Python for System Administrators Documentation

Release 0.1a

Jason McVetta

August 18, 2014

Contents

1 Introduction **1**
- 1.1 Background Assumptions . . . 1
- 1.2 Environment . . . 1
- 1.3 Work In Progress . . . 1

2 Tools **3**
- 2.1 Virtual Environments . . . 3
- 2.2 Virtual Environments for Scripts . . . 4
- 2.3 Eclipse IDE . . . 4
- 2.4 Git - Version Control . . . 5

3 Using Third-Party Libraries **7**
- 3.1 PyPI . . . 7
- 3.2 Installation . . . 7
- 3.3 Finding Packages . . . 7
- 3.4 Popular Packages . . . 7

4 RESTful APIs **9**
- 4.1 What is a RESTful API? . . . 9
- 4.2 Standard Library . . . 10
- 4.3 *Requests* Libary . . . 10

5 Remote control of hosts over SSH **11**
- 5.1 subprocess.Popen() . . . 11
- 5.2 Fabric . . . 11
- 5.3 Task arguments . . . 12

6 Debugging Python Programs **15**
- 6.1 Techniques . . . 15
- 6.2 Debuggers . . . 16
- 6.3 Common Species of Bug . . . 16

7 SOAP APIs **19**
- 7.1 What is SOAP? . . . 19
- 7.2 Suds Library . . . 19

8 LDAP - Remote authentication **21**

CHAPTER 1

Introduction

Welcome to the class notes to *Python for System Administrators*.

The source code for these notes can be found on Github.

The latest version of these notes is published at Read the Docs. It is also available in PDF and ePub formats.

Classroom delivery of this course is available from Silicon Bay Training, who sponsored its development.

1.1 Background Assumptions

This class is designed for system administrators who will be using Python in the course of their work. It assumes students will have the following background knowledge:

- Solid understanding of unix/posix system administration.
- Comfortable working on the command line.
- Basic understanding of networking and tools like SSH.
- Basic understanding of Python language and syntax.

1.2 Environment

The following assumptions are made about the environment in which students will be taking the course:

- Ubuntu 12.04. All package installation etc examples assume you are working on an Ubuntu 12.04 desktop. It should be possible to run all the code examples on other posix-compatible platforms, but additional or different setup may be required.

1.3 Work In Progress

These class notes are a work in progress. Many sections are missing or incomplete. There are still many TODOs:

Todo

Brief description of `pdb`, maybe a simple example.

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/debugging.rst, line 86.)

Todo

Add an example for each (?) species of bug.

http://stackoverflow.com/questions/1011431/common-pitfalls-in-python

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/debugging.rst, line 96.)

Todo

Write entire LDAP section.

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/ldap.rst, line 5.)

Todo

Describe use of standard library for REST API calls

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/restful_apis.rst, line 111.)

Todo

Describe use of *Requests* library for REST API calls

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/restful_apis.rst, line 120.)

Todo

Write entire SOAP section.

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/soap.rst, line 5.)

Todo

Customize (hosts etc) all examples to match student VM setup

(The *original entry* is located in /var/build/user_builds/python-for-system-administrators/checkouts/latest/ssh.rst, line 6.)

CHAPTER 2

Tools

Using the right tools can give a big boost to a programmer's productivity.

The tools described in this section are available on most platforms. Installation instructions for Ubuntu Linux 12.04 are shown.

2.1 Virtual Environments

A virtual environment is a local Python environment isolated from the system-wide environment.

2.1.1 virtualenv

The term "virtualenv" can refer to the command `virtualenv`, used to create a virtual environment, or to the virutal environment itself.

```
$ sudo apt-get install python-virtualenv
```

2.1.2 virtualenvwrapper

`virtualenvwrapper` is a set of extensions to the `virtualenv` tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow, making it easier to work on more than one project at a time without introducing conflicts in their dependencies [1].

```
$ sudo apt-get install virtualenvwrapper
$ mkvirtualenv sysadmin
New python executable in sysadmin/bin/python
Installing distribute.....
Installing pip...............done.
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/sysadmin/bin/predeactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/sysadmin/bin/postdeactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/sysadmin/bin/preactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/sysadmin/bin/postactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/sysadmin/bin/get_env_details
(sysadmin)$ pip freeze # A few packages are installed by default
argparse==1.2.1
distribute==0.6.24
wsgiref==0.1.2
```

[1] http://www.doughellmann.com/projects/virtualenvwrapper/

Note that when the virtualenv is active, its name (in this case "sysadmin") is prepended to the shell prompt:

```
$ # Ordinary shell prompt
(sysadmin)$ # Virtualenv "sysadmin" is active
```

