Str1 = "Apple Inc."

Str2 = "apple Inc."

Result = Str1.lower() == Str2.lower()

print(Result)

True

Str1 = "Apple Inc."

Str2 = "apple Inc"

Result = Str1.lower() == Str2.lower()

print(Result)

False

FuzzyWuzzy Python library

There are many methods of comparing string in python. Some of the main methods are:

1. Using regex
2. Simple compare
3. Using difflib

But one of the very easy method is by using **fuzzywuzzy** library where we can have a score out of 100, that denotes two string are equal by giving similarity index. This article talks about how we start using fuzzywuzzy library.

FuzzyWuzzy is a library of Python which is used for string matching. Fuzzy string matching is the process of finding strings that match a given pattern. Basically it uses [Levenshtein Distance](https://en.wikipedia.org/wiki/Levenshtein_distance" \t "_blank) to calculate the differences between sequences.  
[FuzzyWuzzy](https://github.com/seatgeek/fuzzywuzzy) has been developed and open-sourced by SeatGeek, a service to find sport and concert tickets. Their original use case, as discussed in their [blog.](http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/)

**Requirements of fuzzywuzzy**

* Python 2.4 or higher
* python-Levenshtein

**Install via pip :**

**pip install fuzzywuzzy**

**pip install python-Levenshtein**

**How to use this library ?**

First of import these modules,

|  |
| --- |
| from fuzzywuzzy import fuzz  from fuzzywuzzy import process |

Simple ratio usage :

|  |
| --- |
| fuzz.ratio('geeksforgeeks', 'geeksgeeks')  87    # Exact match  fuzz.ratio('GeeksforGeeks', 'GeeksforGeeks')    100  fuzz.ratio('geeks for geeks', 'Geeks For Geeks ')  80 |

|  |
| --- |
| fuzz.partial\_ratio("geeks for geeks", "geeks for geeks!")  100  # Exclamation mark in second string,  but still partially words are same so score comes 100    fuzz.partial\_ratio("geeks for geeks", "geeks geeks")  64  # score is less because there is a extra  token in the middle of the string. |

Now, token set ratio an token sort ratio:

|  |
| --- |
| # Token Sort Ratio  fuzz.token\_sort\_ratio("geeks for geeks", "for geeks geeks")  100    # This gives 100 as every word is same, irrespective of the position    # Token Set Ratio  fuzz.token\_sort\_ratio("geeks for geeks", "geeks for for geeks")  88   fuzz.token\_set\_ratio("geeks for geeks", "geeks for for geeks")  100  # Score comes 100 in second case because token\_set\_ratio  considers duplicate words as a single word. |

Now suppose if we have list of list of options and we want to find the closest match(es), we can use the **process** module

|  |
| --- |
| query = 'geeks for geeks'  choices = ['geek for geek', 'geek geek', 'g. for geeks']    # Get a list of matches ordered by score, default limit to 5  process.extract(query, choices)  [('geeks geeks', 95), ('g. for geeks', 95), ('geek for geek', 93)]    # If we want only the top one  process.extractOne(query, choices)  ('geeks geeks', 95) |

There is also one more ratio which is used often called **WRatio**, sometimes its better to use WRatio instead of simple ratio as WRatio handles lower and upper cases and some other parameters too.

|  |
| --- |
| fuzz.WRatio('geeks for geeks', 'Geeks For Geeks')  100  fuzz.WRatio('geeks for geeks!!!','geeks for geeks')  100  # whereas simple ratio will give for above case  fuzz.ratio('geeks for geeks!!!','geeks for geeks')  91 |

**Full Code**

|  |
| --- |
| # Python code showing all the ratios together,  # make sure you have installed fuzzywuzzy module    from fuzzywuzzy import fuzz  from fuzzywuzzy import process    s1 = "I love GeeksforGeeks"  s2 = "I am loving GeeksforGeeks"  print "FuzzyWuzzy Ratio: ", fuzz.ratio(s1, s2)  print "FuzzyWuzzy PartialRatio: ", fuzz.partial\_ratio(s1, s2)  print "FuzzyWuzzy TokenSortRatio: ", fuzz.token\_sort\_ratio(s1, s2)  print "FuzzyWuzzy TokenSetRatio: ", fuzz.token\_set\_ratio(s1, s2)  print "FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n'    # for process library,  query = 'geeks for geeks'  choices = ['geek for geek', 'geek geek', 'g. for geeks']  print "List of ratios: "  print process.extract(query, choices), '\n'  print "Best among the above list: ",process.extractOne(query, choices) |

Output:

FuzzyWuzzy Ratio: 84

FuzzyWuzzy PartialRatio: 85

FuzzyWuzzy TokenSortRatio: 84

FuzzyWuzzy TokenSetRatio: 86

FuzzyWuzzy WRatio: 84

List of ratios:

[('g. for geeks', 95), ('geek for geek', 93), ('geek geek', 86)]

Best among the above list: ('g. for geeks', 95)

The FuzzyWuzzy library is built on top of difflib library, python-Levenshtein is used for speed. So it is one of the best way for string matching in python.

