UndefinedOffsetNine

DEVELOPING A TESTING FRAMEWORK FOR FUZZYWUZZY

Originally Developed by: SeatGeek

Project Manager: Ronald Zielaznicki
Developer: Cortney Mood
Developer: Katherine Vaughan
Developer: Courtney Profera

Table of Contents

Chapter 1

The FuzzyWuzzy Project

FuzzyWuzzy is a string matching program that depends on the difflib python library. Difflib gives classes and function for comparing sequences. In the case of FuzzyWuzzy, SequenceMatcher is used and then multiple string processings are condensed into a single operation, in order to compare two strings and produce a percentage of how alike the two strings are. Different methods compare the strings in different ways from comparing if the strings are exactly alike to comparing if the same substrings are present within the strings.

Where is the code located?

The code is originally located in Github so that the reader can view the code as well.

Github: https://github.com/seatgeek/fuzzywuzzy

Difflib is the python library that FuzzyWuzzy uses but does not need to be imported in order to utilize this testing framework. The link is given for reference so that the FuzzyWuzzy code and the framework may be better understood.

Difflib: https://docs.python.org/2/library/difflib.html

And SeatGeek gives a brief explanation as to how everything is pulled together and how each method works although an explanation of the methods will be given within this document.

Explanation:http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/

Initial Installation and Working with fuzzyWuzzy

Below are the environments that have been used while working with this project:

VirtualBox with Ubuntu 14.04, Python 2.7.6, installed using Git
Archlinux, Python 2.7.4, installed using Git
VMware Player with Ubuntu 14.04, Python 2.7.6, installed using Git

In order to install the needed technologies and fuzzyWuzzy using the Ubuntu 14.04 terminal:

~$ sudo apt-get install git

~$ git clone git://github.com/seatgeek/fuzzywuzzy.git fuzzywuzzy

~$ cd fuzzywuzzy

~$ python setup.py install

~$ python

This segment will allow the user to use python commands. The imports will need to be done every time the user leaves python and returns to the command line.

>>>from fuzzyWuzzy import fuzz

>>>from fuzzyWuzzy import process

Initial Observations from Files of fuzzyWuzzy

There were a total of 5 different python files that existed in the fuzzyWuzzy project. The names of the files were fuzz, process, String_processing, Utils, and String Matcher.

The file fuzz.py contains the functions used and called to calculate the string compatibility.

Process.py finds the actual matches within the file system used (which can be an array), since the creators, SeatGeek, are looking to match stings for tickets to different events.

String_processing.py processes the string input. This processing involves eliminating non-letters and non-numbers, stripping whitespace, and altering to upper or lower case letters.

Utils.py involves being able to validate the string to not be 0, ASCII letters, and making sure the type is consistent to be processed and compared.

StringMatcher.py seems to be what pulls everything together. This is from the difflib library.

To Test that FuzzyWuzzy is Running Correctly

If viewing python interaction (>>> will be seen as an indicator on the screen) then type quit() to leave python.

```
~$      cd fuzzywuzzy
~$      python test_fuzzywuzzy.py
```

This should return something close to the following:

```
Ran 41 tests in 0.050s
OK
```

This is mostly to check that the system is installed correctly and, since no errors were returned, that everything is working, although no printed output is available.

Checking the Functionality of FuzzyWuzzy

To check some of the functionality and to explore the FuzzyWuzzy methods here are tests given from the Readme and the results that were received. In some cases they differed from the Readme and this has been noted.

```
~$      python
>>>     from fuzzywuzzy import fuzz
>>>     from fuzzywuzzy import process

>>>     Fuzz.ratio("this is a test", "this is a test!")
97
```

Readme says this should be 96, but around this range is accurate

```
>>>     Fuzz.partial_ratio("this is a test", "this is a test!")
100
```

As predicted in the readme

```
>>>     Fuzz.ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear")
91
```

Readme says 90

```
>>>     Fuzz.token_set_ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear")
100
```

Same as the readme

>>> Fuzz.token_sort_ratio("fuzzy was a bear", "fuzzy fuzzy was a bear")
84

Same as the readme

>>> Fuzz.token_set_ratio("fuzzy was a bear", "fuzzy fuzzy was a bear")
100

Same as readme

Chapter

2

Introduction

This chapter will outline the test plan for testing the fuzzyWuzzy methods. The testing process and requirements will be described as well as how the tests were created. The outcome of the tests will be compared to an expected outcome to determine if the test passed or failed based on the accuracy. With string comparisons, accuracy tends to be the driving requirement.

