8]: f(1) Out[8]: 2

In [9]: def g(x): ...: print 'x is:',x ...:

In [10]: g(1) x is: 1

In [11]: g('hello') x is: hello

IPython.testing.plugin.test_ipdoctest.**doctest_multiline3**()
Multiline examples with blank lines.

In [12]: def h(x): ....: if x>1: ....: return x**2 ....: # To leave a blank line in the input, you must mark it ....: # with a comment character: ....: # ....: # otherwise the doctest parser gets confused. ....: else: ....: return -1 ....:

In [13]: h(5) Out[13]: 25

In [14]: h(1) Out[14]: -1

In [15]: h(0) Out[15]: -1

IPython.testing.plugin.test_ipdoctest.**doctest_run_builtins**()
Check that %run doesn't damage __builtins__ via a doctest.

This is similar to the test_run_builtins, but I want *both* forms of the test to catch any possible glitches in our testing machinery, since that modifies %run somewhat. So for this, we have both a normal test (below) and a doctest (this one).

In [1]: import tempfile

In [3]: f = tempfile.NamedTemporaryFile()

In [4]: f.write('passn')

In [5]: f.flush()

In [7]: %run $f.name

IPython.testing.plugin.test_ipdoctest.**doctest_simple**()
ipdoctest must handle simple inputs

In [1]: 1 Out[1]: 1

In [2]: print 1 1

10.122 testing.plugin.test_refs

10.122.1 Module: testing.plugin.test_refs

Some simple tests for the plugin while running scripts.

10.122.2 Functions

IPython.testing.plugin.test_refs.**doctest_ivars**()
Test that variables defined interactively are picked up. In [5]: zz=1

In [6]: zz Out[6]: 1

IPython.testing.plugin.test_refs.**doctest_refs**()
DocTest reference holding issues when running scripts.

In [32]: run show_refs.py c referrers: [<type 'dict'>]

IPython.testing.plugin.test_refs.**doctest_run**()
Test running a trivial script.

In [13]: run simplevars.py x is: 1

IPython.testing.plugin.test_refs.**doctest_runvars**()
Test that variables defined in scripts get loaded correcly via %run.

In [13]: run simplevars.py x is: 1

In [14]: x Out[14]: 1

IPython.testing.plugin.test_refs.**test_trivial**()
A trivial passing test.

10.123 testing.tools

10.123.1 Module: testing.tools

Generic testing tools that do NOT depend on Twisted.

In particular, this module exposes a set of top-level assert* functions that can be used in place of nose.tools.assert* in method generators (the ones in nose can not, at least as of nose 0.10.4).

Note: our testing package contains testing.util, which does depend on Twisted and provides utilities for tests that manage Deferreds. All testing support tools that only depend on nose, IPython or the standard library should go here instead.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

10.123.2 Functions

IPython.testing.tools.**full_path**()
Make full paths for all the listed files, based on startPath.

Only the base part of startPath is kept, since this routine is typically used with a script's __file__ variable as startPath. The base of startPath is then prepended to all the listed files, forming the output list.

Parameters **startPath** : string

Initial path to use as the base for the results. This path is split

using os.path.split() and only its first component is kept.

files [string or list] One or more files.

Examples

```
>>> full_path('/foo/bar.py',['a.txt','b.txt'])
['/foo/a.txt', '/foo/b.txt']
```

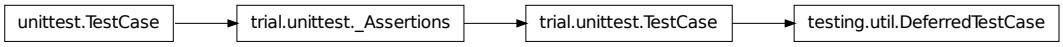
```
>>> full_path('/foo',['a.txt','b.txt'])
['/a.txt', '/b.txt']
```

If a single file is given, the output is still a list: >>> full_path('/foo','a.txt') ['/a.txt']

10.124 testing.util

10.124.1 Module: `testing.util`

Inheritance diagram for `IPython.testing.util`:



This file contains utility classes for performing tests with Deferreds.

