Team 6: Final Report

MH Johnson
Henry Noonan

Table of Contents

Chapter 1: Selection and First Impressions …......... pg 3

Chapter 2: Test Plan …... pg 4

Chapter 3: Testing Framework …........................... pg 7

Chapter 4: Test Suite …... pg 11

Chapter 5: Fault Injection ….................................. pg 18

Chapter 6: Reflections …....................................... pg 19

Chapter 7: How To..pg 20

Chapter 1: Selection and first impressions

We have successfully downloaded and built the Jython codebase on our Ubuntu distro inside of Virtual Box, and we ran all of the existing tests, all of which passed. The process mostly entailed downloading the tools necessary to develop with Jython. This meant downloading things from the bash command line such as the Java Virtual Machine and Development Kit, Mercurial (the version control system of choice for Jython), and Ant (the compiling tool). Once the development environment was successfully set up, downloading and building Jython was as simple as running a few commands. The Jython development guide was incredibly helpful and clear about how to go about downloading and building the Jython codebase. It was also very clear about how to run the existing tests. The tests took about 10-15 minutes to finish running, but they constantly updated inside of our bash terminal, and notified us when all the tests had passed. Overall this was a positive experience thanks to the incredibly helpful Jython development documentation.

Chapter 2: Test Plan

Introduction:

This plan aims to develop and run a series of 25 unit test cases in total on the current, in-development, version of the Jython code base. In developing these tests, we will make use of the functionality of the Jython Bitbucket for ease of browsing the code as well as the JythonDeveloperGuide for clarifications.

The existing Jython code base consists of numerous classes, well beyond the scope of what 25 test cases would cover, so we opt instead to focus on smaller groupings and specific functionalities.

TestItems:

As stated in the introduction, the scope of our tests will be limited to a handful of specific libraries and the functionalities that they represent. At the time of writing, the aim of our test cases revolves around testing the accuracy and robustness of various String and text based functionalities. Beginning with the operations in the Jython String.py class, and subsequently moving on to the URL parsing functionalities of urlparse.py.

Risks:

Given the scope of the Jython project, and the amount of documentation that exists for it and its functions the number of risks posed to our testing should be minimal, although a few potential considerations are as follows:

1. Complex functions – in the case of a few of the libraries in the Jython project, the structure in very complex in terms of the ways in which the various classes will be interacting with each other, we will attempt to minimize this risk by limiting tests to

libraries and functions that are well documented and/or of a simpler architecture

2. Updates to software being tested – Since we are creating test cases based on an in-development version of Jython, it is possible, though unlikely, that the addition of some new functions may interfere with the existing functions that we will be testing.
3. Python – As Jython makes numerous calls to the Python system and objects, it is possible, though unlikely, that some python calls made in functions we test may be changed or obfuscated.

Strategy:

The individual test cases will be written in Python and uploaded to the Team6 GitHub repository, and automated by yet another Python script which will execute all of them in sequence. The outputs from these test cases will be checked against Oracle files, which we will also be stored in the repository.

A test will be considered as passed if it runs to completion and results in an output identical to that which is expected by the oracle

Environment:

These tests will be executed in a Virtual Ubuntu environment, using Java 1.7 and the associated JDK. As such, while these test should reflect the functionality of Jython under any supported environment that cannot be guaranteed.

Notes:

User Testing in Our Own Environment:

1. Run manual tests and compare the actual results with expected results.

2. Run automated tests that will run every time the system is tested.
3. These test will be done in three stages over the next 3 weeks: Unit Testing, Component Testing and Systems Testing.

Notes On Process:

- Challenges:
- Navigating massive code base was difficult
- Accidentally deleted existing github repo files when MH pushed the project up to github
- dealing with file paths, often a script in one directory would have to modify files in a parent directory or distant directory.
- figuring out how to execute a python method in a different directory when the name of the method or file isn't known until runtime

Possible functionality to test:

- jython / lib-python / 2.7 / string.py
    - A library for performing basic operations on Strings
    - Relies heavily on standard python string libraries, is important to test since python could change the behavior of standard string methods and break existing jython code.
- jython / lib-python / 2.7 / urlparse.py
    - Somewhat more complex than the string library, performs basic operations on a url such as parsing out the parameters in a query.

jython / lib-python / 2.7 / decimal.py

- The purpose of this module is to support arithmetic using familiar
  "schoolhouse" rules and to avoid some of the tricky representation
  issues associated with binary floating point

jython / lib-python / 2.7 / pprint.py

- The purpose of this module is to support to pretty-print lists, tuples, & dictionaries recursively.