The Testing Process

In testing the string comparison software FuzzyWuzzy we are testing four different methods. The general environment will be a Linux operating system using python for writing the script and the drivers. There will be a driver for each of the methods, and each driver will have the inputs from the designated test case passed through so that it can be evaluated. The drivers will be run by the script. The script will parse the test cases, run the individual drivers, compare the expected and actual outputs, and create the html file which will display the results of the tests.

Requirements Traceability

Fuzzywuzzy is designed as a way to compare two strings. The methods must compare the accuracy of strings and how closely two strings are to each other in different fashions, from comparing if they are exactly the same or are similar, depending on the specific method. What is testable is this percentage that is produced by the comparison. What will be examined is if the results are accurate to the expected outcomes. Each method is comparing the strings differently so even the same input for the different methods will produce different outputs.

Each of the methods is set to cover a different requirement and the tests have been created to reflect such with each test input set to cover a specific area of the vast input space available for string comparison testing. Ratio was created specifically for finding if strings match exactly using difflib's SequenceMatcher() function. Part_ratio was created to compare a long string to a short string and divide the long string into blocks, using get_matching_block() from difflib in order to see if the shorter string is present in the longer string and how it will match. Token_set_ratio again compares a shorter string to a longer string but splits them into tokens and looks for substring similarities. Token_sort_ratio will match strings even if they are not in the same order but contain the same contents. This is done by tokenizing the string and sorting them alphabetically.

Tested Items

Below is the template for the test cases that will be tested. Each of the test cases will have to deal with string comparisons and can all be found in the index.

1. testNumber:
2. requirement:
3. component:
4. method:
5. inputs:
6. driver:
7. expectedOutput:

The tested inputs will need to be designed to cover the input space as best as possible with four methods and only twenty-five test cases. Since it was strings that were being compared this makes it interesting to select the strings to compare. Empty strings will need to be tested, special characters to be sure that FuzzyWuzzy is not just checking for letters to compare, strings that are identical, strings that have matching substrings, and strings that are not identical at all will need to be test.

Testing Schedule

A total of 25 test cases should be completed by the end of this project. Each member of the team will contribute a test case for each of the 4 methods. There will be additional test cases for the Ratio method. Throughout this semester, there has been a detailed schedule of what specific tasks needed to be accomplished at a given time. A tentative schedule for the semester has been outlined below:

Detailed Schedule

Dates	Specified Tasks
September 9 - September 23	Finish test plan and specify 5/25 test cases
October 10 - October 24	Design/Build an automated testing framework that will be used to implement the test plan
November 6	Finish the automated testing framework and all 25 test cases
November 18	Design and inject 5 faults in the project code that will cause the 5 tests to fail

Test Recording Procedures

The outcome of the tests will be recorded in testReport.html. If the test passes will be indicated by saying "pass" and the word will be highlighted in green. If the test fails, then there will be an indicating "fail" highlighted in red.

Hardware and Software Requirements

Current environments include:

-Operating Systems: VirtualBox with Ubuntu 14.04, VMware Player with Ubuntu 14.04, ---- ArchLinux
-Python 2.7.4, Python 2.7.6
-Git
-Subversion
-Python Difflib library

Constraints

One of the largest constraints is the group members' schedules along with other work that must be completed. Most of the issues come from trying to find time in order to meet. Frequent communication is encouraged through Google Hangouts, Google Docs, texting, and face-to-face meetings. Through as much frequent communication as possible, it is hoped to avoid too much miscommunication and last minute problems, even if it becomes difficult to meet up as a group due to schedules and other work.

Chapter

3

Introduction

Chapter 3 discusses the actual framework such as how it operates, details about how the framework would be run, and how the test cases are laid out to be run in the framework.

