Northern Test Consulting

Taming the Terminal-based Applications and Testing Them

Kristoffer Nordström

@kristoffer_nord

Getting started

0. Copy the folder from the USB stick to your hard drive.

1. Pop in the USB stick provided
2. Open VirtualBox
3. ”Machine” -> ”Add…”
4. Navigate to the USB stick, open the folder and open the .vbox file
5. Start the new virtual machine

User: pft

Password: pft

What is Python?

Long version:

Python is an interpreted, interactive, object-oriented programming language.

It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++.

It is also usable as an extension language for applications that need a programmable interface.

Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later.

Short version:

Python is **fun** and **easy** to learn. It is a programming language very suitable for simple tasks that testers face on a day-to-day basis.

Don’t know any UNIX?

Don’t worry, with these commands you should be able to get around just fine today:

- CTRL + ALT + T
  - Opens a terminal in Ubuntu
- cd <folder_path>
  - Enters the folder_path specified
- ll
  - Lists the content of the current directory
- mkdir <folder_path>
  - Creates a new folder
- rm –rf <folder_path>
  - Removes a folder (can removes files as well, -r not needed)
- cat <filename>
  - Reads the file and prints it input to the terminal
- touch <filename>
  - Creates an empty file

Other information

We will be using:

- A Virtual Machine running Ubuntu Desktop as the OS
- SublimeText 3 as the main Python editor
  - With Anaconda package installed for.
    - Code autocompletion
    - Lint check
  - Use command `subl` to start from terminal
- Bpython for a fancy REPL
- Python 2.7

- All the skeleton code and answers for the exercises are located under the folder '~/TTBA_Exercises/solution/'

And finally…

[Learning goals]

Learning goals for today is to learn the Python syntax, get comfortable with Python and see some concrete examples on how you can make use Pexpect and Python to automate and even check terminal based applications.

You should be able to go home, explore and learn more of Python on your own after today.

[Other]

Bugs may have found their way into the printed material, you're testers I'm sure you will… find them… report a bug… embarrase me…

You are testers I expect you to explore the concepts put forth today.

Fill in the blanks between the lines in the material, experiment, and have some fun.

Also please experiment in the REPL or SublimeText while I'm talking. It's encouraged.

Ask questions…

Speaking of bugs...

The scaling of your display is likely all wrong.

Please start a terminal (CTRL+ALT+T) and then issue these commands:

```
$ gsettings set org.gnome.desktop.interface scaling-factor n
$ gsettings set org.gnome.desktop.interface text-scaling-factor n
```

Getting the environment up and running

REPL
Import
BPython
Indentations
Variables

Running Python from the cmd line

1. Start a terminal
2. Create a temporary folder to hold your scripts and enter it
3. Start SublimeText by typing: *subl helloworld.py*
4. Add the line and save: *print "Hello World"*
5. Switch back to the terminal and run: *python helloworld.py*

”The REPL”

Read – Eval – Print – Loop

Simply start it with the command: python

The REPL is an interactive interpreter that takes single line input, interpets it and returns the result to the user.

To exit the REPL simply provide the end-of-file character (CTRL+D Unix, CTRL+Z Windows). Or use one of the commands quit() or exit()

The REPL is a great way for you to explore new python commands or to do simple one-off instructions intended to be thrown away later.

```
pft@PFT-ubuntu:~# python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print “Hello World”
Hello World
>>> raw_input()
Type here
’Type here’
>>>
```

Start the REPL and print a string to the terminal like the example above.

Import

import_stmt ::= "import" module ["as" name] (",“ module ["as" name])* |
"from" relative_module "import" identifier ["as" name] ("," identifier ["as" name])* |
"from" relative_module "import" "(" identifier ["as" name] ("," identifier ["as" name])* [","] ")" |
"from" module "import" "*"

The way to import modules containing code you need.

```
import shutil
import random as ra
from os import *
```

```
from datetime import datetime
from datetime import datetime as dt
```

Try the examples above, then try importing the module time and call time.sleep(10).

What happened?

BPython

"bpython is a fancy interface to the Python interpreter for Linux, BSD, OS X and Windows (with some work)."

A prefered alternative to the standard REPL included in Python. Offers features such as:

- *In-line syntax highlighting*
- *Readline-like autocomplete with suggestions displayed as you type.*
- *Expected parameter list for any Python function.*
- *"Rewind" function to pop the last line of code from memory and re-evaluate.*
- *Send the code you've entered off to a pastebin.*
- *Save the code you've entered to a file.*
- *Auto-indentation.*

Start `bpython` from the terminal and print again to the terminal a few times.
Next test the "CTRL+S" command and save to a file.
Exit BPython and take a look at the output file, try rerunning the file with Python.

Indentations – Forget {}

Blocks of code in are Python are grouped by indenting the code with either a tab or spaces.

De facto standard is to use 4 spaces of indentation per level.

Also a ':' is used at the end of the parent level (class, if-statement, etc)

```
if myVar == True:
    print "Hit"
else:
    print "Miss"
```

```
for number in range(0,4):
    print number
```

```
while myCondition == 1:
    print "First Level"
    if otherCondition == True:
        print "Other condition was True"
```

Do not mix tabs and spaces in the same file as that is not allowed!