jython/ lib-python / 2.7 / random.py

Chapter 3: Testing Framework

Experience:

The experience for this deliverable was much nicer than the previous, since, misunderstanding what was needed for the previous deliverable, writing the script for executing test cases is something that we had already taken care of. Needless to say this was unpleasant when deliverable 2 was rolling around, but has since made the project outlook very good, with a testing framework which will easily accommodate additional test cases as the project continues

Architecture:

The script clears the temp folder, creates, and subsequently iterates through a list of all text files in the testCases folder. These test cases are written in a specific format to facilitate parsing by the script as follows:

Line1: Test ID #

Line2: Description of the test

Line3: Path to the .py file the tested function resides in

Line4: Function being tested

Line5: Arguments

Line6: Expected output

After parsing, the script makes the method call along with the given argument and compares the results to the expected output given in the test case for each test in the testCases folder. The output of all of these tests is written to an html file.

Script as of Deliverable 3:

```
    #!/usr/bin/python

import subprocess
import os.path
import sys
from importlib import import_module

def getTestCases():
    os.chdir("../testCases/")
    allFiles = os.listdir(".")
    return [TestCase(fileName) for fileName in allFiles if ".txt" in
fileName]

def executeTest(testCase):
    importStatement = convertPathToImport(testCase.modulePath)
    sys.path.append(testCase.modulePath)
    module = __import__(importStatement, fromlist=[testCase.methodName])
    methodToTest = getattr(module, testCase.methodName.replace("()",""))
    result = methodToTest(testCase.inputValue)
    testCase.actualResult = result

    #actual test
    if result == testCase.expectedResult:
        testCase.testPassed = True
    return testCase

def convertPathToImport(path):
    path = path.replace("/", ".")
    path = path.replace(".py","")
    return path.split(".")[-1]

class TestCase:
    def __init__(self, fileName):
        file = open(fileName, 'r')
        fileContents = file.read().split("\n")
        self.id = fileContents[0]
        self.description = fileContents[1]
        self.modulePath = fileContents[2].strip()
        self.methodName = fileContents[3].strip()
        self.inputValue = fileContents[4]
        self.expectedResult = fileContents[5]

        self.actualResult = ""
        self.testPassed = False

        file.close()


def clearTempFolder():
    #go back a directory, remove everyting from temp,
    #if nothing is in temp, the warning is suppressed by sending it to null
    subprocess.call("rm ../temp/* 2>/dev/null", shell=True)
```

```
def generateHtml(testResults):
    html = "<html>"
    html += "<body>"
    html += "<h1>Test Results</h1>"
    html += testResults
    html += "</body>"
    return html + "</html>"

def writeHtmlFile(html):
    os.chdir("..")
    os.chdir("./reports/")
    filename = "test_results.html"
    output = open(filename,'w')
    output.write(html)
    subprocess.call("xdg-open " + filename, shell=True)

def main():
    clearTempFolder()
    testCases = getTestCases()
    outputString = ""
    for testCase in testCases:
        executeTest(testCase)
        if testCase.testPassed:
            outputString += '<p style = "color:green">'
            outputString += "Test case "+testCase.id+" passed!"
            outputString += '</p>'
        else:
            outputString += '<p style = "color:red">'
            outputString += "Test case "+testCase.id+" FAILED!"
            outputString += '</p>'
        outputString += "<ul>"
        outputString += "<li>\tInput: " + testCase.inputValue + "</li>"
        outputString += "<li>\tExpected Result: " + testCase.expectedResult
+ "</li>"
        outputString += "<li>\tActual Output: " +
str(testCase.actualResult) + "</li>"
        outputString += "</ul>"
    htmlBody = generateHtml(outputString)
    writeHtmlFile(htmlBody)


main()
```

Test Cases as of Deliverable 3:

1
this will test the upper method, converting all lower case letters to upper case
/lib-python/2.7/string.py
upper()
caT
CAT

2

this will test the strip method, taking a string and removing any trailing or leading whitespace
/lib-python/2.7/string.py
strip()
      this is a test for trim
this is a test for trim

3
this will test the lower method
/lib-python/2.7/string.py
lower()
CAT
cat

4
this will test the swapcase method changing lower case letters to upper case and upper case letters to lower case
/lib-python/2.7/string.py
swapcase()
teSTING fOr Cases
TEsting FoR cASES

5
this method will test the capwords method, which takes in a string and returns that string with all of the words capitalized
/lib-python/2.7/string.py
capwords()
this iS a tEst.
This Is A Test.

