COLLEGE of CHARLESTON
COMPUTER SCIENCE

YouTube-DL

Matt Adamson
Colton Williams
Matt Bell

Abstract

This project is to create a viable testing framework for the open source project Youtube-DL. The solution was to have a folder of .txt files, containing information of 33 test cases. These files were parsed and executed in a Python file, and outputted into an HTML table.
The result was a more efficient testing system than the current test framework for Youtube-DL, which previously had many different Python testing files and no visual output table.

About Youtube-DL

Youtube-DL is a command line tool that allows users to download videos and music from hundreds of websites. This tool is useful because users are able to download media from multiple sources quickly and without a third-party service. The software is written in Python and open source, so anybody is able to read and clone the code from github.

Testing Framework

The Test Framework involved three components -- a text parser for the .txt files, a method executor, and an output to HTML method. The text parser parsed .txt files which described specific test cases, including necessary information such as arguments for the method being tested. The method executor ran through all data from the text files that were parsed and actually executed the tests. The final parts of the framework convert the text results from testing into html, and then load the html in a browser. This allowed for us to easily see the results of the testing framework.

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

Test Cases

The test cases being fed into the Testing Framework exist in separate .txt files. Each case file contains information such as the test cases' origin file, method to be tested, and expected outputs. The file was seperated by new lines, with the first line being test number, second line being requirement being tested, and so on as detailed in the image below. These test case files are parsed in order to execute the methods later with the specified inputs.

1. Test Number
2. Requirement being tested
3. Component being tested
4. Method being tested
5. Test input
6. Expected outcomes

```
7 lines (6 sloc)  133 Bytes
1  0001
2  converts a RFC defined time string into system timestamp
3  youtube_dl/utils
4  timeconvert
5  Fri, 21 Nov 1997 09:55:06 -0600
6  880127706
```

Testing Output

Output of the test case results was put in a table with notes on each test, as well as if each test passed or failed. Tests that passed were labeled green in the last column, and tests that failed weren't labeled at all. Having the visual representation of the testing framework's results helped us interpret test results more quickly.

Fault Testing

In order to test the effectiveness of our test cases, we injected faults into the project's code to see if our test cases would fail. However, not all of the faults would cause cases to fail, which is why multiple tests of each method were implemented. When the faults injected actually broke the methods, we would see our test cases fail in our testing framework, which confirmed that our testing framework was properly executing the test cases and not just returning everything as a pass. Generally, fault injections are used to confirm that your test cases are thorough enough. So when we injected a fault and it didn't break the test cases, we would look further to see why that had occurred.

Conclusion

The new testing framework is more convenient than the existing project's original testing files. This could provide value to the project by adding a singular testing framework which can run all tests without having to execute a different file to run each test. In the future, this testing framework can be improved by adding more test cases, especially for the main video download function.