One of Python's most useful features is its interactive interpreter. It allows for 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.

IPython

A comprehensive environment for **interactive** and **exploratory** computing

Three Main Components:

- An enhanced interactive Python **shell**.
- A decoupled two-process *communication model*, which allows for multiple clients to connect to a computation kernel, most notably the web-based **notebook**.
- An *architecture* for interactive **parallel computing**.

Some of the many useful features of IPython includes:

- Command history, which can be browsed with the up and down arrows on the keyboard.
- Tab auto-completion.
- In-line editing of code.
- Object introspection, and automatic extract of documentation strings from python objects like classes and functions.
- Good interaction with operating system shell.
- Support for multiple parallel back-end processes, that can run on computing clusters or cloud services like Amazon EC2.

IPython provides a rich architecture for **interactive** computing with:

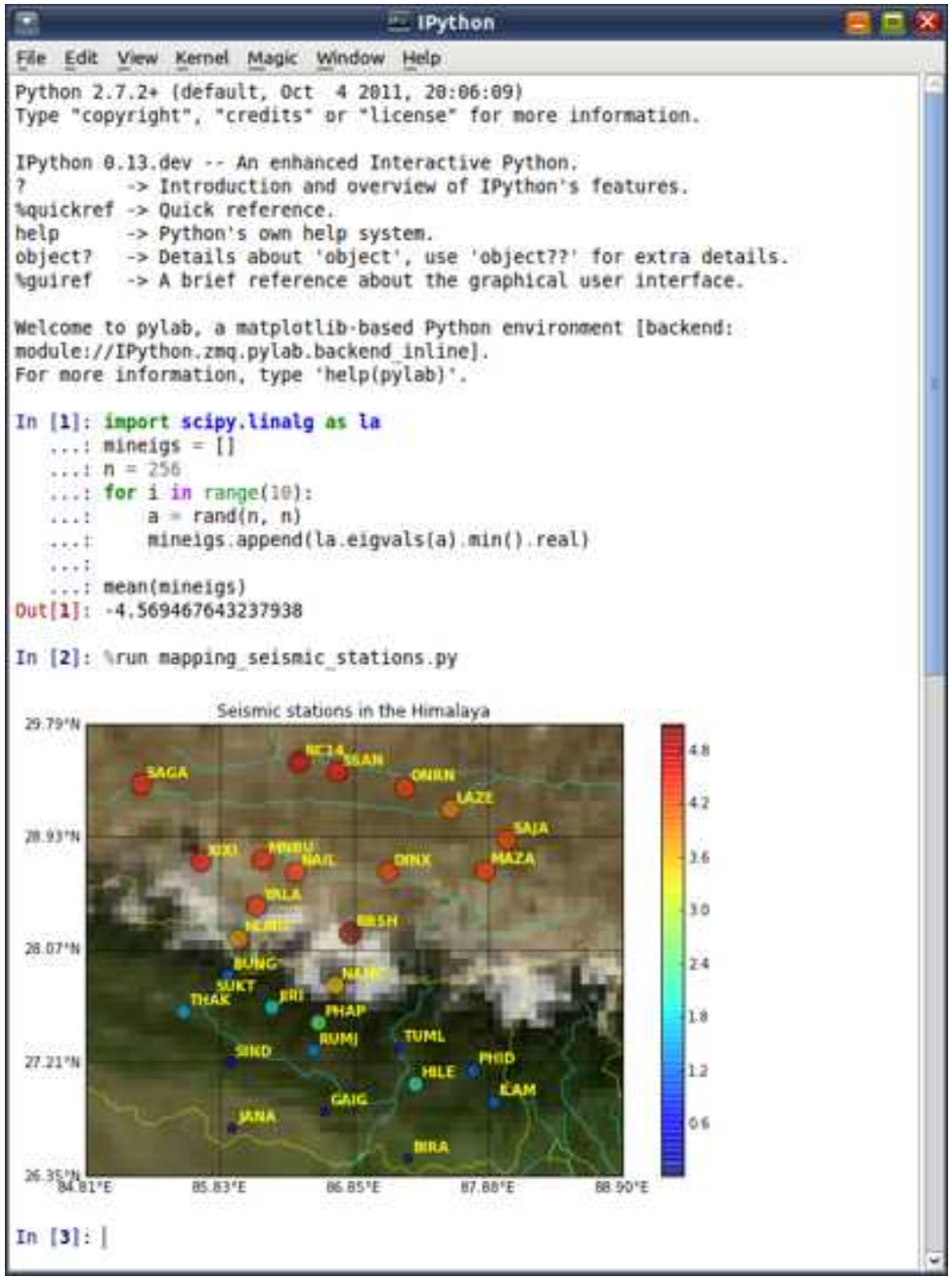
- A powerful interactive shell.
- A kernel for Jupyter.
- Easy to use, high performance tools for parallel computing.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.

