# Checking Word Similarity

Friday Presentation

Shrijan Subedi, November 3rd, 2020

# Spell Checking

## Algorithms / Levenshtein distance

- Levenshtein distance is a string metric for measuring the difference between two sequences.
- Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.
- Levenshtein distance may also be referred to as edit distance.
- The Levenshtein distance can also be computed between two longer strings, but the cost to compute it, which is roughly proportional to the product of the two string lengths, makes this impractical.

# Spell Checking

## Algorithms / Levenshtein distance

The Levenshtein distance between two strings $a, b$ (of length $|a|$ and $|b|$ respectively) is given by $\operatorname{lev}_{a,b}(|a|, |b|)$ where

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\operatorname{lev}_{a,b}(i,j)$ is the distance between the first $i$ characters of $a$ and the first $j$ characters of $b$. $i$ and $j$ are 1-based indices.

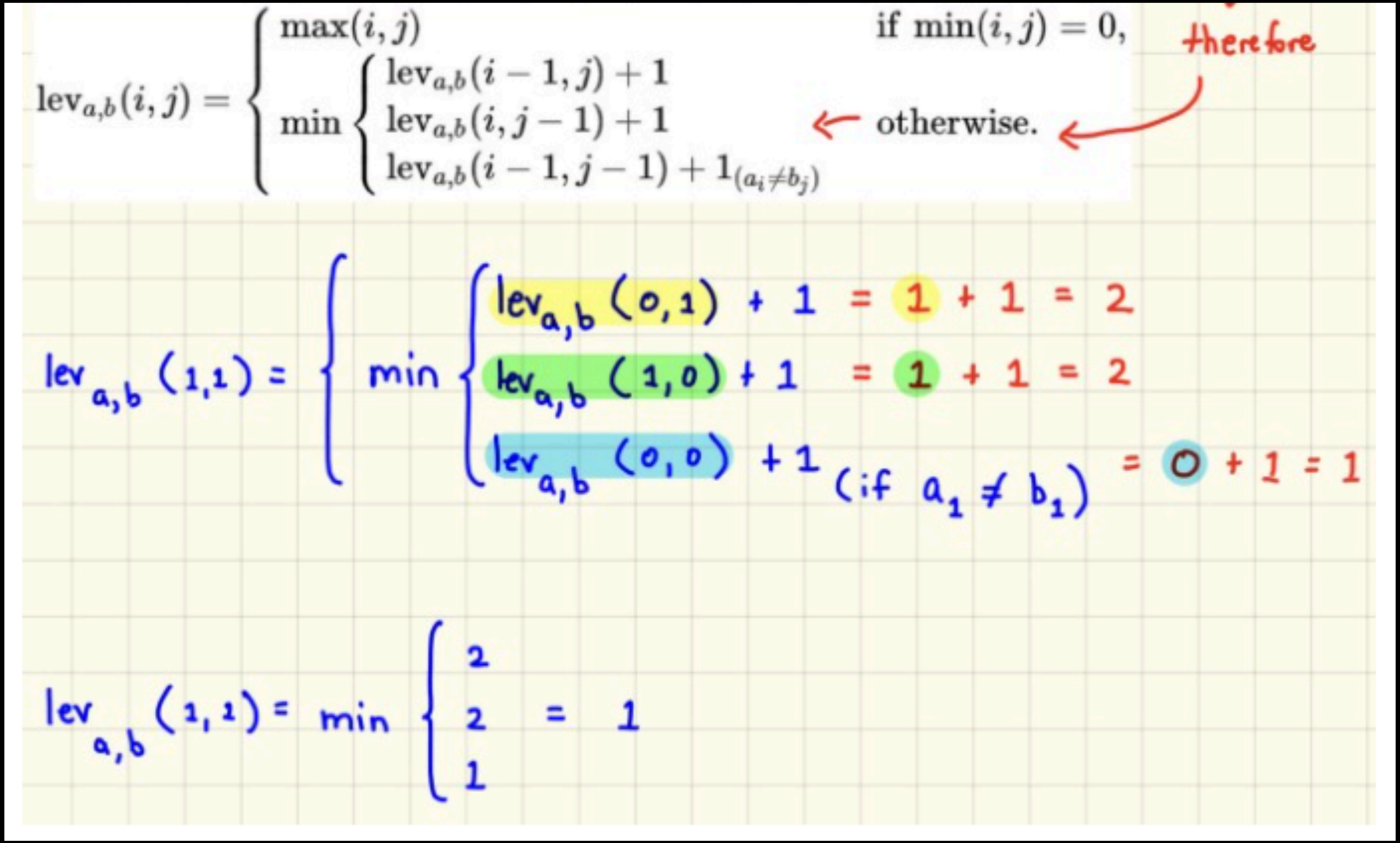
# Spell Checking

## Algorithms / Levenshtein distance / Example

- For example, the Levenshtein distance between “kitten” and “sitting” is 3 since, at a minimum, 3 edits are required to change one into the other.
    - kitten → sitten (substitution of “s” for “k”)
    - sitten → sittin (substitution of “i” for “e”)
    - sittin → sitting (insertion of “g” at the end)

