

Interactive work in Python

IPython's present and future

Fernando Pérez
fperez@colorado.edu

Fast Algorithms Group
Dept. of Applied Mathematics

CU BOULDER

SciPy'04
Caltech, Sept. 3 2004

First things first: BOF (last night) promise

- One of the outcomes of last night's discussion:
 - We need better centralized information about Python and Scientific computing.
 - The Scipy.org website should serve as a holding site for this info
 - It already has a Wiki
- Mail me your science & Python bookmarks/links/pet websites:
 - I'll put up a webpage with the links, organized roughly by category, and with short (2-line) summaries for each.
 - I'll contact python.org for authorization to copy their scientific computing page.
 - The result will be part of the Scipy.org Wiki.
 - From then on, the community better step up to fix my mistakes, enhance it, ...
 - I'm only committing to version ϵ of this.
 - Mail to: Fernando Pérez <fperez@colorado.edu>
- There are other ways to contribute (Joe Harrington's notes), but they require more work. I only offered to do the easy stuff.

Acknowledgements

- Enthought (Eric Jones): hosting IPython (<http://ipython.scipy.org>), the SciPy package, ...
- Nathan Gray (Caltech) and Janko Hauser: authors of the code IPython started from.
- IPython user community: too many to name here.
- John Hunter: matplotlib's author (<http://matplotlib.sourceforge.net>). Much of the recent interesting work for scientific computing happened thanks to John. This opens the door for Envisage integration.

Outline

- Why IPython?
- Design ideas
- Features and demo
- IPython as an adaptable framework
- IPython and plotting (Gnuplot & Matplotlib)
- Status and future

IPython talk from Scipy'03 (some overlap with this one, obviously):

<http://amath.colorado.edu/faculty/fperez/python/scipy03/ipython.pdf>

Why IPython? Overview

The **interactive prompt**: one of Python's greatest strengths.

But: it feels like a half-implemented idea (vs. the Unix shell, or Mathematica's prompt)

IPython is an LGPL Python shell replacement. It tries to be:

1. **A better Python shell**: object introspection, system access, @magic commands, ...
2. **An embeddable interpreter**: debugging, mixing batch-processing with interactive work.
3. **A flexible framework**: you can use it as the base environment for other systems with Python as the underlying language. It is very configurable in this direction.

Portability

- 100% pure Python, works with Python ≥ 2.2
- Developed in **Linux**, it runs under any Unix (including CygWin and Mac OS X).
- **Windows**: OK, but not perfect (issues with readline and ctypes 0.9). I have no Windows machines, so this always lags behind.

Design ideas

A good interactive shell is a key part of a [scientific] computing environment.

Some ideas underlying IPython's design:

- **Make every keystroke count:** do the most with the least typing.
- **Meta-control:** `@magics` control and extend IPython itself.
- **System access:** why should your shell shield you from the OS?
- **Efficient development:**
 - Object introspection: TAB-completion, '?', '??', '@p...' functions.
 - Better tracebacks: colored, longer and with data details.
 - `@run`: `execfile()` on steroids.
 - Profiler: quick and easy profile access via `@prun` and `@run -p`.
 - Debugger: automatic `pdb` triggering on uncaught exceptions.
- **I don't know what you want to do:** easy to extend and customize for specific problems.

Basic interactive features

- `@magic` functions: IPython control, system access, namespace information, etc. This was part of Janko's original work. User-extensible.
- Object introspection with `'?'` and `'??'`.
- TAB-completion in the local namespace and filesystem (via readline).
- Numbered prompts with command history, searching and caching:
 - **Output:** stored in the global `Out` and `_N` (`Out[4]==_4`). For convenience, `_`, `__` and `___` keep the last three results.
 - **Input:** stored in the global `In`. Re-execute code with `'exec In[22:29]+In[34]'`.
 - `@macro`: `'@macro mm 22:29 24'` → type `'mm'` to execute.
 - `@hist` shows previous input history.
 - `Ctrl-p/n`: search previous/next match in history.
- Automatic indentation of typed text (on by default, toggle with `@autoindent`).

- `@edit`: direct access to your `$EDITOR`. This mimics reasonably well multi-line editing capabilities, without the complexity (for me) of a curses interface. IPython can also be used as the Python shell in (X)Emacs.
- Verbose and `colored` exception traceback printouts. Easy to read, they include more information than the default ones. Use `@xmode` to change modes. Based on a text port of Ka Ping Yee's `cgitb` module by Nathan Gray.

- Auto-calling functions:

```
In [13]: /my_fun 0,1    ← The initial '/' is optional for callables
-----> my_fun(0,1)
Out [13]: (0, 1)
```

- Auto-quoting function arguments:

```
In [10]: ,my_fun a b    ← Quotes each argument separately
-----> my_fun("a", "b")
Out [10]: ('a', 'b')
```

- Session logging and restoring (`@logstart`, `@logon/off`, `@runlog`).
- `@save` a group of lines to a given filename.

Development-oriented features

- **Code execution:** `@run` executes (via `execfile`) any Python file:

```
@run [options] your_file [args to your program]
```

`@run` is my main development workhorse:

- IPython's exception tracebacks.
- Easy reloading of code (top-level modules, at least).
- **The debugger:** `@pdb`. Start `pdb` in post-mortem mode at uncaught exceptions.
 - The `pdb` interactive prompt sees the local namespace.
 - Walk up and down the stack of your dead program, print variables, call code, ...
 - This can save massive amounts of debugging time compared to other methods.
- **The profiler:**
 - `@run -p`: profile complete programs.
 - `@prun`: profile single Python expressions (like function calls).
- **Recursive reloading:** `@dreload`. It helps interactive use, but it's not perfect.