10.124.2 `DeferredTestCase`

class `IPython.testing.util.`**DeferredTestCase**(*methodName='runTest'*)

Bases: `twisted.trial.unittest.TestCase`

__init__()

Construct an asynchronous test case for C{methodName}.

@param methodName: The name of a method on C{self}. This method should be a unit test. That is, it should be a short method that calls some of the assert* methods. If C{methodName} is unspecified, L{runTest} will be used as the test method. This is mostly useful for testing Trial.

assertDeferredEquals()

Calls assertEquals on the result of the deferred and expectedResult.

chainDeferred can be used to pass in previous Deferred objects that have tests being run on them. This chaining of Deferred's in tests is needed to insure that all Deferred's are cleaned up at the end of a test.

assertDeferredRaises()

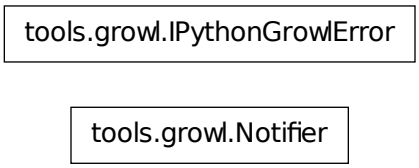
Calls assertRaises on the Failure of the deferred and expectedException.

chainDeferred can be used to pass in previous Deferred objects that have tests being run on them. This chaining of Deferred's in tests is needed to insure that all Deferred's are cleaned up at the end of a test.

10.125 tools.growl

10.125.1 Module: tools.growl

Inheritance diagram for IPython.tools.growl:



10.125.2 Classes

IPythonGrowlError

class IPython.tools.growl.**IPythonGrowlError**
Bases: exceptions.Exception

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Notifier

class IPython.tools.growl.**Notifier**(*app_name*)
Bases: object

__init__()

notify()

notify_deferred()

10.125.3 Functions

IPython.tools.growl.**notify**()

IPython.tools.growl.**notify_deferred**()

IPython.tools.growl.**start**()

10.126 tools.utils

10.126.1 Module: `tools.utils`

Generic utilities for use by IPython's various subsystems.

10.126.2 Functions

IPython.tools.utils.**extractVars**()
Extract a set of variables by name from another frame.

Parameters

- **names*: strings One or more variable names which will be extracted from the caller's

frame.

Keywords

- *depth*: integer (0) How many frames in the stack to walk when looking for your variables.

Examples:

In [2]: def func(x): ...: y = 1 ...: print extractVars('x','y') ...:

In [3]: func('hello') {'y': 1, 'x': 'hello'}

IPython.tools.utils.**extractVarsAbove**()
Extract a set of variables by name from another frame.

Similar to extractVars(), but with a specified depth of 1, so that names are exctracted exactly from above the caller.

This is simply a convenience function so that the very common case (for us) of skipping exactly 1 frame doesn't have to construct a special dict for keyword passing.

IPython.tools.utils.**list_strings**()
Always return a list of strings, given a string or list of strings as input.

Examples In [7]: list_strings('A single string') Out[7]: ['A single string']

In [8]: list_strings(['A single string in a list']) Out[8]: ['A single string in a list']

In [9]: list_strings(['A','list','of','strings']) Out[9]: ['A', 'list', 'of', 'strings']

IPython.tools.utils.**marquee**()
Return the input string centered in a 'marquee'.

Examples In [16]: marquee('A test',40) Out[16]: '********** A test **********'

In [17]: marquee('A test',40,'-') Out[17]: '———————- A test ———————-'

In [18]: marquee('A test',40,' ') Out[18]: ' A test '

IPython.tools.utils.**shexp**()

Expand $VARS and ~names in a string, like a shell

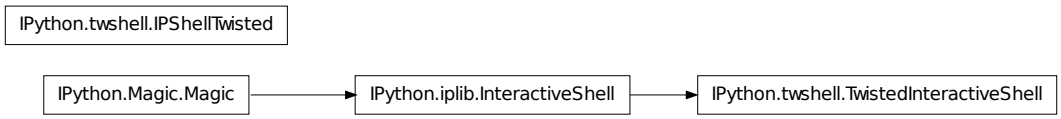
Examples In [2]: os.environ['FOO']='test'

In [3]: shexp('variable FOO is $FOO') Out[3]: 'variable FOO is test'

10.127 twshell

10.127.1 Module: twshell

Inheritance diagram for IPython.twshell:



Twisted shell support.