# Spell Checking

## Algorithms / Levenshtein distance / Example



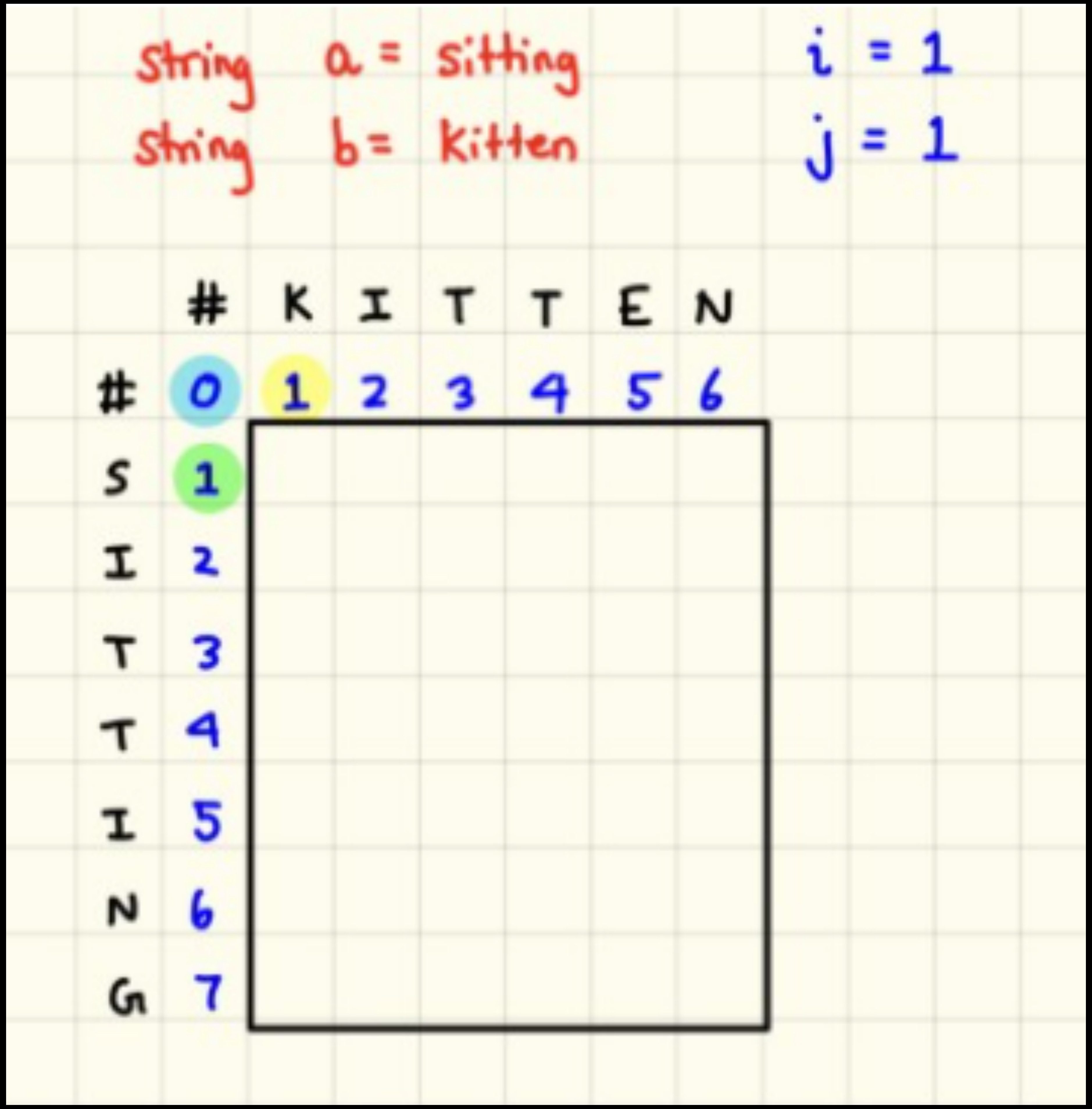
$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min\begin{cases}\mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i\neq b_j)}\end{cases} & \leftarrow \text{otherwise.} \leftarrow \text{therefore} \end{cases}$$

$$\mathrm{lev}_{a,b}(1,1) = \left\{ \min\begin{cases}\mathrm{lev}_{a,b}(0,1)+1 = 1+1 = 2 \\ \mathrm{lev}_{a,b}(1,0)+1 = 1+1 = 2 \\ \mathrm{lev}_{a,b}(0,0)+1_{(\text{if } a_1\neq b_1)} = 0+1 = 1\end{cases}\right.$$

