MINDFUL LEARNING

python

A language is not worth knowing unless it teaches you to think differently

- Larry Wall, Randal Schwartz

Professional

python

Systems Programming

python

datetime Module: time

- **datetime** contains functions and classes for working with dates and times, separately and together
- Time values are represented with the time class. Times have attributes for hour, minute, second, and microsecond. They can also include time zone information

```
import datetime

t = datetime.time(1, 2, 3)
print t
print 'hour  :', t.hour
print 'minute:', t.minute
print 'second:', t.second
print 'microsecond:', t.microsecond
print 'tzinfo:', t.tzinfo
```

MINDFUL LEARNING

python

datetime Module: date

- Calendar date values are represented with the date class. Instances have attributes for year, month, and day.
- It is easy to create a date representing today's date using the today() class method

```
import datetime

today = datetime.date.today()
print today
print 'ctime:', today.ctime()
print 'tuple:', today.timetuple()
print 'ordinal:', today.toordinal()
print 'Year:', today.year
print 'Mon :', today.month
print 'Day :', today.day
```

Combining Date and Time

- Use the datetime class to hold values consisting of both date and time components. As with date, there are several convenient class methods to make creating datetime instances from other common values.

```
import datetime

print 'Now    :', datetime.datetime.now()
print 'Today  :', datetime.datetime.today()
print 'UTC Now:', datetime.datetime.utcnow()

d = datetime.datetime.now()
for attr in [ 'year', 'month', 'day', 'hour',
  'minute', 'second', 'microsecond']:
    print attr, ':', getattr(d, attr)
```

python

Parsing Date and Time

- The default string representation of a datetime object uses the ISO 8601 format (YYYY-MM-DDTHH:MM:SS.mmmmmm).
- Alternate formats can be generated using **strftime()**
- Similarly, if your input data includes timestamp values parsable with **time.strptime()**, then **datetime.strptime()** is a convenient way to convert them to datetime instances.

MINDFUL LEARNING

python

Parsing Date and Time

```
import datetime
format = "%a %b %d %H:%M:%S %Y"
today = datetime.datetime.today()
print 'ISO      :', today
s = today.strftime(format)
print 'strftime:', s
d = datetime.datetime.strptime(s, format)
print 'strptime:', d.strftime(format)
```

https://docs.python.org/2/library/datetime.html
All the format specifiers can be seen here

python

File System and Directories

- Access to your file system occurs mostly through the Python **os** module
- **os** module is actually a front-end to the real module that is loaded, a module that is clearly operating system dependent
- This "real" module may be one of the following: posix (Unix-based, i.e., Linux, MacOS X, *BSD, Solaris, etc.), nt (Win32), mac (old MacOS), dos (DOS), os2 (OS/2), etc.
- You should never import those modules directly
- Just import os and the appropriate module will be loaded, keeping all the underlying work hidden from sight

Refer:
https://docs.python.org/3/library/os.html
https://docs.python.org/2/library/os.html

MINDFUL LEARNING

python

Important Functions in **os** Module

Function	Purpose
mkdir()/makedirs()	Create directory(ies)
rmdir()/removedirs()	Remove directory(ies)
getcwd()/getcwdu()	Return current working directory/same but in Unicode
listdir()	List files in directory
chroot()	Change root direcory of current process
chdir()/fchdir()	Change working directory/via a file descriptor
access()	Verify permission modes
chmod()	Change permission modes
remove()/unlink()	Delete file
rename()/renames()	Rename file
open()	Low-level operating system open [for files, use the **standard open() built-in** functions
read()/write()	Read/write data to a file descriptor

os.path Module

- A second module that performs specific pathname operations is also available which called **os.path**
- The **os.path** module is accessible through the **os** module
- Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries

Important Functions in **os.path** Module

Function	Purpose
basename()	Remove directory path and return leaf name
dirname()	Remove leaf name and return directory path
join()	Join separate components into single pathname
split()	Return (dirname(), basename()) tuple
splitdrive()	Return (drivename, pathname) tuple
splitext()	Return (filename, extension) tuple
getatime()	Return last file access time
getctime()	Return file creation time
getmtime()	Return last file modification time
getsize()	Return file size (in bytes)
exists()	Does pathname (file or directory) exist?
isabs()	Is pathname absolute?
isdir()	Does pathname exist and is a directory?
isfile()	Does pathname exist and is a file?
islink()	Does pathname exist and is a symbolic link?
ismount()	Does pathname exist and is a mount point?
samefile()	Do both pathnames point to the same file?

