PCS2 交互环境之 iPython

概述

iPython 是一个 Python 的交互式 Shell，比默认的 Python Shell 好用得多，功能也更强大。她支持语法高亮、自动完成、代码调试、对象自省，支持 Bash Shell 命令，内置了许多很有用的功能和函式等，非常容易使用。

应用

目前，最新稳定的 iPython 版本是 0.8.4 版，支持多种操作系统，如 GNU/Linux、Unix、Mac OS X、Windows 等，这里详细介绍在 GNU/Linux 和 Windows 下的安装过程。其他系统下的安装过程可参见 *http://ipython.scipy.org/moin/Download*（精巧地址：*http://bit.ly/1sRzxv*）上的具体说明。

Windows 下的 iPython 安装

在 Windows 下安装 iPython 可分为以下几步：

1. 下载 ipython-0.8.4.win32-setup.exe 和 pyreadline-1.5-win32-setup.exe。

    （1）下载 ipython-0.8.4.win32：*http://ipython.scipy.org/dist/ipython-0.8.4.win32-setup.exe*
    精巧地址：*http://bit.ly/YqbkJ*

    （2）下载 pyreadline-1.5-win32：*http://ipython.scipy.org/dist/pyreadline-1.5-win32-setup.exe*
    精巧地址：*http://bit.ly/2JFKDM*

2. 双击运行安装程序，只需经过 4 步即可完成安装。

    第一步，显示一些提示信息如图 PCS 2-1，直接点击“下一步”。

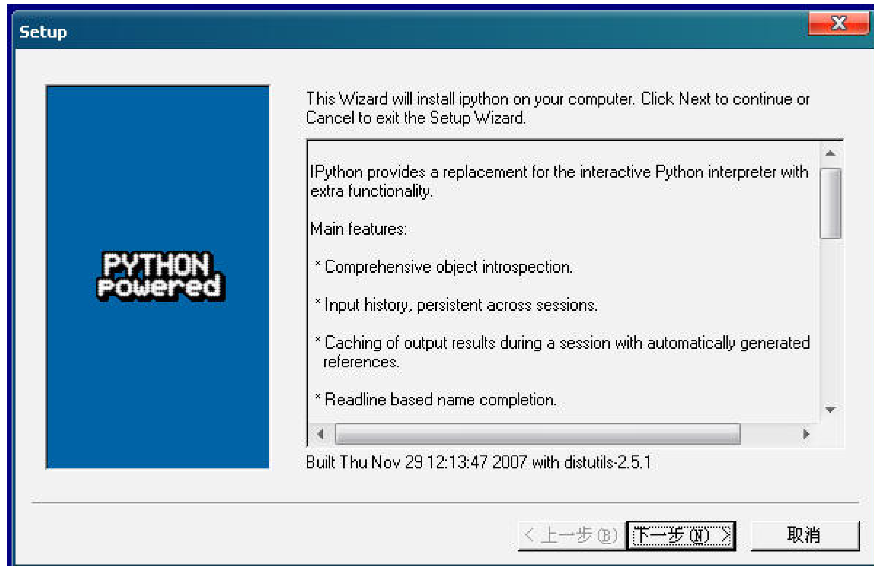

图 PCS 2-1

第二步，输入 Python 安装目录及 iPython 安装位置，因为之前把 Python 默认安装在“C:\Python25\”了，所以这里两个都为默认设置（如图 PCS 2-2 所示），直接点击“下一步”。若之前 Python 不是在“C:\Python25\”，则需要改变为 Python 所在的目录。