$$\mathrm{lev}_{a,b}(1,1) = \min\begin{cases}2 \\ 2 \\ 1\end{cases} = 1$$

# Spell Checking

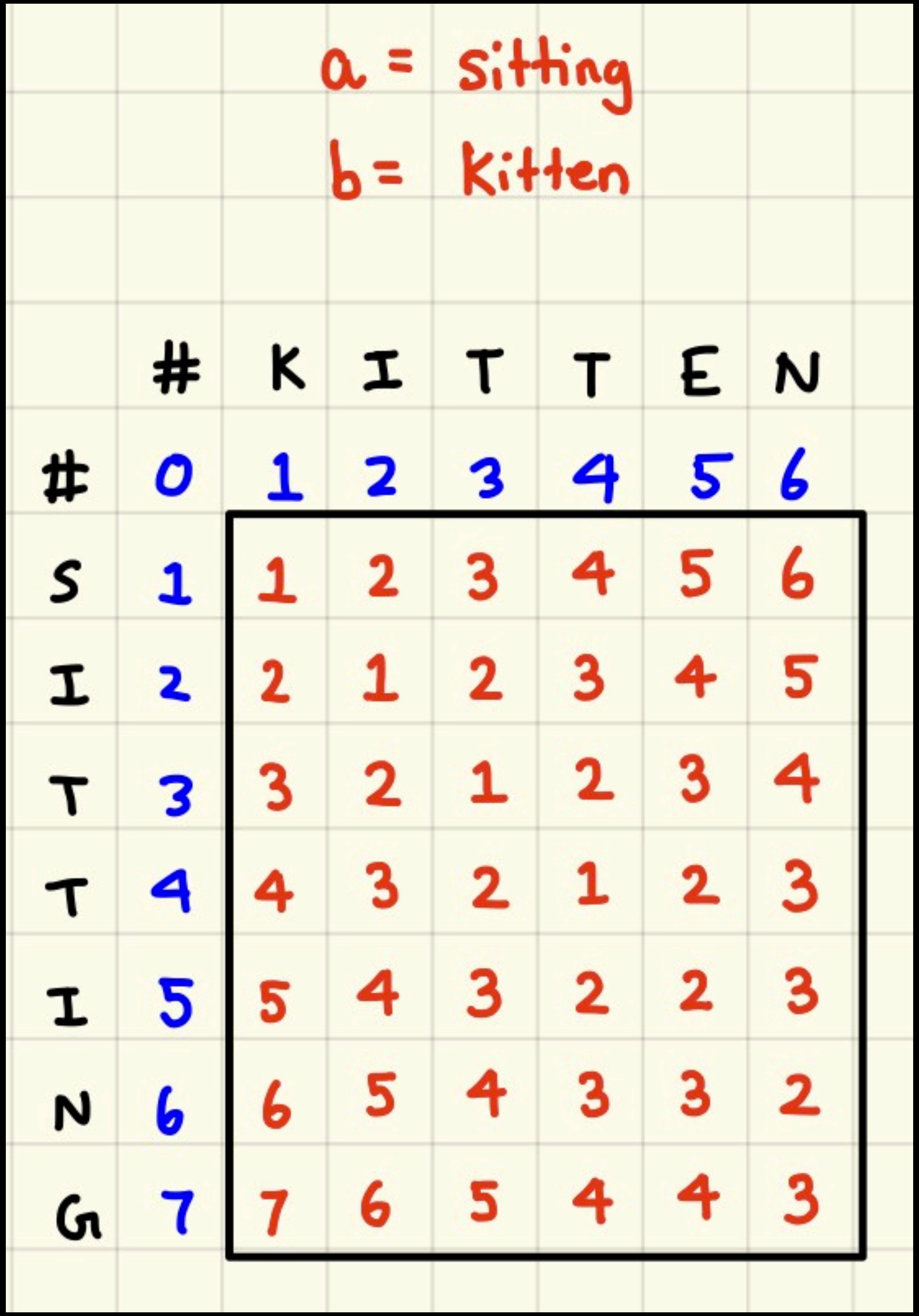
## Algorithms / Levenshtein distance / Example



String a = sitting
String b = kitten

i = 1
j = 1

|   | # | K | I | T | T | E | N |
|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| S | 1 |  |  |  |  |  |  |
| I | 2 |  |  |  |  |  |  |
| T | 3 |  |  |  |  |  |  |
| T | 4 |  |  |  |  |  |  |
| I | 5 |  |  |  |  |  |  |
| N | 6 |  |  |  |  |  |  |
| G | 7 |  |  |  |  |  |  |



a = sitting
b = kitten

|   | # | K | I | T | T | E | N |
|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| S | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| I | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| T | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| T | 4 | 4 | 3 | 2 | 1 | 2 | 3 |
| I | 5 | 5 | 4 | 3 | 2 | 2 | 3 |
| N | 6 | 6 | 5 | 4 | 3 | 3 | 2 |
| G | 7 | 7 | 6 | 5 | 4 | 4 | 3 |

## Spell Checking

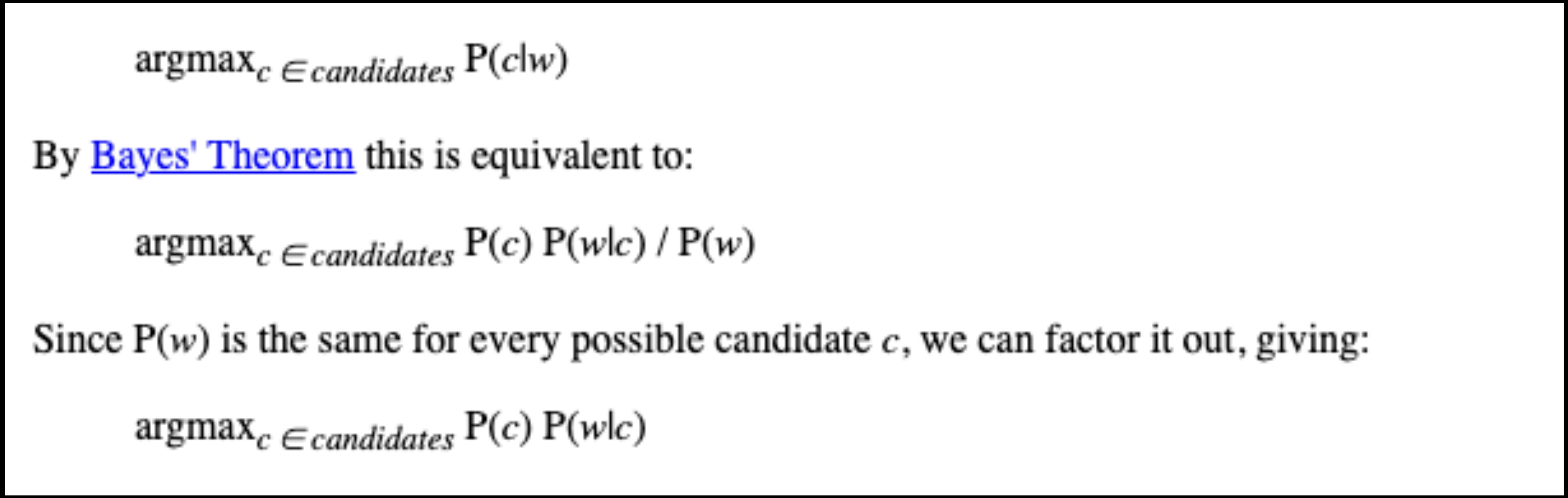
### Algorithms / Norvig Spell Corrector

- Norvig Spell Corrector is Spelling Corrector implementation suggested by Peter Norvig in 2007
- It makes use of Probability theory to compute the most probable correction for a word.
- It is limited to an edit distance of 2 and makes use of delete, transpose, replace and insert to achieve the spelling correction.
- Code

