CHARMING THE SNAKE

PYTHON FOR SYSTEM ADMINS

TONY WILLIAMS

SYSTEM ENGINEER

WHAT WE WILL TALK ABOUT

- A short introduction to Python
- IPython
  - IPython as shell
  - IPython for coding
- System Administration
  - More Python
  - Talking to JSS

PYTHON AS A CALCULATOR

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>>
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>>
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

STRINGS

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result.

\ can be used to escape quotes:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
>>> print _
'"Isn't," she said.'
```

TRIPLE QUOTES

String literals can span multiple lines and include white space. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a at the end of the line. The following example:

```
print ("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

produces the following output (note that the initial newline is not included):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

CONCATENATION

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>>
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>>
>>> 'Py' 'thon'
'Python'
```

This only works with two literals though, not with variables or expressions:

```
>>>
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use +:

```
>>>
>>> prefix + 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>>
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

BOOLEANS

MOVING ON TO IPYTHON

STARTING IPYTHON

command	description
ipython	run IPython
ipython qtconsole	runs ipython in QT window
ipython notebook	runs the IPython notebook server
ipython --help	IPython man page
ipython --help-all	IPython man page with *all* command line options.

The four most helpful commands, as well as their brief description, are 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.

LISTS

- Surrounded by ""
- Ordered
- Indexed at 0
- Can contain any type or mixed types

```
lst = ["apple", "banana", "orange", 22, "pear"]
lst[0]
lst[2]
lst[3] + 7
```

SLICING LISTS

```
lst[1:3]
lst[:2]
lst[3:]
lst[:-2]
lst[-3:]
```

DICTIONARIES

- A key and a value (think of a plist)
- Surrounded by "{}"
- Unordered
- keys
  - Case sensitive
  - Unique
  - New value overwrites

```
dict = {'x': 12, 'y': 22, 'z': 2}
dict
dict['x']
```

FLOW CONTROL

DECISIONS, DECISIONS

```
num = 3
if num > 2:
    print "Big"
else:
    print "Small"
```

LOOPS

```
tm = 1
while tm >= 10:
    print tm
    tm = tm + 1
```

```
for x in range(1,11):
    print x
```

```
collection = ['hey', 5, 'd']
for x in collection:
    print x
```

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print x
```

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.

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.

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.

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.

HISTORY

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

Input and output history are kept in variables called In and Out, keyed by the prompt numbers, e.g. In[4]. 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.

EXPLORE THE MAGIC FUNCTIONS

%BOOKMARK

`bookmark` is a directory bookmarking system.

command	description
bookmark name	set bookmark to current dir
bookmark name dir	set bookmark to dir
bookmark -l	list all bookmarks
bookmark -d name	remove bookmark
bookmark -r	remove all bookmarks

Then `cd -b <name>` or just `cd <name>` if there is no directory called AND there is such a bookmark defined. (The latter is why I usually use two or three letter bookmark names.)

%CD

The `cd` magic is necessary (and nicely enhanced) as the system `cd` won't work. It keeps a history of the directories visited.

command	description
cd	changes to ~
cd 'dir'	changes to directory 'dir'
cd -	changes to previous directory
cd -<n>	changes to directory <n> in directory history
cd --foo	changes to directory that matches 'foo' in history
cd -b <name>	jump to bookmark <name>

`%dhist` prints the directory history and `%dhist <n>` prints the last <n> entries.

THE DIRECTORY STACK

As well as a history of directories IPython also has a directory stack.

command	description
dirs	list directory stack
pushd	push the current directory onto the stack
pushd <dir>	push the current directory and cd to <dir>
popd	pop the top directory off the stack and cd to it

%EDIT

`edit` opens things in the editor you have defined in your $EDITOR environment variable. ###

PROMPT CUSTOMIZATION

Here are some prompt configurations you can try out interactively by using the %config magic:

```
%config PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}\
[{color.LightCyan}\Y1{color.LightBlue}]{color.Green}|\#> '
%config PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
%config PromptManager.out_template = r'<\#> '
```

You can change the prompt configuration to your liking permanently by editing ipython_config.py:

```
c.PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}\
[{color.LightCyan}\Y1{color.LightBlue}]{color.Green}|\#> '
c.PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
c.PromptManager.out_template = r'<\#> '
```

Read more about the configuration system for details on how to find ipython_config.py.

STRING LISTS

String lists (IPython.utils.text.SList) are a handy way to process output from system commands. They are produced by `var = !cmd` syntax.