How the Framework Operates

The framework operates through a script that runs a set of drivers, which are written in python. There is a driver for each method that is being tested: ratio, partial_ratio, token_set_ratio, and token_sort_ratio.

```
def run():
        clearFiles()
        with open(OUTPUT_DIRECTORY + "/testReport.html", 'w') as outputFile:
                setUpHTML(outputFile)
                for file in os.listdir(TEST_CASE_DIRECTORY):
                        testCase = parseTestCase(file)
                        actualOutput = runDriver(testCase)
                        compareResult = compareExpectedToActual(actualOutput, testCase['expectedOutput'])
                        outputFile.write(createFormattedHTML(testCase, actualOutput, compareResult))
                finishHTML(outputFile)
        webbrowser.open(OUTPUT_DIRECTORY + "/testReport.html")

run()
```

The main run() method in runAllTests.py

The script, runAllTests.py consists of methods that allow for the framework to work. Initially, the method clearFiles() will check if there is already a file with the name 'testReport.html' present, this is the file where the testing results are placed. If the file is present then it will be erased so that another blank file of the same name can be created in its place.

Then, with testReport.html set as the output file the basic html for the file, setUpHTML(outputFile), is written so the output is formatted into a table for ease of reading. The script moves to the TestCase directory where it parses the test cases, with parseTestCase(), to determine the test number, requirement, component, method, inputs, driver, and expected output.

From there each driver is run, runDriver(testCase), based on the driver given in the test case file. The driver evaluates the actual output received from the specified fuzzywuzzy method that is being tested and returns the actual output to be displayed in the testReport.html. The actual output and expected output are compared through compareExpectedToActual(actualOutput, expectedOutput) and a pass or fail rating is determined and sent to the html file.

Framework Directory Structure

/UndefinedOffsetNine_TESTING
/docs
  /..
/oracles
  /..
/project
  /..
/reports
    testReport.html
    /scripts
    runAllTests.py
    /..
/temp
    /..
    /testCases
    Test1.txt
    Test2.txt
    …
    /testCasesExecutables
    fuzzPartialRatioDriver.py
    fuzzRatioDriver.py
    fuzzTokenSetRatioDriver.py
    fuzzTokenSortRatioDriver.py
  /..

How to Use the Framework

The FuzzyWuzzy testing framework was developed to work in a Linux environment, being run through the terminal. It has been tested both using Ubuntu 14.04 and Archlinux but this walk through is for Ubuntu 14.04.

SVN, Git, python 2.7.4 or 2.7.6, and FuzzyWuzzy will need to be installed.

To install SVN

~$ sudo apt-get install subversion

To install Git

~$ sudo apt-get install git

To install python

```
~$    sudo apt-get install python2.7
```

To install FuzzyWuzzy

```
~$    git clone git://github.com/seatgeek/fuzzywuzzy.git fuzzywuzzy
~$    cd fuzzywuzzy
~$    python setup.py install
```

To access the functions of FuzzyWuzzy directly the following commands would need to be executed. Within the drivers though are the proper import statements needed so to use this framework these are not needed.

```
~$    python
>>> from fuzzywuzzy import fuzz
>>> from fuzzywuzzy import process
```

To run the framework within Terminal

```
~$    svn checkout https://svn.cs.cofc.edu/repos/CSCI3622014/UndefinedOffsetNine/
~$    cd UndefinedOffsetNine
~$    cd UndefinedOffsetNine_Testing
~$    python ./scripts/runAllTests.py
```

Either 'File not found' or 'file found, clearing...' will appear, then an internet browser should open to display testReport.html which displays the test number, requirement, component, method, inputs, driver name, expected output, actual output, and pass or fail status. If this does not open automatically, the file can be found in the reports directory

Test Cases

The test case files are labeled as Test#.txt where # is the test case number.

The template used for the test cases

testNumber:
requirement:
component:
method:
inputs:
driver:
expectedOutput:

Here is a sample test case, in this case Test1.txt

testNumber: 1
requirement: Compare null to String values
component: fuzz
method: partial_ratio
inputs: "","testing empty string"
driver: fuzzPartialRatioDriver
expectedOutput: 0

Inputs Chosen

With string comparisons it can be tricky to divide up the input space to try to cover a wide range of territory to ensure that the testing presents a satisfactory sample that will prove that the method either works or does not. Basic ground that needed to be covered included empty strings, strings made of special characters, strings that matched exactly, strings with matching substrings, and strings that are completely unalike. It was decided that the for the most part, the four methods would be presented with the same input since each method compares strings differently and thus, in some cases, would produce differing results based on the input.

Chapter 4

Results of Testing

The overall result was that FuzzyWuzzy passed all of the tests that were presented within the testing framework. While it understandably handled basic strings well, it also was able to compare empty strings, and strings that included special characters like a colon.