# Spell Checking

## Algorithms / Norvig Spell Corrector

- The call correction(w) tries to choose the most likely spelling correction for w.
- We are trying to find the correction c, out of all possible candidate corrections, that maximizes the probability that c is the intended correction, given the original word w:
  -



$$\text{argmax}_{c \in candidates} \, P(c|w)$$

By Bayes' Theorem this is equivalent to:

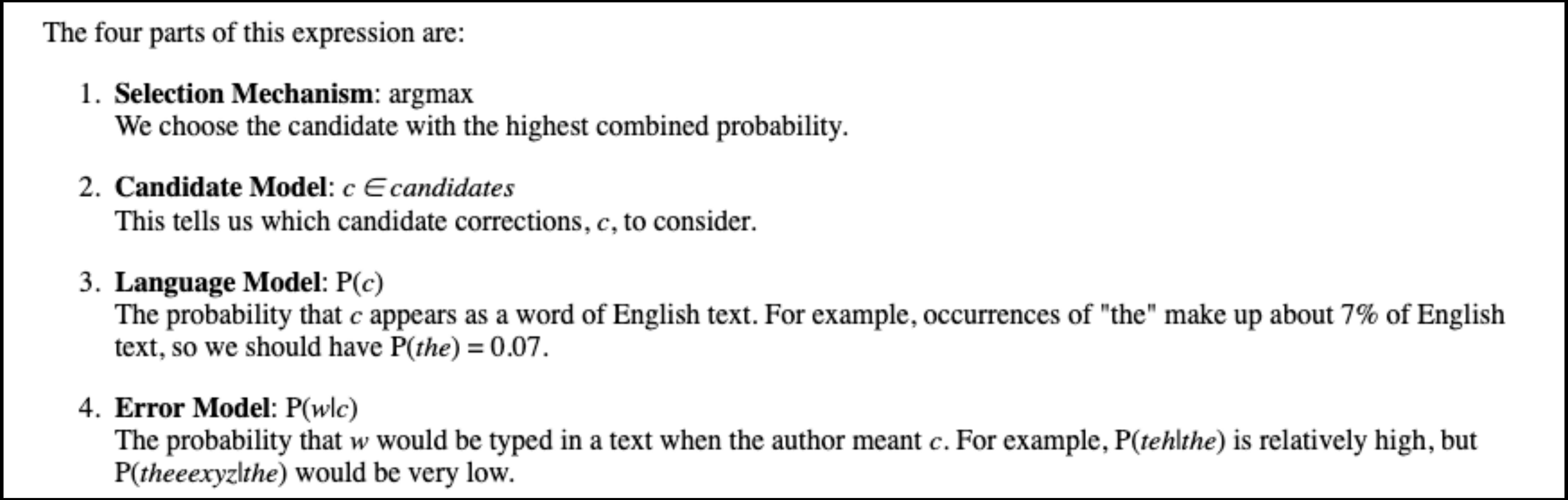
$$\text{argmax}_{c \in candidates} \, P(c) \, P(w|c) \, / \, P(w)$$

Since P($w$) is the same for every possible candidate $c$, we can factor it out, giving:

$$\text{argmax}_{c \in candidates} \, P(c) \, P(w|c)$$

## Spell Checking

### Algorithms / Norvig Spell Corrector


The four parts of this expression are:

1. **Selection Mechanism**: argmax
   We choose the candidate with the highest combined probability.

2. **Candidate Model**: $c \in candidates$
   This tells us which candidate corrections, $c$, to consider.

3. **Language Model**: $P(c)$
   The probability that $c$ appears as a word of English text. For example, occurrences of "the" make up about 7% of English text, so we should have $P(the) = 0.07$.

4. **Error Model**: $P(w|c)$
   The probability that $w$ would be typed in a text when the author meant $c$. For example, $P(teh|the)$ is relatively high, but $P(theeexyz|the)$ would be very low.

# Spell Checking

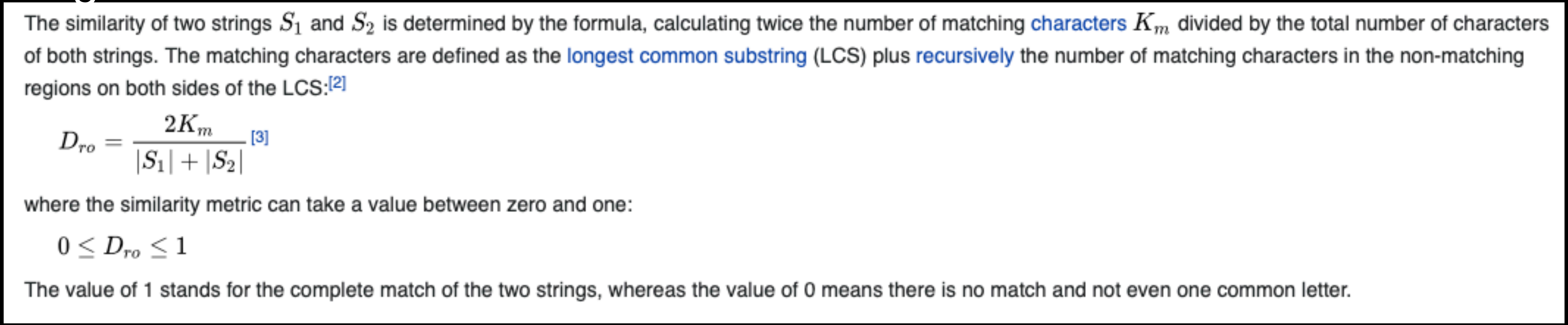
## Algorithms / Gestalt Pattern Matching

