IPython

Getting the most out of working interactively in Python

http://ipython.scipy.org

Brian E. Granger
Tech-X
bgranger@txcorp.com

Fernando Pérez
Applied Math, U. Colorado
fperez@colorado.edu

CU BOULDER

PyCon'07
Addison, Texas; Feb. 24 2007

Acknowledgements

- Nathan Gray and Janko Hauser: authors of the code IPython started from.

- IPython community: too many people to list, but a few must be mentioned

  - Ville Vainio (trunk)
  - Jörgen Stenarson (pyreadline for win32)
  - Walter Dörwald (ipipe system)
  - Benjamin Ragan-Kelley (distributed/parallel support, see next talk)

- Enthought (Eric Jones): hosting IPython, the SciPy package, ...

- $$$:

  - Tech-X Corporation (B. Granger)
  - US Department of Energy and Oak Ridge National Laboratory (F. Pérez)

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

- Fernando found Python in 2001 as a Perl/sh/awk/sed/C/C++/IDL/Mathematica refugee while a Physics graduate student.
- David Beazley's slides: `sys.displayhook` → a Mathematica-like prompt.
- An article on O'Reilly's site → 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.
- Fernando put together his modifications and theirs, as an 'afternoon hack' (famous last words). Six weeks later (with little sleep or thesis progress), IPython 0.1 was out.
- Eventually, Fernando did finish his thesis...

Where we are today

- Enthought hosts IPython (http://ipython.scipy.org), with proper SVN, mailing lists, Trac and Moin sites, etc. Thanks!

- A crude LOC count (IPython currently has zero extension code):

```
# In trunk:
find | grep 'py$' | egrep -v '/build/|\.svn|external' | xargs wc -l
28761 total
# In the saw branch (current development, see the next talk):
find | grep 'py$' | egrep -v '/build/|\.svn|external' | xargs wc -l
15150 total
```

- Packaged by all the major Linux distros and Fink. We ship Win32 installer. egg OK.

- Widely used, stable. A number of projects offer it as a shell, sometimes with extensive customizationss: SAGE, Django, TurboGears, PyRAF from the Hubble Telescope, CASA from NRAO, Ganga from CERN, ...

- Requires: Python $>=$ 2.3.

IPython's goals

In its simplest form, IPython is a BSD-licensed Python shell replacement.

In broader terms, 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 use it as the base environment for other systems with Python as the underlying language. It is very configurable in this direction.

4. A system for interactive control of distributed/parallel computing systems: next talk.

5. An interactive component we can plug into GUIs, browser-based shells, etc.

Design ideas

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

Some key ideas underlying IPython's design:

- Every keystroke counts: it's an interactive shell, after all.
- Meta-control: the 'magic' functions control IPython itself while it runs.
- System-level access: direct access to files, commands, etc.
- 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.

Interactive Demos

(which will only cover a few features, there's plenty more, see the docs!)

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 scipy` ← *Load `ipythonrc-scipy` config*

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

  `ipythonrc` ⊂ `ipythonrc-math` ⊂ `ipythonrc-scipy`
  (base config) (calculator) (full SciPy)

- 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, ...)

Random goodies we won't have time to talk about

These tools may be useful to your projects and everyday use of Python:

- ✔ `demo:` The demos presented here use the `IPython.demo` module for interactively presenting Python scripts to an audience.

- ✔ `irunner:` pexpect wrapper to run a file containing input for any interactive system with a recognizable (by a regexp) prompt.

- ✔ `pycolor:` read your source nicely highlighted at the terminal.

- ✔ `ipdoctest:` easily generate doctest files (for non-docstring uses), merge them into a standard unittest suite, use IPython syntax in them. In the `saw` development branch.

- ✔ `ultraTB/CrashHandler:` nicer tracebacks for your code, enormously detailed post-mortem reports (many bugs caused by user code fixed just with these tracebacks).

Current status and future development

The user's perspective

- ✔ Fairly good (we think ;-) Users seem to like it, and we use it a lot (we eat our own dog food).
- ✔ Trunk is stable, and fairly bug-free. Very detailed post-crash reports help a lot.
- ✔ Customizations are easy to do. Pretty much everything is customizable.
- ✔ Documentation is pretty thorough (∼90 pages manual, Wiki FAQ and Cookbook).
- ✔ We try to be responsive on the mailing lists.

The developer's perspective

✘ It was Fernando's first Python program ever - it shows.

✘ Knew next to nothing about OO. That shows too.

✘ The internals are a mess in need of a major cleanup. We're doing it...

✘ No unit tests. But with the new ipdoctest module, this is changing.

✔ Volunteers are welcome. We now have a good team, but more hands are good!

✔ The next talk will discuss some of the new developments...

Thanks!

Any Questions?

Extra Slides

Outline

- Why IPython? A short bit of history
- IPython's goals and design ideas
- Feature overview and demo
  - Workflow.
  - GUI Toolkits, plotting.
  - System access, ipipe.
  - An extensible framework.
- 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!

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 '??', wildcard search '*foo*?'
- Object introspection with %pdoc, %pdef, %pfile, %psource, %pinfo.
- TAB-completion in the local namespace and filesystem (via readline). Extensible.
- 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 (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.