IPython

New design notes

Fernando Pérez

6th May 2005

1 Introduction

This is a draft document with notes and ideas for the IPython rewrite. The section order and structure of this document roughly reflects in which order things should be done and what the dependencies are. This document is mainly a draft for developers, a pdf version is provided with the standard distribution in case regular users are interested and wish to contribute ideas.

A tentative plan for the future:

- 0.6.x series: in practice, enough people are using IPython for real work that I think it warrants a higher number. This series will continue to evolve with bugfixes and incremental improvements.
- 0.7.x series: (maybe) If resources allow, there may be a branch for 'unstable' development, where the architectural rewrite may take place.

However, I am starting to doubt it is feasible to keep two separate branches. I am leaning more towards a LyX-like approach, where the main branch slowly transforms and evolves. Having CVS support now makes this a reasonable alternative, as I don't have to make pre-releases as often. The active branch can remain the mainline of development, and users interested in the bleeding-edge stuff can always grab the CVS code.

Ideally, IPython should have a clean class setup that would allow further extensions for special-purpose systems. I view IPython as a base system that provides a great interactive environment with full access to the Python language, and which could be used in many different contexts. The basic hooks are there: the magic extension syntax and the flexible system of recursive configuration files and profiles. But with a code as messy as the current one, nobody is going to touch it.

2 Immediate TODO and bug list

Things that should be done for the current series, before starting major changes.

- Fix any bugs reported at the online bug tracker.
- **Redesign the output traps.** They cause problems when users try to execute code which relies on sys.stdout being the 'true' sys.stdout. They also prevent scripts which use raw_input() to work as command-line arguments.
  The best solution is probably to print the banner first, and then just execute all the user code straight with no output traps at all. Whatever comes out comes out. This makes the ipython code actually simpler, and eliminates the problem altogether.

These things need to be ripped out, they cause no end of problems. For example, if user code requires acces to stdin during startup, the process just hangs indefinitely. For now I've just disabled them, and I'll live with the ugly error messages.

- The prompt specials dictionary should be turned into a class which does proper namespace management, since the prompt specials need to be evaluated in a certain namespace. Currently it's just globals, which need to be managed manually by code below.

- Fix coloring of prompts: the pysh color strings don't have any effect on prompt numbers, b/c these are controlled by the global scheme. Make the prompts fully user-definable, colors and all. This is what I said to a user:
  As far as the green \#, this is a minor bug of the coloring code due to the vagaries of history. While the color strings allow you to control the coloring of most elements, there are a few which are still controlled by the old ipython internal coloring code, which only accepts a global 'color scheme' choice. So basically the input/output numbers are hardwired to the choice in the color scheme, and there are only 'Linux', 'LightBG' and 'NoColor' schemes to choose from.

- Clean up FakeModule issues. Currently, unittesting with embedded ipython breaks because a FakeModule instance overwrites __main__. Maybe ipython should revert back to using __main__ directly as the user namespace? Handling a separate namespace is proving *very* tricky in all corner cases.

- Make the output cache depth independent of the input one. This way one can have say only the last 10 results stored and still have a long input history/cache.

- Fix the fact that importing a shell for embedding screws up the command-line history. This can be done by not importing the history file when the shell is already inside ipython.

- Lay out the class structure so that embedding into a gtk/wx/qt app is trivial, much like the multithreaded gui shells now provide command-line coexistence with the gui toolkits. See `http://www.livejournal.com/users/glyf/32396.html`

- Get Holger's completer in, once he adds filename completion.

Lower priority stuff:

- Add @showopt/@setopt (decide name) for viewing/setting all options. The existing option-setting magics should become aliases for setopt calls.

- It would be nice to be able to continue with python stuff after an @ command. For instance "@run something; test_stuff()" in order to test stuff even faster. Suggestion by Kasper Souren <Kasper.Souren@ircam.fr>

- Run a 'first time wizard' which configures a few things for the user, such as color_info, editor and the like.

- Logging: @logstart and -log should start logfiles in ~.ipython, but with unique names in case of collisions. This would prevent ipython.log files all over while also allowing multiple sessions. Also the -log option should take an optional filename, instead of having a separate -logfile option.
  In general the logging system needs a serious cleanup. Many functions now in Magic should be moved to Logger, and the magic @s should be very simple wrappers to the Logger methods.