XXX - This module is missing proper docs.

10.127.2 Classes

IPShellTwisted

class IPython.twshell.**IPShellTwisted**(*argv=None*, *user_ns=None*, *debug=1*, *shell_class=<class 'IPython.twshell.TwistedInteractiveShell'>*)

Run a Twisted reactor while in an IPython session.

Python commands can be passed to the thread where they will be executed. This is implemented by periodically checking for passed code using a Twisted reactor callback.

__init__()

mainloop()

run()

TwistedInteractiveShell

class IPython.twshell.**TwistedInteractiveShell**(*name*, *usage=None*, *rc=Struct({'__allownew': True, 'args': None, 'opts': None})*, *user_ns=None*, *user_global_ns=None*, *banner2=''*, ***kw*)

Bases: IPython.iplib.InteractiveShell

Simple multi-threaded shell.

__init__()
Similar to the normal InteractiveShell, but with threading control

kill()
Kill the thread, returning when it has been shut down.

runcode()
Execute a code object.

Multithreaded wrapper around IPython's runcode().

runsource()
Compile and run some source in the interpreter.

Modified version of code.py's runsource(), to handle threading issues. See the original for full docstring details.

10.127.3 Functions

IPython.twshell.**hijack_reactor**()
Modifies Twisted's reactor with a dummy so user code does not block IPython. This function returns the original 'twisted.internet.reactor' that has been hijacked.

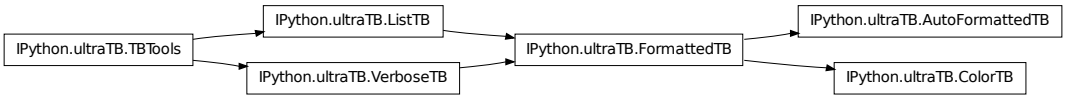
NOTE: Make sure you call this *AFTER* you've installed the reactor of your choice.

IPython.twshell.**install_gtk2**()
Install gtk2 reactor, needs to be called bef

10.128 ultraTB

10.128.1 Module: ultraTB

Inheritance diagram for IPython.ultraTB:



ultraTB.py – Spice up your tracebacks!

- ColorTB

I've always found it a bit hard to visually parse tracebacks in Python. The ColorTB class is a solution to that problem. It colors the different parts of a traceback in a manner similar to what you would expect from a syntax-highlighting text editor.

Installation instructions for ColorTB: import sys,ultraTB sys.excepthook = ultraTB.ColorTB()

- VerboseTB

I've also included a port of Ka-Ping Yee's "cgitb.py" that produces all kinds of useful info when a traceback occurs. Ping originally had it spit out HTML and intended it for CGI programmers, but why should they have all the fun? I altered it to spit out colored text to the terminal. It's a bit overwhelming, but kind of neat, and maybe useful for long-running programs that you believe are bug-free. If a crash *does* occur in that type of program you want details. Give it a shot–you'll love it or you'll hate it.

Note:

> The Verbose mode prints the variables currently visible where the exception happened (shortening their strings if too long). This can potentially be very slow, if you happen to have a huge data structure whose string representation is complex to compute. Your computer may appear to freeze for a while with cpu usage at 100%. If this occurs, you can cancel the traceback with Ctrl-C (maybe hitting it more than once).
>
> If you encounter this kind of situation often, you may want to use the Verbose_novars mode instead of the regular Verbose, which avoids formatting variables (but otherwise includes the information and context given by Verbose).

Installation instructions for ColorTB: import sys,ultraTB sys.excepthook = ultraTB.VerboseTB()

Note: Much of the code in this module was lifted verbatim from the standard library module 'traceback.py' and Ka-Ping Yee's 'cgitb.py'.