A Practical Example: **os** and **os.path**

- This example exercises some of the functionality found in the os and os.path modules.
- It creates a test file, populates a small amount of data in it, renames the file, and dumps its contents.
- Other auxiliary file operations are performed as well, mostly pertaining to directory tree traversal and file pathname manipulation
- Study the operations carefully justifying the output

MINDFUL LEARNING

python

Solution

```
# ospathx__demo.py: Shows os.path module's operations
#!/usr/bin/env python

import os
for tmpdir in ('/tmp', 'E:\WINDOWS\Temp'):
    if os.path.isdir(tmpdir):
        break
    else:
        print ('no temp directory available')
        tmpdir = ''

if tmpdir:
    os.chdir(tmpdir)
    cwd = os.getcwd()
    print ('*** current temporary directory')
    print (cwd)

    print ('*** creating example directory...')
    os.mkdir('example')
    os.chdir('example')
    cwd = os.getcwd()
    print ('*** new working directory:')
    print (cwd)
    print ('*** original directory listing:')
    print (os.listdir(cwd))
```

Solution

MINDFUL LEARNING

python

```
print ('*** creating test file...')
    file = open('test', 'w')
    file.write('foo\n')
    file.write('bar\n')
    file.close()
    print ('*** updated directory listing:')
    print (os.listdir(cwd))

    print ("*** renaming 'test' to 'filetest.txt'")
    os.rename('test', 'filetest.txt')
    print ('*** updated directory listing:')
    print (os.listdir(cwd))

    path = os.path.join(cwd, os.listdir(cwd)[0])
    print ('*** full file pathname:')
    print (path)
    print ('*** (pathname, basename) == ')
    print (os.path.split(path))
    print ('*** (filename, extension) == ')
    print (os.path.splitext(os.path.basename(path)))
```

Common path name manipulation functions:

split

splitext

join

Solution

MINDFUL LEARNING

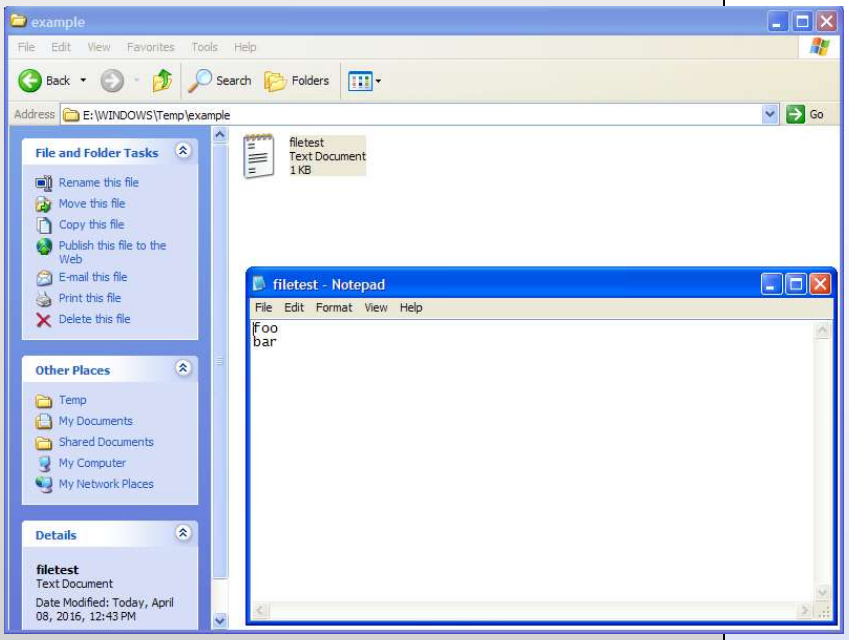
python

```
print ('*** displaying file contents:')
file = open(path)
allLines = file.readlines()
file.close()
for eachLine in allLines:
    print (eachLine)

print ('*** deleting test file')
os.remove(path)
print ('*** updated directory listing:')
print (os.listdir(cwd))
os.chdir(os.pardir)
print ('*** deleting test directory')
os.rmdir('example')
print ('*** DONE')
```