If later you have logged out, and want to activate this virtualenv, you can use the `workon` command:

```
$ workon sysadmin
(sysadmin)$
```

You can deactivate the virtualenv with the `deactivate` command:

```
(sysadmin)$ deactivate
$ # Back to normal shell prompt
```

2.1.3 Location of Virtualenvs

By default, `virtualenvwrapper` stores your virtualenvs in `~/.virtualenvs`. However you can control this by setting the `WORKON_HOME` environment variable. This could potentially be used for shared virtualenvs, perhaps with group write permission.

```
export WORKON_HOME=/path/to/virtualenvs
```

2.2 Virtual Environments for Scripts

There are several ways you can run scripts that rely on a virtualenv:

- Use Fabric's `prefix()` context manager when calling the script remotely:

  ```
  def task():
      with prefix('workon sysadmin'):
          run('uptime')
          run('uname -a')
  ```

- Have whatever is calling your script (`cron` etc) call `workon` first.
- Specify your virtualenv's Python interpreter directly in the script's bangline.
- Use a bash script as a wrapper. Ugly, but sometimes convenient.

2.3 Eclipse IDE

Eclipse is a powerful IDE - an integrated development environment. It provides valuable tools for understanding, browsing, and refactoring your code.

Out of the box, Eclipse does not support Python. However Eclipse is a plugin-based system, and there are excellent tools available for Python development.

2.3.1 Aptana / PyDev

The Python plugin for Eclipse, called *PyDev*, is now part of Aptana Studio. Aptana can be installed as a seperate download, or as an Eclipse plugin. For convenience we will download the whole application.

http://aptana.com/products/studio3/download

2.3.2 Installing Eclipse Plugins

Each Eclipse plugin has an *Update Site* URL, from which it can be installed.

To install a plugin in Eclipse, choose `Install New Software...` from the `Help` menu. Click the `Add...` button to add a new plugin repository. Put the plugin's *Update Site* URL in the `Location:` field.

Once you have added the plugin repository, check the box of the plugin you want to install. Click `Next >`, then click thru until it is installed. Normally Eclipse will want to restart itself after a new plugin has been installed.

2.3.3 Vwrapper

Vrapper is an Eclipse plugin providing VI-keys support. Only install this plugin if you are *certain* you want it.

Update site:

```
http://vrapper.sourceforge.net/update-site/stable
```

2.4 Git - Version Control

Even host-specific scripts should be version controlled. If no central VCS repository is available, Git can create a local repository. A local repo provides less safety against data loss than a remote central repo; but is still a huge step up from using *no* version control.

CHAPTER 3

Using Third-Party Libraries

3.1 PyPI

PyPI is the *Python Package Index*. It provides a central index of Python packages, any of which can be installed with a single command.

Not to be confused with PyPy, an alternative Python interpreter.

3.2 Installation

Use the `pip` command to install libraries into a virtual environment.

For example:

```
$ workon sysadmin # Activate virtualenv
(sysadmin)$ pip install ipython
```

If you really must install a library system-wide, *please* first look and see if a reasonable version of the library is available from your system's package manager.

3.3 Finding Packages

Google (Bing, etc) is your friend. So is Pip:

```
$ pip search mysql
sqlbean                   - A auto maping ORM for MYSQL and can bind with memcached
chartio                   - Setup wizard and connection client for connecting MySQL/PostgreSQL databa
tiddlywebplugins.mysql2   - MySQL-based store for tiddlyweb
MySQL-python              - Python interface to MySQL
lovely.testlayers         - mysql, postgres nginx, memcached cassandra test layers for use with zope.
zest.recipe.mysql         - A Buildout recipe to setup a MySQL database.
...
```

3.4 Popular Packages

Here are some popular, useful packages:

Package	Description
boto	Python interface to Amazon Web Services
django	Web application framework
fabric	SSH library and command line tool
flask	Lightweight web framework
gevent	High speed event handling library
gunicorn	Production webserver
ipython	Interactive Python shell
psycopg2	PostgreSQL driver
python-ldap	Object-oriented API to access LDAP directory servers
requests	HTTP for humans
restkit	HTTP resource kit
sphinx	Documentation tool

3.4.1 Django Plugins

These packages extend/enhance the Django framework:

Package	Description
docutils	Documentation utilities
django-bootstrap-form	Format Django forms to look nice with Twitter Bootstrap
django-celery	Distributed task queue
django-debug-toolbar	In-browser debugging utility
django-heroku-memcacheify	Automatic Django memcached configuration on Heroku.
django-heroku-postgresify	Automatic Django database configuration on Heroku.
django-social-auth	Authenticate using social account
django-sslify	Require SSL site-wide
django-storages	S3 storage for file attachments
django-templated-email	Template-based email
newrelic	Cloud-based monitoring service
pillow	Heroku-compatible imaging library
py-bcrypt	More secure password hashing
south	Database migration service

CHAPTER 4

RESTful APIs

4.1 What is a RESTful API?

A 'RESTful API' is a remote API that follows the *REST* style of software architecture.