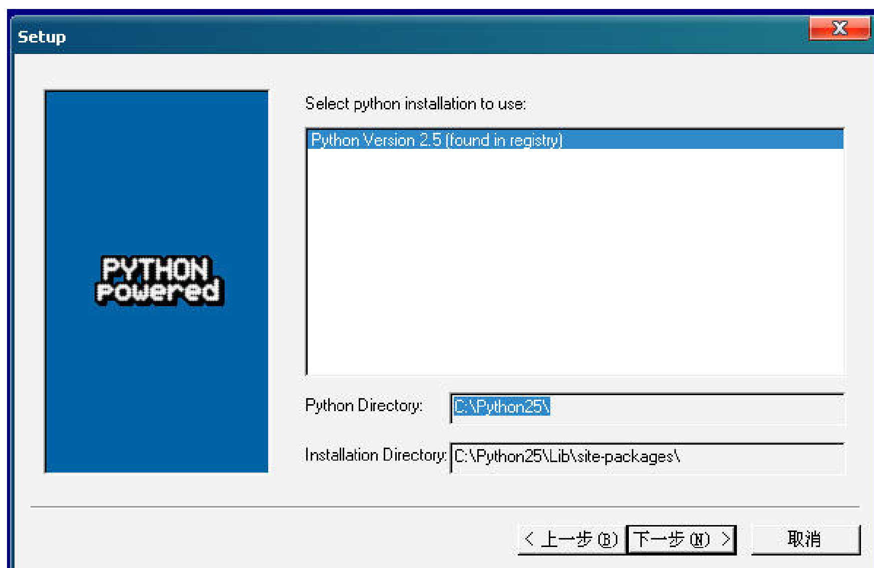

图 PCS 2-2

第三步，出现进度条，如图 PCS 2-3 所示。须等待一会儿，完毕后点击“下一步”。

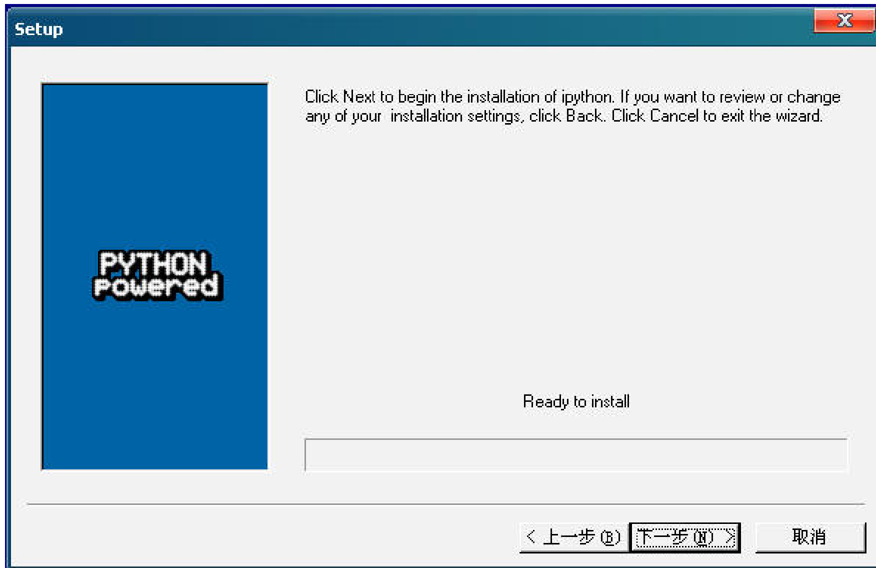

图 PCS 2-3

第四步，显示完成（如图 PCS 2-4 所示），成功安装 iPython。

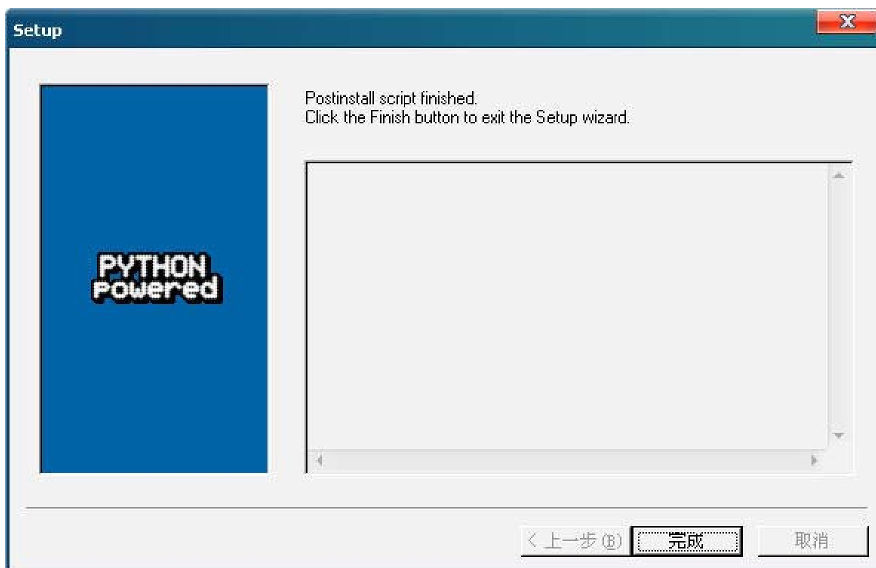

图 PCS 2-4

3. 安装 pyreadline。直接双击安装，如图 PCS 2-5 所示：

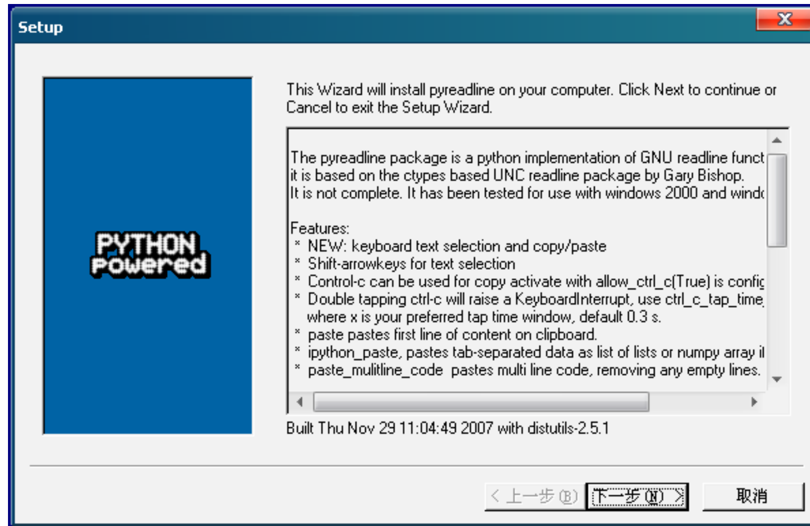

图 PCS 2-5

接下去，只须点击“下一步”直至安装完毕即可。

4. 设置环境变量 Path，在它里面加上 Python 安装目录下面的 scripts 目录，如图 PCS 2-6 所示。

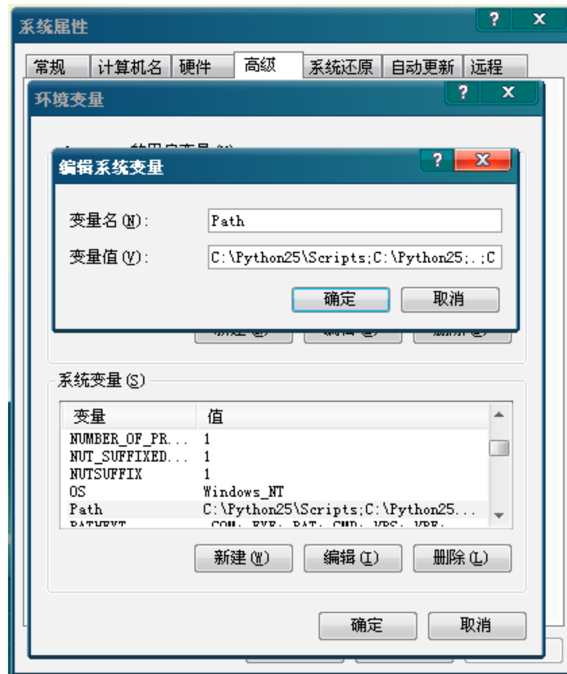

图 PCS 2-6

5. 安装完成后，进入 Windows 命令行，输入 ipython，即可以进入 iPython 交互环境如图 PCS 2-7 所示。

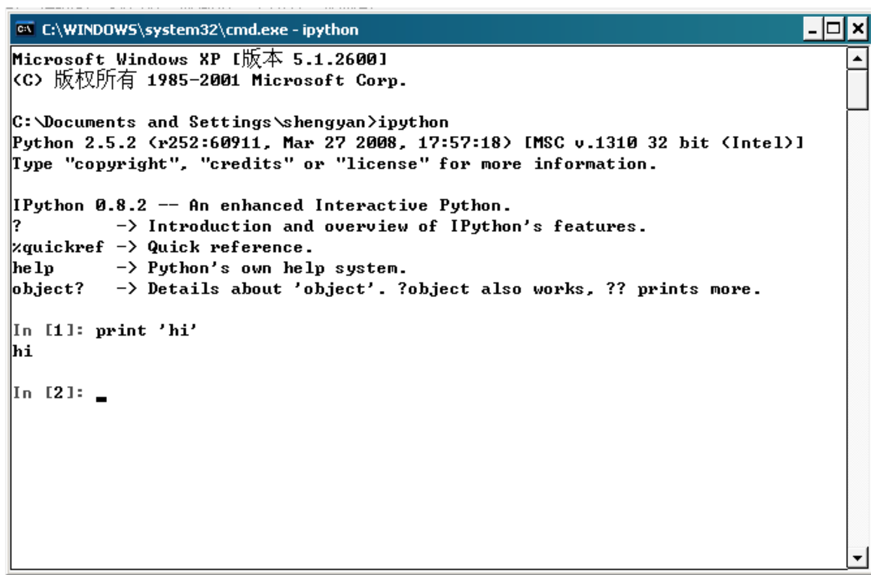

图 PCS 2-7

GNU/Linux 下的 iPython 安装

1. 下载 ipython-0.8.4.tar.gz。

   （1）下载 ipython-0.8.4.tar.gz 包：*http://ipython.scipy.org/dist/ipython-0.8.4.tar.gz*

   精巧地址：*http://bit.ly/1VaGJw*

2. 进入终端，输入如下几条命令，分别是解压、进入目录、编译、安装，一切 ok 的话就顺利安装了 iPython。

```
~$ tar xvzf ipython-0.8.4.tar.gz
~$ cd ipython-0.8.4
~/ipython-0.8.4$ python setup.py build  #这步可以省略
~/ipython-0.8.4$ sudo python setup.py install
```