"IndentationError: unexpected indent"

Other languages typically use {} to enclose blocks of code. Not so in Python.

Try importing support for curly braces from the module __future__:

```
from __future__ import braces
```

Objects

Objects are Pythons abstraction for all data.
You can assign values to objects such as strings, integers, floats, but also more complex objects such as lists, dictionaries, classes, and even methods.

```
#A String
obj = ’Python For Testers’

#An integer
numb = 9
```

```
#A list
My_list = [’FirstItem’, ’SecondItem’, ’AndSoOn’]

#A float
Float_obj = 0.9
```

From the REPL try assigning values to objects and then print them *(`print obj`)*.

Also adding two numbers together and see the result *(`numb + Float_obj`)*, assign the result of that to another object *(`result = numb + Float_obj`)*. Other modifiers you can try are: -, /, %
Explain the difference between % and / when used on two integers.

What happens if you add two or more strings?
Explore the operators ’+=’ and ’-=’ *(myvalue += value)*

Concatenating strings

A lot of times we want to print a string that is comprised of several strings.

In Python it’s really easy to do, all it takes is the use of the + operator:

```
firstName = ’Kristoffer’
lastName = ’Nordström’

print firstName + ’ ’ + lastName
```

Scope in Python

```
outer_name = ”Kristoffer”
if condition is True:
    inner_name = ”Nordstrom”
    print outer_name + ’ ’ + inner_name
else:
    other_name = ”Andersson”
    print ”inner_name not available”

print ”inner_name or other_name maybe available”
```

Casting to other types

Sometimes we need to explicitly cast the type of the object to another type in order to do something.

Most common is to cast something into the type string when printing objects.

You can print single integer, but if you want to print a string together with (for example) an integer you need to cast it before concatenating the objects.

```
number = 2
greeting = ’Hello ’

#Prints ”Hello 2 times”
print greeting + str(number)  + ’ times’
```

But we also sometimes need to convert from for example a float to an integer.

```
number = 2.0
print number
print int(number)
```

Exercise – Hello {Name}

Write a small Python script that asks for your name and then prints out:

Hallo {Name} en welkom naar Nieuwegein!

Use a raw_input(), print, and concatenating strings.

Northern Test Consulting

Exercise v.1

Exercise the installer with PExpect

Pexpect General
Pexpect expect
Pexpect sendline

Pexpect

Pexpect makes Python a better tool for controlling other applications.

Pexpect is a pure Python module for spawning child applications; controlling them; and responding to expected patterns in their output.

Pexpect can be used for automating interactive applications such as ssh, ftp, passwd, telnet, etc. It can be used to a automate setup scripts for duplicating software package installations on different servers. It can be used for automated software testing.

Source: http://pexpect.sourceforge.net/doc/

Install Python Packages – Distutils

Distutils is a standard as to how a developer can choose to distibuted the python module in source code form.

If you download a Python module, it comes in a *.tar.gz or *.zip file, and if you find a setup.py file when you extract it then you have encountered a Python module distributed with Distutils.

To install the package simply execute:

```
python setup.py install
```

In the folder /opt/PFT/pexpect/ you find the file pexpect-2.3.tar.gz.
Unpack it (`tar xvzf pexpect-2.3.tar.gz`) and enter the extracted folder.

Next install your first 3rd party python module using the command above.
In bpython try importing pexpect to validate it installed properly.

Pexpect - Expect

Pexpect works by that it starts an external process and then ”expects” certain output from that terminal based process.

At that point it can send input back into the process depending on the output encountered.

It supports regular expressions in the output that it expects from the process started.

If you want what is happening with pexpect to be written to the screen you simply set logfile to sys.stdout (that could be a file object instead for logging).

```
import pexpect, sys

child = pexpect.spawn('uname -a')
child.logfile = sys.stdout

child.expect('.*Linux.*')
print ”We're on Linux! Halleluja!”
```

Pexpect - Sendline

Once pexpect finds a line it expects, it then has the option of sending a line back into the process.

It is simply done by invoking the command: sendline()

You can send any string you want to, if you want to send an empty line, i.e an enter stoke just send \n.

```
import pexpect, sys

child = pexpect.spawn('python ~/myscript.py')
child.logfile = sys.stdout

child.expect('What is your name:')
child.sendline('Kristoffer')
child.expect('Hello Kristoffer, press enter to continue:')
child.sendline('\n')
```

Exercise - Pexpect

If you run the python script installer.py (under ~/TTBA_Exercises/) you will find a terminal based installer for a 3rd party tool.

Explore it and learn about its behavior.

Then write a python script (runner.py) that uses pexpect and exercises the installer and installs the software to the system.