4.1.1 REST - Representational State Transfer

REpresentational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant Web service design model. [1]

The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

Conforming to the **REST constraints** is generally referred to as being "RESTful".

4.1.2 The REST Constraints

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components free to design: [2]

Client–server

A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

Stateless

The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all of the information necessary to service the request, and any session state is held in the client.

Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well^managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

[1] http://en.wikipedia.org/wiki/Representational_state_transfer
[2] http://en.wikipedia.org/wiki/Representational_state_transfer#Constraints

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load^balancing and by providing shared caches. They may also enforce security policies.

Code on demand (optional)

Servers are able temporarily to extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client^side scripts such as JavaScript.

Uniform interface

The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are detailed below.

The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be considered RESTful.

Complying with these constraints, and thus conforming to the REST architectural style enables any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

4.1.3 JSON - Javascript Object Notation

JSON, or *JavaScript Object Notation*, is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages. [3]

The JSON format was originally specified by Douglas Crockford, and is described in RFC 4627. The official Internet media type for JSON is `application/json`. The JSON filename extension is `.json`.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

4.2 Standard Library

Todo

Describe use of standard library for REST API calls

4.3 *Requests* Libary

Todo

Describe use of *Requests* library for REST API calls

Citations

[3] http://en.wikipedia.org/wiki/JSON

CHAPTER 5

Remote control of hosts over SSH

Todo

Customize (hosts etc) all examples to match student VM setup

5.1 subprocess.Popen()

It is possible to control a local ssh session using `subprocess.Popen()` if no libraries are available. This is a super primative way to do things, and not recommended if you can avoid it.

Here is an example: [1]

```
import subprocess
import sys

HOST="www.example.org"
# Ports are handled in ~/.ssh/config since we use OpenSSH
COMMAND="uname -a"

ssh = subprocess.Popen(["ssh", "%s" % HOST, COMMAND],
                       shell=False,
                       stdout=subprocess.PIPE,
                       stderr=subprocess.PIPE)
result = ssh.stdout.readlines()
if result == []:
    error = ssh.stderr.readlines()
    print >>sys.stderr, "ERROR: %s" % error
else:
    print result
```

5.2 Fabric

Fabric is a library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

It provides a basic suite of operations for executing local or remote shell commands (normally or via sudo) and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution.

[1] https://gist.github.com/1284249

5.2.1 Basic Usage

Typical use involves creating a Python file named `fabfile.py`, containing one or more functions, then executing them via the fab command-line tool. Below is a small but complete `fabfile.py` containing a single task:

```
from fabric.api import run

def host_type():
    run('uname -s')
```

Once a task is defined, it may be run on one or more servers, like so:

```
(sysadmin)$ fab -H applebox,linuxbox host_type
[applebox] run: uname -s
[applebox] out: Darwin
[linuxbox] run: uname -s
[linuxbox] out: Linux

Done.
Disconnecting from localhost... done.
Disconnecting from linuxbox... done.
```

5.3 Task arguments

It's often useful to pass runtime parameters into your tasks, just as you might during regular Python programming. Fabric has basic support for this using a shell-compatible notation: `<task name>:<arg>,<kwarg>=<value>,...`. It's contrived, but let's extend the above example to say hello to you personally: [2]

```
def hello(name="world"):
   print("Hello %s!" % name)
```

By default, calling `fab hello` will still behave as it did before; but now we can personalize it:

```
(sysadmin)$ fab hello:name=Jeff
Hello Jeff!

Done.
```

Those already used to programming in Python might have guessed that this invocation behaves exactly the same way:

```
(sysadmin)$ fab hello:Jeff
Hello Jeff!

Done.
```

For the time being, your argument values will always show up in Python as strings and may require a bit of string manipulation for complex types such as lists. Future versions may add a typecasting system to make this easier.