- Color schemes

The colors are defined in the class TBTools through the use of the ColorSchemeTable class. Currently the following exist:

- NoColor: allows all of this module to be used in any terminal (the color

escapes are just dummy blank strings).

- Linux: is meant to look good in a terminal like the Linux console (black

or very dark background).

- LightBG: similar to Linux but swaps dark/light colors to be more readable

in light background terminals.

You can implement other color schemes easily, the syntax is fairly self-explanatory. Please send back new schemes you develop to the author for possible inclusion in future releases.

10.128.2 Classes

AutoFormattedTB

class IPython.ultraTB.**AutoFormattedTB**(*mode='Plain'*, *color_scheme='Linux'*, *tb_offset=0*, *long_header=0*, *call_pdb=0*, *include_vars=0*)

Bases: IPython.ultraTB.FormattedTB

A traceback printer which can be called on the fly.

It will find out about exceptions by itself.

A brief example:

AutoTB = AutoFormattedTB(mode = 'Verbose',color_scheme='Linux') try:

...

except: AutoTB() # or AutoTB(out=logfile) where logfile is an open file object

__init__()

text()

ColorTB

class IPython.ultraTB.**ColorTB**(*color_scheme='Linux'*, *call_pdb=0*)

Bases: IPython.ultraTB.FormattedTB

Shorthand to initialize a FormattedTB in Linux colors mode.

__init__()

FormattedTB

class IPython.ultraTB.**FormattedTB**(*mode='Plain'*, *color_scheme='Linux'*, *tb_offset=0*, *long_header=0*, *call_pdb=0*, *include_vars=0*)

Bases: IPython.ultraTB.VerboseTB, IPython.ultraTB.ListTB

Subclass ListTB but allow calling with a traceback.

It can thus be used as a sys.excepthook for Python > 2.1.

Also adds 'Context' and 'Verbose' modes, not available in ListTB.

Allows a tb_offset to be specified. This is useful for situations where one needs to remove a number of topmost frames from the traceback (such as occurs with python programs that themselves execute other python code, like Python shells).

__init__()

context()

plain()

set_mode()
Switch to the desired mode.

If mode is not specified, cycles through the available modes.

text()
Return formatted traceback.

If the optional mode parameter is given, it overrides the current mode.

verbose()

ListTB

class IPython.ultraTB.**ListTB**(*color_scheme='NoColor'*)
Bases: IPython.ultraTB.TBTools

Print traceback information from a traceback list, with optional color.

Calling: requires 3 arguments: (etype, evalue, elist)

as would be obtained by: etype, evalue, tb = sys.exc_info() if tb:

elist = traceback.extract_tb(tb)

else: elist = None

It can thus be used by programs which need to process the traceback before printing (such as console replacements based on the code module from the standard library).

Because they are meant to be called without a full traceback (only a list), instances of this class can't call the interactive pdb debugger.

__init__()

text()
Return a color formatted string with the traceback info.

TBTools

class IPython.ultraTB.**TBTools**(*color_scheme='NoColor'*, *call_pdb=False*)
Basic tools used by all traceback printer classes.

__init__()

color_toggle()
Toggle between the currently active color scheme and NoColor.

set_colors()
Shorthand access to the color table scheme selector method.

VerboseTB

class IPython.ultraTB.**VerboseTB**(*color_scheme='Linux'*, *tb_offset=0*, *long_header=0*, *call_pdb=0*, *include_vars=1*)
Bases: IPython.ultraTB.TBTools

A port of Ka-Ping Yee's cgitb.py module that outputs color text instead of HTML. Requires inspect and pydoc. Crazy, man.

Modified version which optionally strips the topmost entries from the traceback, to be used with alternate interpreters (because their own code would appear in the traceback).