- Gestalt Pattern Matching is a string-matching algorithm for determining the similarity of two strings.
- It was developed in 1983 by John W. Ratcliff and John A. Obershelp
- It is used in Python's Difflib library
- Due to the unfavourable runtime behaviour of this similarity metric, three methods have been implemented. Two of them (real_quick_ratio( ) and quick_ratio( ) ) return an upper bound in a faster execution time. real_quick_ratio( ) only compares length of two substrings

# Spell Checking

## Algorithms / Gestalt Pattern Matching

- Algorithm:



The similarity of two strings $S_1$ and $S_2$ is determined by the formula, calculating twice the number of matching characters $K_m$ divided by the total number of characters of both strings. The matching characters are defined as the longest common substring (LCS) plus recursively the number of matching characters in the non-matching regions on both sides of the LCS:[2]

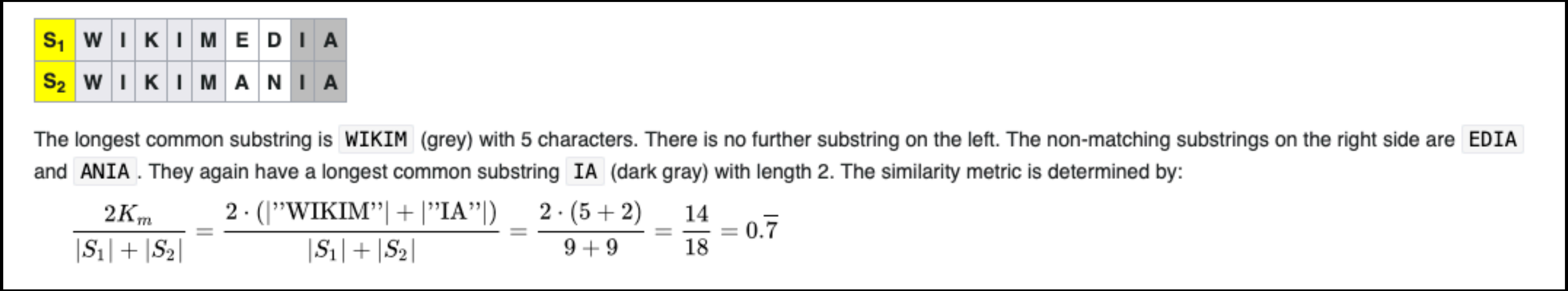
$$D_{ro} = \frac{2K_m}{|S_1| + |S_2|}$$ [3]

where the similarity metric can take a value between zero and one:

$$0 \leq D_{ro} \leq 1$$

The value of 1 stands for the complete match of the two strings, whereas the value of 0 means there is no match and not even one common letter.

- Example:



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | W | I | K | I | M | E | D | I | A |
| $S_2$ | W | I | K | I | M | A | N | I | A |

The longest common substring is `WIKIM` (grey) with 5 characters. There is no further substring on the left. The non-matching substrings on the right side are `EDIA` and `ANIA`. They again have a longest common substring `IA` (dark gray) with length 2. The similarity metric is determined by:

$$\frac{2K_m}{|S_1| + |S_2|} = \frac{2 \cdot (|\text{''WIKIM''}| + |\text{''IA''}|)}{|S_1| + |S_2|} = \frac{2 \cdot (5 + 2)}{9 + 9} = \frac{14}{18} = 0.\overline{7}$$

## Spelling Libraries

### Enchant

- Enchant is a library that wraps a number of different spelling libraries and programs with a consistent interface.
- We can use a wide range of spelling libraries, including some specialised for particular languages, without needing to program to each library's interface.
- Enchant has 6 backends: Hunspell, Nuspell, GNU Aspell, Hspell, Voikko, Zemberek, AppleSpell

## Spelling Libraries

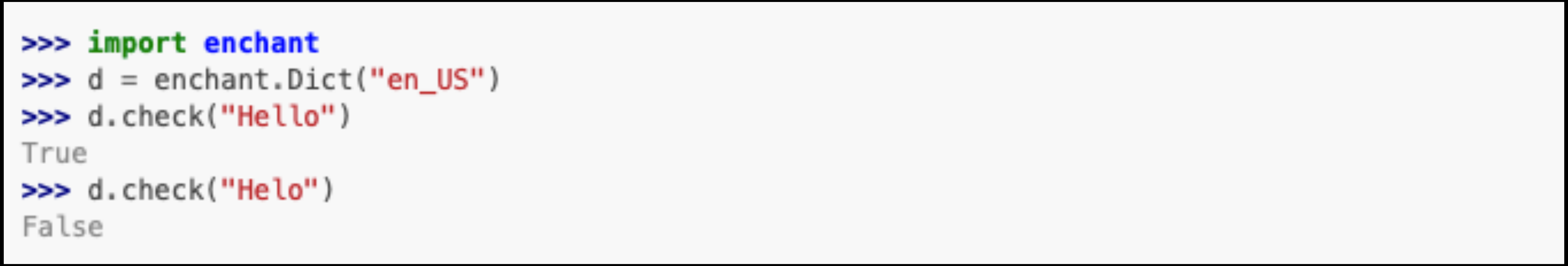
**Enchant / PyEnchant**

- PyEnchant is a spellchecking library for Python, based on the Enchant library.
- PyEnchant combines all the functionality of the underlying Enchant library with the flexibility of Python.
- PyEnchant is compatible with Python versions 3.5 and requires the following to be fulfilled:
  - Enchant C library : libenchant(macOS,linux), glib2(windows)
  - Dictionaries for particular language: enchant.list_languages lists all available dictionaries. If not present installation may be required.