3. 在 Ubuntu 下，输入命令 sudo apt-get install ipython 即可完成安装，非常方便。

使用 iPython

在顺利安装好 iPython 之后，就可以进入该交互环境使用了。下面介绍在 GNU/Linux 下 iPython 的使用方式，Windows 下也是很类似的。下面的内容很多参考自：

使用 iPython 增强交互式体验：*http://forum.ubuntu.org.cn/viewtopic.php?p=447255&sid=*

5ba2eaa5af49eaca994976f9c285b819

精巧地址：*http://bit.ly/1v24Sl*

其英文原文“Enhanced Interactive Python with iPython”：*http://www.onlamp.com/pub/a/python/2005/01/27/ipython.html*

精巧地址：*http://bit.ly/1h699v*

非常不错的资料！

进入终端，输入 ipython，即可进入如图 PCS 2-8 所示的 iPython 交互环境。

```
shengyan@LIZZIE:~/openbookprojectnew$ ipython
Python 2.5.2 (r252:60911, May  7 2008, 15:19:09)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: print 'hi'
hi

In [2]: 
```

图 PCS 2-8

注意：若是第一次运行 ipython 时会自动生成配置目录 $HOME/.ipython，它里面包含的一些配置文件适用于不同的环境：ipythonrc、ipythonrc-math、ipythonrc-numeric、ipythonrc-physics、ipythonrc-pysh、ipythonrc-scipy 及 ipythonrc-tutorial。先可以不用管它，直接使用默认就好了。

在交互环境和在 Python 默认交互环境中一样，编写代码进行调试、测试等。但比默认 Python 环境好的几点如下所示。

1. Magic。iPython 有一些“magic”关键字：

```
