IPython

An enhanced interactive Python

Fernando Pérez

`fperez@colorado.edu`

Fast Algorithms Group

Dept. of Applied Mathematics

CU Boulder

SciPy'03

Caltech, Sept. 12 2003

Outline

- Why IPython? A short bit of history
- IPython's goals
- Design ideas
- Feature overview and demo
- An extensible framework
- IPython, Numeric and plotting
- Status and future development

Note: I'll bounce between slides introducing features and interactive usage.

This will be an easy, laid back talk: interrupt me!

Why IPython? A short bit of history

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)

A 2-minute history of IPython

- I found Python in 2001 as a Perl/sh/awk/sed/C/C++/IDL/Mathematica refugee.
- David Beazley's slides: `sys.displayhook` $\rightarrow$ a Mathematica-like prompt.
- An article on O'Reilly's site $\rightarrow$ Janko Hauser's and Nathaniel Gray's work.
- Nathan's: `$PYTHONSTARTUP` enhancements.
- Janko's: a full shell built on top of the `code.InteractiveConsole` module.
- I put together my modifications and theirs, as an 'afternoon hack' (famous last words). Six weeks later (with little sleep or thesis progress), IPython 0.1 was out.
- As of this year, SciPy hosts IPython (`http://ipython.scipy.org`), it has public mailing lists, bug tracking and public CVS access. Thanks!

IPython's goals

An LGPL Python shell replacement. It tries to be:

1. A better Python shell: object introspection, system access, 'magic' command system for adding functionality when working interactively, . . .

2. An embeddable interpreter: useful for debugging and for mixing batch-processing with interactive work.

3. A flexible framework: you can be 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 $>=$ 2.1

- Developed in **Linux**, it runs under any Unix (including CygWin and Mac OS X).

- **Windows**: OK, but not perfect. I don't use Windows, and I've fallen way behind on keeping up with user efforts to contribute Windows patches. Next item on my list (for version 0.5.1)

Design ideas

A good shell is a necessity for a solid, pleasant scientific computing environment

Some key ideas underlying IPython's design:

- Fluid work: do the most with the least typing. It's an interactive shell, after all.
- Meta-control: the 'magic' functions control IPython itself while it runs.
- System-level access: scientific computing often requires data file manipulations.
- Pleasant 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`.
  - Debugger: automatic `pdb` triggering on exceptions.
- Adaptability: be easily extensible and customizable for specific problem domains.

Basic interactive features

- 'Magic' functions (prefixed with '@'): IPython control , system access, namespace information, etc. This was part of Janko's original work. User-extensible (example).
- Object introspection with '?' and '??'.
- Object introspection with @pdoc, @pdef, @pfile, @psource, @pinfo.
- 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' $\longrightarrow$ type 'mm' to execute.
  - @hist shows previous input history.
  - Ctrl-P/N: search previous/next match in history.

- Automatic indentation of typed text (off 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
-------> 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`).

System access

IPython is *NOT* trying to replace a system shell (though people *have* asked :).

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 lsext ls *.%s
In [5]: lsext lyx
ipython.lyx  numerics.lyx
```

(*Note*: the alias system is a nice example of Python's dynamism. An alias is auto-generated code, compiled and added as a method to the current IPython instance while it runs).

- Support for directory traversal (`@cd`, `@dhist`, `@popd`, `@pushd`, `@ds`).
- Lines starting with '`!`' are passed directly to the system shell.

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.

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).

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.
- Other parts are also customizable (magics, pompts, object info, . . .)

IPython, Numeric/SciPy, Gnuplot & MayaVi

A very nice environment for scientific computing.

- Chaco: I'm rooting for it, but I need to get work done in the meantime ;-)
- Gnuplot: a solid, stable tool for 2d plots. Excellent, publication-quality PostScript.
- Gnuplot.py: Python support for Gnuplot (`http://gnuplot-py.sourceforge.net`).
- MayaVi: amazing visualization tool for 2d arrays and 3d data (see Prabhu's talk).
- IPython.GnuplotRuntime: better syntax and options for creating plots via Gnuplot. Mainly meant for scripting. Most of my additions have made it back into the Gnuplot.py mainline (as of very recently, still only in CVS).
- IPython.GnuplotInteractive: additional facilities for quick interactive plotting:

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

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

Current status and future development

The user's perspective

✔ Fairly good (I think ;-) Users seem to like it, and I use it a lot myself.

✔ Stable, and fairly bug-free. If it crashes, it generates a very detailed post-mortem traceback which can be mailed to me. No known crashes in 0.5.0.

✔ Customizations are easy to do (albeit inelegant).

✔ Documentation is pretty thorough ($\sim$60 pages manual).

✘ Windows is the weak spot. Mostly my fault (I don't kept up with user-supplied patches). Though it would help a *lot* if Windows had a decent, standards compliant terminal!

The developer's perspective

- ✘ It was my first Python program ever - it shows.
- ✘ I also knew next to nothing about OO. That shows too.
- ✘ The internals are a mess in need of a major cleanup.
- ✘ No unit tests. Not a single one in ∼13000 LOC. Fun...
- ✘ I have very little time for it. I try to keep up with bug fixes, but the cleanup would be a serious undertaking.
- ✘ Since it works for me (as a user), a major rewrite will happen slooooowly.
- ✔ Volunteers are welcome.
- ✔ Eventually, the goal is to optionally integrate with PyCrust or other graphical shells (without losing command-line functionality).
- ✔ Perhaps inclusion in the stdlib? I think Python should ship with a better shell. But the current IPython code base is too messy for this.