Charming The Snake
Python For System Admins

Tony Williams
Systems 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.'
```

Unicode

Python 2.x supports Unicode but you have to tell it a string *is* Unicode

```
ustring = u'1024 \u00D7 768'
print ustring
```

Booleans

`True` is `1` or `1.0` and everything else is `False`.

operator	function
==	equals
!=	not equals
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Moving On To IPython

Starting IPython

command	description
ipython	run IPython
jupyter qtconsole	runs ipython in QT window
jupyter notebook	runs the IPython notebook server
ipython --help	IPython man page

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 = 4
if num > 3:
    print "Biggest"
elif num > 3:
    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
```

Functions

- Use `def` keyword
- May use `return` keyword

```
def hello_world():
    print("Hello World!")

hello_world()
```

```
def times_two(x):
    return x * 2

times_two(21)
```

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

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 and place them on the clipboard.

%edit

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

```
%edit times_two
```

More About Lists

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`:

```
In [4]: lines = !ls -l
```

Now, let's take a look at the contents of 'lines':

```
In [5]: lines
Out[5]:
['total 22888',
 '-rw-r--r--@  1 tonyw  staff  10129637 26 May 19:45 Input-Font.zip',
 '-rw-r--r--@  1 tonyw  staff      1901 29 May 09:04 README.md',
 '-rw-r--r--@  1 tonyw  staff     53078 27 Apr 10:17 XW16-Empty.jpg',
 '-rw-r--r--@  1 tonyw  staff    332729 27 Apr 09:01 XW16-Empty.pxm',
 '-rw-r--r--   1 tonyw  staff         1 29 May 09:01 empty_1.txt',
 '-rw-r--r--   1 tonyw  staff         1 29 May 09:01 empty_2.txt',
 '-rw-r--r--   1 tonyw  staff         1 29 May 09:01 empty_3.txt',
 '-rw-r--r--@  1 tonyw  staff     38783 26 May 19:29 index.html',
 '-rwxr-xr-x@  1 tonyw  staff       369 26 May 19:36 pandoc-print.sh',
 '-rwxr-xr-x@  1 tonyw  staff       302 26 May 19:37 pandoc.sh',
 '-rw-r--r--   1 tonyw  staff     38516 26 May 20:19 print.html',
 'drwxr-xr-x@ 16 tonyw  staff       544 26 May 11:31 reveal.js',
 '-rw-r--r--@  1 tonyw  staff     19288 26 May 19:56 slides.md',
 '-rw-r--r--@  1 tonyw  staff    588576 26 May 20:20 slides.pdf',
 '-rw-r--r--@  1 tonyw  staff     95532 26 May 20:23 slides_notes.pdf',
 '-rw-r--r--@  1 tonyw  staff    379863 26 May 19:45 solarized.zip']
```

SLists have special properties

```
lines.p
lines.n
lines.s
```

SLists also inherit all the properties of strings. Let's have a look at some.

Let’s filter out the ‘slides’ lines:

```
In [6]: lines.grep('slides',prune=1)
Out[6]:
['total 22888',
 '-rw-r--r--@  1 tonyw  staff  10129637 26 May 19:45 Input-Font.zip',
 '-rw-r--r--@  1 tonyw  staff      1901 29 May 09:04 README.md',
 '-rw-r--r--@  1 tonyw  staff     38783 26 May 19:29 index.html',
 '-rwxr-xr-x@  1 tonyw  staff       369 26 May 19:36 pandoc-print.sh',
 '-rwxr-xr-x@  1 tonyw  staff       302 26 May 19:37 pandoc.sh',
 '-rw-r--r--   1 tonyw  staff     38516 26 May 20:19 print.html',
 'drwxr-xr-x@ 16 tonyw  staff       544 26 May 11:31 reveal.js',
 '-rw-r--r--@  1 tonyw  staff    379863 26 May 19:45 solarized.zip']
```

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

```
In [8]: Out[6].fields(8,0)
Out[8]:
['total',
 'Input-Font.zip -rw-r--r--@',
 'README.md -rw-r--r--@',
 'index.html -rw-r--r--@',
 'pandoc-print.sh -rwxr-xr-x@',
 'pandoc.sh -rwxr-xr-x@',
 'print.html -rw-r--r--',
 'reveal.js drwxr-xr-x@',
 'solarized.zip -rw-r--r--@']
```