%Exit, %Pprint, %Quit, %alias, %autocall, %autoindent, %automagic,
%bookmark, %cd, %color_info, %colors, %config, %dhist, %dirs, %ed,
%edit, %env, %hist, %logoff, %logon, %logstart, %logstate, %lsmagic,
%macro, %magic, %p, %page, %pdb, %pdef, %pdoc, %pfile, %pinfo, %popd,
%profile, %prun, %psource, %pushd, %pwd, %r, %rehash, %rehashx, %reset,
%run, %runlog, %save, %sc, %sx, %system_verbose, %unalias, %who,
%who_ls, %whos, %xmode
```

iPython 会检查传给它的命令是否包含 magic 关键字。如果命令是一个 magic 关键字，iPython 就自己来处理。如果不是 magic 关键字，就交给 Python 去处理。如果

automagic 打开（默认），不需要在 magic 关键字前加%符号。相反，如果 automagic 是关闭的，则%是必须的。在命令提示符下输入命令 magic 就会显示所有 magic 关键字列表，以及它们简短的用法说明。良好的文档对于一个软件的任何一部分来说都是重要的，从在线 iPython 用户手册到内嵌文档（%magic），iPython 当然不会在这方面有所缺失。下面介绍些常用的 magic 函式，如：

%bg function

把 function 放到后台执行，例如：%bg myfunc(x, y, z=1)，之后可以用 jobs 将其结果取回，myvar = jobs.result(5)或 myvar = jobs[5].result。另外，jobs.status()可以查看现有任务的状态。

%ed 或 %edit

编辑一个文件并执行，如果只编辑不执行，用 ed -x filename 即可。

%env

显示环境变量

%hist 或 %history

显示历史记录

%macro name n1-n2 n3-n4 ... n5 .. n6 ...

