IPython - Interactive Python Shell: An introduction

Ferry Boender

Revision History

Revision 1.0 October 29, 2007 Revised by: FB

1. Introduction

Python has an interactive shell, which you can start by simply starting python:

```
[todsah@jib]~$ python
Python 2.4.4 (#2, Apr  5 2007, 20:11:18)
[GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello')
hello
```

This is a nice and very powerful way of using Python, but it's a bit limited. So you might want to check out IPython (http://ipython.scipy.org/). IPython is also an interactive Python shell, but with lot's of stuff added, such as tab completion, colors, dynamic object introspection, sessions, command history, etc.

To start it, simply run the IPython command:

```
[todsah@jib]~$ ipython
Python 2.4.4 (#2, Apr  5 2007, 20:11:18)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.0 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

You'll be dropped at a prompt (`In [1]:`) where you can enter python commands, just like in the normal interactive python interpreter. Let's walk through a couple of IPython's best features:

2. Tab completion

IPython has smart tab completion for lots of things such as modules, classes, methods and directory/files. For instance, we can define a class:

```
In [7]: class Foo:
   ...:     """ Example class """
   ...:     def bar(self, name):
   ...:         """ Example method """
   ...:         print("Hello %s!" % (name))
   ...:
```

Then, when we want to create an instance of that class, we can use tab completion to find the possible completion names:

```
In [8]: f = F[TAB]
False               FloatingPointError
Foo                 FutureWarning
In [8]: f = Foo()
```

Tab completion will also help you with methods in classes:

```
In [11]: Foo.[TAB]
Foo.__doc__     Foo.__module__  Foo.bar
```

Another example of tab completion is when you need to specify a filename:

```
In [9]: pw = file('/etc/pa[TAB]
/etc/pam.conf      /etc/pam.d         /etc/pango
/etc/paper.config  /etc/papersize     /etc/passwd
/etc/passwd-

In [9]: pw = file('/etc/pas[TAB]
/etc/passwd   /etc/passwd-

In [9]: pw = file('/etc/passwd')
```

3. Documentation

We can also use IPython features to retrieve information on all kinds of stuff by appending a question mark behind it:

```
In [10]: Foo?
Type:           classobj
String Form:    __main__.Foo
Namespace:      Interactive
```

```
File:           /var/lib/python-support/python2.4/IPython/FakeModule.py
Docstring:
    Example class

In [15]: file?
Type:           type
Base Class:
String Form:
Namespace:      Python builtin
Docstring:
    file(name[, mode[, buffering]]) -> file object

    Open a file.  The mode can be 'r', 'w' or 'a' for reading (default),
    writing or appending.  The file will be created if it doesn't exist
    when opened for writing or appending; it will be truncated when
    opened for writing.  Add a 'b' to the mode for binary files.
    Add a '+' to the mode to allow simultaneous reading and writing.
    If the buffering argument is given, 0 means unbuffered, 1 means line
    buffered, and larger numbers specify the buffer size.
    Add a 'U' to mode to open the file for input with universal newline
    support.  Any line ending in the input file will be seen as a 'n'
    in Python.  Also, a file so opened gains the attribute 'newlines';
    the value for this attribute is one of None (no newline read yet),
    'r', 'n', 'rn' or a tuple containing all the newline types seen.

    'U' cannot be combined with 'w' or '+' mode.

    Note:  open() is an alias for file().
```

The above works on almost anything from modules to methods. Documentation in another form can also be retrieved using the help function (also available in the default python interpreter):

```
In [11]: help Foo
Help on class Foo in module __main__:

class Foo
 |  Example class
 |
 |  Methods defined here:
 |
 |  bar(self, name)
 |      Example method
 |-> help(Foo)
```

Another little something that could come in handy: Searching the namespaces for wildcards:

```
In [55]: ?*oo
Foo

In [56]: ?*ile
compile
execfile
```

```
file
```

4. Editing in an editor

IPython provides a couple of interesting 'magic\' functions. These do anything from acting as an actual system shell (ls, cd, etc) as well as providing access to some settings and other features. One such feature is the ability to call an external editor to quickly edit multi-line Python code:

```
In [19]: %edit
IPython will make a temporary file named: /tmp/ipython_edit_vpMbMW.py

[EDITOR IS LAUCHED]
Editing... done. Executing edited code...
Out[19]: 'class Foo:\n\tdef Bar(self):\n\t\treturn("hello, world!")\n\n'
```

It shows the code we typed in the editor as a single line of output and runs it. class Foo is now defined. We can revisit the code we typed in the editor by using the output number given: Out[19]:

```
In [20]: %edit _19
IPython will make a temporary file named: /tmp/ipython_edit_r7EqK7.py
Editing... done. Executing edited code...
Out[20]: 'class Foo:\n\tdef Bar(self):\n\t\treturn("goodbye world!")\n\n'
```

By the way, `_19` is nothing more than a special variable that contains the output of command 19 in the interactive shell:

```
In [21]: print _19
class Foo:
        def Bar(self):
                return("hello, world!")
```

`%edit _19` simply says we want to edit the contents of that variable. That %means we can also do things such as:

```
In [22]: s = "This is some string\nWhat should we do with it?"