For your convenience you can find the questions that the installer asks in the file questions.txt

The installer can exit with:

- 'Finished installation successfully'

Exercise v.2

Expecting different answers

Casting types
if/else
Booleans
Lists and Dicts
Pexpect multiple choices

If/Else

if_stmt ::= "if" expression ":" suite
              ("elif" expression ":" suite)*
              ["else" ":" suite]

An ”if” statement is used to decide if a block of code (”suite”) is to be executed.
Can be followed by ”elif” and ”else”.

```
x = -1
name = 'Lisa'

if x < 0:
    print "X is negative"

if name == 'Lisa':
    print "Name is Lisa"
```

```
fullName = "Kristoffer Nordstrom"

if "Nord" in fullName:
    print "Name contains string Nord"

if "Schoots" not in fullName:
    print "Schoots not part of Name"
```

```
if name == 'Robert':
    print "Hi Robert"
elif name == 'Ann':
    print "Welcome Ann"
else:
    print "Howdy stranger"
```

Try out the examples above in sublimetext and a .py file

Boolean

A boolean is a data type that evalutes to the values ”True” or ”False”.

```
condition = True
if condition is True:
    print ”Condition was True”
```

```
condition = False
if condition is True:
    print ”Condition was True”
else:
    print ”Condition was False”
```

```
condition = True
if condition is not False:
    print ”Condition was True”
```

```
first = False
second = True

if first is False and second is True:
    print ”Conditions was met”
```

Try out the examples above in the terminal

Python Data Types – List & Dict

Python offers several data types to store data, two of them are lists and dicts.

A list contains just that, a list of data objects (strings, numbers, booleans, even methods) that you can pass around.

A dict is as dictionary where every data object also has a key.

```
Names = [’James’, ’Michael’, ’Paul’]
print Names
Names.append(’Huib’)
Names.append(’Kristoffer’)
print Names
print Names[0]
print Names[2]
Names.remove(’Kristoffer’)
print Names
print len(names)
```

```
Person = {'First': 'Bilbo', 'Last': 'Bagger'}
print Person
Person['Address'] = 'Bag End, Bagshot Row'
print Person
print Person['First'] + ' lives at ' + Person['Address']
Person.pop('Address')
print len(Person)
print Person.keys()
```

Use the code examples above to experiment with creating your own lists and dicts.
The dict method keys() returns a list, can you use that in a for loop?

Pexpect – Expect

Pexpect also supports expecting multiple options and then allowing you to act according to the result.

Instead of supplying one string of what to expect, simply supply a list of strings that are valid results.

The result of *expect()* is then an integer indicating which of the strings that was encountered.

```
import pexpect, sys

child = pexpect.spawn('uname -a')
child.logfile = sys.stdout

result = child.expect(['.*ubuntu.*', '.*redhat.*'])
if result == 0:
    print "We're on Ubuntu!"
elif result == 1:
    print "We're on RedHat!"
```

Exercise – v.2

Extend your runner.py to expect the different possible results from the installer and that prints the result depending on what occured.

Be aware that the installer can exit with the different messages:

- 'Finished installation successfully‘
- 'An error occurred during installation'

Northern Test Consulting

Exercise v.3

Fetch the questions/answers from a list

For-loops

Zip()

For loop and Range()

for_stmt ::= "for" target_list "in" expression_list ":" suite

The for loop is your way to iterate over several objects in a collection.
It is also the way you repeat a block of code X number of times using the **range()** method.

range() returns a list (collection) of numbers in the range specified.

```
for i in range(0,5):
    print i #Will print numbers 0 to 4
```

```
Names = ['Luke', 'Leia', 'Chewbacca']
for name in Names:
    print name #Will print names in list
```

Try exploring the range() method in BPython, using parameters start, stop, and step for the method.
And then use it in a for-loop.

Also see if you can create a list of names and in a for-loop print different messages depending on what the name is.

zip()

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables.

The returned list is truncated in length to the length of the shortest argument sequence.

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
    [(1, 4), (2, 5), (3, 6)]
```

What if you have two different lists and want to iterate over them both at the same time?
Could you use the output of zip(x,y) in a for loop?

Exercise – v.3

Extend your runner.py to fetch the questions and answers from two lists (answers & questions) in a for-loop that calls sendline() and expect()

```
questions = ['Installation path \[/opt/esconfs\]:',
             'Full or light installation \[Full/Light\]:',
             'Deploy web server  \[Y/N\]:',
             'On which port \[80\]:',
             'Admin user account \[admin\]:',
             'Admin acount password \[\]:',
             'Backup previous data \[Y/N\]:',
             'Backup path \[~/esconfs_backup/\]:',
             'Install with these setting \[Y/N\]:']

answers = ['/tmp/esconfs',
             'Full',
             'Y',
             '80',
             'admin',
             'esconfs',
             'Y',
             '\n',
             'Y']
```

Exercise v.4

Structuring our code

Classes
Methods
Terminal Colors

Methods()

Methods are your way of creating logical groups of code that you want executed.

If you are executing the same lines of code multiple times, perhaps with different parameters, then consider creating a separate method for that.

Multiple parameters into a method are simply comma-separated.

Use `return` to hand back data from the method.

This

```
name = ”Adam”
print ”Hello ” + name
name = ”Bob”
print ”Hello ” + name
name = ”Cecilia”
print ”Hello ” + name
name = ”Donna”
print ”Hello ” + name
```

Could be this…

```
def Hello(name):
    print ”Hello ” + name

Hello(”Adam”)
Hello(”Bob”)
Hello(”Cecilia”)
Hello(”Donna”)
```

Or even this

```
def Hello(name):
    print ”Hello ” + name

names = [’Adam’, ’Bob’, ’Carl’, ’Donna’]

for name in names:
    Hello(name)
```

Try creating your own method that takes none, one, and several arguments as input.

Mix the input arguments as strings, numbers, lists etc.

Classes

Classes are your way of storing data and logic in one container.

Declaring a class is like specifying the blueprint.

Instantiating it into an object is like building the house from the blueprint.

Classes can therefore contain both variables and methods. The ”self” variable refers to the class-object itself.