创建一个名称为 name 的宏，执行 name 就是执行 n1-n2 n3-n4 ... n5 .. n6 ... 这些代码。

%pwd

显示当前目录

%pycat filename

用语法高亮显示一个 Python 文件（不用加.py 后缀名）

%save filename n1-n2 n3-n4 ... n5 .. n6 ...

将执行过多代码保存为文件

%time statement

计算一段代码的执行时间

%timeit statement

自动选择重复和循环次数计算一段代码的执行时间，太方便了。

2. iPython 中用！表示执行 shell 命令，用$将 Python 的变量转化成 Shell 变量。通过这两个符号，就可以做到和 Shell 命令之间的交互，可以非常方便地做许多复杂的工作。比如可以很方便地创建一组目录：

```
for i in range(10):
    s = "dir%s" % i
    !mkdir $s
```

不过写法上还是有一些限制，$ 后面只能跟变量名，不能直接写复杂表达式，$"dir%s"%i 就是错误的写法了，所以要先完全产生 Python 的变量以后再用。例如：

```
for i in !ls:
    print i
```

这样的写法也是错的，可以这样：

```
a = !ls
for i in a:
    print i
```

还有一点需要说明，就是执行普通的 Shell 命令中如果有 $ 的话需要用两个 $。比如原来的 echo $PATH 现在得写成!echo $$PATH。

3. Tab 自动补全。iPython 一个非常强大的功能是 tab 自动补全。标准 Python 交互式解释器也可以 tab 自动补全：

```
~$ python
Python 2.5.1 (r251:54863, Mar  7 2008, 04:10:12)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rlcompleter, readline
>>> readline.parse_and_bind('tab: complete')
>>> h
hasattr  hash    help    hex
```

标准 Python 交互式解释器和 iPython 都支持“普通”自动补全和菜单补全。

使用自动补全，要先输入一个匹配模型，然后按 Tab 键。如果是“普通”自动补全模式（默认），按 Tab 键后会：

- 匹配模型按最大匹配展开；
- 列出所有匹配的结果。

例如：

```
In [1]: import os

In [2]: os.po
os.popen os.popen2 os.popen3 os.popen4

In [2]: os.popen
```

输入 os.po 然后按 Tab 键，os.po 被展开成 os.popen（就像在 In [2]:提示符显示的那样），并显示 os 所有以 po 开头的模块、类和函式，它们是 popen、popen2、popen3 和 popen4。

而菜单补全稍有不同。关闭默认 Tab 补全，使用菜单补全，需修改配置文件 $HOME/.ipython/ipythonrc。

注释掉：

```
readline_parse_and_bind tab: complete
```

取消注释：

```
readline_parse_and_bind tab: menu-complete
```

不同于“普通”自动补全的显示，当前命令所有匹配列表的菜单补全会随着每按一次 Tab 键而循环显示匹配列表中的项目。例如：

```
In [1]: import os

In [2]: os.po
```

结果是：

```
In [3]: os.popen
```

接下来每次按 Tab 键就会循环显示匹配列表中的其他项目：popen2、popen3、popen4，最后回到 po。菜单补全模式下查看所有匹配列表的快捷键是 Ctrl+L。

4. 自省。Python 有几个内置的函式用于自省。iPython 不仅可以调用所有标准 Python 函式，对于那些 Python shell 内置函式同样适用。典型的使用标准 Python shell 进行自省是使用内置的 dir()函式：

```
>>> import SimpleXMLRPCServer
>>> dir(SimpleXMLRPCServer.SimpleXMLRPCServer)
['__doc__', '__init__', '__module__', '_dispatch',
'_marshaled_dispatch', 'address_family', 'allow_reuse_address',
'close_request', 'fileno', 'finish_request', 'get_request',
'handle_error', 'handle_request', 'process_request',
'register_function', 'register_instance',
'register_introspection_functions', 'register_multicall_functions',
'request_queue_size', 'serve_forever', 'server_activate', 'server_bind',
'server_close', 'socket_type', 'system_listMethods',
'system_methodHelp', 'system_methodSignature', 'system_multicall',
'verify_request']
```