3 Lighten the code

If we decide to base future versions of IPython on Python 2.3, which has the new Optik module (called optparse), it should be possible to drop DPyGetOpt. We should also remove the need for Itpl. Another area for trimming is the Gnuplot stuff: much of that could be merged into the mainline project.

Double check whether we really need FlexCompleter. This was written as an enhanced rlcompleter, but my patches went in for python 2.2 (or 2.3, can't remember).

With these changes we could shed a fair bit of code from the main trunk.

4 Unit testing

All new code should use a testing framework. Python seems to have very good testing facilities, I just need to learn how to use them. I should also check out QMTest at `http://www.codesourcery.com/qm/qmtest`, it sounds interesting (it's Python-based too).

5 Configuration system

Move away from the current ipythonrc format to using standard python files for configuration. This will require users to be slightly more careful in their syntax, but reduces code in IPython, is more in line with Python's normal form (using the $PYTHONSTARTUP file) and allows much more flexibility. I also think it's more 'pythonic', in using a single language for everything.

Options can be set up with a function call which takes keywords and updates the options Struct.

In order to maintain the recursive inclusion system, write an 'include' function which is basically a wrapper around safe_execfile(). Also for alias definitions an alias() function will do. All functionality which we want to have at startup time for the users can be wrapped in a small module so that config files look like:

```
from IPython.Startup import *
...
set_options(automagic=1,colors='NoColor',...)
...
include('mysetup.py')
...
alias('ls ls --color -l')
...  etc.
```

Also, put **all** aliases in here, out of the core code.

The new system should allow for more seamless upgrading, so that:

- It automatically recognizes when the config files need updating and does the upgrade.
- It simply adds the new options to the user's config file without overwriting it. The current system is annoying since users need to manually re-sync their configuration after every update.
- It detects obsolete options and informs the user to remove them from his config file.

Here's a copy of Arnd Baecker suggestions on the matter:

1.) upgrade: it might be nice to have an "auto" upgrade procedure: i.e. imagine that IPython is installed system-wide and gets upgraded, how does a user know, that an upgrade of the stuff in

~/.ipython is necessary ? So maybe one has to a keep a version number in ~/.ipython and if there is a mismatch with the started ipython, then invoke the upgrade procedure.

2.) upgrade: I find that replacing the old files in ~/.ipython (after copying them to .old not optimal (for example, after every update, I have to change my color settings (and some others) in ~/.ipython/ipthonrc). So somehow keeping the old files and merging the new features would be nice. (but how to distinguish changes from version to version with changes made by the user ?) For, example, I would have to change in GnuplotMagic.py gnuplot_mouse to 1 after every upgrade ...

This is surely a minor point - also things will change during the "BIG" rewrite, but maybe this is a point to keep in mind for this ?

3.) upgrade: old, sometimes obsolete files stay in the ~/.ipython subdirectory. (hmm, maybe one could move all these into some subdirectory, but which name for that (via version-number ?) ?)

5.1 Command line options

It would be great to design the command-line processing system so that it can be dynamically modified in some easy way. This would allow systems based on IPython to include their own command-line processing to either extend or fully replace IPython's. Probably moving to the new optparse library (also known as optik) will make this a lot easier.

6 OS-dependent code

Options which are OS-dependent (such as colors and aliases) should be loaded via include files. That is, the general file will have:

```
if os.name == 'posix':
include('ipythonrc-posix.py')
elif os.name == 'nt':
include('ipythonrc-nt.py')...
```

In the `-posix`, `-nt`, etc. files we'll set all os-specific options.

7 Merging with other shell systems

This is listed before the big design issues, as it is something which should be kept in mind when that design is made.

The following shell systems are out there and I think the whole design of IPython should try to be modular enough to make it possible to integrate its features into these. In all cases IPython should exist as a stand-alone, terminal based program. But it would be great if users of these other shells (some of them which have very nice features of their own, especially the graphical ones) could keep their environment but gain IPython's features.

IDLE — This is the standard, distributed as part of Python.

pyrepl — `http://starship.python.net/crew/mwh/hacks/pyrepl.html`. This is a text (curses-based) shell-like replacement which doesn't have some of IPython's features, but has a crucially useful (and hard to implement) one: full multi-line editing. This turns the interactive interpreter into a true code testing and development environment.

PyCrust — `http://sourceforge.net/projects/pycrust`. Very nice, wxWindows based system.

PythonWin — `http://starship.python.net/crew/mhammond`. Similar to PyCrust in some respects, a very good and free Python development environment for Windows systems.