```
class MyClass:

    def __init__(self, counter): #Constructor

        self.value = counter

    def otherMethod(self):

        print ”Initial value: ” + str(self.value)


if __name__ == ’__main__’:

    m = MyClass(3)

    m.otherMethod()
```

```
class OtherClass: #Here is the blueprint below

    counter = 0

    def testMethod(self):

        print "Hello"

    def otherMethod(self):

        self.counter += 1

        print "I have been called " +
str(self.counter) + " times"


if __name__ == '__main__':

    m = OtherClass() #Here we create the house

    m.testMethod()

    m.otherMethod()

    m.otherMethod()
```

Terminal Colors

When working in a terminal a nice feature for your scripts can be to color code the output depending on the context.

Simply print the ANSI escape sequences beginning with the code for the desired color and then the end sequence to reset to normal.

Working but not so pretty…

```
print '\033[92m' + "In green" + '\033[0m'
print '\033[94m' + "In blue" + '\033[0m'
```

The code above will work in Unix:es (Solaris, OS X , Linux) and windows (provided you enable ansi.sys)

To enable ansi.sys in Windows:

http://support.microsoft.com/kb/101875

Hint: a finished module/code snippet is available under ~/TTBA_Exercises/

Much better

```
class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'

print bcolors.OKGREEN + "In green" + bcolors.ENDC
print bcolors.OKBLUE + "In blue" + bcolors.ENDC
```

Exercise – v.4

Create a class (positiveflow.py) that contains the lists of questions and answers, and that also provides two methods for retrieving these lists.

- getQuestions()
- getAnswers()

Break away the code responsible for executing (sendline() & expect()) into a separate class responsible for that (class: executor.py).

Provide a method runTestCase that takes as input parameters:

- which cmd to use to start the installer
- A list of the questions
- A list of the answers

Fetch the lists in runner from positiveflow and the send them into executor.

Exercise v.5

Checking if the installer succeeded

os.path.join
os.path.isfile
os.path.isdir

os – path manipulation

Python runs on different systems and the scripts you write are cross-platform. But only as much as your paths in the code.

The os.path module contains methods for you to create code that runs on different OS:es where folders are separated by ”\” or ”/”.

```
import os.path

print os.path.join('one',
                   'two',
                   'three')
```

```
import os.path

base, end = os.path.split(’/one/two/three’)

print base
print end
```

Try the examples above and watch the output.

Then try the method os.path.splitext() on the string below and explain what it does:

’c:\\tmp\\myfile.txt’

os – verify file/folder

os.path provides two methods to check if a file or folder exists:

- os.path.isdir()
- os.path.isfile()

```
import os.path

result = os.path.isfile('/temp.log')
if result is True:
    print "File /temp.log exist."
else:
    print "File /temp.log does not exist."
```

```
import os.path

result = os.path.isdir('/tmp/')
if result is True:
    print "Folder /tmp/ exist."
else:
    print "Folder /tmp/ does not exist."
```

Try the examples above and watch the output.

Exercise – v.5

Create a class (verifier.py) that verifies if the installation passed or failed based on the existance of the following test artifacts:

Folder: {InstallationPath}

File: {InstallationPath}/success.log

In the __init__() method for the class set the installation path as a member variable.

Provide a method (verifyInstallation()) that checks if the artifacts above exists.

Northern Test Consulting

Exercise v.6

Breaking apart test code and test data

File read

File Open - Read

file_object = open(*name, 'o'*)

file_object.read()

file_object.readline()

```
f = open('test.txt', 'r')
entire_file = f.read()
f.close()
```

```
f = open('test.txt', 'r')
first_line = f.readline()
#If readline() returns empty string it means EOF
f.close()
```

```
f = open('test.log', 'r')
for line in f:
    print line
f.close()
```

The last example where a for loop is used to iterate over the file object. This is memory efficient, fast, and leads to simple code

Open the file: /opt/PFT/somefile

String strip()

A string variable contains a method called *strip()* which removes trailing whitespaces and newlines from strings.

Create a string below with a trailing \n and print it.

Use strip() and print the result of that operation.

File Write

file_object = open(*name, 'w|a' [, buffering]*)

file_object.write(string)

file_object.close()

Open a file for writing or appending data.
buffering: '0' means unbuffered, '1' line buffered, other numbers for buffered bytes.
'wb' for writing binary data.
write() takes a string of data as input to be written to the file object.