因为 dir()是一个内置函式，在 iPython 中也能很好地使用它们。但是 iPython 的操作符?和??功能还要强大：

```
In [3]: import SimpleXMLRPCServer

In [4]: ? SimpleXMLRPCServer
Base Class:      <type 'module'>
String Form: <module 'SimpleXMLRPCServer' from
'/usr/lib/python2.5/SimpleXMLRPCServer.pyc'>
Namespace:       Interactive
File:           /usr/lib/python2.5/SimpleXMLRPCServer.py
Docstring:
```

```
    Simple XML-RPC Server.

    This module can be used to create simple XML-RPC servers
    by creating a server and either installing functions, a
    class instance, or by extending the SimpleXMLRPCServer
    class.

    It can also be used to handle XML-RPC requests in a CGI
    environment using CGIXMLRPCRequestHandler.

    A list of possible usage patterns follows:

    1. Install functions:

    server = SimpleXMLRPCServer(("localhost", 8000))
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
:
```

?操作符会截断长的字符串。相反，??不会截断长字符串，如果有源代码的话还会以语法高亮形式显示它们。

5. 历史。当在 iPython shell 下交互地输入了大量命令、语句等，就像这样：

```
In [1]: a = 1

In [2]: b = 2

In [3]: c = 3

In [4]: d = {}

In [5]: e = []

In [6]: for i in range(20):
   ...:     e.append(i)
   ...:     d[i] = b
   ...:
```

可以输入命令“hist”快速查看那些已输入的历史记录：

```
In [7]: hist
1: a = 1
2: b =2
3: c = 3
```

```
4: d = {}
5: e = []
6:
for i in range(20):
    e.append(i)
    d[i] = b

7: _ip.magic("hist ")
```

要去掉历史记录中的序号（这里是 1 至 7），可使用命令"hist -n"：

```
In [8]: hist -n
a = 1
b =2
c = 3
d = {}
e = []
for i in range(20):
    e.append(i)
    d[i] = b

_ip.magic("hist ")
_ip.magic("hist -n")
```

这样就可方便地将代码复制到一个文本编辑器中。要在历史记录中搜索，可以先输入一个匹配模型，然后按 Ctrl+P 键。找到一个匹配后，继续按 Ctrl+P 键就会向后搜索再上一个匹配，按 Ctrl+N 键则是向前搜索最近的匹配。

6. 编辑。如果想在 Python 提示符下试验一个想法，经常要通过编辑器修改源代码（甚至是反复修改）。在 iPython 下输入 edit 就会根据环境变量$ EDITOR 调用相应的编辑器。如果$EDITOR 为空，则会调用 vi（Unix）或记事本（Windows）。要回到 iPython 提示符，直接退出编辑器即可。如果是保存并退出编辑器，输入编辑器的代码会在当前名字空间下被自动执行。如果不想这样，可使用 edit+X。如果要再次编辑上次最后编辑的代码，使用 edit+P。在上一个特性里，提到使用 hist−n 可以很容易地将代码拷贝到编辑器。一个更简单的方法是 edit 加 Python 列表的切片（slice）语法。假定 hist 输出如下：

```
In [29]: hist
1 : a = 1
2 : b = 2
3 : c = 3
4 : d = {}
5 : e = []
```

```
6 :
for i in range(20):
e.append(i)
d[i] = b

7 : %hist
```