Output

```
 >>>
no temp directory available
*** current temporary directory
E:\WINDOWS\Temp
*** creating example directory...
*** new working directory:
E:\WINDOWS\Temp\example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing:
['filetest.txt']
*** full file pathname:
E:\WINDOWS\Temp\example\filetest.txt
*** (pathname, basename) ==
('E:\\WINDOWS\\Temp\\example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```



Snap before deleting

Executing Non-Python Programs

- **os** module provides several functions to run non-python programs such as binary executables and shell scripts
- All that is required is
  - A valid execution environment
  - Permissions to file access and execution
  - Shell scripts must be able to access their interpreters
  - Binaries must be accessible

os Module Functions for Running External Programs

MINDFUL LEARNING

python

os Module Function	Description
system(cmd)	Execute program cmd given as string, wait for program completion, and return the exit code (on Windows, the exit code is always 0)
fork()	Create a child process that runs in parallel to the parent process [usually used with exec*()]; return twice... once for the parent and once for the child (Unix/Linux Only)
execl(file, arg0, arg1, ...)	Execute file with argument list arg0, arg1, etc.
execv(file, arglist)	Same as execl() except with argument vector (list or tuple) arglist
execle(file, arg0, arg1, ...env)	Same as execl() but also providing environment variable dictionary env
execve(file, arglist, env)	Same as execle() except with argument vector arglist
execlp(cmd, arg0, arg1,...)	Same as execl() but search for full file pathname of cmd in user search path
execvp(cmd, arglist)	Same as execlp() except with argument vector arglist
execlpe(cmd, arg0, arg1,...env)	Same as execlp() but also providing environment variable dictionary env
execvpe(cmd, arglist, env)	Same as execvp() but also providing environment variable dictionary env

os Module Functions for Running External Programs

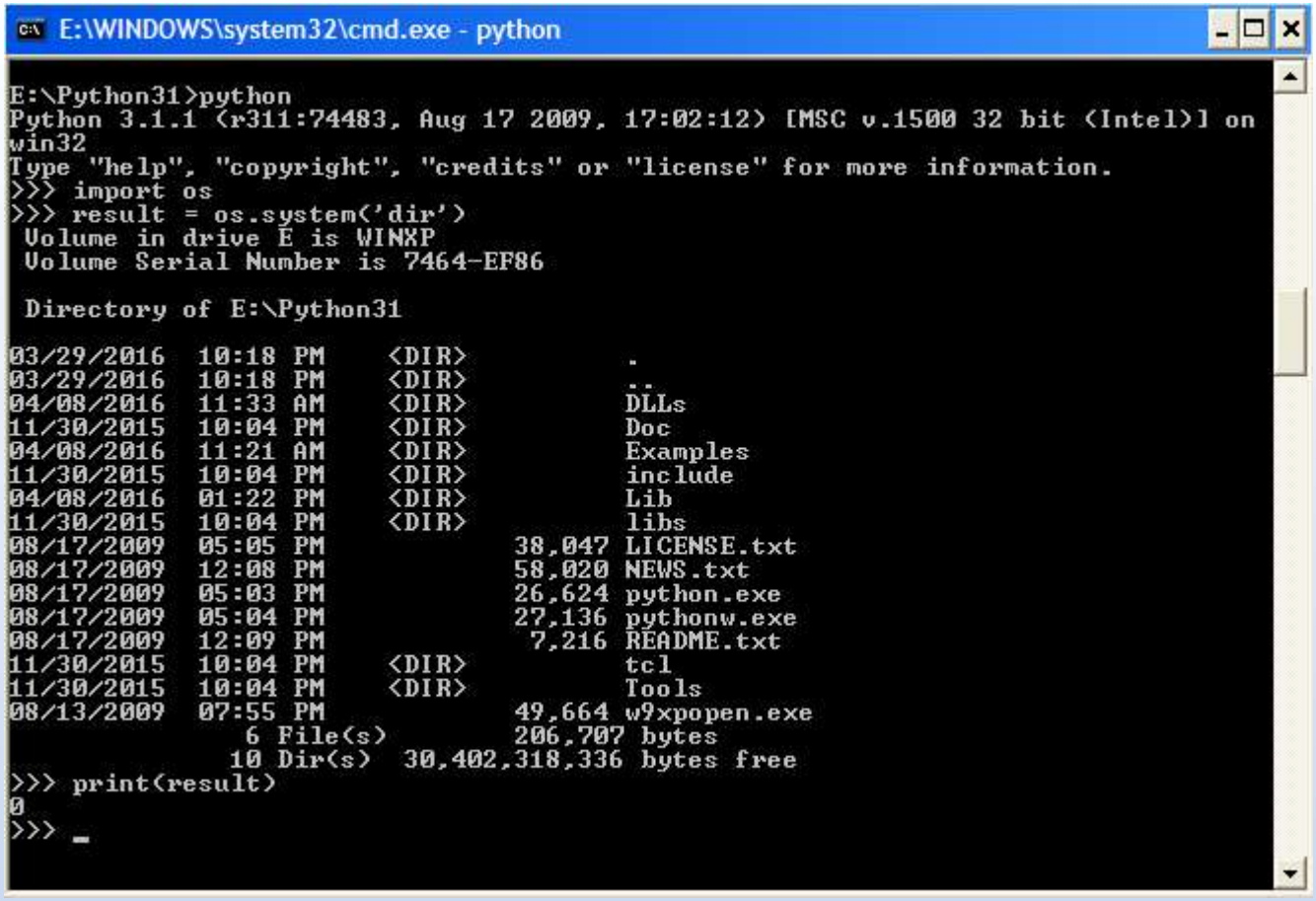
MINDFUL LEARNING