8 Class design

This is the big one. Currently classes use each other in a very messy way, poking inside one another for data and methods. ipmaker() adds tons of stuff to the main __IP instance by hand, and the mix-ins used (Logger, Magic, etc) mean the final __IP instance has a million things in it. All that needs to be cleanly broken down with well defined interfaces amongst the different classes, and probably no mix-ins.

The best approach is probably to have all the sub-systems which are currently mixins be fully independent classes which talk back only to the main instance (and **not** to each other). In the main instance there should be an object whose job is to handle communication with the sub-systems.

I should probably learn a little UML and diagram this whole thing before I start coding.

8.1 Magic

Now all methods which will become publicly available are called Magic.magic_name, the magic_ should go away. Then, Magic instead of being a mix-in should simply be an attribute of __IP:

__IP.Magic = Magic()

This will then give all the magic functions as __IP.Magic.name(), which is much cleaner. This will also force a better separation so that Magic doesn't poke inside __IP so much. In the constructor, Magic should get whatever information it needs to know about __IP (even if it means a pointer to __IP itself, but at least we'll know where it is. Right now since it's a mix-in, there's no way to know which variables belong to whom).

Build a class MagicFunction so that adding new functions is a matter of:

```
my_magic = MagicFunction(category = 'System utilities')
my_magic.__call__ = ...
```

Features:

- The class constructor should automatically register the functions and keep a table with category sections for easy sorting/viewing.
- The object interface must allow automatic building of a GUI for them. This requires registering the options the command takes, the number of arguments, etc, in a formal way. The advantage of this approach is that it allows not only to add GUIs to the magics, but also for a much more intelligent building of docstrings, and better parsing of options and arguments.

Also think through better an alias system for magics. Since the magic system is like a command shell inside ipython, the relation between these aliases and system aliases should be cleanly thought out.

8.2 Color schemes

These should be loaded from some kind of resource file so they are easier to modify by the user.

9 Hooks

IPython should have a modular system where functions can register themselves for certain tasks. Currently changing functionality requires overriding certain specific methods, there should be a clean API for this to be done.

9.1 whos hook

This was a very nice suggestion from Alexander Schmolck <a.schmolck@gmx.net>:

2. I think it would also be very helpful if there where some sort of hook for "whos" that let one customize display formaters depending on the object type.

For example I'd rather have a whos that formats an array like:

```
Variable Type Data/Length
------------------------------
a array size:  4x3 type:  'Float'
```

than

```
Variable Type Data/Length
------------------------------
a array [[ 0.  1.  2.  3<...> 8.  9.  10.  11.]]
```

10 Parallel support

For integration with graphical shells and other systems, it will be best if ipython is split into a kernel/client model, much like Mathematica works. This simultaneously opens the door for support of interactive parallel computing. Currenlty %bg provides a threads-based proof of concept, and Brian Granger's XGrid project is a much more realistic such system. The new design should integrates ideas as core elements. Some notes from Brian on this topic:

1. How should the remote python server/kernel be designed? Multithreaded? Blocking? Connected/disconnected modes? Load balancing?

2. What APi/protocol should the server/kernel expose to clients?

3. How should the client classes (which the user uses to interact with the cluster) be designed?

4. What API should the client classes expose?

5. How should the client API be wrapped in a few simple magic functions?

6. How should security be handled?

7. How to work around the issues of the GIL and threads?

I think the most important things to work out are the client API (#4) the server/kernel API/protocol (#2) and the magic function API (#5). We should let these determine the design and architecture of the components.

One other thing. What is your impression of twisted? I have been looking at it and it looks like a _very_ powerful set of tools for this type of stuff. I am wondering if it might make sense to think about using twisted for this project.

11 Manuals

The documentation should be generated from docstrings for the command line args and all the magic commands. Look into one of the simple text markup systems to see if we can get latex (for reLyXing later) out of this. Part of the build command would then be to make an update of the docs based on this, thus giving more complete manual (and guaranteed to be in sync with the code docstrings).

[PARTLY DONE] At least now all magics are auto-documented, works farily well. Limited Latex formatting yet.

11.1 Integration with pydoc-help

It should be possible to have access to the manual via the pydoc help system somehow. This might require subclassing the pydoc help, or figuring out how to add the IPython docs in the right form so that help() finds them.