First, we acquire the output of `ls -l`:

```
[Q:doc/examples]|2> lines = !ls -l
 ==
['total 23',
 '-rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py',
 '-rw-rw-rw- 1 ville None 1927 Sep 30  2006 example-embed-short.py',
 '-rwxrwxrwx 1 ville None 4606 Sep  1 17:15 example-embed.py',
 '-rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py',
 '-rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py',
 '-rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py',
 '-rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc']
```

Now, let's take a look at the contents of 'lines' (the first number is the list element number):

```
[Q:doc/examples]|3> lines
                <3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py
2: -rw-rw-rw- 1 ville None 1927 Sep 30  2006 example-embed-short.py
3: -rwxrwxrwx 1 ville None 4606 Sep  1 17:15 example-embed.py
4: -rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py
5: -rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py
6: -rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py
7: -rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc
```

Now, let's filter out the 'embed' lines:

```
[Q:doc/examples]|4> l2 = lines.grep('embed',prune=1)
[Q:doc/examples]|5> l2
               <5> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30  2006 example-demo.py
2: -rwxrwxrwx 1 ville None 1017 Sep 30  2006 example-gnuplot.py
3: -rwxrwxrwx 1 ville None  339 Jun 11 18:01 extension.py
4: -rwxrwxrwx 1 ville None  113 Dec 20  2006 seteditor.py
5: -rwxrwxrwx 1 ville None  245 Dec 12  2006 seteditor.pyc
```

Now, we want strings having just file names and permissions:

```
[Q:doc/examples]|6> l2.fields(8,0)
                <6> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total
1: example-demo.py -rw-rw-rw-
2: example-gnuplot.py -rwxrwxrwx
3: extension.py -rwxrwxrwx
4: seteditor.py -rwxrwxrwx
5: seteditor.pyc -rwxrwxrwx
```

Note how the line with ‘total’ does not raise IndexError.

If you want to split these (yielding lists), call fields() without arguments:

```
[Q:doc/examples]|7> _.fields()
                <7>
[['total'],
 ['example-demo.py', '-rw-rw-rw-'],
 ['example-gnuplot.py', '-rwxrwxrwx'],
 ['extension.py', '-rwxrwxrwx'],
 ['seteditor.py', '-rwxrwxrwx'],
 ['seteditor.pyc', '-rwxrwxrwx']]
```

If you want to pass these separated with spaces to a command (typical for lists of files), use the .s property:

```
[Q:doc/examples]|13> files = l2.fields(8).s
[Q:doc/examples]|14> files
               <14> 'example-demo.py example-gnuplot.py extension.py seteditor.py seteditor.pyc
[Q:doc/examples]|15> ls $files
example-demo.py  example-gnuplot.py  extension.py  seteditor.py  seteditor.pyc
```

SLists are inherited from normal python lists, so every list method is available:

```
[Q:doc/examples]|21> lines.append('hey')
```

MORE WITH LISTS

Let's start with a list of files.

```
cd /Applications/Utilities
utils = !ls
```

We now have an Slist. Have a look at it.

```
utils.p
utils.n
utils.s
```

What's the problem with `utils.s`?

FIXING THE SPACES

We're going to use `map`

```
def quote(str):
    return '"' + str + '"'

new = map(quote, utils)
them = ' '.join(new)
them
```

LIST COMPREHENSION INSTEAD

List comprehensions are a neat trick

```
u = ['"' + i +'"' for i in utils]
u
```

TALKING TO JSS

First we get our JSS object

```
import jss
jss_prefs = jss.JSSPrefs()
j = jss.JSS(jss_prefs)
```

Get the computer list

```
j.Computer()
```

Notice that python-jss pretty prints the list. Note also that it doesn't retrieve all the information, just the name and id.

Put the result in a variable and format it yourself.

```
computers = j.Computer()
for i in computers:
    print "id:"+str(i.id)+" name:"+i.name
```

Now get the record of one computer. This will get **all** the record.

```
example = j.Computer(193)
example
```

We can view that with `less` using the `page` magic.

```
page example
```

Some information is easily retrieved

```
example.serial_number
example.mac_addresses
```

Other information requires some XML work. XML is a tree of nodes and we have to work with those nodes.

Let's get a list of installed applications. `findall()` will return a list of nodes that match our search string. `find()` returns a single child node that matches.

```
x = example.findall('.//application')
for i in x:
    nm = i.find('name')
    ver = i.find('version')
    path = i.find('path')
    print nm.text, ver.text, path.text
```

Rather than print it let's gather the info.

```
o = []
for i in x:
    nm = i.find('name')
    ver = i.find('version')
    path = i.find('path')
    o.append(' '.join([nm.text, ver.text, path.text]))
o
```

FURTHER EXAMPLES

```
model = comp.findall('.//hardware/model_identifier')
model[0].text
os = comp.findall('.//os_version')
os[0].text
```

Lets get *all* the computer records. (*This might take a while.*)

```
all_computers = j.Computer().retrieve_all()
```

Now iterate over them

```
for computer in all_computers:
    model = computer.findtext('model')
    os = computer.findtext('os_version')
```

LET'S EXPLORE