IPython Tip Sheet

To make better use of ipython, do the following one-time operations:

1. Edit your ~/.bashrc file, and add the following lines in some appropriate place:

   ```
   export LESS = "- R"
   export EDITOR=emacs
   ```

   This will tell the pager program less to interpret "raw" control sequences appropriately, and to use emacs as your default editor in certain situations. IPython uses raw control sequences to make colored text in its displays.
2. Start up a new terminal window to get the updated environment variables, or, alternatively, type `. ~/.bashrc` to update the current shell.
3. Edit your ~/ipython/ipythonrc file, and search for the set of commands describing xmode. Comment out (using #) the lines for "xmode Plain" and "xmode Context", and uncomment the line for "xmode Verbose", to get more useful information about python errors.

IPython "magic" commands are conventionally prefaced by `%`, but if the flag `%automagic` is set to on, then one can call magic commands without the preceding `%`. (`%automagic` appears to be on by default.)

The following ipython commands may be of use to you.

- ~/ipython/.ipythonrc: Any further tweaks to your configuration that you want to make permanent for all ipython sessions should involve editing the appropriate information in your ipythonrc file.

- `%run`: You've already been using this, but it's included here for completeness. `%run module` or `%run module.py` will execute the python code in the file `module.py`, and bring everything in that module's namespace into the current interactive namespace, which is different than if you had typed `import module`. NOTE: If both `module` and `module.py` exist, `%run module` will execute run the former (which may not be a python source file) rather than the latter.

- TAB-completion: At any point while typing an ipython input line, you can hit the TAB key and ipython will expand out all possible completions based on what you've already typed. This is useful both to save you from having to type long function names (i.e., type out a few of the first characters and then hit TAB), or to allow you to see what methods or attributes are attached to a particular object. For example, if (in the SmallWorldNetworks module), you type:

  ```
  g = MakeSmallWorldNetwork(100, 4, 0.1)
  g.TAB   # i.e., hit the TAB key after typing "g."
  ```

  then you will be presented with the possible completions of the command `g.` :

  ```
  g.AddEdge      g.GetNeighbors   g.HasNode      g.__doc__      g.__module__
  g.AddNode      g.GetNodes       g.__class__    g.__init__     g.neighbor_dict
  ```

- `?` and `??`: Typing `?` after a name will give you information about the object attached to that name, e.g., if I type `g.AddEdge?`, where `g` is the graph I constructed above, then I see:

  ```
  Type:           instancemethod
  Base Class:     <type 'instancemethod'>
  String Form:    <bound method UndirectedGraph.AddEdge of <Networks.UndirectedGraph instance at 0x369ea30>>
  Namespace:      Interactive
  File:           /Users/myers/teaching/ComputationalMethods/ComputerExercises/PythonSoftware/Networks.py
  Definition:     g.AddEdge(self, node1, node2)
  Docstring:
      Add node1 and node2 to network first
      Adds new edge
      (appends node2 to neighbor_dict[node1] and vice-versa, since it's
      an undirected graph)
      Do so only if old edge does not already exist
      (node2 not in neighbor_dict[node1])
  ```

  Typing two question marks lists the not only the summary information produced above, but also shows the source code used to define the object of interest.

- `%who` and `%whos`: These magic functions list objects, functions, etc. that have been added in the current namespace, as well as modules that have been imported. `%who` simply lists names of such objects, while `%whos` additionally lists type and data information (you might want to make your terminal window wide enough to capture all the output, since it doesn't do a great job with formatting the text), e.g.:

  ```
  Variable                       Type                 Data/Info
  ------------------------------------------------------------------
  AddRandomEdges                 function             <function AddRandomEdges at 0x34a8230>
  FindAverageAveragePathLength   function             <function FindAverageAver<...>ePathLength at 0x34a8130>
  FindAverageClusteringCoefficient function            <function FindAverageClus<...>Coefficient at 0x34a8030>
  GetClustering_vs_p             function             <function GetClustering_vs_p at 0x34a8570>
  GetPathLength_vs_p             function             <function GetPathLength_vs_p at 0x34a80f0>
  MakePathLengthHistograms       function             <function MakePathLengthHistograms at 0x34a8170>
  MakeRingGraph                  function             <function MakeRingGraph at 0x34a8270>
  MakeSmallWorldNetwork          function             <function MakeSmallWorldNetwork at 0x34a81f0>
  MultiPlot                      module               <module 'MultiPlot' from <...>nSoftware/MultiPlot.pyc'>
  NetGraphics                    module               <module 'NetGraphics' fro<...>oftware/NetGraphics.pyc'>
  Networks                       module               <module 'Networks' from 'Networks.pyc'>
  ```

```
Percolation                     module                  <module 'Percolation' from 'Percolation.py'>
PlotClustering_vs_p             function                <function PlotClustering_vs_p at 0x34a8730>
PlotPathLength_vs_p             function                <function PlotPathLength_vs_p at 0x34a80b0>
PlotScaledPathLength_vs_pZL     function                <function PlotScaledPathL<...>ngth_vs_pZL at 0x34a8070>
PlotWattsStrogatzFig2           function                <function PlotWattsStrogatzFig2 at 0x34a8770>
SmallWorldBetweenness           function                <function SmallWorldBetweenness at 0x34a87f0>
SmallWorldSimple                function                <function SmallWorldSimple at 0x34a81b0>
TestBetweennessSimple           function                <function TestBetweennessSimple at 0x34a87b0>
g                               Networks.UndirectedGraph <Networks.UndirectedGraph instance at 0x369ea30>
numpy                           module                  <module 'numpy' from '/sw<...>ages/numpy/__init__.pyc'>
os                              module                  <module 'os' from '/sw/lib/python2.5/os.pyc'>
pylab                           module                  <module 'pylab' from '/sw<...>site-packages/pylab.pyc'>
random                          module                  <module 'random' from '/s<...>ib/python2.5/random.pyc'>
```