System access

Just enough functionality to allow fluid system access while using Python.

- Magics which mimic system commands (@cd, @cat, @clear, @env, @ls, @less, @mkdir, @mv, ...)
- You can define new system aliases with @alias
 - New aliases appear as new magic functions.
 - You can put your favorite aliases in your IPython configuration file.
 - Aliases can even have parameters:

```
In [4]: alias lsextr ls *.%s
```

```
In [5]: lsextr lyx
```

```
ipython.lyx  numerics.lyx
```

- Support for directory traversal (@cd, @dhist, @popd, @pushd, @dirs).
- New @bookmark system: access to often visited directories.
- @sc a=ls *.py to capture the output of 'ls *.py'.
- Lines starting with '!' go to the system shell. '!!' returns the output as a list.
- !cmd \$foo expands Python var foo for the shell (via str(foo))

PySH: IPython as a (quasi) system shell

- Long requested by users, finally here (thanks to user contributions).
- IPython is very flexible: pysh consists of 36 lines of unique (non-doc) code.
- **Just an IPython profile:** `$ipython -p pysh`. It @aliases all of `$PATH`.
- Not a real shell: **no job control!!!**
- Still pretty cool: python syntax + full shell access.
- Pipes and 'cmd &' are OK: the whole line goes to `os.system()`
- Customizable prompts (a la bash).
- New special syntax: `$` and `$$` as first char
 - `$var=cmd` - capture output of cmd into Python string
 - `$$var=cmd` - capture output of cmd into Python list (split on `'\n'`)
 - In shell command, `$var` expands Python var (as a string).

A simple PySH example

```
maqroll[ipython] |1> print "This is still %(lang)s" % {'lang':'Python!'}
This is still Python!
maqroll[ipython] |2> 99**3
      <2> 970299
maqroll[ipython] |3> _ + 55
      <3> 970354
maqroll[ipython] |4> $$pyfiles=ls *py
maqroll[ipython] |5> pyfiles
      <5> ['ipython_win_post_install2.py',
'ipython_win_post_install.py', 'pywin32_postinstall.py', 'setup.py']
maqroll[ipython] |6> for f in pyfiles:
      |.>     if len(f)>12:
      |.>         wc -l $f
      |.>
171 ipython_win_post_install2.py
130 ipython_win_post_install.py
257 pywin32_postinstall.py
```

Embedding IPython into other programs

You can call IPython as a Python shell inside your own programs.

The resulting shell opens within the surrounding local/global namespaces.

Great for:

- Debugging: print variables, execute code, plot things right at the trouble spot.
- Providing interactive abilities for your programs (very useful for data analysis).
- Studying programs you didn't write (pop a shell and start TABbing).

It's as simple as:

```
from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
... Your code here ...
ipshell() ← Opens IPython in your program at this point
... More code ...
ipshell() ← It can be called multiple times
```

An extensible framework

- Plain Python customisation is clunky: `$PYTHONSTARTUP`.
- IPython has extensive customization options in `~/.ipython/ipythonrc`
- Configuration 'profiles':

```
$ ipython -p numeric ← Load ipythonrc-numeric config
```

These configuration files can include others: a base config for most options, plus specific settings for particular uses:

```
ipythonrc ⊂ ipythonrc-math ⊂ ipythonrc-numeric  
(base config)      (calculator)          (full Numeric)
```

- Extensible input syntax. You can define filters that preprocess user input before execution (try `ipython -p tutorial`). Very useful to make tools tailored for special application domains (entry of quantities with units, DNA sequences, ...).
- Other parts are also customizable (magics, prompts, object info, ...)
- A profile can turn IPython into a special purpose tool with minimal effort.

IPython & plotting: Gnuplot & Matplotlib

Until now: Gnuplot

- A solid, stable tool for 2d plots, with good Python support.
- `IPython.GnuplotRuntime`: better syntax and options for creating plots via Gnuplot. Mainly meant for scripting.
- `IPython.GnuplotInteractive`: additional facilities for quick interactive plotting:

In [1]: `x=frange(0,2*pi,npts=500)` ← `frange` is part of IPython's numeric utils.

In [2]: `plot x,sin(x**2),'o'` ← plots $\sin(x^2)$ vs x , and $f(x) = 0$.

A new option: Matplotlib (<http://matplotlib.sourceforge.net>).

- Great feature set (see John's talk), was lacking a bit in the interactive dept.
- A new IPython option: `-pylab`. Possible thanks to John's extensive help.
- Automatic configuration of threading mode to match backend configuration (Tk, GTK, WXPython).
- Use `@run` to execute matplotlib scripts interactively (no blocking).

Status

- ✓ Fairly widely used (part of SUSE, available for OSX-Fink, Debian, Gentoo, ...)
- ✓ Stable, and pretty bug-free (detailed post-crash tracebacks mailed to me).
- ✓ Easy (albeit inelegant) to customize.
- ✓ Well documented (~70 pages manual, HTML & PDF).
- ✗ **The internals are a mess** in need of a major cleanup.
- ✗ **No unit tests**. Not a single one in ~14000 LOC.
 - Writing tests for interactive stuff is not easy (though that's not an excuse :)

Where to go from here?

Cleanup starting soon

- Finish generic threading code, useful for GTK/WX projects. **Help!!!**
- New @decorators (official as of 2 days ago): change @magic character.
- Internal refactoring, code removal. Python 2.3 (getopt, logging)
- Easy to plug into GUI shells (PyCrust, others).
- Integration with Enthought's Envisage.
 - I *really* like this idea (think IDL & matlab, but with such a better language).
- A Mathematica notebook-style environment ... Envisage? LyX? T_EXMacS?
 - This would be great, but it's a longer term idea.