__init__()
Specify traceback offset, headers and color scheme.

Define how many frames to drop from the tracebacks. Calling it with tb_offset=1 allows use of this handler in interpreters which will have their own code at the top of the traceback (VerboseTB will first remove that frame before printing the traceback info).

debugger()
Call up the pdb debugger if desired, always clean up the tb reference.

Keywords:

•force(False): by default, this routine checks the instance call_pdb

flag and does not actually invoke the debugger if the flag is false. The 'force' option forces the debugger to activate even if the flag is false.

If the call_pdb flag is set, the pdb interactive debugger is invoked. In all cases, the self.tb reference to the current traceback is deleted to prevent lingering references which hamper memory management.

Note that each call to pdb() does an 'import readline', so if your app requires a special setup for the readline completers, you'll have to fix that by hand after invoking the exception handler.

handler()

text()
Return a nice text document describing the traceback.

10.128.3 Functions

IPython.ultraTB.**findsource**()
Return the entire source file and starting line number for an object.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of all the lines in the file and the line number indexes a line in that list. An IOError is raised if the source code cannot be retrieved.

FIXED version with which we monkeypatch the stdlib to work around a bug.

IPython.ultraTB.**fix_frame_records_filenames**()
Try to fix the filenames in each record from inspect.getinnerframes().

Particularly, modules loaded from within zip files have useless filenames attached to their code object, and inspect.getinnerframes() just uses it.

IPython.ultraTB.**inspect_error**()
Print a message about internal inspect errors.

These are unfortunately quite common.

10.129 upgrade_dir

10.129.1 Module: upgrade_dir

A script/util to upgrade all files in a directory

This is rather conservative in its approach, only copying/overwriting new and unedited files.

To be used by "upgrade" feature.

10.129.2 Functions

IPython.upgrade_dir.**showdiff**()

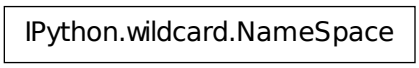
IPython.upgrade_dir.**upgrade_dir**()
Copy over all files in srcdir to tgtdir w/ native line endings

Creates .upgrade_report in tgtdir that stores md5sums of all files to notice changed files b/w upgrades.

10.130 wildcard

10.130.1 Module: wildcard

Inheritance diagram for IPython.wildcard:



Support for wildcard pattern matching in object inspection.

Authors

- Jörgen Stenarson <jorgen.stenarson@bostream.nu>

10.130.2 Class

10.130.3 NameSpace

class IPython.wildcard.**NameSpace**(*obj*, *name_pattern='*'*, *type_pattern='all'*, *ignore_case=True*, *show_all=True*)

Bases: object

NameSpace holds the dictionary for a namespace and implements filtering on name and types

__init__()

filter()
Return dictionary of filtered namespace.

get_ns()
Return name space dictionary with objects matching type and name patterns.

get_ns_names()
Return list of object names in namespace that match the patterns.

ns
Return name space dictionary with objects matching type and name patterns.

ns_names
List of objects in name space that match the type and name patterns.

10.130.4 Functions

IPython.wildcard.**create_typestr2type_dicts**()
Return dictionaries mapping lower case typename to type objects, from the types package, and vice versa.

IPython.wildcard.**is_type**()
is_type(obj,typestr_or_type) verifies if obj is of a certain type or group of types takes strings as parameters of the for 'tuple'<->TupleType 'all' matches all types. TODO: Should be extended for choosing more than one type

IPython.wildcard.**list_namespace**()
Return dictionary of all objects in namespace that matches type_pattern and filter.

IPython.wildcard.**show_hidden**()
Return true for strings starting with single _ if show_all is true.

10.131 winconsole

10.131.1 Module: winconsole

Set of functions to work with console on Windows.

IPython.winconsole.**get_console_size**()
Return size of current console.

This function try to determine actual size of current working console window and return tuple (sizex, sizey) if success, or default size (defaultx, defaulty) otherwise.

Dependencies: ctypes should be installed.