Output from `%whos` can be restricted to objects of a specified type; e.g., typing `%whos function` will print out only those objects in the namespace that are functions.

- `%hist`: This presents a list of the last several input command lines. (It presents ipython magic commands not as you typed them but as they were processed through the _ip.magic system; `%hist -r` will show the command lines exactly as you typed them.) Previous input commands are stored in the list `In`; e.g., `In[37]` will show the string associated with input line 37, and `exec In[37]` will actually (re)execute that line. Similarly, the output of each of the previous commands is stored in the `Out` variable, indexed by the line number. Finally, while typing input to a command line, hit <CONTROL>-P and ipython will present you with all previous command lines that began with the text you have typed (more efficient than paging through *all* previous commands with the UP arrow); repeatedly typing <CONTROL>-P cycles through this list.

- `%macro`: Assign a name to a set of input commands, so that they can be executed all together using the assigned name, e.g., if I start a session with the following 5 lines:

  ```
  %run SmallWorldNetworks
  g = MakeSmallWorldNetwork(100, 4, 0.1)
  NetGraphics.DisplayCircleGraph(g)
  distances = Networks.FindPathLengthsFromNode(g, 0)
  print distances
  ```

  then I can make a macro called `runswn` (short for RunSmallWorldNetworks) that executes everything from lines 1 through 5, inclusive:

  ```
  %macro runswn 1-5
  ```

  It is obviously of use to run `%hist` to find which specific line numbers need to be included in the macro. Line numbers need not be contiguous, e.g., `%macro runswn 1-5 7 12-15` will assemble a macro from the specified disjoint set of command lines.

- `%edit`: This will open an editor (whatever the shell variable EDITOR is set to, see above, or vi/vim if no variable is set) containing the specified material, based on what arguments are provided, and will execute that code once the editor is exited. Therefore this is better for making small changes and testing things out, rather than keeping a large source file open in an editor as we usually do. Here are some examples of its use:

  ```
  %edit SmallWorldNetworks.py    # opens the file SmallWorldNetworks.py
  %edit MakeSmallWorldNetwork    # opens the source file containing the function MakeSmallWorldNetwork
  %edit runswn                   # opens the runswn macro defined above in a temporary file
  %edit 1-5 7 12-15              # opens a temporary file containing the input lines 1-5, 7, and 12-15
  ```

- `%lsmagic`: This lists all ipython magic commands. Have a peek and see if there are other functions that may be of use, such as:
  - `%store` (stores variables, functions, etc. that you've defined in your .ipython/ipythonrc file for use in future sessions)
  - `%pdb` (configures ipython to automatically open the python debugger pdb when an error occurs)
  - `%time` and `%timeit` (timing functions to see how long expressions take to execute)
  - `%logstart`, `%logon`, `%logoff`, and `%logstate` (to log ipython input and/or output to files)
  - `%cd`, `%pushd`, `%popd`, and `%bookmark` (to change directories, manipulate directory stacks, and create directory "bookmarks")

- See also the IPython "Quick Tips" at the IPython web site at http://ipython.scipy.org/doc/manual/node4.html .

An aside on debugging and profiling

The python `pdb` module implements a debugger, typically used by calling `pdb.run` on a quoted python expression, e.g.:

```
import pdb
pdb.run('Networks.FindPathLengthsFromNode(g, 0)')
```

This puts you into the debugger, from which you can do the usual sort of things (list source code, set breakpoints, step one line at a time, continue until a breakpoint or exception is reached, etc.). Type `?` at the pdb prompt for a list of available commands. As noted above, setting `%pdb on` within ipython will make it such that the pdb debugger will automatically be started at the point of an exception, once it is encountered.

Similarly, the python `profile` module is useful for identifying how much time is spent in various functions. The syntax is similar to that for debugging:

```
import profile
profile.run('Networks.FindPathLengthsFromNode(g, 0)')
```