现在要将第 4、5、6 句代码导出到编辑器，只要输入：

```
edit 4:7
```

7. Debugger 接口。iPython 的另一特性是它与 Python debugger 的接口。在 iPython Shell 下输入 magic 关键字 pdb 就会在产生一个异常时开关自动 debugging 功能。在 pdb 自动呼叫启用的情况下，当 Python 遇到一个未处理的异常时 Python debugger 就会自动启动。debugger 中的当前行就是异常发生的那一行。iPython 的作者说有时候当他需要在某行代码处 debug 时，他会在开始 debug 的地方放一个表达式 1/0。启用 pdb，在 iPython 中运行代码。当解释器处理到 1/0 那一行时，就会产生一个 ZeroDivisionError 异常，然后它就从指定的代码处被带到一个 debugging session 中了。
8. 运行。有时候在一个交互式 Shell 中，如果可以运行某个源文件中的内容将会很有用。运行 magic 关键字 run 带一个源文件名就可以在 iPython 解释器中运行一个文件了（例如 run <源文件> <运行源文件所需参数>）。参数主要有以下这些：
   - -n 阻止运行源文件代码时{{{__name__}}}变量被设为"{{{__main__}}}"。这会防止 if {{{__name__}}} == "{{{__main__}}}":块中的代码被执行。
   - -i 源文件在当前 iPython 的名字空间下运行而不是在一个新的名字空间中。如果你需要源代码可以使用在交互式 session 中定义的变量，它会很有用。
   - -p 使用 Python 的 profiler 模块运行并分析源代码。使用该选项的代码不会运行在当前名字空间。
9. 宏。宏允许用户为一段代码定义一个名字，这样可在以后使用这个名字来运行这段代码。就像在 magic 关键字 edit 中提到的，列表切片法也适用于宏定义。假设有一个历史记录如下：

```
In [3]: hist
1: l = []
2:
for i in l:
print i
```

可以这样来定义一个宏：

```
In [4]: macro print_l 2
```

```
Macro `print_l` created. To execute, type its name (without quotes).
Macro contents:
for i in l:
print i
```

运行宏：

```
In [5]: print_l
------> print_l()
```

在这里，列表 l 是空的，所以没有东西被输出。但这其实是一个很强大的功能，赋予列表 l 某些实际值，再次运行宏就会看到不同的结果：

```
In [7]: l = range(5)

In [8]: print_l
------> print_l()
0
1
2
3
4
```

当运行一个宏时就好像你重新输入了一遍包含在宏 print_l 中的代码。它还可以使用新定义的变量 l。由于 Python 语法中没有宏结构（也许永远也不会有），在一个交互式 shell 中它更显得是一个有用的特性。

10. 环境（Profiles）。就像早先提到的那样，iPython 安装了多个配置文件用于不同的环境。配置文件的命名规则是 ipythonrc-。要使用特定的配置启动 iPython，需要这样：

```
ipython -p
```

一个创建自己环境的方法是在$HOME/.ipython 目录下创建一个 iPython 配置文件，名字就叫做 ipythonrc-，这里是你想要的环境名字。如果同时进行好几个项目，而这些项目又用到互不相同的特殊的库，这时候每个项目都有自己的环境就很有用了。也可以为每个项目建立一个配置文件，然后在每个配置文件中 import 该项目中经常用到的模块。

11. 使用操作系统的 Shell。使用默认的 iPython 配置文件，有几个 Unix Shell 命令（当然，是在 Unix 系统上），cd、pwd 和 ls 都能像在 bash 下一样工作。运行其他的 shell 命令需要在命令前加!或!!。使用 magic 关键字%sc 和%sx 可以捕捉 shell 命令的输出。pysh 环境可以被用来替换掉 shell。使用-p pysh 参数启动的 iPython，可以接受并执行用户$PATH 中的所有命令，同时还可以使用所有的 Python 模块、Python 关键字和内置函式。例如，想要创建 500 个目录，命名规则是从 d_0_d 到 d_499_d，可以使用

-p pysh 启动 iPython，然后就像这样：

```
[~/ttt]|1> for i in range(500):
                  |.>    mkdir d_${i}_d
                  |.>
```

这就会创建 500 个目录：

```
[~/ttt]|2> ls -d d* | wc -l
500
```

注意这里混合了 Python 的 range 函式和 Unix 的 mkdir 命令。虽然 ipython -p pysh 提供了一个强大的 shell 替代品，但它缺少正确的 job 控制。在运行某个很耗时的任务时按下 Ctrl+Z 键将会停止 iPython session 而不是那个子进程。

最后，退出 iPython。可输入 Ctrl+D 键（会要求你确认），也可以输入 Exit 或 Quit（注意大小写）退出而无须确认。

小结

经过本文对 iPython 的特性及其基本使用方法的介绍，已经充分感受到 iPython 的强大功能了吧！那么，就把它作为一个有利的工具帮助我们开发吧！对于进一步的配置和更多的用途，有兴趣的读者可以在以下网络上发掘更丰富的资料。

- iPython 网站：*http://ipython.scipy.org*
- iPython 英文文档：*http://ipython.scipy.org/moin/Documentation*

  精巧地址：*http://bit.ly/GdPZU*
- iPython 中一些 magic 函式：*http://guyingbo.javaeye.com/blog/111142*

  精巧地址：*http://bit.ly/3GVxE8*