CHAPTER

ELEVEN

LICENSE AND COPYRIGHT

11.1 License

IPython is licensed under the terms of the new or revised BSD license, as follows:

11.2 About the IPython Development Team

Fernando Perez began IPython in 2001 based on code from Janko Hauser <jhauser@zscout.de> and Nathaniel Gray <n8gray@caltech.edu>. Fernando is still the project lead.

The IPython Development Team is the set of all contributors to the IPython project. This includes all of the IPython subprojects. Here is a list of the currently active contributors:

- Matthieu Brucher
- Ondrej Certik
- Laurent Dufrechou
- Robert Kern
- Brian E. Granger
- Fernando Perez (project leader)
- Benjamin Ragan-Kelley
- Ville M. Vainio
- Gael Varoququx
- Stefan van der Walt
- Tech-X Corporation
- Barry Wark

If your name is missing, please add it.

11.3 Our Copyright Policy

IPython uses a shared copyright model. Each contributor maintains copyright over their contributions to IPython. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the IPython source code, in its entirety is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire IPython Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the IPython repositories.

11.4 Miscellaneous

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

Versions of IPython up to and including 0.6.3 were released under the GNU Lesser General Public License (LGPL), available at http://www.gnu.org/copyleft/lesser.html.

CHAPTER

TWELVE

CREDITS

IPython is led by Fernando Pérez.

As of this writing, the following developers have joined the core team:

- [Robert Kern] <rkern-AT-enthought.com>: co-mentored the 2005 Google Summer of Code project to develop python interactive notebooks (XML documents) and graphical interface. This project was awarded to the students Tzanko Matev <tsanko-AT-gmail.com> and Toni Alatalo <antont-AT-an.org>.
- [Brian Granger] <ellisonbg-AT-gmail.com>: extending IPython to allow support for interactive parallel computing.
- [Benjamin (Min) Ragan-Kelley]: key work on IPython's parallel computing infrastructure.
- [Ville Vainio] <vivainio-AT-gmail.com>: Ville has made many improvements to the core of IPython and was the maintainer of the main IPython trunk from version 0.7.1 to 0.8.4.
- [Gael Varoquaux] <gael.varoquaux-AT-normalesup.org>: work on the merged architecture for the interpreter as of version 0.9, implementing a new WX GUI based on this system.
- [Barry Wark] <barrywark-AT-gmail.com>: implementing a new Cocoa GUI, as well as work on the new interpreter architecture and Twisted support.
- [Laurent Dufrechou] <laurent.dufrechou-AT-gmail.com>: development of the WX GUI support.
- [Jörgen Stenarson] <jorgen.stenarson-AT-bostream.nu>: maintainer of the PyReadline project, necessary for IPython under windows.

The IPython project is also very grateful to:

Bill Bumgarner <bbum-AT-friday.com>: for providing the DPyGetOpt module which gives very powerful and convenient handling of command-line options (light years ahead of what Python 2.1.1's getopt module does).

Ka-Ping Yee <ping-AT-lfw.org>: for providing the Itpl module for convenient and powerful string interpolation with a much nicer syntax than formatting through the '%' operator.

Arnd Baecker <baecker-AT-physik.tu-dresden.de>: for his many very useful suggestions and comments, and lots of help with testing and documentation checking. Many of IPython's newer features are a result of discussions with him (bugs are still my fault, not his).

Obviously Guido van Rossum and the whole Python development team, that goes without saying.

IPython's website is generously hosted at http://ipython.scipy.orgby Enthought (http://www.enthought.com). I am very grateful to them and all of the SciPy team for their contribution.

Fernando would also like to thank Stephen Figgins <fig-AT-monitor.net>, an O'Reilly Python editor. His Oct/11/2001 article about IPP and LazyPython, was what got this project started. You can read it at: http://www.onlamp.com/pub/a/python/2001/10/11/pythonnews.html.