Some comments from Arnd and my reply on this topic:

> ((Generally I would like to have the nice documentation > more easily accessable from within ipython ... > Many people just don't read documentation, even if it is > as good as the one of IPython))

That's an excellent point. I've added a note to this effect in new_design. Basically I'd like help() to naturally access the IPython docs. Since they are already there in html for the user, it's probably a matter of playing a bit with pydoc to tell it where to find them. It would definitely make for a much cleaner system. Right now the information on IPython is:

-ipython –help at the command line: info on command line switches

-? at the ipython prompt: overview of IPython

-magic at the ipython prompt: overview of the magic system

-external docs (html/pdf)

All that should be better integrated seamlessly in the help() system, so that you can simply say:

help ipython -> full documentation access

help magic -> magic overview

help profile -> help on current profile

help -> normal python help access.

12 Graphical object browsers

I'd like a system for graphically browsing through objects. `@obrowse` should open a widged with all the things which `@who` lists, but cliking on each object would open a dedicated object viewer (also accessible as `@oview <object>`). This object viewer could show a summary of what `<object>?` currently shows, but also colorize source code and show it via an html browser, show all attributes and methods of a given object (themselves openable in their own viewers, since in Python everything is an object), links to the parent classes, etc.

The object viewer widget should be extensible, so that one can add methods to view certain types of objects in a special way (for example, plotting Numeric arrays via grace or gnuplot). This would be very useful when using IPython as part of an interactive complex system for working with certain types of data.

I should look at what PyCrust has to offer along these lines, at least as a starting point.

13 Miscellaneous small things

- Collect whatever variables matter from the environment in some globals for __IP, so we're not testing for them constantly (like $HOME, $TERM, etc.)

14 Session restoring

I've convinced myself that session restore by log replay is too fragile and tricky to ever work reliably. Plus it can be dog slow. I'd rather have a way of saving/restoring the *current* memory state of IPython. I tried with pickle but failed (can't pickle modules). This seems the right way to do it to me, but it will have to wait until someone tells me of a robust way of dumping/reloading *all* of the user namespace in a file.

Probably the best approach will be to pickle as much as possible and record what can not be pickled for manual reload (such as modules). This is not trivial to get to work reliably, so it's best left for after the code restructuring.

The following issues exist (old notes, see above paragraph for my current take on the issue):

- magic lines aren't properly re-executed when a log file is reloaded (and some of them, like clear or run, may change the environment). So session restore isn't 100% perfect.
- auto-quote/parens lines aren't replayed either. All this could be done, but it needs some work. Basically it requires re-running the log through IPython itself, not through python.
- _p variables aren't restored with a session. Fix: same as above.

15 Tips system

It would be nice to have a tip() function which gives tips to users in some situations, but keeps track of already-given tips so they aren't given every time. This could be done by pickling a dict of given tips to IPYTHONDIR.

16 TAB completer

Some suggestions from Arnd Baecker:

a) For file related commands (ls, cat, ...) it would be nice to be able to TAB complete the files in the current directory. (once you started typing something which is uniquely a file, this leads to this effect, apart from going through the list of possible completions ...). (I know that this point is in your documentation.)

More general, this might lead to something like command specific completion ?

Here's John Hunter's suggestion:

The *right way to do it* would be to make intelligent or customizable choices about which namespace to add to the completion list depending on the string match up to the prompt, eg programmed completions. In the simplest implementation, one would only complete on files and directories if the line preceding the tab press matched 'cd ' or 'run ' (eg you don't want callable showing up in 'cd ca<TAB>')

In a more advanced scenario, you might imaging that functions supplied the TAB namespace, and the user could configure a dictionary that mapped regular expressions to namespace providing functions (with sensible defaults). Something like

```
completed = {
'^cd\s+(.*)' : complete_files_and_dirs,
'^run\s+(.*)' : complete_files_and_dirs,
'^run\s+(-.*)' : complete_run_options,
}
```

I don't know if this is feasible, but I really like programmed completions, which I use extensively in tcsh. My feeling is that something like this is eminently doable in ipython.

/JDH

For something like this to work cleanly, the magic command system needs also a clean options framework, so all valid options for a given magic can be extracted programatically.

17 Debugger

Current system uses a minimally tweaked pdb. Fine-tune it a bit, to provide at least:

- Tab-completion in each stack frame. See email to Chris Hart for details.
- Object information via ? at least. Break up magic_oinfo a bit so that pdb can call it without loading all of IPython. If possible, also have the other magics for object study: doc, source, pdef and pfile.
- Shell access via !
- Syntax highlighting in listings. Use py2html code, implement color schemes.