## Spelling Libraries

### Enchant / PyEnchant / Creating and Using Dictionary

- The most important object in the PyEnchant module is the Dict object, which represents a dictionary. It’s used to check spellings and get suggestions.
- Example:



```
>>> import enchant
>>> d = enchant.Dict("en_US")
>>> d.check("Hello")
True
>>> d.check("Helo")
False
```

- If not dictionary can be found, it raises an exception. We can check if a dictionary exists using .dict_exists(“name”)

# Spelling Libraries

## Enchant / PyEnchant / Personal Word Lists

- We can check words against a custom list of correctly-spelled words known as a Personal Word List. It is a file of words to be considered, one word per line.
- >>> pwl = enchant.request_pwl_dict(“mywords.txt”)
- The personal word list Dict object can be used in the same way as Dict objects
- PyEnchant also provides the class DictWithPWL which can be used to combine a language dictionary and a personal word list file.
- >>> d2 = enchant.DictWithPWL(“en_US","mywords.txt")

## Spelling Libraries

### Enchant / PyEnchant / Checking Spellings

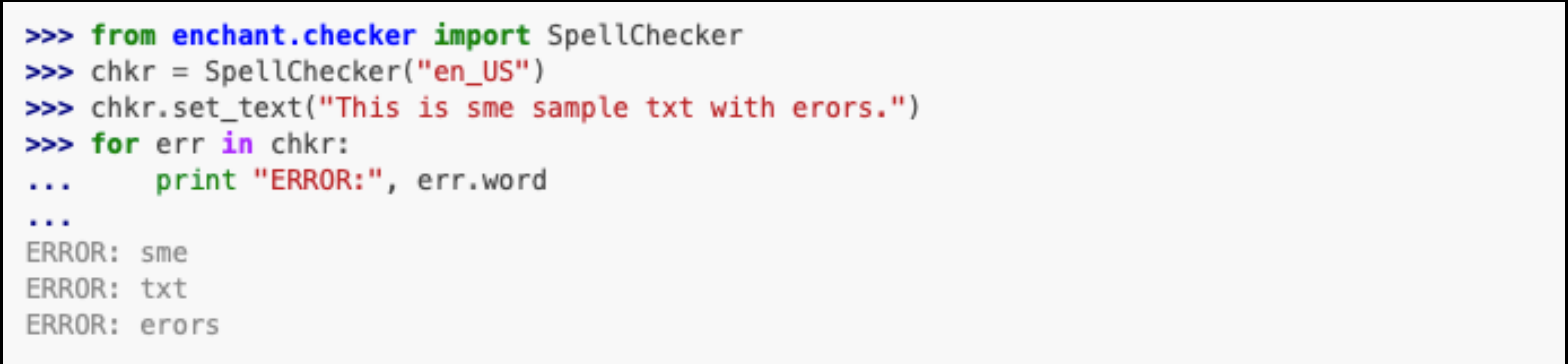
- We can use dictionary.suggest(‘word’) to get suggestions for a misspelled word.
- This returns a list of suggestions that are probable to the given word with first index having the highest probability.
- >>> pwl.suggest(‘helo’)

```
[‘hello’,’halo’,……]
```

# Spelling Libraries

## Enchant / PyEnchant / Checking Entire Blocks of Text

- The module enchant.checker provides a class SpellChecker which is designed to handle paragraphs.
- SpellChecker objects are created in the same way as Dict objects - by passing a language tag to the constructor
- SpellChecker object can be used as an iterator over the spelling mistakes in the text.

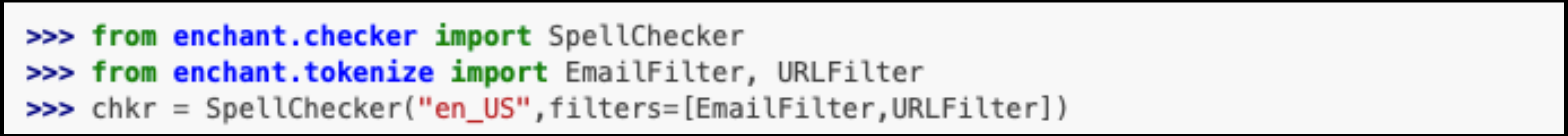


```
>>> from enchant.checker import SpellChecker
>>> chkr = SpellChecker("en_US")
>>> chkr.set_text("This is sme sample txt with erors.")
>>> for err in chkr:
...     print "ERROR:", err.word
...
ERROR: sme
ERROR: txt
ERROR: erors
```