Tested Methods

With this project it was chosen that the testing remain within the realm of the fuzz.py file and the actual string comparisons. process.py allows for the user to compare strings but it pulls the best match from an array instead of giving the percentage of similarity between two strings. It seemed that it would be easier to test between two strings because humans are able to see the comparison between the two strings instead of dealing with an array of strings. fuzz.py includes the methods ratio, partial_ratio, token_sort_ratio, and token_set_ratio. Each method goes about testing in a slightly different fashion, pulling from the difflib python library.

ratio uses SequenceMatcher() from the difflib library, SequenceMatcher(isjunk, string1, string2) where isjunk is asking if the element should be ignored, usually set to None. Then difflib's ratio() is used to return the calculated similarity as a float.

partial_ratio also uses SequenceMatcher() as well as get_matching_blocks(). This method has each block representing a sequence of matching characters in the string. So instead of comparing multiple bits of a long string to a short string to try to find if it is a match, the blocks will automatically compare the substring, using SequenceMatcher() and ratio() yet again to find the percentage.

token_sort_ratio calls _token_sort. This method allows for checking similarities between strings even if they are out of order. If a string is not in a proper order the similarity results will be low, even if the substrings are all included in each string. _token_sort splits the string and pulls the individual tokens so that they can be sorted and joined back together. It is then that partial_ratio and ratio are called on the strings.

token_set_ratio is similar to token_sort_ratio although the tokens are split up and compared so that there can be a comparison of substrings specifically. By pulling the tokens and checking the difference, the method is able to find when a larger string may include a shorter string as part of a substring. Again, partial_ratio and ratio_are used to produce the actual percentages.

Tested Inputs

Not all inputs can be tested to see what kind of ratio they will produce as the possibilities are infinite. The initial set of tests compare a string with an empty string (Test1-Test5). Test6-Test10 are used to compare the special character ':' to a string to be sure that the testing framework does not break when this is done because within parseTestCase() in our framework, lines are split on ':'. The next set of test cases, Test11-Test20 compare strings that are exactly alike and strings that are approximately 50% alike. Test21-Test24 are used to compare strings that are completely different (0%). Test25 is used to compare strings that have 25% similarity. This is used to check that the comparisons are occurring correctly for "ideal" situations and what would be a typical use scenario.

Sample Output

Test Number	Requirement	Component	Method	Inputs	Driver	Expected Output	Actual Output	Pass/Fail
1	Compare empty and nonempty values. Method compares the shorter string to blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and get_matching_blocks().	fuzz	partial_ratio	"","testing empty string"	fuzzPartialRatioDriver	0	0	Pass
10	Comparing special character : to a string. Method checks for similarities even out of order by sorting the tokens alphabetically and then comparing the strings.	fuzz	token_sort_ratio	":","String"	fuzzTokenSortRatioDriver	0	0	Pass
11	Compare identical strings. Method uses difflib's SequenceMatcher() to compare if the two strings are exactly alike.	fuzz	ratio	"completely alike","completely alike"	fuzzRatioDriver	100	100	Pass
12	Test two identical strings. Method compares the shorter string to blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and get_matching_blocks().	fuzz	partial_ratio	"completely alike","completely alike"	fuzzPartialRatioDriver	100	100	Pass
13	Compare identical strings. Comparison of substring, splits into tokens and compares each substring specifically for comparing short strings to long strings to find substring similarities.	fuzz	token_set_ratio	"completely alike","completely alike"	fuzzTokenSetRatioDriver	100	100	Pass
14	Compare identical strings. Method checks for similarities even out of order by sorting the tokens alphabetically and then comparing the strings.	fuzz	token_sort_ratio	"completely alike","completely alike"	fuzzTokenSortRatioDriver	100	100	Pass
15	Compare two strings with identical substrings. Method uses difflib's SequenceMatcher() to compare if the two strings are exactly alike.	fuzz	ratio	"abc123","abcdef"	fuzzRatioDriver	50	50	Pass

Chapter

5

Injecting faults into FuzzyWuzzy

5 faults were injected into FuzzyWuzzy to determine if the framework could detect each of the faults. This was done in two ways. The first was to inject individual faults into the project. The second multiple faults were injected. The same faults were injected in the same places in both cases. The only difference being in the first only one fault was injected at a time, in the second all the faults were injected at once.