IPython

IPython is an interactive shell that addresses the limitation of the standard python interpreter, and it is a work-horse for scientific use of python. It provides an interactive prompt to the python interpreter with a greatly improved user-friendliness.

It comes with a browser-based **notebook** with support for code, text, mathematical expressions, inline plots and other rich media.

You don't need to know anything beyond Python to start using IPython – just type commands as you would at the standard Python prompt. But IPython can do much more than the standard prompt...



IPython

Beyond the Terminal ...

- The REPL as a Network **Protocol**
- **Kernels**
  - Execute Code
- **Clients**
  - Read input
  - Present Output

Simple abstractions enable rich, sophisticated clients

The Four Most Helpful Commands

The four most helpful commands is shown to you in a banner, every time you start IPython:

Command	Description
`?`	Introduction and overview of IPython's features.
`%quickref`	Quick reference.
`help`	Python's own help system.
`object?`	Details about `object`, use `object??` for extra details.

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>` to view the object's attributes. Besides Python objects and keywords, tab completion also works on file and directory names.

Running and Editing

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 (unlike imported modules, which have to be specifically reloaded). IPython also includes `dreload`, a recursive reload function.

`%run` has special flags for timing the execution of your scripts (`-t`), or for running them under the control of either Python's pdb debugger (`-d`) or profiler (`-p`).

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.

Magic Functions ...

The following examples show how to call the builtin **`%timeit`** magic, both in line and cell mode:

```
In [1]: %timeit range(1000)
100000 loops, best of 3: 7.76 us per loop

In [2]: %%timeit x = range(10000)
   ...: max(x)
   ...:
1000 loops, best of 3: 223 us per loop
```

The builtin magics include:

- Functions that work with code: **`%run`**, **`%edit`**, **`%save`**, **`%macro`**, **`%recall`**, etc.
- Functions which affect the shell: **`%colors`**, **`%xmode`**, **`%autoindent`**, **`%automagic`**, etc.
- Other functions such as **`%reset`**, **`%timeit`**, **`%%writefile`**, **`%load`**, or **`%paste`**.

Exploring 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. To get specific information on an object, you can use the **magic** commands **`%pdoc`**, **`%pdef`**, **`%psource`** and **`%pfile`**.

Magic Functions

IPython has a set of predefined **magic functions** that you can call with a command line style syntax. There are two kinds of magics, line-oriented and cell-oriented.

- Line magics are prefixed with the **`%`** character and work much like OS command-line calls: they get as an argument the rest of the line, where arguments are passed without parentheses or quotes.
- Cell magics are prefixed with a double **`%%`**, and they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument.

System Shell Commands

To run any command at the system shell, simply prefix it with `!`. You can capture the output into a Python list. To pass the values of Python variables or expressions to system commands, prefix them with `$`.

System Aliases

It's 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 (`%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.

Magic Functions ...

You can always call them using the `%` prefix, and if you're calling a line magic on a line by itself, you can omit even that: `run thescript.py`. You can toggle this behavior by running the `%automagic` magic.

Cell magics must always have the `%%` prefix.

A more detailed explanation of the magic system can be obtained by calling `%magic`, and for more details on any magic function, call `%somemagic?` to read its docstring. To see all the available magic functions, call `%lsmagic`.

System Shell Commands ...

```
!ping www.bbc.co.uk

files = !ls                    # capture

!grep -rF $pattern ipython/*  # passing vars
```

History ...

Input and output history are kept in variables called In and Out, keyed by the prompt numbers. The last three objects in output history are also kept in variables named _, __ and ___.

You can use the **`%history`** magic function to examine past input and output. Input history from previous sessions is saved in a database, and IPython can be configured to save output history.

Several other magic functions can use your input history, including **`%edit`**, **`%rerun`**, **`%recall`**, **`%macro`**, **`%save`** and **`%pastebin`**. You can use a standard format to refer to lines:

```
%pastebin 3 18-20 ~1/1-5
```

This will take line 3 and lines 18 to 20 from the current session, and lines 1-5 from the previous session.

History

IPython stores both the commands you enter, and the results it produces. You can easily go through previous commands with the up- and down-arrow keys, or access your history in more sophisticated ways.

Debugging

After an exception occurs, you can call **`%debug`** to jump into the Python debugger (pdb) and examine the problem. Alternatively, if you call **`%pdb`**, IPython will automatically start the debugger on any uncaught exception. 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. This can be an efficient way to develop and debug code, in many cases eliminating the need for print statements or external debugging tools.

You can also step through a program from the beginning by calling **`%run -d theprogram.py`**.