python

os Module Function	Description
spawn*(mode, file, args[, env])	spawn*() family executes path in a new process given args as arguments and possibly an environment variable dictionary env;mode is a magic number indicating various modes of operation
wait()	Wait for child process to complete [usually used with fork() and exec*()]
waitpid(pid, options)	Wait for specific child process to complete [usually used with fork() and exec*()]
popen(cmd, mode='r ', buffering=-1)	Execute cmd string, returning a file-like object as a communication handle to the running program, defaulting to read mode and default system buffering
startfile(path)	Execute path with its associated application

Refer: https://docs.python.org/3/library/os.htm I

Example: os.system()



python

Executable Object Statements

- Python provides a number of BIFs (built-in functions) supporting callables and executable objects, including the **exec** statement
- These functions let the programmer execute code objects as well as generate them using the **compile()** BIF
- Important functions are listed in the next slide

MINDFUL LEARNING

python

Executable Object BIFs

Built-in Function or Statement	Description
compile(string, file, type)	Creates a code object from string of type type; file is where the code originates from (usually set to ""). The type argument specifies what kind of code must be compiled; it can be '**exec**' if source consists of a sequence of statements, '**eval**' if it consists of a single expression, or '**single**' if it consists of a single interactive statement
eval(obj, globals=globals(), locals=locals())	Evaluates obj, which is either an expression compiled into a code object or a string expression; global and/or local namespace may also be provided
exec (obj)	Executes obj,a single Python statement or set of statements, either in code object or string format; obj may also be a file object (opened to a valid Python script)

Refer: https://docs.python.org/3/library/functions.html

MINDFUL LEARNING

python

Examples of Executable BIFs