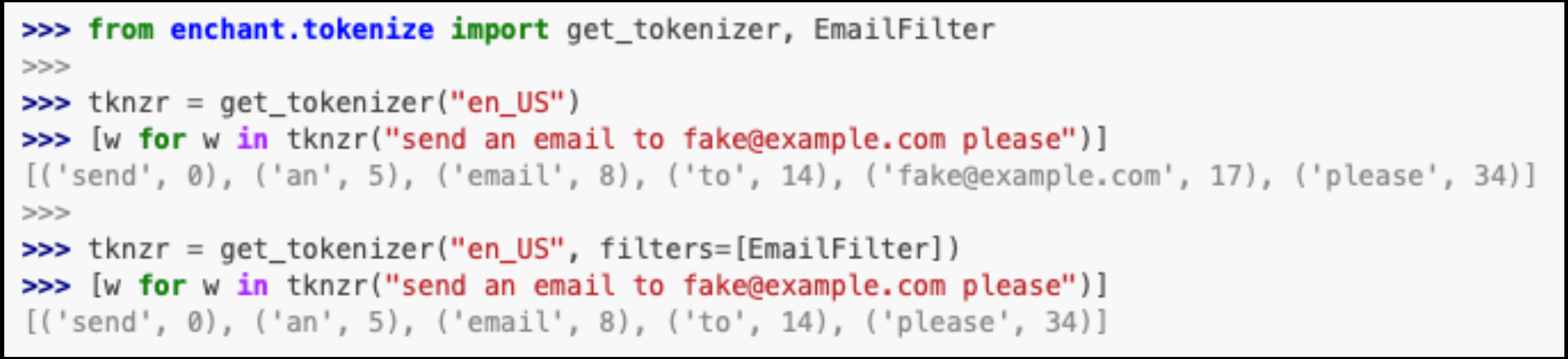
## Spelling Libraries

### Enchant / PyEnchant / Filtering

- We can use filters to ignore certain word forms, by passing a list of filters in as a keyword argument.
- We may use filters with SpellChecker class or with get_tokenizer function.



```
>>> from enchant.checker import SpellChecker
>>> from enchant.tokenize import EmailFilter, URLFilter
>>> chkr = SpellChecker("en_US",filters=[EmailFilter,URLFilter])
```

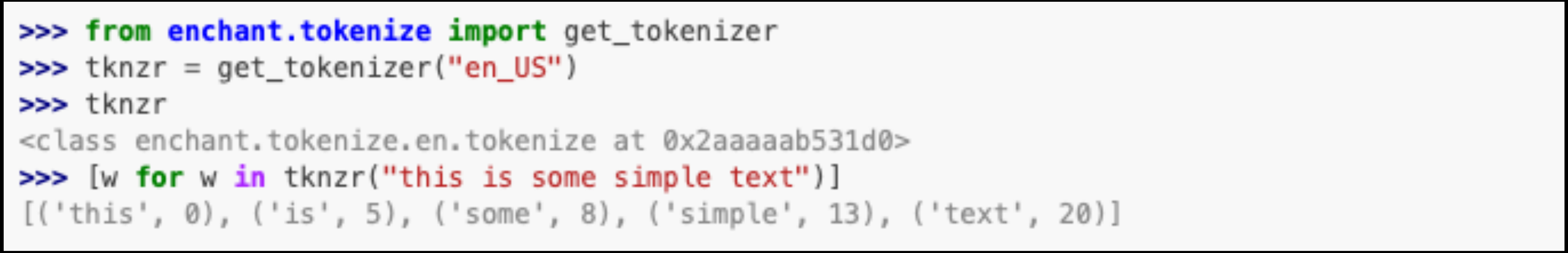


```
>>> from enchant.tokenize import get_tokenizer, EmailFilter
>>>
>>> tknzr = get_tokenizer("en_US")
>>> [w for w in tknzr("send an email to fake@example.com please")]
[('send', 0), ('an', 5), ('email', 8), ('to', 14), ('fake@example.com', 17), ('please', 34)]
>>>
>>> tknzr = get_tokenizer("en_US", filters=[EmailFilter])
>>> [w for w in tknzr("send an email to fake@example.com please")]
[('send', 0), ('an', 5), ('email', 8), ('to', 14), ('please', 34)]
```

## Spelling Libraries

### Enchant / PyEnchant / Tokenizer

- Tokenization is splitting a body of text up into its constitutive words, each of which is then passed to a Dict object for checking.
- PyEnchant provides the enchant.tokenize module to assist with this task.



```
>>> from enchant.tokenize import get_tokenizer
>>> tknzr = get_tokenizer("en_US")
>>> tknzr
<class enchant.tokenize.en.tokenize at 0x2aaaaab531d0>
>>> [w for w in tknzr("this is some simple text")]
[('this', 0), ('is', 5), ('some', 8), ('simple', 13), ('text', 20)]
```

## Spelling Libraries

**difflib /**

- Difflib is a Python module that contains several easy-to-use functions and classes that allow users to compare sets of data.
- The module presents the results of these sequence comparisons in a human-readable format
- It utilizes deltas to display the differences.
- Much of the time difflib is used to compared string sequences, however, it can also be used to compare other data types as long as they are hashable (i.e. if its hash value doesn’t change during the duration of its lifetime.

## Spelling Libraries

### difflib / Sequence Matcher Class

- SequenceMatcher compares two given strings and returns data that presents how similar the two strings are.
- We use .ratio( ) object with the SequenceMatcher. This determines the ratio of characters that are similar in the two strings and the result is then returned as a decimal
- It takes two parameters, a and b , the two strings to be compared.
- Example:
    - >>> seq = SequenceMatcher(a=str1, b=str2)
    - >>> print(seq.ratio())

## Spelling Libraries

**difflib / Sequence Matcher Class / Algorithm**

- Extension of Gestalt pattern matching algorithm by Ratcliff and Obershelp
- Find the longest contiguous matching subsequence that contains no “junk” elements (i.e blank lines or whitespaces etc).
- The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence.
- This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.
- SequenceMatcher is quadratic time O(N^2) for the worst case; best case time is linear.