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

```
In [9]: _.fields()
Out[9]:
[['total'],
 ['Input-Font.zip', '-rw-r--r--@'],
 ['README.md', '-rw-r--r--@'],
 ['index.html', '-rw-r--r--@'],
 ['pandoc-print.sh', '-rwxr-xr-x@'],
 ['pandoc.sh', '-rwxr-xr-x@'],
 ['print.html', '-rw-r--r--'],
 ['reveal.js', 'drwxr-xr-x@'],
 ['solarized.zip', '-rw-r--r--@']]
```

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

```
In [10]: Out[6].fields(8).s
Out[10]: 'Input-Font.zip README.md index.html pandoc-print.sh pandoc.sh print.html
reveal.js slides.md slides.pdf slides_notes.pdf solarized.zip'
```

More with Slists

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

The basic syntax is [expression `for` item `in` list `if` conditional]

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

Standard Libraries

- re – regular expressions
- datetime, calendar & time – time & date functionality
- random – for non-secure randomness
- sys – system specific functionality, specifically access to command line args, python path & exit

File APIs

These parts of the standard library allow you to interact with files & the file system.

- os – provides basics like open, mkdir, stat, rmdir, remove and walk
- os.path – everything needed for path manipulation, including join, dirname, basename & exists
- tempfile – create temporary files & directories
- glob – unix style pattern matching (i.e. *.gif)
- shutil – high level file ops like copy, move, copytree and rmtree (you can lose metadata)

Running Commands

Subprocess module

Call external shell commands

	Returns
subprocess.call	Return code
subprocess.check_call	Return code or exception
subprocess.check_output	Output string or exception

```
import subprocess
cmd_str = '/usr/bin/dsmemberutil checkmembership \
-U tonyw -G admin'
cmd = cmd_str.split()

retcode = subprocess.call(cmd)
checkcode = subprocess.check_call(cmd)
output = subprocess.check_output(cmd)
```

Working With Files

Joining Paths

```
import os
silverlight_plugin_path = os.path.join("/", \
    "Library", \
    "Internet Plug-Ins", \
    "Silverlight.plugin")
print(silverlight_plugin_path)
/Library/Internet Plug-Ins/Silverlight.plugin
```

Manipulating Paths

```
os.path.basename(silverlight_plugin_path)
'Silverlight.plugin'
os.path.dirname(silverlight_plugin_path)
'/Library/Internet Plug-Ins'
os.path.splitext("com.apple.Safari.plist")
('com.apple.Safari', '.plist')
```

Tests on Files

```
os.path.exists(silverlight_plugin_path)
True
os.path.isdir(silverlight_plugin_path)
True
os.path.islink("/etc")
True
```

glob

```
import glob
osx_install = glob.glob("/Applications/" \
    "Install*OS X*.app")
print(osx_install)
['/Applications/Install Mac OS X Lion.app',
 '/Applications/Install OS X Mountain
Lion.app']
```

PyObjC

```
from AppKit import NSScreen

width = NSScreen.mainScreen().frame().size.width
height = NSScreen.mainScreen().frame().size.height

print height, width
900.0 1440.0
```

Read a preference

```
from CoreFoundation import CFPreferencesCopyValue, \
  kCFPreferencesCurrentUser, kCFPreferencesAnyHost

v = CFPreferencesCopyValue('NSNavLastRootDirectory', 'com.apple.Finder', \
  kCFPreferencesCurrentUser, kCFPreferencesAnyHost)
print(v)
```

Write a preference value

```
CFPreferencesSetValue('FXConnectToLastURL', 'smb://example.com', 'com.apple.Finder', \
  kCFPreferencesCurrentUser, kCFPreferencesAnyHost)

# Required to force preferences to sync to .plist on disk
CFPreferencesSynchronize('com.apple.Finder', kCFPreferencesCurrentUser,
  kCFPreferencesAnyHost)
```

Talking To The JSS

```
pip install python-jss
```

Then create preferences using `default`.

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. python-jss uses the ElementTree module.

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 on your JSS.*)

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

Now iterate over them

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

Further Places

- Dive Into Python - A good tutorial for experience programmers
- Python Programming For Beginners - Good tutorial for writing command line tools.
- PyObjC home page
- python-jss - python-jss home page on github
- ElementTree - ElementTree at python docs
- ElementTree overview - ElementTree tutorial