```
f = open('test.txt', 'w')
f.write('Data into file')
f.close()
```

```
f = open('test.bin', 'wb')
f.write(binary_data)
f.close()
```

```
f = open('test.log', 'w', 0)
f.write("Dark Side\nLight Side")
f.close()
```

Try opening a file and writing a string of text into it.
Write several times into a file, perhaps inside a for-loop…
Experiment with the modes 'w' and 'a', what's the difference`?

Exercise – v.6

Rename the class positiveflow.py into testcase.py and rename the class accordingly.

Next change the __init__ to take an argument for which testcase file to read from, and then in the constructor open the file and read every other line to a answers-list and a questions-list.

Bonus points if you add a check to see if the testcase file exists.

Leave getAnswers() and getQuestions() like they are.

Northern
Test Consulting

Exercise v.7

Working with 3rd party modules

Email capability

Mailgun class

In ~/TTBA_Exercises/ there is a module called mailgun.py that you should copy into your working folder.

Import the Mailgun module (*from mailgun import Mailgun*) and create a mailgun object.

The module has a method called sendEmail() that takes four input parameters (strings):

-sender: *The mail address of the sender*

-to: *The mail address of the recipient*

-subject: *The subject of the mail*

-message: *The message included in the email*

```
from mailgun import Mailgun

mg = Mailgun()
mg.sendEmail('my@address.com',
             'your@address.com,
             'Fancy title',
             'Staggering content')
```

Exercise – v.7

Use the mailgun module to extend your verifier class so that it sends an email if the test fails or passes.

Northern Test Consulting

Exercise v.8

Gather test artifacts

Os
Shutil
Zipfiles
Timestamps

OS

This module provides a portable way of using operating system dependent functionality.

If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module.

```
import os


print os.environ

os.putenv('Test', 'Value')

print os.environ
```

```
import os


print os.getcwd()

os.chdir('/tmp/')

print os.getcwd()

os.makedirs('my_folder')

print "Content of /tmp/:"

for item in os.listdir('.'):

    print item
```

Also investigate the commands:

- os.getlogin()
- os.chmod()
- os.chown()

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

shutil

The shutil module offers a number of high-level operations on files and collections of files.

In particular, functions are provided which support file copying and removal.

```
import shutil
shutil.copy('src_file.txt', '/tmp/destination_file.txt')
```

```
import shutil
shutil.move('/tmp/src_folder/', '/tmp/dest_folder/')
```

Try using the method shutil.rmtree(path) to delete the folder '/tmp/dest_folder/'.

DateTime

A common need is to get the current date and time in order to create a timestamp.

The DateTime module provides such a tool.

You can also use the method strftime() to format your timestamp.

```
import datetime


N = datetime.datetime.now()

print N

print N.hour

print N.minute

print N.second
```

```
from datetime import datetime


N = datetime.now()

Timestamp =  N.strftime('%Y%m%d-%H%M%S')

print Timestamp
```

Experiment using hours, minutes, or something else in timedelta()

Also have a look here for strftime parameters:
http://linux.die.net/man/3/strftime

```
import datetime


N = datetime.datetime.now()

Diff = datetime.timedelta(days=3)

print N+Diff
```

ZipFile –Zip archives

Python offers the support to create zip archives.

Limitations:
The zipfile module does not support ZIP files with appended comments, or multi-disk ZIP files.
It does support ZIP files larger than 4 GB that use the ZIP64 extensions.

```
import zipfile

zipf = zipfile.ZipFile('myfile.zip', 'w', zipfile.ZIP_DEFLATED)
zipf.write('file.txt')
zipf.close()
```

```
import zipfile

zipf = zipfile.ZipFile('myfile.zip', 'a', zipfile.ZIP_DEFLATED)
zipf.write('fileTwo.txt')
zipf.close()
```

Create the files file.txt and fileTwo.txt and then try the code examples above.

Next from the terminal run: unzip

Exercise – v.8

Extend the class verifier with a method *gatherTestArtifacts()* that creates a temporary folder, and copies the artifacts of an installation (success.log or fail.log) into the temporary file into the folder.

Then it should create a zipfile archive with the timestamp of the following format:

Testartifact_20150603-140213.zip

Next use the mailgun method send *sendEmailWithAttachement()* in the mailgun module instead and attach the zipfile created to the email.

This method takes the same four input parameters as before but also a fifth method specifying the path to the file add as an attachement.

Northern Test Consulting

Exercise v.9

Verify REST API

Requests

Install Python Packages – pip

Pip is a package management system used to install and manage software packages written in Python.

Many packages can be found in the Python Package Index (PyPI).

The Python Package Index or PyPI is the official third-party software repository for the Python programming language.

Python developers intend it to be a comprehensive catalog of all open source Python packages.

The strength of pip is that it is very easy to use.

```
pip install <package-name>
```

Install the following package using pip:

- requests

JSON – JavaScript Object Notation

JSON is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML.

Python contains excellent support for serializing data objects to and from JSON.