## Spelling Libraries

### difflib / Sequence Matcher Class / .ratio

- Return a measure of the sequences’ similarity as a float in the range [0, 1].
- Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. This is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.
- We may use quick_ratio( )->return upper bound on ratio( ) quickly or real_quick_ratio() -> return upper bound on ratio( ) very quickly.
- The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation

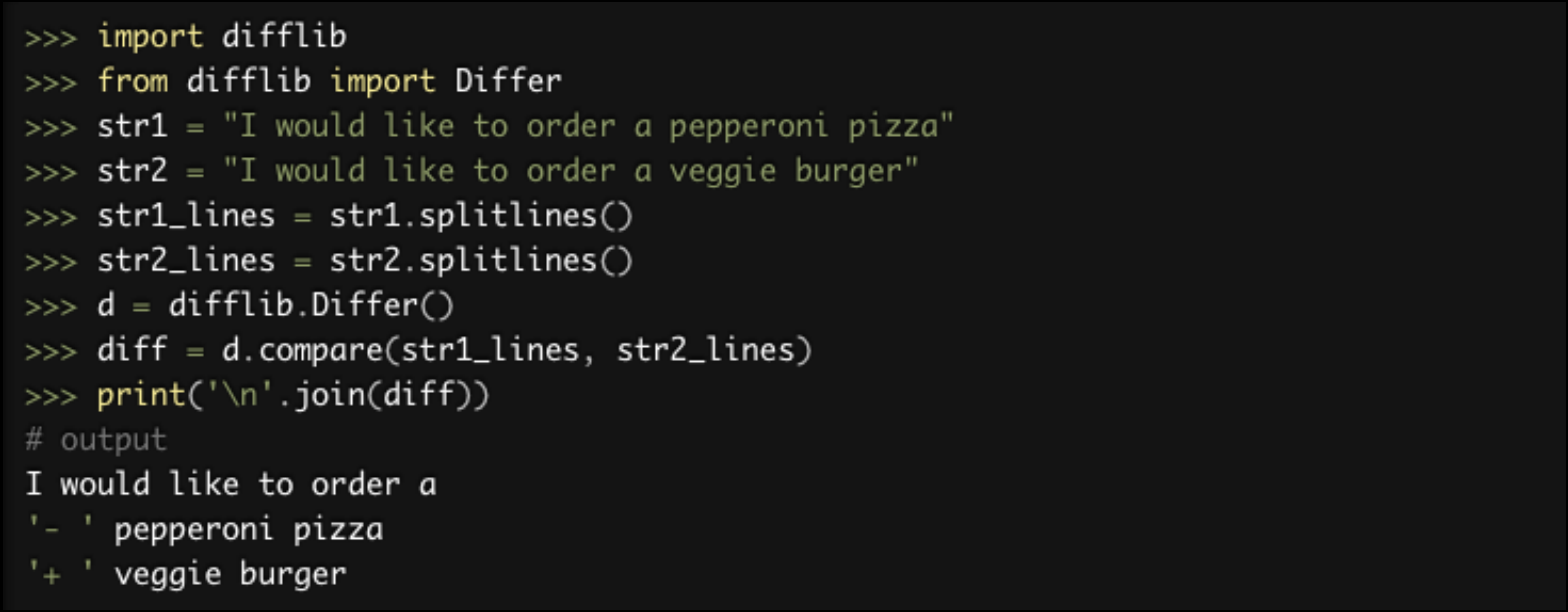
## Spelling Libraries

### difflib / Differ Class

- The Differ class is the opposite of SequenceMatcher; it takes in lines of text and finds the differences between the strings.
- When adding new characters to the second string in a comparison between two strings, a '+ ' will appear before the line that has received the additional characters.
- Deleting some of the characters that were visible in the first string will cause '- ' to pop up before the second line of text.

## Spelling Libraries

### difflib / Differ Class



```
>>> import difflib
>>> from difflib import Differ
>>> str1 = "I would like to order a pepperoni pizza"
>>> str2 = "I would like to order a veggie burger"
>>> str1_lines = str1.splitlines()
>>> str2_lines = str2.splitlines()
>>> d = difflib.Differ()
>>> diff = d.compare(str1_lines, str2_lines)
>>> print('\n'.join(diff))
# output
I would like to order a
'- ' pepperoni pizza
'+ ' veggie burger
```

## Spelling Libraries

**difflib / get_close_matches**

- get_close_matches is a tool that will take in arguments and return the closest matches to the target string.
- We use it as:
  - get_close_matches(target_word, list_of_possibilities, n=result_limit, cutoff)
- Example:
  - get_close_matches('bat', ['baton', 'chess', 'bat', 'bats', 'fireflies', ‘batter'])
  - ['bat', 'bats', 'baton']

## Spelling Libraries

**difflib / get_close_matches**

- Arguments:

  word:  the word to check spelling of

  possibilities:  a list of sequences against which to match word

  n:  maximum number of close matches to return. Takes int n>0

  cutoff:  minimum possibility score to return word .Takes float [0, 1]

## Spelling Libraries

### difflib / Other features

- difflib.ndiff -> Compare a and b (lists of strings); return a Differ-style delta
- class difflib.HtmlDiff -> This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights.
- difflib.context_diff -> Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

## References

- https://iq.opengenus.org/difflib-module-in-python/
- http://pyenchant.github.io/pyenchant/tutorial.html
- https://abiword.github.io/enchant/
- https://towardsdatascience.com/sequencematcher-in-python-6b1e6f3915fc
- https://docs.python.org/3/library/difflib.html#difflib.get_close_matches