**The FuzzyWuzzy Package**

This package may have a funny name, but it can be your best friend when the standard Levenshtein distance ratio of similarity between two strings falls short. So far the example that I have been using with "Apple Inc." and "apple Inc" has been relatively simple. After all, there is just one full stop/period of difference if you turn both strings to lower case. However, what happens when something is spelled out of order? What happens when something has considerable spelling variation, but yet it refers to the same thing? That's where the FuzzyWuzzy package comes in since it has functions that allow our fuzzy matching scripts to handle these sorts of cases.

Let's start simple. FuzzyWuzzy has, just like the Levenshtein package, a ratio function that computes the standard Levenshtein distance similarity ratio between two sequences. You can see an example below:

from fuzzywuzzy import fuzz

Str1 = "Apple Inc."

Str2 = "apple Inc"

Ratio = fuzz.ratio(Str1.lower(),Str2.lower())

print(Ratio)

95

That ratio of similarity is the same as we expected given the other examples above. However, fuzzywuzzy has more powerful functions that allow us to deal with more complex situations such as substring matching. Here is an example:

Str1 = "Los Angeles Lakers"

Str2 = "Lakers"

Ratio = fuzz.ratio(Str1.lower(),Str2.lower())

Partial\_Ratio = fuzz.partial\_ratio(Str1.lower(),Str2.lower())

print(Ratio)

print(Partial\_Ratio)

50

100

fuzz.partial\_ratio() is capable of detecting that both strings are referring to the Lakers. Thus, it yields 100% similarity. The way this works is by using an "optimal partial" logic. In other words, if the short string has length kk and the longer string has the length mm, then the algorithm seeks the score of the best matching length-kk substring.

Nevertheless, this approach is not foolproof. What happens when the strings comparison the same, but they are in a different order? Luckily for us, fuzzywuzzy has a solution. You can see the example below:

Str1 = "united states v. nixon"

Str2 = "Nixon v. United States"

Ratio = fuzz.ratio(Str1.lower(),Str2.lower())

Partial\_Ratio = fuzz.partial\_ratio(Str1.lower(),Str2.lower())

Token\_Sort\_Ratio = fuzz.token\_sort\_ratio(Str1,Str2)

print(Ratio)

print(Partial\_Ratio)

print(Token\_Sort\_Ratio)

59

74

100

The fuzz.token functions have an important advantage over ratio and partial\_ratio. They tokenize the strings and preprocess them by turning them to lower case and getting rid of punctuation. In the case of fuzz.token\_sort\_ratio(), the string tokens get sorted alphabetically and then joined together. After that, a simple fuzz.ratio() is applied to obtain the similarity percentage. This allows cases such as court cases in this example to be marked as being the same.

Still, what happens if these two strings are of widely differing lengths? Thats where fuzz.token\_set\_ratio() comes in. Here is an example:

Str1 = "The supreme court case of Nixon vs The United States"

Str2 = "Nixon v. United States"

Ratio = fuzz.ratio(Str1.lower(),Str2.lower())

Partial\_Ratio = fuzz.partial\_ratio(Str1.lower(),Str2.lower())

Token\_Sort\_Ratio = fuzz.token\_sort\_ratio(Str1,Str2)

Token\_Set\_Ratio = fuzz.token\_set\_ratio(Str1,Str2)

print(Ratio)

print(Partial\_Ratio)

print(Token\_Sort\_Ratio)

print(Token\_Set\_Ratio)

57

77

58

95

95% similarity is that magic? No, it's just string preprocessing under the hood. In particular, fuzz.token\_set\_ratio() takes a more flexible approach than fuzz.token\_sort\_ratio(). Instead of just tokenizing the strings, sorting and then pasting the tokens back together, token\_set\_ratio performs a set operation that takes out the common tokens (the intersection) and then makes fuzz.ratio() pairwise comparisons between the following new strings:

* s1 = Sorted\_tokens\_in\_intersection
* s2 = Sorted\_tokens\_in\_intersection + sorted\_rest\_of\_str1\_tokens
* s3 = Sorted\_tokens\_in\_intersection + sorted\_rest\_of\_str2\_tokens

The logic behind these comparisons is that since Sorted\_tokens\_in\_intersection is always the same, the score will tend to go up as these words make up a larger chunk of the original strings or the remaining tokens are closer to each other.

Finally, the fuzzywuzzy package has a module called process that allows you to calculate the string with the highest similarity out of a vector of strings. You can see how this works below:

from fuzzywuzzy import process

str2Match = "apple inc"

strOptions = ["Apple Inc.","apple park","apple incorporated","iphone"]

Ratios = process.extract(str2Match,strOptions)

print(Ratios)

# You can also select the string with the highest matching percentage

highest = process.extractOne(str2Match,strOptions)

print(highest)

[('Apple Inc.', 100), ('apple incorporated', 90), ('apple park', 67), ('iphone', 30)]

('Apple Inc.', 100)

you have learned how to approximately match strings and determine how similar they are.

You leaned fuzzy string matching to map manually entered company names to the account names present in employer's Salesforce CRM ("Apple Inc." to "apple inc" was actually one of the mappings).

However, the usefulness of this technique does not end up here. There are applications in fields such as spell checking or bioinformatics to match DNA sequences, so then there is definitely more uses for fuzzy matching.