Chapter 4: Testing Suite

Experience:

The primary complication in finishing the test case suite was realizing the limitations of our existing testing framework, which we had initially assumed would be sufficient to easily implement the rest of the test cases. As it was however the framework could only take in arguments as strings, which was rarely a problem in testing Jython's string and url parsing functionalities, but needed to be expanded upon in order to work with numerical inputs as well. As such the formatting for our test cases was expanded upon to include “” to designate strings, as well as an escape character ($,) to differentiate commas used in strings from commas used to separate multiple arguments to a function. It was also noted that there is some inconvenience in testing functions which return lists and tuples, in that the expected output line of the test case must be input as a string precisely as python outputs the results to a string rather than iterating through the resultant list.

Nevertheless, in addition to implementing the additional test cases, the deliverable left us necessarily producing a more fleshed out and useful testing framework, shown below.

Script as of Deliverable 4:

```
    #!/usr/bin/python

from sys import platform as _platform
import subprocess
import os.path
import sys
from importlib import import_module

def getTestCases():
    os.chdir("../testCases/")
    allFiles = os.listdir(".")
    return [TestCase(fileName) for fileName in allFiles if ".txt" in
fileName and fileName[-1] != '~']

def executeTest(testCase):
    importStatement = convertPathToImport(testCase.modulePath)
    sys.path.append(testCase.modulePath)
    module = __import__(importStatement, fromlist=[testCase.methodName])
    #Just gets method name itself, not parameters or other signature
    methodNameTrimmed =
testCase.methodName[0:testCase.methodName.find('(')]
    methodToTest = getattr(module, methodNameTrimmed)
    result = methodToTest(*testCase.inputValue)
    testCase.actualResult = result

    #actual test
    if str(result) == testCase.expectedResult:
        testCase.testPassed = True
    return testCase

def convertPathToImport(path):
    path = path.replace("/", ".")
    path = path.replace(".py","")
    return path.split(".")[-1]

class TestCase:
    def __init__(self, fileName):
        file = open(fileName, 'r')
        fileContents = file.read().split("\n")
        self.id = fileContents[0]
        self.description = fileContents[1]
        self.modulePath = fileContents[2].strip()
        self.methodName = fileContents[3].strip()
        self.inputValue = fileContents[4].split('$,')
        #go through parameters and pluck out strings vs numbers
        for i in range(len(self.inputValue)):
            if('"' in self.inputValue[i]):
                self.inputValue[i] = self.inputValue[i][1:-1]
            else: #else it's a number
                self.inputValue[i] = eval(self.inputValue[i])
        self.expectedResult = fileContents[5]

        self.actualResult = ""
        self.testPassed = False

        file.close()


def clearTempFolder():
    #go back a directory, remove everyting from temp,
    #if nothing is in temp, the warning is suppressed by sending it to null
    subprocess.call("rm ../temp/* 2>/dev/null", shell=True)

def generateHtml(testResults):
    html = "<html>"
    html += "<body>"
    html += "<h1>Test Results</h1>"
    html += testResults
    html += "</body>"
    return html + "</html>"

def writeHtmlFile(html):
    os.chdir("..")
    os.chdir("./reports/")
    filename = "test_results.html"
    output = open(filename,'w')
    output.write(html)
    if _platform == "linux" or _platform == "linux2":
        # linux:
        subprocess.call("xdg-open " + filename, shell=True)
    elif _platform == "darwin":
        # OS X:
        subprocess.call("open " + filename, shell=True)


def main():
    clearTempFolder()
    testCases = getTestCases()
    testCases.sort(key= lambda x: eval(x.id))
    outputString = ""
    for testCase in testCases:
        executeTest(testCase)
        if testCase.testPassed:
            outputString += '<p style = "color:green">'
            outputString += "Test case "+testCase.id+" passed!"
            outputString += '</p>'
        else:
            outputString += '<p style = "color:red">'
            outputString += "Test case "+testCase.id+" FAILED!"
            outputString += '</p>'
        outputString += "<ul>"
        outputString += "<li>\tInputs: " + str(testCase.inputValue) +
"</li>"
        outputString += "<li>\tExpected Result: " + testCase.expectedResult
+ "</li>"
        outputString += "<li>\tActual Output: " +
str(testCase.actualResult) + "</li>"
        outputString += "</ul>"
    htmlBody = generateHtml(outputString)
    writeHtmlFile(htmlBody)


main()
```

Test Cases:

1
The upper method shall take in a string and return that same string entirely in upper case letters
/lib-python/2.7/string.py
upper(s)

"caT"
CAT

2
The strip method shall take in a string and return that same string without leading or trailing whitespaces
/lib-python/2.7/string.py
strip(s)
"   this is a test for trim   "
this is a test for trim

3
The lower method shall take in a string and return that same string entirely in lower case letters
/lib-python/2.7/string.py
lower(s)
"CAT"
cat

4
The swapcase method shall take in a string and return the same string with lower case letters capitalized and visa versa
/lib-python/2.7/string.py
swapcase(s)
"teSTING fOr Cases"
TEsting FoR cASES

5
The capwors method shall take in a string, and return that string with all of the words capitalized
/lib-python/2.7/string.py
capwords(s)
"this iS a tEst."
This Is A Test.