The five injections were placed mainly in the function ratio, and in the function partial_ratio since a large number of other functions depends on them. Injection **One** was placed at line 50, the function would return 1 instead of 0 at this point. **Two** was placed at line 53, multiply by 101 instead of 100. **Three** at line 65, returned 1 instead of 0. **Four** at line 91, there was a comparison of r to .005 instead of .995. **Five** at line 96, multiplying by 50 instead of 100 was done.

During single fault injections, the project managed to run well enough except for in the specific cases where the fault injection occurred. The testing framework managed to detect that a fault was occurred in each case. During the multiple fault injection, FuzzyWuzzy became far more unstable. With each fault injected the ability to predict the outcome became more difficult. The testing framework managed to detect all the faults, but it was far more difficult to trace back the fault since there were multiple faults.

Initially, it was only planned to have the boundary test cases where a null string was being compared to a full string to fail. However, even after the multiple injections were implemented some of the boundary test cases were still passing. Specifically for token_set_ratio, some of the test cases that were expected to fail did not fail, like TestCase3.

So even while being able to detect every fault, it is not guaranteed that the fault will be found even for test cases that were expected to fail. It is also possible that as projects become larger, the effect of a fault on a system could be extremely difficult to detect and find if the fault affects a large part of the system.

Chapter

6

Introduction

Throughout this project the team working on it has had some interesting experiences and has learned quite a bit. Each individual has learned differently due to the differences in experience and expertise. In general though, it can be agreed upon that this has been a rewarding experience providing information that will be helpful later on in life.

Cortney Mood

I feel like working on this project has been both stressful and beneficial. I also feel like this was a pedestal to prepare me for the real world. Being that we were not able to choose our own teams, I feel like this sculpted me into learning how to work in a group and deal with people that have different personalities and different levels of programming. In addition to that, you have to learn how to cooperate and communicate effectively when you are working as a team. If you do not, then you will lose the support and trust of your team members in the process. Otherwise, deadlines are not going to be met and things are not going to get done when they are supposed to. I will also state that I have gained my knowledge as far as working with Ubuntu and subversion being that I have never used them before. Even though the journey was not easy, I will take this learning experience and apply it to my future endeavors.

Courtney Profera

It has been a great experience working with a group on a system that was unknown to me. I feel like this project helped me to understand what I types of group projects I would be doing after I leave college. This project has helped me learn that even though you may not be able to meet up all the time, that you can still work and get things done as a group as long as you communicate. It has also taught me that I will be working with people in groups whose skills are at different levels and that I should not be afraid to ask how to do something if I don't know the answer. I had a really rough start on this project because I was using a laptop that was less than subpar, but a few weeks in I had bought a new laptop and things seemed smoother after everything was installed correctly. It has been really great working with this team and my knowledge has expanded a great deal since the beginning because of my teammates.

Katherine Vaughan

This project has been both rewarding and served as a lesson. It has been rewarding in that I have learned so much while working. I was able to continue with my knowledge of working with Linux and review Python which I have not been able to work with for quite some time. I also get the wonderful ability to say that I have learned what goes into testing and how to build something made specifically for testing whereas previously I was creating the object that needed to be tested. It is also rewarding to be able to work with others of varying experiences and how we were able to learn from each other and work through the problems presented to us. The project has also been a lesson in patience and in general working as a group. I have found patience in trying to find meeting times that can work for a group with schedules that are less than ideal. I have found patience in planning out a project, even if the plan does not always actually work, to ty to divide up the time so that a reasonable amount is done at a time instead of when the pressure is crashing down. I, also have found patience in dealing with a VirtualBox which during the time of this project has had to be recreated twice, and runs rather slow. I found the experience of this project to be exceptionally rewarding.

Ronald Zielaznicki

Working in this group as a project manager has taught me how invaluable face to face meetings are and how much of a difference they can make. When we started out on this project our communication was haphazard at best, but as soon as we started meeting regularly we came together as a team and started pushing forward with the project. Also, this project is the first semi-major Python project I've dealt with. While I knew Python's syntax well enough, this project helped me to understand some of the languages semantics such as lists and dictionaries. The constant learning coupled with the deadlines made this experience a hectic one but a beneficial one.

