Testing Framework for Youtube_dl

CSCI-362 Software Engineering, Fall 2016

Designed by Matt Bell, Matt Adamson, Colton Williams

Table of Contents

Chapter 1: Overview

Experiences

Youtube-dl is a command line tool written in Python that can download videos from hundreds of websites. This is useful as users are able to download videos from a variety of sources with one tool, and it's open source. When starting work on the project, we first installed VirtualBox to provide a Linux environment to run the project's source code. Next, we cloned the GitHub repository for Youtube-dl. Luckily, Youtube-dl's only dependency was having Python 3 installed, which comes preinstalled with Ubuntu, so we were good to go from there. Once downloaded, using the program is fairly simple as well. Being a command line tool, Youtube-dl has no User interface, unlike many projects. To download a video with the tool, open a terminal and type "youtube-dl [VideoID]", with VideoID being the url of the online video to download. Once you've done that though, that's it, and the video will begin downloading.

In regards to testing, the project includes a testing framework by default. Youtube-dl came with an existing group of test case files, with each file containing tests for each python file in the project. So for example, for the file utils.py, there is a test file named test_utils.py which contains test cases for all of the methods in utils.py. Most of the files executed without trouble, though the tests had little documentation. In regards to the quantity of tests, most methods included at least three or more tests, and every file for the project had a corresponding test file for testing the methods.

Chapter 2: Testing Plan

Test Case Plan

Youtube-dl contains a folder with Python test case files. Each file tests for a specific feature, as well as a file that runs all of the test case .py files. We plan to take advantage of the existing testing structure, creating relevant files and folders as needed. Current tests will be analyzed in order to find untested or under tested sections of code. This section is where we will focus most of our testing efforts in order to ensure the most reliable product possible. In addition, we created eight test cases that use the actual terminal command of Youtube-dl to download the videos.

Requirements Traceability

- timeConvert(str)→ returns an epoch time from an RFC String
- sanitize_filename(str) → sanitizes a string so it could be used as a file name
- clean_html(str)→ remove tags from HTML code
- parse_age_limit(str)→ converts a string into the minimum viewing age of a video
- parse_duration(str)→ converts string into float of video duration.

Tested Items

The methods to be tested are functions insides of the utils.py which assist the main video downloading functionality. These methods inlcude timeConvert(), sanitize_filename(), clean_html(), parse_age_limit(), and parse_duration().

Testing Schedule

- Sept. 27 - Specify five test cases.
- Oct. 1 - Implement the first five test cases.
- Oct. 8 - Create and implement ten total test cases.
- Oct. 15 - develop automatic testing framework.
- Oct. 22 - Create and implement fifteen total test cases.
- Oct. 29 - Create and implement 20 total test cases.
- Nov. 5 - Create and implement 25 total test cases, test with 5 faults injected as well

Test Recording Procedures

Following the current setup of the project's test cases, we will create a file for each feature we plan to test, and fill it with test cases for that feature's methods. We will also make another file which will call all of our homemade test cases.

Hardware and Software Requirements

Aside from laptops, no hardware is required. For the software side of the project, we will use Virtual Box for Linux, Python's IDLE environment to implement test cases, as well as various video sites to download videos from in order to test the program.

Constraints

The project will not have any monetary constraints. The largest constraint of the project is the amount of available time that our group has to meet each week. Also, our test framework assumes that the websites we are downloading from are currently available with a valid internet connection.