18 A Python-based system shell - pysh?

Note: as of IPython 0.6.1, most of this functionality has actually been implemented.

This section is meant as a working draft for discussions on the possibility of having a python-based system shell. It is the result of my own thinking about these issues as much of discussions on the ipython lists. I apologize in advance for not giving individual credit to the various contributions, but right now I don't have the time to track down each message from the archives. So please consider this as the result of a collective effort by the ipython user community.

While IPyhton is (and will remain) a python shell first, it does offer a fair amount of system access functionality:

- ! and !! for direct system access,

- magic commands which wrap various system commands,

- @sc and @sx, for shell output capture into python variables,

- @alias, for aliasing system commands.

This has prompted many users, over time, to ask for a way of extending ipython to the point where it could be used as a full-time replacement over typical user shells like bash, csh or tcsh. While my interest in ipython is such that I'll concentrate my personal efforts on other fronts (debugging, architecture, improvements for scientific use, gui access), I will be happy to do anything which could make such a development possible. It would be the responsibility of someone else to maintain the code, but I would do all necessary architectural changes to ipython for such an extension to be feasible.

I'll try to outline here what I see as the key issues which need to be taken into account. This document should be considered an evolving draft. Feel free to submit comments/improvements, even in the form of patches.

In what follows, I'll represent the hypothetical python-based shell ('pysh' for now) prompt with '>>'.

18.1 Basic design principles

I think the basic design guideline should be the following: a hypothetical python system shell should behave, as much as possible, like a normal shell that users are familiar with (bash, tcsh, etc). This means:

1. System commands can be issued directly at the prompt with no special syntax:

>> ls

>> xemacs

should just work like a user expects.

2. The facilities of the python language should always be available, like they are in ipython:

>> 3+4
7

3. It should be possible to easily capture shell output into a variable. bash and friends use backquotes, I think using a command (@sc) like ipython currently does is an acceptable compromise.

4. It should also be possible to expand python variables/commands in the middle of system commands. I thihk this will make it necessary to use $var for name expansions:

>> var='hello' # var is a Python variable
>> print var hello # This is the result of a Python print command
>> echo $var hello # This calls the echo command, expanding 'var'.

5. The above capabilities should remain possible for multi-line commands. One of the most annoying things I find about tcsh, is that I never quite remember the syntactic details of looping. I often want to do something at the shell which involves a simple loop, but I can never remember how to do it in tcsh. This often means I just write a quick throwaway python script to do it (Perl is great for this kind of quick things, but I've forgotten most its syntax as well).

It should be possible to write code like:

>> for ext in ['.jpg','.gif']:
.. ls file$ext

And have it work as 'ls file.jpg;ls file.gif'.

18.2 Smaller details

If the above are considered as valid guiding principles for how such a python system shell should behave, then some smaller considerations and comments to keep in mind are listed below.

- it's ok for shell builtins (in this case this includes the python language) to override system commands on the path. See tcsh's 'time' vs '/usr/bin/time'. This settles the 'print' issue and related.

- pysh should take

foo args

as a command if (foo args is NOT valid python) and (foo is in $PATH).

If the user types

>> ./foo args

it should be considered a system command always.

- _, __ and ___ should automatically remember the previous 3 outputs captured from stdout. In parallel, there should be _e, __e and ___e for stderr. Whether capture is done as a single string or in

list mode should be a user preference. If users have numbered prompts, ipython's full In/Out cache system should be available.

But regardless of how variables are captured, the printout should be like that of a plain shell (without quotes or braces to indicate strings/lists). The everyday 'feel' of pysh should be more that of bash/tcsh than that of ipython.

- filename completion first. Tab completion could work like in ipython, but with the order of priorities reversed: first files, then python names.

- configuration via standard python files. Instead of 'setenv' you'd simply write into the os.environ[] dictionary. This assumes that IPython itself has been fixed to be configured via normal python files, instead of the current clunky ipythonrc format.

- IPython can already configure the prompt in fairly generic ways. It should be able to generate almost any kind of prompt which bash/tcsh can (within reason).