Appendix

Test Cases

Test1.txt

```
testNumber: 1
requirement: Compare empty and nonempty values. Method compares the shorter string
to blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and
get_matching_blocks().
component: fuzz
method: partial_ratio
inputs: "","testing empty string"
driver: fuzzPartialRatioDriver
expectedOutput: 0
```

Test2.txt

```
testNumber: 2
requirement: Compare empty to nonempty strings. Method uses difflib's
SequenceMatcher() to compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: "","testing empty string"
driver: fuzzRatioDriver
expectedOutput: 0
```

Test3.txt

```
testNumber: 3
requirement: Compare empty to nonempty string. Method checks for similarities even
out of order by sorting the tokens alphabetically and then comparing the strings.
component: fuzz
method: token_set_ratio
inputs: "","testing empty string"
driver: fuzzTokenSetRatioDriver
expectedOutput: 0
```

Test4.txt

```
testNumber: 4
requirement: Compare empty to nonempty string. Comparison of substring, splits
into tokens and compares each substring specifically for comparing short strings
to long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: "Fun Stuff",""
driver: fuzzTokenSetRatioDriver
expectedOutput: 0
```

Appendix

Test5.txt

```
testNumber: 5
requirement: Compare null to String values. Method checks for similarities even
out of order by sorting the tokens alphabetically and then comparing the strings.
component: fuzz
method: token_sort_ratio
inputs: "","testing empty string"
driver: fuzzTokenSortRatioDriver
expectedOutput: 0
```

Test6.txt

```
testNumber: 6
requirement: Comparing special character : to a string. Method compares the
shorter string to blocks (substrings) of the longer string using difflib's
SeuqnceMatcher() and get_matching_blocks().
component: fuzz
method: partial_ratio
inputs: ":","String"
driver: fuzzPartialRatioDriver
expectedOutput: 0
```

Test7.txt

```
testNumber: 7
requirement: Comparing special character : to a string. Method uses difflib's
SequenceMatcher() to compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: ":","String"
driver: fuzzRatioDriver
expectedOutput: 0
```

Text8.txt

```
testNumber: 8
requirement: Comparing special character : to a string. Comparison of substring,
splits into tokens and compares each substring specifically for comparing short
strings to long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: ":","String"
driver: fuzzTokenSetRatioDriver
expectedOutput: 0
```

Appendix

Test9.txt

```
testNumber: 9
requirement: Comparing special character : to a string. Comparison of substring,
splits into tokens and compares each substring specifically for comparing short
strings to long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: ":","String"
driver: fuzzTokenSetRatioDriver
expectedOutput: 0
```

Test10.txt

```
testNumber: 10
requirement: Comparing special character : to a string. Method checks for
similarities even out of order by sorting the tokens alphabetically and then
comparing the strings.
component: fuzz
method: token_sort_ratio
inputs: ":","String"
driver: fuzzTokenSortRatioDriver
expectedOutput: 0
```

Test11.txt

```
testNumber: 11
requirement: Compare identical strings. Method uses difflib's SequenceMatcher() to
compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: "completely alike","completely alike"
driver: fuzzRatioDriver
expectedOutput: 100
```

Test12.txt

```
testNumber: 12
requirement: Test two identical strings. Method compares the shorter string to
blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and
get_matching_blocks().
component: fuzz
method: partial_ratio
inputs: "completely alike","completely alike"
driver: fuzzPartialRatioDriver
expectedOutput: 100
```

Appendix

Test13.txt

```
testNumber: 13
requirement: Compare identical strings. Comparison of substring, splits into
tokens and compares each substring specifically for comparing short strings to
long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: "completely alike","completely alike"
driver: fuzzTokenSetRatioDriver
expectedOutput: 100
```

Test14.txt

```
testNumber: 14
requirement: Compare identical strings. Method checks for similarities even out of
order by sorting the tokens alphabetically and then comparing the strings.
component: fuzz
method: token_sort_ratio
inputs: "completely alike","completely alike"
driver: fuzzTokenSortRatioDriver
expectedOutput: 100
```

Test15.txt

```
testNumber: 15
requirement: Compare two strings with identical substrings. Method uses difflib's
SequenceMatcher() to compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: "abc123","abcdef"
driver: fuzzRatioDriver
expectedOutput: 50
```