6
The unquote method shall decode an encoded parameter of a url, replacing all %20 with spaces
/lib-python/2.7/urlparse.py
unquote(s)
"abc%20def"
abc def

7
The replace method takes in a string, and replaces all instances of a given substring with instances of a second given substring
/lib-python/2.7/string.py
replace(s, old, new)
"here is a test"$,"test"$,"TEST"
here is a TEST

8
The replace method takes in a string, and replaces all instances of a given substring with instances of a second given substring
/lib-python/2.7/string.py
replace(s, old, new)
"she sells seashells by the seashore"$,"she"$,""
 sells sealls by the seashore

9

Return the lowest index in s where substring sub is such that sub is contained within s[start,end]. arguments start and end are interpreted as in slice notation.
/lib-python/2.7/string.py
find(s, *args)
"looking for the first instance of the letter g"$,"g"
6

10
The count method takes in a string and returns a sum of all instances of a given substring occuring within that string
/lib-python/2.7/string.py
count(s, *args)
"example string with a good   number of characters in it"$," "
12

11
Return the lowest index in s where substring sub is such that sub is contained within s[start,end]. arguments start and end are interpreted as in slice notation.
/lib-python/2.7/string.py
find(s, *args)
"the letter after p will not be found in this string"$,"q"
-1

12
The gcd method takes two integers returns the greatest common denominator of the two integers given, for use by Jython's fraction functions
/lib-python/2.7/fractions.py
gcd(a,b)
12$,18
6

13
Return a right-justified version of s, in a field of specified width, padded with spaces as needed. The string is never truncated. If specified the fillchar is used instead of spaces.
/lib-python/2.7/string.py
rjust(s, width, *args)
"test"$,7
```
   test
```

14
The atoi method takes in a string representing an integer in a given base and returns the integer in base 10
/lib-python/2.7/string.py
atoi(s,base=10)
"00101100"$,2
44

15
The atoi method takes in a string representing an integer in a given base and returns the integer in base 10
/lib-python/2.7/string.py
atoi(s,base=10)
"1BC"$,16
444

16
The expandtabs method, given two arguments, will change the number of whitespaces that \t expands to when processing the given text
/lib-python/2.7/string.py
expandtabs(s, tabsize=8)
"hello   world"$,1

hello world

17
Return a left-justified version of s, in a field of specified width, padded with spaces as needed. The string is never truncated. If specified the fillchar is used instead of spaces.
/lib-python/2.7/string.py
ljust(s, width, *args)
"test"$,7
test

18
Pad a numeric string x with zeros on the left, to fill a field of the specified width. The string x is never truncated.
/lib-python/2.7/string.py
zfill(x, width)
123$,8
00000123

19
Return a center version of s, in a field of the specified width. padded with spaces as needed. The string is never truncated. If specified the fillchar is used instead of spaces.
/lib-python/2.7/string.py
center(s, width, *args)
"test"$,10
   test

20
The wrap method takes in a string and wraps it such that it fits in lined of no more than the given width, these lines are returned as elements of a list
/lib-python/2.7/textwrap.py
wrap(text, width=70, **kwargs)
"hello world and all who inhabit it"$,8
['hello', 'world', 'and all', 'who', 'inhabit', 'it']

21
The unquote method shall decode an encoded parameter of a url, replacing all %20 with spaces
/lib-python/2.7/urlparse.py
unquote(s)
"abc def"
abc def

22
The wrap method takes in a string and wraps it such that it fits in lined of no more than the given width, these lines are returned as elements of a list
/lib-python/2.7/textwrap.py
wrap(text, width=70, **kwargs)
"hello world and all who inhabit it"$,15
['hello world and', 'all who inhabit', 'it']

23
Pad a numeric string x with zeros on the left, to fill a field of the specified width. The string x is never truncated.
/lib-python/2.7/string.py
zfill(x, width)
123$,0
123

24
Return the highest index in s where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
/lib-python/2.7/string.py
rfind(s, *args)
"looking for the last instance of the letter g"$,"g"
44

25
The replace method takes in a string, and replaces all instances of a given substring with instances of a second given substring
/lib-python/2.7/string.py
replace(s, old, new)
"how will   it handle   a   t on of useless space   ?"$," "$,"_"
how_will___it_handle___a___t_on__of_useless_space____?