```
>>> def info():
        print('This is a pythonic style :)')


>>> info()
This is a pythonic style :)
>>> callable(info)
True
```

callable(): Removed in 3.0
brought back in 3.2

```
>>> code = """
def gcd(x, y):
    while y != 0:
        (x, y) = (y, x % y)
    return x

print('GCD Calculator')
a = int(input('Enter A : '))
b = int(input('Enter B : '))
print(gcd(a, b))
"""
>>> executable = compile(code, '', 'exec')
>>> exec(executable)
GCD Calculator
Enter A : 15
Enter B : 45
15
```

Executing Other Python Programs

- Other python programs can be executed by calling execfile() in python 2.7.x or exec() in python 3.x from with in the function

```
>>> exec(open('E:\Python31\Examples\height_of_tree.py').read())
Enter the angle in degrees: 45
Enter the distance in meters : 76
Height of the tree will be approximately 76.00 meters
```

Executing Other Python Programs

- Another module-related trick lets you both import a file as a module and run it as a standalone program and is widely used in Python files
- Each module has a built-in attribute called **__name__**, which
- Python creates and assigns automatically as follows:
  - If the file is being run as a top-level program file, **__name__** is set to the string **"__main__"** when it starts.
  - If the file is being imported instead, **__name__** is set to the module's name as known by its clients.
  - In effect, a module's **__name__** variable serves as a usage mode flag, allowing its code tobe leveraged as both an importable library and a top-level script.

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Refer: https://docs.python.org/2/using/cmdline.html for command line options

MINDFUL LEARNING

python

Example

```
# info.py
def quote():
    print("""He who has injured thee was either stronger or weaker than thee.
             If weaker, spare him; if stronger, spare thyself.""")
    print('William Shakespeare')

if __name__ == '__main__':  # Only when run
    quote()                 # Not when imported
```

```
>>>
He who has injured thee was either stronger or weaker than thee.
             If weaker, spare him; if stronger, spare thyself.
William Shakespeare
```

Run

```
# test_import.py
import info
print("Testing __name__ == '__main__'")
```

```
>>>
Testing __name__ == '__main__'
```

Imported

MINDFUL LEARNING

python

Example

```
# info.py
def quote():
    print("""He who has injured thee was either stronger or weaker than thee.
             If weaker, spare him; if stronger, spare thyself.""")
    print('William Shakespeare')

# if __name__ == '__main__':  # Only when run
quote()                       # Not when imported
```

```
>>>
He who has injured thee was either stronger or weaker than thee.
             If weaker, spare him; if stronger, spare thyself.
William Shakespeare
```

Run

```
# test_import.py
import info
print("Testing __name__ == '__main__'")
```

```
>>>
He who has injured thee was either stronger or weaker than thee.
             If weaker, spare him; if stronger, spare thyself.
William Shakespeare
Testing __name__ == '__main__'
```

Imported

Command Line Arguments

- The **sys** module also provides access to any command-line arguments via **sys.argv**
  - **sys.argv** is the list of command-line arguments
  - **len(sys.argv)** is the number of command-line arguments
- Command line arguments are extremely useful for chaining different programs where the output of one program is piped into input of another program

Terminating Execution

- The primary way to exit a program immediately and return to the calling program is the **exit()** function found in the **sys** module

  Syntax: **sys.exit()**

python

Module **subprocess**

- The subprocess module provides a consistent interface to creating and working with additional processes.
- It offers a higher-level interface than some of the other available modules, and is intended to replace functions such as:
  - os.system(),
  - os.spawn*(),
  - os.popen*(),
  - popen2.*()
  - commands.*()

Running External Command

- To run an external command without interacting with it, such as one would do with *os.system()*, use the call() function

```
import subprocess

subprocess.call(['ls', '-l'], shell=True)
```

```
subprocess.call('ls -l')
```

This will not work because subprocess will look for executable file called 'ls –l' which obviously is not found

```
subprocess.call('ls -l', shell=True)
```

Here subprocess will run this in the shell, 'ls –l' is passed as a shell command string

```
subprocess.call(['ls', '-l'])
```

'-l' is passed as command line argument

Getting the Return Code

- The return value from call() is the exit code of the program.
- The caller is responsible for interpreting it to detect errors.
- The check_call() function **works like** call() except that the exit code is checked, and if it indicates an error happened then a **CalledProcessError** exception is raised.

MINDFUL LEARNING

python

Getting the Return Code