Test16.txt

```
testNumber: 16
requirement: Compare two similar strings. Method compares the shorter string to
blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and
get_matching_blocks().
component: fuzz
method: partial_ratio
inputs: "Hello","Hello world"
driver: fuzzPartialRatioDriver
expectedOutput: 100
```

Appendix

Test17.txt

```
testNumber: 17
requirement: Compare two strings with identical substrings. Method uses difflib's
SequenceMatcher() to compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: "Hello","HelloWorld"
driver: fuzzRatioDriver
expectedOutput: 67
```

Test18.txt

```
testNumber: 18
requirement: Compare two strings with identical substrings. Comparison of
substring, splits into tokens and compares each substring specifically for
comparing short strings to long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: "Hello","HelloWorld"
driver: fuzzTokenSetRatioDriver
expectedOutput: 67
```

Test19.txt

```
testNumber: 19
requirement: Compare two strings with identical substrings. Method checks for
similarities even out of order by sorting the tokens alphabetically and then
comparing the strings.
component: fuzz
method: token_set_ratio
inputs: "Hello","Hello World"
driver: fuzzTokenSetRatioDriver
expectedOutput: 100
```

Test20.txt

```
testNumber: 20
requirement: Compare two strings with identical substrings. Method checks for
similarities even out of order by sorting the tokens alphabetically and then
comparing the strings.
component: fuzz
method: token_sort_ratio
inputs: "Hello","HelloWorld"
driver: fuzzTokenSortRatioDriver
expectedOutput: 67
```

Appendix

Test21.txt

```
testNumber: 21
requirement: Compare different strings. Method uses difflib's SequenceMatcher() to
compare if the two strings are exactly alike.
component: fuzz
method: ratio
inputs: "Alphabet", "Corny"
driver: fuzzRatioDriver
expectedOutput: 0
```

Test22.txt

```
testNumber: 22
requirement: Compare different strings. Method compares the shorter string to
blocks (substrings) of the longer string using difflib's SeuqnceMatcher() and
get_matching_blocks().
component: fuzz
method: partial_ratio
inputs: "Alphabet","Corny"
driver: fuzzPartialRatioDriver
expectedOutput: 0
```

Test23.txt

```
testNumber: 23
requirement: Compare different strings. Comparison of substring, splits into
tokens and compares each substring specifically for comparing short strings to
long strings to find substring similarities.
component: fuzz
method: token_set_ratio
inputs: "Alphabet","Corny"
driver: fuzzTokenSetRatioDriver
expectedOutput: 0
```

Test24.txt

```
testNumber: 24
requirement: Compare different strings. Method checks for similarities even out of
order by sorting the tokens alphabetically and then comparing the strings.
component: fuzz
method: token_sort_ratio
inputs: "Alphabet","Corny"
driver: fuzzTokenSortRatioDriver
expectedOutput: 0
```

Appendix

Test25.txt

```
testNumber: 25
requirement: Compare strings with identical substrings Method uses difflib's
SequenceMatcher() to compare if the two strings are exactly alike.
component: ratio
method: ratio
inputs: "Courtney","Combasil"
driver: fuzzRatioDriver
expectedOutput: 25
```

Drivers

ratio

```
import os, sys

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_DIR = os.path.dirname(CURRENT_DIR +"/../")
sys.path.append(PROJECT_DIR)

from project.fuzz import ratio

def main(arguments):
        actualOutput = eval ("ratio(" + arguments + ")")
        return actualOutput
```

partial_ratio

```
import os, sys

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_DIR = os.path.dirname(CURRENT_DIR +"/../")
sys.path.append(PROJECT_DIR)

from project.fuzz import partial_ratio

def main(arguments):
        actualOutput = eval ("partial_ratio(" + arguments + ")")
        return actualOutput
```

token_set_ratio

```
import os, sys

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_DIR = os.path.dirname(CURRENT_DIR +"/../")
sys.path.append(PROJECT_DIR)

from project.fuzz import token_set_ratio

def main(arguments):
        actualOutput = eval ("token_set_ratio(" + arguments + ")")

        return actualOutput
```

Token_sort_ratio

```
import os, sys

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_DIR = os.path.dirname(CURRENT_DIR +"/../")
sys.path.append(PROJECT_DIR)

from project.fuzz import token_sort_ratio

def main(arguments):
        actualOutput = eval ("token_sort_ratio(" + arguments + ")")

        return actualOutput
```