Five Eventual Test Cases

These are the first five cases we implemented once we framework was operational:

```
0001
converts a RFC defined time string into system timestamp
youtube_dl/utils
timeconvert
Fri, 21 Nov 1997 09:55:06 -0600
880127706

0002
sanitizes a string so it could be used as a file name
youtube_dl/utils
sanitize_filename
this is a file name       to sanitize?}{[]\|#'$';True;False
this_is_a_file_name_______to_sanitize__________

0003
Clean an HTML snippet into a readable string
youtube_dl/utils
clean_html
<!doctype html> <html lang="en"> <head> <meta charset="utf-8">  </head> <body> <h1>Test Output</h1> <p>$output</p> </body>
</html>
Test Output $output

0004
formats seconds into hours mins and seconds
youtube_dl/utils
formatSeconds
23456
6:30:56

0005
takes an iso8601 and returns UNIX timestamp from the given date
youtube_dl/utils
parse_iso8601
1997-07-16T19:20:30+01:00
869077230
```

Chapter 3: The Testing Framework

Experiences

The parse test case component of our framework was written first. This proved easy enough, however, the component that gave us the most trouble was the execution of the test cases. The issue we faced was that the names of the methods to be tested were passed in at runtime and we needed to find a way to execute these unknown methods that had an unknown number of parameters. First, a temporary method called fakeExecuteTestCases() was created to generate a dummy output list as a placeholder to use so we could continue work on the html generation and browser loading components. Initially we generated and html list to display output. It became apparent that that format wasn't concise enough and we quickly modified our method to generate an html table instead. After more research and consultation we came across the getattr() method in Python. We used this to return a method, unknow at runtime, contained in the youtube_dl project. We then used positional arguments to pass any number of parameters that the method expected.The massive headache we faced for several hours was quickly resolved with this snippet of code:

```
returnInput = str(getattr(utils,listTests[index +
2])(*paramaterList))
```

Architectural description

Our testing framework logic is contained in topLevelScript.py. This file is composed of four methods and a main. The first method, parseTextFile(), reads through the folder containing the test case text files and parses them into a List. This List contains String representations of the tests' id numbers, requirements, components being tested, method names to be called, parameters, and expected outcomes. This list is then passed into the second method of the program, executeTestCase(). This method then

executes all of the tests contained in the input List. This method inserts the actual output of the external method call as well as a boolean Pass value at the end of that methods data in the List. This List is then returned and is passed to the third method textToHtml(), which writes the list to an CSS styled HTML table inside of TestReport.html. After textToHtml() returns in main() loadOutputInBrowser() is called, loading TestReport.html inside of the user's default browser. This entire evolution is captured in Python code below:

```
def main():
      testInputList = parseTextFile()
      outputList = executeTestCase(testInputList)
      textToHtml(outputList)
      loadOutputInBrowser()
```

How-to Documentation

In order to use our framework test cases must be specified inside of the testCases folder in a .txt file. The file must be line delimited and follow the following example format;

```
Test Case #
Requirement being tested
Component
Method
Inputs separated by ;
Expected outcome(s)
```

After files are in the directory, RunAllTests.py may be run from the scripts folder in order to start the testing framework and execute test cases.

Example Test Cases

```
0001
converts a RFC defined time string into system timestamp
youtube_dl/utils
timeconvert
Fri, 21 Nov 1997 09:55:06 -0600
880127706
```

```
0002
sanitizes a string so it could be used as a file name
youtube_dl/utils
sanitize_filename
this is a file name       to sanitize?}{[]\|#'$';True;False
This_is_a_file_name_______to_sanitize__________
```

After test cases have been loaded into text files the framework can be executed by using the sh command on runAllTests.sh inside of scrips.

Chapter 4: About Test Cases

Experiences

Out of our thirty-three test cases, twenty five of them involving tests for methods that manipulate either string or integer inputs. For example, the method formatSeconds in the file utils.py takes an integer input, which is the number of seconds, and returns an output in the form of hours, minutes, and seconds. Methods such as this were preferable because figuring out the correct output and comparing it to the actual output was relatively simple. For example, we were able to calculate most of the outputs by hand and then check them against the actual output in order to double check that the function was working correctly.

Eight of our thirty-three test cases involving downloading videos using the main.py video-download functionality. Unlike the other test cases from utils.py, these tests must be executed via command line. Initially, this gave us some trouble as we weren't sure how we were going to execute the methods without a system call. However, Python's os.system() method executes commands through the operating system, just as if you were using the terminal. This proved to be the solution to our issue, and allow for these eight tests to work properly.