```
import json

badGuy = {"name":"Sauron", "Occupation":"Necromancer",
"Address": "First tower on the left in Dol Guldur",
"Interests": ["Hunting hobbits", "Polishing rings"],
"Rings": 0}

print badGuy
print json.dumps(badGuy)
print json.dumps(badGuy, sort_keys=True, indent=4)

f = open('storage.json', 'w')
f.write(json.dumps(badGuy, sort_keys=True, indent=4))
f.close()
```

```
import json

f = open('storage.json', 'rb')
villain = json.loads(f.read())

print villain
```

Play around with the two exercises above in order to learn how to deal with data object, seralizing to json and back.
Use other data types as well to see how they are represented:
- Booleans
- Integers
- Floats

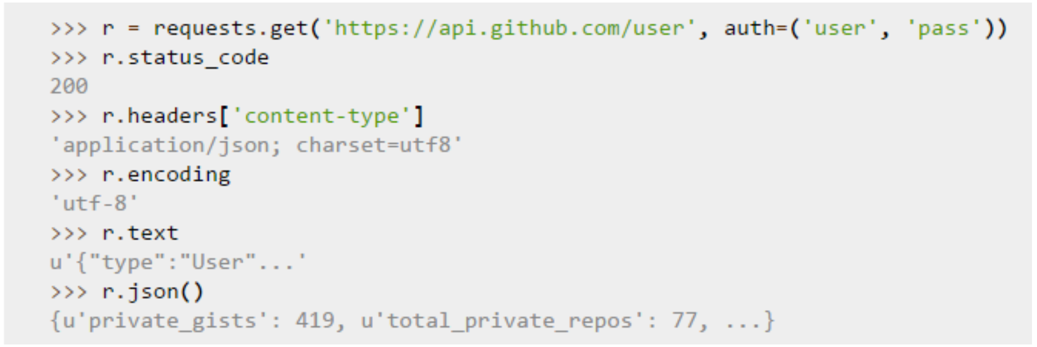
Requests: HTTP for Humans

" Requests is an Apache2 Licensed HTTP library, written in Python, for human beings.

Python's standard urllib2 module provides most of the HTTP capabilities you need, but the API is thoroughly broken. It was built for a different time — and a different web.
It requires an enormous amount of work (even method overrides) to perform the simplest of tasks.

Things shouldn't be this way. Not in Python."

http://docs.python-requests.org/



```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Try using requests.get() towards the URLs below and inspect the response objects:

- http://localhost/pft
- http://localhost/despicableme

Requests – For REST API:s

Calling REST API:s using requests and Python couldn't be easier.

```
>>> r = requests.get('https://api.github.com/events')
>>> r = requests.post("http://httpbin.org/post")
>>> r = requests.put("http://httpbin.org/put")
>>> r = requests.delete("http://httpbin.org/delete")
```

Including parameters.

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get("http://httpbin.org/get", params=payload)
>>> print(r.url)
http://httpbin.org/get?key2=value2&key1=value1
```

Try calling these two REST API:s including a JSON-payload containing the string "name".
Hint: {"name": "Kristoffer"}

[GET] http://localhost/sayhello
[POST] http://localhost/createentry

After the POST request, look under the folder "/opt/PFT/rest/

Try/Except

If an error occurs in your code then an exceptions going to be *thrown* from the code and the execution is going to be halted.

Standard practice is to surround such blocks of code with a try/except statement in order for our code to have a chance to catch the exception being thrown in order to decide what to do with it.

It is also possible to catch specific and even several exceptions

```
try:
    ...
    {Your code that can throw exceptions}
    ...
except ValueError:
    print "Oops!  That was no valid number.  Try again..."
except (RuntimeError, TypeError, NameError):
    pass
except:
    print ”An unspecified error occured”
```

Exercise – v.9

Extend the class verifier and the method *verifyInstallation()* so that it makes a call to the REST API that the installer starts.

The REST API is located at the following address as a simple GET:

http://localhost:3369/ttba

You will need to surround the REST call with a try/except block to catch if the REST API isn't running.

Northern Test Consulting

Exercise v.10

Command line
input

optparse

Optparse – Command line input

Often you want your python script to support command line arguments as inputs in order to make them more reusable and flexible.

While there are a lot of interesting 3rd part modules, optparse is the de facto choice from the stdlib shipped with Python.

```
import optparse

par = optparse.OptionParser()

par.add_option("-n",
               "--noarg",
               action="store_true",
               default=False)

par.add_option("-w",
               "--withargs",
               action="store",
               help="Sets withargs to value provided",
               dest="withargs")

options, args = par.parse_args()

print options.noarg
print options.withargs
```

```
import optparse

par = optparse.OptionParser()

par.add_option("--withargs2",
               action="store",
               help='Sets withargs2 to value provided, 3 by default',
               default=3,
               type="int")

par.add_option("-c",
               "--choice",
               help='Select a valid value: "First, Second, Third"',
               type="choice",
               choices=['First', 'Second', 'Third'])

options, args = par.parse_args()

print options.choice
print options.withargs2
```

Try updating the data digger exercise to support different files as input arguments.
Add the keywords as options as well.

Exercise – v.10

Add command line input to your runner class.
It should accept the input of which testcase file to run:
Suggested input format:

```
Usage: runner.py [options]

Options:
  -h, --help            show this help message and exit
  -t TESTNAME, --testcase=TESTNAME Specify the name of the testcase file to run [Default: positiveflow.txt]
```

Ftplib – FTP Sessions

In some instances you want to interact with an FTP server for a reason or other using Python.

The module ftplib offers such support.

storbinary() and retrbinary() is used to upload/download, just open a file object in binary read/write mode and supply.

cwd() changes the current working directory on the FTP server.