```
import subprocess
subprocess.check_call(['false'])
```

The **false** command always exits with a non-zero status code, which check_call() interprets as an error. You can try with 'ls –l'

```
$ python subprocess_check_call.py

Traceback (most recent call last):
  File "subprocess_check_call.py", line 11, in <module>
    subprocess.check_call(['false'])
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.
7/subprocess.py", line 511, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['false']' returned non-zero e
xit status 1
```

MINDFUL LEARNING

python

Capturing Output

- Use check_output() to capture the output for later processing

```
import subprocess
output = subprocess.check_output(['ls', '-1'])
print 'Have %d bytes in output' % len(output)
print output
```

Example

- This script runs a series of commands in a subshell.
- Messages are sent to standard output and standard error before the commands exit with an error code

```
import subprocess

output = subprocess.check_output(
    'echo to stdout; echo to stderr 1>&2; exit 1',
    shell=True,
    )
print 'Have %d bytes in output' % len(output)
print output
```

Working with Pipes

- Module subprocess defines a class called **Popen** which helps in opening a pipe to or from command
- The underlying process creation and management in subprocess module is handled by the **Popen** class
- It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions
- **subprocess.Popen()** executes a child program in a new process

subprocess.Popen()

MINDFUL LEARNING

python

- Arguments

```
class subprocess.Popen(args, bufsize=0, executable=None,
stdin=None, stdout=None, stderr=None, preexec_fn=None,
close_fds=False, shell=False, cwd=None, env=None,
universal_newlines=False, startupinfo=None, creationflags=0)
```

python

Methods in Popen()

- poll()
  - Check if child process has terminated.
  - Returns returncode attribute.
- wait()
  - Wait for child process to terminate.
  - Returns returncode attribute.
- communicate(input=None)
  - Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached.
  - Wait for process to terminate.
  - The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.
  - communicate() returns a tuple (stdout, stderr).

MINDFUL LEARNING

python

Attributes in Popen()

- stdin
  - If the stdin argument is **PIPE**, this attribute is a file object that provides input to the child process. Otherwise, it is None.
- stdout
  - If the stdout argument is **PIPE**, this attribute is a file object that provides output from the child process. Otherwise, it is **None**.
- stderr
  - If the stderr argument is **PIPE**, this attribute is file object that provides error output from the child process. Otherwise, it is **None**.
- pid
  - The process ID of the child process
- returncode
  - The child return code.
  - A None value indicates that the process hasn't terminated yet.
  - A negative value -N indicates that the child was terminated by signal N (UNIX only).

Example

- The following script will print the message on th console

```
import subprocess
p = subprocess.Popen(["echo", "hello world"],
     stdout=subprocess.PIPE)
print p.communicate()
>>>('hello world ', None)
```

MINDFUL LEARNING

python

Example

- Let's write our own ping program where we first ask the user for input, and then perform the ping request to that host.

```
import subprocess

# Ask the user for input
host = raw_input("Enter a host to ping: ")

# Set up the echo command and direct the output to a pipe
p1 = subprocess.Popen(['ping', '-c 2', host],
   stdout=subprocess.PIPE)

# Run the command
output = p1.communicate()[0]
print output
```

Mindful Learning

python

Example

- This time we use the host command for the previous example

```
target = raw_input("Enter an IP or Host to ping:")

host = subprocess.Popen(['host', target], stdout =
  subprocess.PIPE).communicate()[0]

print host
```

Working with Shell Scripts

- Shell scripts can be invoked in the following way as shown in the example

```
import os
arglist = 'arg1 arg2 arg3'
bashCommand = "/bin/bash script.sh " + arglist
os.system(bashCommand)
```

- It can also be done with **call()** in the **subprocess** module with **shell=True**

python

SSH Connection with Python

- It is possible to control a local ssh session using **subprocess.Popen()** if no libraries are available