Chapter 5: Fault Injection

Experiences

In order to inject our faults, we chose the methods which were working on a string or integer and returning some sort of data based on that input. With this in mind, it was easy to find functions which could have faults injected, as simple changes such as replacing a > with a < cause almost all inputs to return invalid outputs. For our first fault, we changed the method timeConvert() and removed a "not", which caused the return value to be None. For the second fault, we changed the method sanitize_filename() and changed an "or" to an "and", which resulted in incorrect filename sanitation. For our third fault, we changed a > to a < which made the return type be formatted incorrectly. For the fourth injection, we changed a * to a +, resulting in incorrect time durations to be calculated and returned. For our last injection, we removed a call to strip() which resulted in the input not getting formatted correctly for return.

The biggest issue throughout the course of fault injection was injecting faults that didn't result in tests failing. This was an issue with most of the methods actually, but tinkering with them further resulted in getting them to break eventually. The important thing to note here is that your function could be broken even if you're getting correct output.

Faults Injected

1) For the first fault, the original line was on line 443 in timeConvert():

```
if timetuple is not None:
```

We changed this to:

```
if timetuple is None:
```

which caused test cases 1 and 11 to fail as expected, with a return value of None.

2) For the second fault, the original line was on line 456 in sanitize_filename():

```
if char == '?' or ord(char) < 32 or ord(char) == 127:
```

We changed this to:

```
if char == '?' and ord(char) < 32 or ord(char) == 127:
```

which caused test cases 2 and 12 to fail.

3) For the third fault, the original line was on line 636 in formatSeconds():

```
if secs > 3600:
```

We changed this to:

```
if secs < 3600:
```

which caused test cases 4 and 14 to fail.

4) For the fourth fault, the original line was on line 1788 in parse_duration():

```
duration += float(mins) * 60
```

We changed this to:

```
duration += float(mins) + 60
```

which caused test cases 18 and 19 to fail.

5) For the fifth fault, the original line was on line 1697 in url_basename():

```
return path.strip('/').split('/')[-1]
```

We changed this to:

```
return path.strip('/')[-1]
```

which caused test cases 21 and 22 to fail.

Chapter 6: Lessons Learned, Evaluations, Suggestions

Lessons Learned/Team Self Evaluation

All in all, a number of valuable lessons have been learnt throughout the semester as we've progressed on our project. While we did not create the testing framework in the same way we had envisioned at the beginning of the semester, it still works as we had intended for it to, which is a victory in itself. Using github and having a strong understanding of version control proved integral to our success, as that was the main way that we collaborated and shared code. For example, we could split up tasks into two parts, and one of us could work on the first while the other worked on the second separately, and once both were finished we could merge them. This sped up the process of working, however this wasn't how we did all of our work. When we found ourselves in a tricky situation, we often pair-programmed with one person at the computer and another person looking along, error checking as the first writes code. This helped us solve some of our trickiest problems, and saved us a lot of time when doing things like creating the test cases themselves, as we were able to breeze through the creation.

If there was anything that could have led to more success within the team, it would have been spending more time in-person rather than working remotely. While working remotely was nice for convenience, it often proved a hindrance in regards communication efforts. The worst case of this was when we had two people working on the same code, and we lost time since two people were working on the same method independently. While this didn't necessarily set us back, it didn't help us at all either. With that in mind -- when we did work in person, communication had little to no barriers, and workflow improved substantially.

Assignment Evaluations & Suggestions

In regards to the amount of work required to complete the deliverables, the workload for each assignment seemed fair. However, having blog posts due on the same day as deliverables made those days particularly difficult to handle. Perhaps moving back the due date for the blog posts to the class before deliverables would result in higher quality deliverables, as we got a majority of our work done around crunch time. The spacing out of assignments

seemed just about right, any more might have been overbearing, any less would have seemed like too few. It would've been nice to have access to some past projects to see how they worked. However, being able to hear other groups' issues in class and see their presentations helped our understanding of the concepts, which proved as a nice substitute.