```
import ftplib

session = ftplib.FTP('127.0.0.1', ’user', ’password')
session.cwd('/tmp/')
trfile = open(filepath, 'rb')
session.storbinary('STOR uploadfile.txt’, trfile)
```

```
import ftplib

session = ftplib.FTP('127.0.0.1', ’user', ’password')
session.cwd('/tmp/')
trfile = open('downloadfile.txt', 'wb')
session.retrbinary('RETR downloadfile.txt’, trfile.write)
```

Create the files (downloadfile.txt and uploadfile.txt) in /tmp and try the examples above.

Exercise - Gather Test Artifacts

Write a simple script that copies the two test artifacts (see below) into '/tmp/testartifacts/':

Files: /opt/PFT/firstLarge, /opt/PFT/secondLarge

Then creates a zip archive with a filename formated with a timestamp like this:

'testartifacts_20141103_153000.zip'

After that the script should connect to the FTP service (on localhost) with credentials:

Username: tester

Password: python

And uploads the created zip archive into:

Location: /upload

```
import os, shutil, ftplib
from zipfile import ZipFile
from datetime import datetime

#Check if script is invoked from command line:

    #Create a folder to copy the artifacts into
    #Copy the files

    #Create a filename with the timestamp
    #And extend the path to include your folder

    #Open a zipfile for writing
    #Write files into it. Dont forget to close

    #Open an ftp session and upload the file
```

Time

Perhaps the most interesting method available in the time module for a tester is sleep().

time.sleep() pauses the execution of your program for the amount of seconds specified.

```
import time

#Sleep for ten seconds
time.sleep(10)
```

```
import time

#Sleep for one ten milliseconds
time.sleep(0.01)
```

Try writing a for loop that iterates 100 times, prints something to the screen, followed by a 100 millisecond sleep inside the for-loop

While-loop

Another type of control flow is the while loop.

It will ***do something while a condition is true***.

```
while processStuff():
    print "Method still has stuff to process"
```

```
counter = 0
while counter <= 100:
    print "Counter is: " + str(counter)
    counter += 1

    if counter <= 50:
        print "Counter equal to or less than fifty"
    elif counter > 50 and counter < 100:
        print "Counter more than fifty, less than one hundred"
    else:
        print "Counter one hundred, exiting while loop"
```

Make a while loop that compares a text string as its condition.

Exercise - Data Digger

Write a simple script that opens two different files and reads from them in a continous loop.

If a specific keyword is encountered in the specific file then that line should be printed to the console.

Use different color coding for lines found in the different files.

Also start "generator.py" in separate terminal.

First file: /opt/PFT/DataDigger/first.log

Second file: /opt/PFT/DataDigger/second.log

```
#import datetime here
import time, random

class bcolors:
    #Fill in the rest here
class reader:
    def __init__(self, filename, color, keyword):
        self.filename = filename
        #Open the file here for reading
        #Also store the keyword & color here

    #define a method here called getLineFromFile()
        #read a line in the file
        #check if line is not empty & contains keyword
            #print the line found with color coding

if __name__ == '__main__':
    #create a 'reader' object for the first file
    #create a 'reader' object for the second file
    while True:
        #readLine for the first file here
        time.sleep(0.1)
        #readLine for the second file here
        time.sleep(0.1)
```

Northern
Test Consulting

Exercise Data Generation

CSV Files

Tar Files

os.walkdir()

CSV - Comma-separated values

A very common data format you will come across is CSV, a file containing comma-separated values. Python offers the csv module for you to work with such data.

You can create a CSV-reader and/or a CSV-writer depending on your needs.

```
import csv
csvfile = open('file.csv', 'wb')
Writer = csv.writer(csvfile, delimiter=',')
Writer.writerow(['Adam', 'Bob', 'Cecilia', 'Donna'])
Writer.writerow(range(0,10))
Writer.writerow(['Testers'] * 5)
csvfile.close()
```

```
import csv
csvfile = open(’file.csv’, ’rb’)
Reader = csv.reader(csvfile, delimiter=’,’)
for row in Reader:
    print row
csvfile.close()
```

Save the examples above as python files and first run the writer example, look into the generated file.

Then run the reader example.

TAR – Tape Archive

In the unix world a lot of data is stored in so-called ”TAR-files”. Python offers a convenient method of extracting files and folders stored in tar archive.

You need to open the TAR-file and work with the resulting TAR-object when extracting or writing a new TAR archive.