And last but not least, all the kind IPython users who have emailed new code, bug reports, fixes, comments and ideas. A brief list follows, please let us know if we have ommitted your name by accident:

- Justin Riley <justin.t.riley-AT-gmail.com> Contributions to parallel support, Amazon EC2, Sun Grid Engine, documentation.
- Satrajit Ghosh <satra-AT-mit.edu> parallel computing (SGE and much more).
- Darren Dale <dsdale24-AT-gmail.com>, traits-based configuration system, Qt support.
- Dan Milstein <danmil-AT-comcast.net>. A bold refactoring of the core prefilter stuff in the IPython interpreter.
- [Jack Moffit] <jack-AT-xiph.org> Bug fixes, including the infamous color problem. This bug alone caused many lost hours and frustration, many thanks to him for the fix. I've always been a fan of Ogg & friends, now I have one more reason to like these folks. Jack is also contributing with Debian packaging and many other things.
- [Alexander Schmolck] <a.schmolck-AT-gmx.net> Emacs work, bug reports, bug fixes, ideas, lots more. The ipython.el mode for (X)Emacs is Alex's code, providing full support for IPython under (X)Emacs.
- [Andrea Riciputi] <andrea.riciputi-AT-libero.it> Mac OSX information, Fink package management.
- [Gary Bishop] <gb-AT-cs.unc.edu> Bug reports, and patches to work around the exception handling idiosyncracies of WxPython. Readline and color support for Windows.
- [Jeffrey Collins] <Jeff.Collins-AT-vexcel.com> Bug reports. Much improved readline support, including fixes for Python 2.3.
- [Dryice Liu] <dryice-AT-liu.com.cn> FreeBSD port.
- [Mike Heeter] <korora-AT-SDF.LONESTAR.ORG>
- [Christopher Hart] <hart-AT-caltech.edu> PDB integration.
- [Milan Zamazal] <pdm-AT-zamazal.org> Emacs info.
- [Philip Hisley] <compsys-AT-starpower.net>
- [Holger Krekel] <pyth-AT-devel.trillke.net> Tab completion, lots more.
- [Robin Siebler] <robinsiebler-AT-starband.net>
- [Ralf Ahlbrink] <ralf_ahlbrink-AT-web.de>
- [Thorsten Kampe] <thorsten-AT-thorstenkampe.de>
- [Fredrik Kant] <fredrik.kant-AT-front.com> Windows setup.
- [Syver Enstad] <syver-en-AT-online.no> Windows setup.