```
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
```

pxssh

- This class extends **pexpect.spawn** to specialize setting up SSH connections.
- This adds methods for login, logout and expecting shell prompt
- It does various tricky things to handle many sit uations in the SSH login process

http://pexpect.sourceforge.net/pxssh.html

MINDFUL LEARNING

python

Example

```
import pxssh
import getpass
try:
    s = pxssh.pxssh()
    hostname = raw_input('hostname: ')
    username = raw_input('username: ')
    password = getpass.getpass('password: ')
    s.login (hostname, username, password)
    s.sendline ('uptime')   # run a command
    s.prompt()              # match the prompt
    print s.before          # print everything before the prompt.
    s.sendline ('ls -l')
    s.prompt()
    print s.before
    s.sendline ('df')
    s.prompt()
    print s.before
    s.logout()
except pxssh.ExceptionPxssh, e:
    print "pxssh failed on login."
    print str(e)
```

fabric

- Fabric is a Python library used for interacting with SSH and computer systems to automate a wide range of tasks, varying from application deployment to general system administration
- One of the key areas for using Fabric is automating the everyday tasks of system (and server) administration. These jobs include pretty much everything that relates to:
  - Building a server
  - Its maintenance
  - Monitoring

fabric

- Deploying an application (regardless of it being a web site, an API, or a server) usually means:
  - setting up a system from scratch (or from a snapshot taken in time),
  - preparing it by updating everything
  - downloading dependencies
  - setting up the file structure and permissions
  - followed by finally uploading your codebase - or downloading it using a SCM such as Git.
- Being able to automate these tasks in a logical manner becomes invaluable
- This is where Fabric comes to your aid

python

Fabric's Integration with SSH

- **run** *(fabric.operations.run)*
  - *Running SSH commands*
- **sudo** *(fabric.operations.sudo)*
  - *Running with superuser previlages*
- **local** *(fabric.operations.local)*
  - *Allows to run commands on the local machine*
- **get** *(fabric.operations.get)*
  - *Download files*
- **put** *(fabric.operations.put)*
  - *Upload files*
- **prompt** *(fabric.operations.prompt)*
  - *Create a prompt to the user*
- **reboot** *(fabric.operations.reboot)*
  - *Reboots the machine*

Mindful Learning

python

Fabric's Context Managers

- **cd** *(fabric.context_managers.cd)*
  - *Changing directory state in the SSH session*
- **lcd** *(fabric.context_managers.lcd)*
  - *Changing directory state in the local machine*
- **path** *(fabric.context_managers.path)*
  - *Allows to alter the path variable*
- **settings** *(fabric.context_managers.settings)*
  - *Temporarily override the environment variables*
- **prefix** *(fabric.context_managers.prefix)*
  - *Allows prefixing run and sudo commands*

MINDFUL LEARNING

python

Basic Usage

- Typical use involves:
  - creating a Python module containing one or more functions
  - then executing them via the **fab** command-line tool.
- Below is a small but complete "fabfile" containing a single task:

```
#fabfile.py
from fabric.api import run

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

```
$ fab -H localhost,linuxbox host_type
[localhost] run: uname -s
[localhost] out: Darwin
[linuxbox] run: uname -s
[linuxbox] out: Linux

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

MINDFUL LEARNING

python

Example

```
#fabfile.py
# Fabfile to:
#    - update the remote system(s)
#    - download and install an application

# Import Fabric's API module
from fabric.api import *

env.hosts = [
    'server.domain.tld',
  # 'ip.add.rr.ess
  # 'server2.domain.tld',
]
# Set the username
env.user   = "root"

# Set the password [NOT RECOMMENDED]
# env.password = "passwd"

def update_upgrade():
    """
        Update the default OS installation's
        basic default tools.
                                              """
    run("aptitude    update")
    run("aptitude -y upgrade")
```

MINDFUL LEARNING

python

Example

```
def install_memcached():
    """ Download and install memcached. """
    run("aptitude install -y memcached")

def update_install():

    # Update
    update_upgrade()

    # Install
    install_memcached()
```

```
# Automate everything!
fab update_install
```