```
import tarfile

tar = tarfile.open('myCollection.tar', 'w')
for name in ['firstFile.log', 'secondFile.log', 'folder']:
    tar.add(name)
tar.close()
```

Or…

```
import tarfile, os

tar = tarfile.open('myCollection.tar', 'w')
for name in os.listdir('.'):
    tar.add(name)
tar.close()
```

```
import tarfile

tar = tarfile.open('myCollection.tar')
tar.extractall()
tar.close()
```

Create a temp-folder and in there create the files and folder in the example above.
Hint: use command touch.

Then run the write-example, verify a TAR-file has been created.
Remove the files and folder and run the extractAll() example.

os – walk() directory

os.walk() is a way to traverse a folder and its files recursively.

os.walk(*top*, *topdown*=*True*, *onerror*=*None*, *followlinks*=*False*)

```
import os

for root, dirs, files in os.walk('/opt/PFT/os_walk/'):
    print 'In folder: ' + root
    print 'Contains files: ' + str(files)
    print 'And directories: ' + str(dirs)
    print "------\n"
```

os.walk() is a generator that yields the file names in a directory tree by walking the tree.
Try the example above and think about the objects returned by os.walk()

Exercise – Data Generation

Write a simple tool that creates a structure of 5 folders and within those folder another set of 5 folders.

In each of the outermost folders there should be a file called 'data.txt' that has the format:

Firstname Lastname

State

Also the folder should contain a jpg picture with a picture of a person.

In the exercise folder you will find CSV files containing names and states in the US.

There is also a tar/gzip:ed archive containing faces of different persons.

```
/tmp/testdata/
    0/
        0/
            data.txt
            face.jpg
        1/
            data.txt
            face.jpg
        2/
        3/
        4/
    1/
        0/
        1/
        2/
        3/
        4/
    2/
    3/
    4/
```

Outline

```
import os, shutil, csv, tarfile, random

def addTestDataToFolder(folder, names, states, files):
    #Retrieve a random first and last name from names, and state from states
    #Write the data into data.txt in folder
    #Find a random picture from files
    #Copy the picture into the data folder

def getRowsFromCSVAsList(filename):
    #Open CSV, in a for loop iterate over all items, add all items into list
    return generated_list

If __name__ == '__main__':
    getRowsFromCSVAsList('names.csv')
    #The same but for states.csv
    #Extract the tar archive, use os.walk() to add all file-paths into a list (files)
    #Create folder (/tmp/testdata)
    for firstLvlFolder in range(0,5):
        #Create "folder/firstLvlFolder" - use os.makedirs & os.path.join
        for secondLvlFolder in range(0,5):
            #Create second level folder
            addTestDataToFolder(destFolder, names, states, files)
```

Northern Test Consulting

Command line input

optparse

Exercise HTTP + REST + Monitor System

pip

JSON

Requests

psutil

Psutil – Monitor your system

"psutil (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network) in Python.

It is useful mainly for system monitoring, profiling and limiting process resources and management of running processes. It implements many functionalities offered by command line tools such as: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap.

It currently supports Linux, Windows, OSX, FreeBSD and Sun Solaris, both 32-bit and 64-bit architectures, with Python versions from 2.4 to 3.4. PyPy is also known to work."

https://github.com/giampaolo/psutil

Experiment with psutil and explore some of the following functions.

- `psutil.cpu_percent(interval=1)`
- `psutil.cpu_count()`
- `psutil.virtual_memory()`
- `psutil.swap_memory()`
- `psutil.disk_partitions()`
- `psutil.disk_usage('/')`
- `psutil.disk_io_counters(perdisk=False)`
- `psutil.net_io_counters(pernic=True)`
- `psutil.net_connections()`
- `psutil.users()`
- `psutil.boot_time()`

- `psutil.pids()`

- `p = psutil.Process(<PID_Number>)`
- `p.name()`
- `p.exe()`
- `p.cwd()`
- `p.cmdline()`
- `p.status()`
- `p.username()`
- `p.create_time()`

Exercise - HTTP + REST + Monitor System

Write two scripts. The first script should in a loop continously call the REST API:s (be nice and sleep a bit in between calls…)

- *[GET] http://localhost/eatcpu*

  No JSON parameters necessary
- *[GET] http://localhost/hogmemory*

  JSON Parameter: {'MemoryAmount':[Bytes]}

The second script should monitor your system for its CPU and memory usage over a 30 seconds period and take a sample every 2 seconds.

Store the samples and create a simple HTML report afterwards in report.html (Hint: Just open a file and write HTML strings into it)

What info could be of interest in the report?

Run the two scripts in parallell, one to load the system, and the other to monitor the system.

Python and SOAP requests

"Simple Object Access Protocol"

Python and SOAP requests?

"SOAP, originally an acronym for Simple Object Access protocol, is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on otherapplication layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission."

http://en.wikipedia.org/wiki/SOAP

With the use of a third party library Python can handle Soap requests with ease.

There are several third party modules availble. At the writing time 'PySimpleSoap' seems to be the easiest and most stable.

```
# sudo pip install PySimpleSoap


from pysimplesoap.client import SoapClient

client = SoapClient(wsdl=' http://www.webservicex.net/country.asmx?WSDL', trace=false)


client.help('GetCurrencyByCountry') #Strictly not needed if you know how APIs work

client.GetCurrencyByCountry('Sweden')
```

Thanks for your time

If you ever have any Python questions after taking Python For Testers you will always be welcome to contact Kristoffer and he will gladly help you.

Email: *kristoffer.nordstrom@northerntest.se*

Twitter: *@kristoffer_nord*

Homepage: *http://www.northerntest.se/*