5.3.1 Library Usage

In addition to use via the fab tool, Fabric's components may be imported into other Python code, providing a Pythonic interface to the SSH protocol suite at a higher level than that provided by e.g. the ssh library (which Fabric itself uses.) [3]

[2] http://docs.fabfile.org/en/1.4.2/tutorial.html#task-arguments
[3] http://stackoverflow.com/a/8166050

Consider the case where we want to collect average uptime from a list of hosts:

```
from fabric import tasks

env.hosts = ['localhost', 'sunflower.heliotropic.us']
pattern = re.compile(r'up (\d+) days')

# No need to decorate this function with @task
def uptime():
    res = run('uptime')
    match = pattern.search(res)
    if match:
        days = int(match.group(1))
        env['uts'].append(days)

def main():
    env['uts'] = []
    tasks.execute(uptime)
    uts_list = env['uts']
    if not uts_list:
        return # Perhaps we should print a notice here?
    avg = sum(uts_list) / float(len(uts_list))
    print '-' * 80
    print 'Average uptime: %s days' % avg
    print '-' * 80

if __name__ == '__main__':
    main()
```

CHAPTER 6

Debugging Python Programs

6.1 Techniques

6.1.1 The `print` statement

Sometimes the quickest/easiest solution is just to throw a few `print` statements into your code. Not a good idea for complex problems. Some very good programmers disdain this technique, calling it sloppy. However others really very good programmers, such as Rob Pike (early contributor to Unix, father of Plan 9, UTF-8, and Go) is said to approve of this method.

6.1.2 IPython

IPython is an interactive interpreter shell with more features than the default Python REPL shell.

While not used for *debugging* per se, an interactive session can be a good way to understand and explore small bits of code.

```
(venv)$ pip install ipython
```

6.1.3 Debugger

A debugger is a computer program that lets you run your program, line by line and examine the values of variables or look at values passed into functions and let you figure out why it isn't running the way you expected it to. [1]

6.1.4 Logging

While running code in a debugger offers the maximum visibility into its operation, it may be difficult or impossible to correctly simulate a production environment (including number of connections, network topography, load, etc) while a program is running inside a debugger.

Logging statements offer an alternative way to gain insight into your code's operation. Verbose logging can be especially helpful for understanding bugs in highly concurrent code, where it would be difficult to inspect each thread/process in an ordinary debugger.

Logging is an essential part of 12-Factor methodology for building modern applications.

[1] http://cplus.about.com/od/glossar1/g/debugdefinition.htm

6.2 Debuggers

6.2.1 Eclipse

Eclipse offers a graphical debugger, making it easy to explore your code and set breakpoints while running in the debugger.

Note: Eclipse's PyDev debugger is powerful, but *notoriously* flakey. *Expect a few hiccups!*

6.2.2 `pdb`

Todo

Brief description of `pdb`, maybe a simple example.

6.3 Common Species of Bug

Todo

Add an example for each (?) species of bug.

http://stackoverflow.com/questions/1011431/common-pitfalls-in-python

6.3.1 Indentation

Usually results from mixture of tabs & spaces, causing the actual scope of some lines to be different than it appears on the programmer's screen.

6.3.2 Wrong Number of Arguments

A function is called with the wrong number (too few or too many) arguments, causing an exception to be thrown.

6.3.3 Wrong Package

You type `import foobar`, and the module imported is `foobar.py` in the local folder, not the `foobar` module in the standard library.

6.3.4 Catchign Multiple Exceptions

Be careful catching multiple exception types: [2]

```
try:
    raise KeyError("hmm bug")
except KeyError, TypeError:
    print TypeError
```

[2] http://stackoverflow.com/q/1011431/164308

This prints "hmm bug", though it is not a bug; it looks like we are catching exceptions of both types, but instead we are catching KeyError only as variable TypeError.

PythonTutor can help us visualize what is happening.

The correct way to catch multiple exceptions is to put them in parentheses:

```
try:
    raise KeyError("hmm bug")
except (KeyError, TypeError):
    print TypeError
```

Visualize the correct program flow.

6.3.5 Unqualified `except:` block

Do you really want to catch *all* exceptions? Can your `except` block *really* recover from all the exceptions it catches?

6.3.6 Populating Arrays

When you need a population of arrays you might be tempted to type something like this: [3]

```
>>> a=[[1,2,3,4,5]]*4
```

And sure enough it will give you what you expect when you look at it

```
>>> from pprint import pprint
>>> pprint(a)

[[1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5]]
```

But don't expect the elements of your population to be seperate objects:

```
>>> a[0][0] = 2
>>> pprint(a)

[[2, 2, 3, 4, 5],
 [2, 2, 3, 4, 5],
 [2, 2, 3, 4, 5],
 [2, 2, 3, 4, 5]]
```

Unless this is what you need...

Visualize what is happening.

It is worth mentioning a workaround:

```
a = [[1,2,3,4,5] for i in range(4)]
```

[3] http://stackoverflow.com/a/1025447/164308

CHAPTER 7

SOAP APIs

Todo

Write entire SOAP section.

7.1 What is SOAP?

7.2 Suds Library

CHAPTER 8

LDAP - Remote authentication

Todo

Write entire LDAP section.