In [23]: edit s
IPython will make a temporary file named: /tmp/ipython_edit_VelPpw.py
Editing... done. Executing edited code...
```

```
   File "/tmp/ipython_edit_VelPpw.py", line 1
     This is some string
                       ^
SyntaxError: invalid syntax

WARNING: Failure executing file: </tmp/ipython_edit_VelPpw.py>
```

```
Out[23]: 'This is some string\nWhat should we do with it?\nEdit it!\n'
```

This will give us a better view of the contents of the variable 's\', which can be useful for stuff retrieved from a file, for instance. IPython won't reassign the edited text to the variable, but this can be easily accomplished by assigning the output of `Out[23]` to the variable `s`:

```
In [24]: s = _23

In [25]: s
Out[25]: 'This is some string\nWhat should we do with it?\nEdit it!\n'
```

Another special variable is '_\' which is the last output received. This can be used to consecutively edit a piece of code until it's free of syntax errors:

```
In [42]: %edit
IPython will make a temporary file named: /tmp/ipython_edit_nWOuiK.py
Editing... done. Executing edited code...

   File "/tmp/ipython_edit_nWOuiK.py", line 2
     de bar(self):
        ^
SyntaxError: invalid syntax

WARNING: Failure executing file: </tmp/ipython_edit_nWOuiK.py>
Out[42]: 'class Foo:\n\tde bar(self):\n\t\tprint "Hello, world"\n\n'

In [43]: %edit _
IPython will make a temporary file named: /tmp/ipython_edit_U-ylh8.py
Editing... done. Executing edited code...
Out[43]: 'class Foo:\n\tdef bar(self):\n\t\tprint "Hello, world"\n\n'
```

5. Debugging

IPython also has a built-in debugger that can be used to inspect tracebacks in a post-mortem. Let's take the following code:

```
def f1(param1):
        f2(param1)
def f2(param2):
        if type(param2) != type(int):
                raise ValueError("expected an integer", 1)
```

When we %edit this, and execute it, and then call `f1('foo\')` it will generate an exception:

```
In [53]: f1('foo')
```

```
exceptions.ValueError                                Traceback (most recent call last)

/home/todsah/

/tmp/ipython_edit_qrYEq0.py in f1(param1)
...-> 2         f2(param1)
      3 def f2(param2):
      4         if type(param2) != type(int):
      5                 raise ValueError("expected an integer", 1)
      6

/tmp/ipython_edit_qrYEq0.py in f2(param2)
      2         f2(param1)
      3 def f2(param2):
      4         if type(param2) != type(int):
...-> 5                 raise ValueError("expected an integer", 1)
      6

ValueError: ('expected an integer', 1)
```

Now, we can activate the post-mortem debugger and inspect various things:

```
In [54]: %debug
> /tmp/ipython_edit_qrYEq0.py(5)f2()
      4         if type(param2) != type(int):
...-> 5                 raise ValueError("expected an integer", 1)
      6

ipdb>
```

Here, we've landed at the debugger prompt. The help command can help us:

```
ipdb> help

Documented commands (type help ):
========================================
EOF    break  condition  disable  help    list  pdoc   r       u        where
a      bt     cont       down     ignore  n     pinfo  return  unalias
alias  c      continue   enable   j       next  pp     s       up
args   cl     d          exit     jump    p     q      step    w
b      clear  debug      h        l       pdef  quit   tbreak  whatis

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

People familiar with other debuggers such as GDB will notice familiar stuff. I won't go into the details, but consider this little session as an example of what you can with the debugger:

```
ipdb> p param2
'foo'
ipdb> p type(param2)
<type 'str'>
ipdb> up
> /tmp/ipython_edit_qrYEq0.py(2)f1()
      1 def f1(param1):
----> 2         f2(param1)
      3 def f2(param2):
ipdb> p param1, type(param1)
('foo', <type 'str'>)
```

This can be used in combination with the %run command to execute a python file:

```
In [67]: %run raise.py
```

```
---------------------------------------------------------------------------
exceptions.Exception                                 Traceback (most recent call last)

/home/todsah/raise.py
      1 #!/usr/bin/python
      2
----> 3 raise Exception("Some error!", 1)
      4
      5

Exception: ('Some error!', 1)
WARNING: Failure executing file:
```

(ignore that last line. It did in fact succeed at executing the file. It's just that an error occurred while executing it).

6. Other stuff

The reload() method reloads a module. This can come in handy when you're writing a module and keep changing it and you want those changes to be reloaded by IPython:

```
In [75]: import sys

In [76]: reload(sys)
```

7. Conclusion

IPython is a very versatile tool for interactive python programming. It allows you to quickly inspect modules, classes, methods, documentation and a lot more. Not sure if that list comprehension is going to give you exactly what you want? Fire up IPython and just try it out! Whenever I'm writing Python code, I'll always have an IPython shell open for experimenting with code or for finding information. Try it, and you'll find that IPython is an indispensable tool.

8. Document license

Author:

```
Ferry Boender
<ferry (DOT) boender (AT) electricmonk (DOT) nl>
```