Chapter 5: Fault Injection

The goal of this deliverable was to inject faults into our local copy of the code to be tested. The hurdle for us was the realization that some of the functions we were testing happened to be named the same thing as python built in functions, so our script was actually executing the built in functions, rather than our local copy of jython's version of those functions. To fix this, we added a more strict path definition to the requirements of our framework. In the future, we hope to allow the user to specify the location of their copy of jython, but as of right now, our framework assumes that the copy of jython being tested is in the following path: /users/<username>/jython

The following faults were injected into our copy of jython to cause our tests to fail.

1) Test case #5 file: jython/lib-python/2.7/string.py Line number: 26 Changed From: return (sep or ' ').join(x.capitalize() for x in s.split(sep)) Changed To: return (sep or ' ').join(x.lower() for x in s.split(sep))

2) Test cases #6, #21 file: jython/lib-python/2.7/urlparse.py Line number: 325 Changed From: res = s.split('%20') Changed To: res = s.split(' ')

3) Test case #12 file: jython/lib-python/2.7/fractions.py Line number: 25 Changed From: a, b = b, a%b Changed To: a, b = b, a//b

4) Test cases #18 file: jython/lib-python/2.7/string.py Line number: 467 Changed From: return x.zfill(width) Changed To: return x.zfill(width * 2)

5) Test cases #3 file: jython/lib-python/2.7/string.py Line number: 226 Changed From: return s.lower() Changed To: return s.upper()

As a result of the following fault injections, test cases 3, 5, 6, 12, 18, and 21 all fail.

Chapter 6: Reflections

After a number of challenges through the development process, the current version of the Automated Testing Framework is functioning as intended within the scope of the planned 25 test case suite. The Framework even includes some additional functionalities beyond the scope of what we had initially envisioned, including additional acceptable data types in our test case template as well as out $, delimiter for including multiple arguments to the same function, and the easily legible table format of our listing of the test case passes and failures. Additionally, it is worth mentioning that the test script as well as the test case template, make no explicit mentions of Jython, so it should be possible to use the script and template to test any program, so long as it is written in python and contains at least some standalone functions.

Going forward with this design, although it does function within our scope of testing, there are a couple of shortcomings which could hinder its application on a larger scale. Additional developments to our framework might include support for additional data types for input/output, specifically, tuples and lists, in order to test a wider variety of functions. Additionally the ability to take mock objects as inputs or outputs would be beneficial in testing functions which do not stand alone. Finally, the way that the current Testing Framework operates, results in the storage of all the test cases in memory, although this is a non-issue within the context of our 25 test-case suite it could hinder the framework's applicability to larger test suites comprised of hundreds or thousands of tests.

How To:

Instructions

Dependencies

The first step to install this framework is to install Jython. Installing jython requires some dependencies. First, you will need to have python installed (it should be installed by default assuming a clean ubuntu OS).

To clone the testing framework onto your local machine, you will need to have git installed. In a terminal, enter the following command.

```
sudo apt-get install git
```

Jython uses the mercurial version control system. To install this on a unix system, enter the following command in a terminal:

```
sudo apt-get install mercurial
```

Cloning Jython

Once you have Mercurial installed, you can clone a copy of jython onto your machine.

Take note where you install jython, since you will need to write the path to the root directory of jython in a text file later.

`cd` into the directory that jython will be installed in, then run the following command:

```
hg clone http://hg.python.org/jython
```

Downloading Test Framework

`cd` into the directory that you wish to put the framework in and, using git, execute the following command:

```
git clone https://github.com/CSCI-362-02-2015/Team6.git
```

Document your jython path

`cd` into your downloaded copy of jython. Enter the `pwd` command, this will print out the current directory. For example, if jython is on your desktop, the jython path will print out as:

```
/home/<username>/Desktop/jython
```

Copy this path, and paste it into a text file titled `codebase_path.txt` located in the testing framework folder in the following path:

```
/TestAutomation/docs/codebase_path.txt
```

Run the script

`cd` into the `TestAutomation` directory of the framework and enter the following command to run the framework:

```
python ./scripts/runAllTests.py
```

The testing framework will now run and display the results in an HTML file!

Test the script by injecting faults

See deliverable #5/Chapter 5 of the final report, for exactly where to inject faults in the jython codebase to cause tests to fail.