- Keep the Magics system. They provide a lightweight syntax for configuring and modifying the state of the user's session itself. Plus, they are an extensible system so why not give the users one more tool which is fairly flexible by nature? Finally, having the @magic optional syntax allows a user to always be able to access the shell's control system, regardless of name collisions with defined variables or system commands.

But we need to move all magic functionality into a protected namespace, instead of the current messy name-mangling tricks (which don't scale well).

19 Future improvements

- When from <mod> import * is used, first check the existing namespace and at least issue a warning on screen if names are overwritten.
- Auto indent? Done, for users with readline support.

19.1 Better completion a la zsh

This was suggested by Arnd:

> > More general, this might lead to something like

> > command specific completion ?

>

> I'm not sure what you mean here.

Sorry, that was not understandable, indeed ...

I thought of something like

- cd and then use TAB to go through the list of directories

- ls and then TAB to consider all files and directories

- cat and TAB: only files (no directories ...)

For zsh things like this are established by defining in .zshrc

compctl -g '*.dvi' xdvi

compctl -g '*.dvi' dvips

compctl -g '*.tex' latex

compctl -g '*.tex' tex

...

20 Outline of steps

Here's a rough outline of the order in which to start implementing the various parts of the redesign. The first 'test of success' should be a clean pychecker run (not the mess we get right now).

- Make Logger and Magic not be mixins but attributes of the main class.
  - Magic should have a pointer back to the main instance (even if this creates a recursive structure) so it can control it with minimal message-passing machinery.
  - Logger can be a standalone object, simply with a nice, clean interface.
  - Logger currently handles part of the prompt caching, but other parts of that are in the prompts class itself. Clean up.
- Change to python-based config system.
- Move make_IPython() into the main shell class, as part of the constructor. Do this *after* the config system has been changed, debugging will be a lot easier then.
- Merge the embeddable class and the normal one into one. After all, the standard ipython script *is* a python program with IPython embedded in it. There's no need for two separate classes (*maybe* keep the old one around for the sake of backwards compatibility).

21 Ville Vainio's suggestions

Some notes sent in by Ville Vainio `<vivainio@kolumbus.fi>` on Tue, 29 Jun 2004. Keep here for reference, some of it replicates things already said above.

Current ipython seems to "special case" lots of stuff - aliases, magics etc. It would seem to yield itself to a simpler and more extensible architecture, consisting of multple dictionaries, where just the order of search is determined by profile/prefix. All the functionality would just be "pushed" to ipython core, i.e. the objects that represent the functionality are instantiated on "plugins" and they are registered with ipython core. i.e.

def magic_f(options, args): pass

m = MyMagic(magic_f) m.arghandler = stockhandlers.OptParseArgHandler m.options = ... # optparse options, for easy passing to magic_f and help display

note that arghandler takes a peek at the instance, sees options, and proceeds # accordingly. Various arg handlers can ask for arbitrary options. # some handler might optionally glob the filenames, search data folders for filenames etc.

ipythonregistry.register(category = "magic", name = "mymagic", obj = m)

I bet most of the current functionality could easily be added to such a registry by just instantiating e.g. "Magic" class and registering all the functions with some sensible default args. Supporting

legacy stuff in general would be easy - just implement new handlers (arg and otherwise) for new stuff, and have the old handlers around forever / as long as is deemed appropriate. The 'python' namespace (locals() + globals()) should be special, of course.

It should be easy to have arbitrary number of "categories" (like 'magic', 'shellcommand','projectspecific_myproject', 'projectspecific_otherproject'). It would only influence the order in which the completions are suggested, and in case of name collision which one is selected. Also, I think all completions should be shown, even the ones in "later" categories in the case of a match in an "earlier" category.

The "functionality object" might also have a callable object 'expandarg', and ipython would run it (with the arg index) when tab completion is attempted after typing the function name. It would return the possible completions for that particular command... or None to "revert to default file completions". Such functionality could be useful in making ipython an "operating console" of a sort. I'm talking about:

> > lscat reactor # list commands in category - reactor is "project specific" category

r_operate

> > r_operate <tab> start shutdown notify_meltdown evacuate

> > r_operate shutdown <tab>

1 2 5 6 # note that 3 and 4 are already shut down

> > r_operate shutdown 2

Shutting down.. ok.

> > r_operate start <tab>

2 3 4 # 2 was shut down, can be started now

> > r_operate start 2

Starting.... ok.

I'm talking about having a super-configurable man-machine language here! Like cmd.Cmd on steroids, as a free addition to ipython!