- [Richard] <rxe-AT-renre-europe.com> Global embedding.
- [Hayden Callow] <h.callow-AT-elec.canterbury.ac.nz> Gnuplot.py 1.6 compatibility.
- [Leonardo Santagada] <retype-AT-terra.com.br> Fixes for Windows installation.
- [Christopher Armstrong] <radix-AT-twistedmatrix.com> Bugfixes.
- [Francois Pinard] <pinard-AT-iro.umontreal.ca> Code and documentation fixes.
- [Cory Dodt] <cdodt-AT-fcoe.k12.ca.us> Bug reports and Windows ideas. Patches for Windows installer.
- [Olivier Aubert] <oaubert-AT-bat710.univ-lyon1.fr> New magics.
- [King C. Shu] <kingshu-AT-myrealbox.com> Autoindent patch.
- [Chris Drexler] <chris-AT-ac-drexler.de> Readline packages for Win32/CygWin.
- [Gustavo Cordova Avila] <gcordova-AT-sismex.com> EvalDict code for nice, lightweight string interpolation.
- [Kasper Souren] <Kasper.Souren-AT-ircam.fr> Bug reports, ideas.
- [Gever Tulley] <gever-AT-helium.com> Code contributions.
- [Ralf Schmitt] <ralf-AT-brainbot.com> Bug reports & fixes.
- [Oliver Sander] <osander-AT-gmx.de> Bug reports.
- [Rod Holland] <rhh-AT-structurelabs.com> Bug reports and fixes to logging module.
- [Daniel 'Dang' Griffith] <pythondev-dang-AT-lazytwinacres.net> Fixes, enhancement suggestions for system shell use.
- [Viktor Ransmayr] <viktor.ransmayr-AT-t-online.de> Tests and reports on Windows installation issues. Contributed a true Windows binary installer.
- [Mike Salib] <msalib-AT-mit.edu> Help fixing a subtle bug related to traceback printing.
- [W.J. van der Laan] <gnufnork-AT-hetdigitalegat.nl> Bash-like prompt specials.
- [Antoon Pardon] <Antoon.Pardon-AT-rece.vub.ac.be> Critical fix for the multithreaded IPython.
- [John Hunter] <jdhunter-AT-nitace.bsd.uchicago.edu> Matplotlib author, helped with all the development of support for matplotlib in IPyhton, including making necessary changes to matplotlib itself.
- [Matthew Arnison] <maffew-AT-cat.org.au> Bug reports, '%run -d' idea.
- [Prabhu Ramachandran] <prabhu_r-AT-users.sourceforge.net> Help with (X)Emacs support, threading patches, ideas...
- [Norbert Tretkowski] <tretkowski-AT-inittab.de> help with Debian packaging and distribution.
- [George Sakkis] <gsakkis-AT-eden.rutgers.edu> New matcher for tab-completing named arguments of user-defined functions.
- [Jörgen Stenarson] <jorgen.stenarson-AT-bostream.nu> Wildcard support implementation for searching namespaces.

- [Vivian De Smedt] <vivian-AT-vdesmedt.com> Debugger enhancements, so that when pdb is activated from within IPython, coloring, tab completion and other features continue to work seamlessly.
- [Scott Tsai] <scottt958-AT-yahoo.com.tw> Support for automatic editor invocation on syntax errors (see http://www.scipy.net/roundup/ipython/issue36).
- [Alexander Belchenko] <bialix-AT-ukr.net> Improvements for win32 paging system.
- [Will Maier] <willmaier-AT-ml1.net> Official OpenBSD port.
- [Ondrej Certik] <ondrej-AT-certik.cz>: set up the IPython docs to use the new Sphinx system used by Python, Matplotlib and many more projects.
- [Stefan van der Walt] <stefan-AT-sun.ac.za>: support for the new config system.

BIBLIOGRAPHY

[Twisted] Twisted matrix. http://twistedmatrix.org

[ZopeInterface] http://pypi.python.org/pypi/zope.interface

[Foolscap] Foolscap network protocol. http://foolscap.lothar.com/trac

[pyOpenSSL] pyOpenSSL. http://pyopenssl.sourceforge.net

[Capability] Capability-based security, http://en.wikipedia.org/wiki/Capability-based_security

[PBS] Portable Batch System. http://www.openpbs.org/

[SGE] Sun Grid Engine. http://www.sun.com/software/sge/

[LSF] Load Sharing Facility. http://www.platform.com/

[SSH] SSH-Agent http://en.wikipedia.org/wiki/Ssh-agent

[MPI] Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/

[mpi4py] MPI for Python. mpi4py: http://mpi4py.scipy.org/

[OpenMPI] Open MPI. http://www.open-mpi.org/

[PyTrilinos] PyTrilinos. http://trilinos.sandia.gov/packages/pytrilinos/

[RFC5246] <http://tools.ietf.org/html/rfc5246>

[Bazaar] Bazaar. http://bazaar-vcs.org/

[Launchpad] Launchpad. http://www.launchpad.net/ipython

[reStructuredText] reStructuredText. http://docutils.sourceforge.net/rst.html

[Sphinx] Sphinx. http://sphinx.pocoo.org/

[Nose] Nose: a discovery based unittest extension. http://code.google.com/p/python-nose/

PYTHON MODULE INDEX

i

INDEX

Symbols

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

X

Z