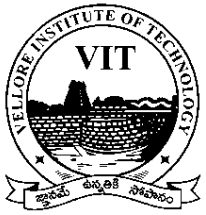


VIT-AP UNIVERSITY

Natural Language Processing

(Course Code: CSE 3015)

Module-1:Lecture-2: Text Preprocessing

Gundimeda Venugopal, Professor of Practice, SCOPE

Text Preprocessing Steps

Text preprocessing includes the following:

- Text Wrangling
- Text cleansing
- Sentence splitter
- Tokenization
- Stemming
- Lemmatization
- Stop word removal
- Rare word removal
- Spell correction

Text Wrangling

❖Text wrangling is the process of cleaning and preparing raw text data for analysis or training. It's a common step in Natural Language Processing (NLP) pipelines.

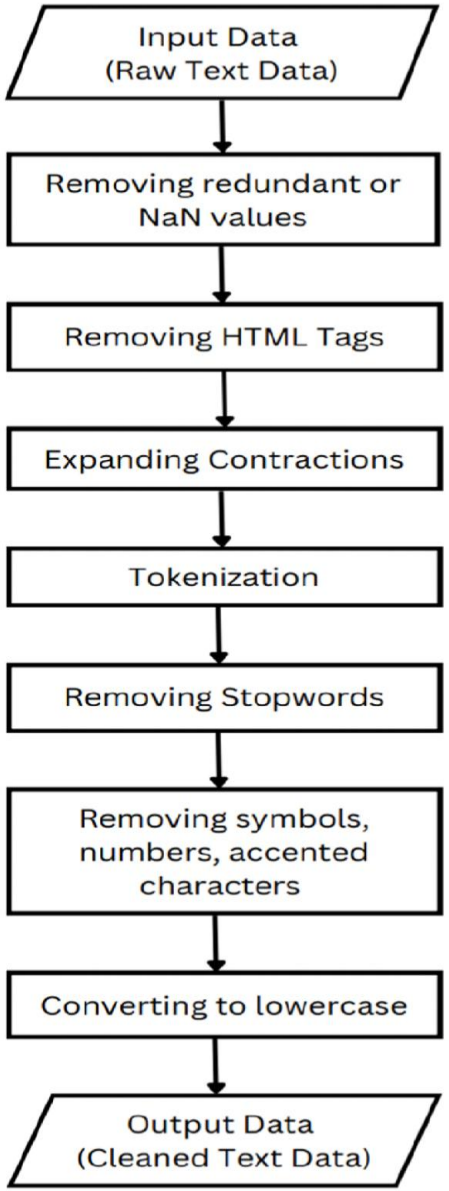
❖ Some common text wrangling / Preprocessing techniques:

- Tokenization: Breaks text into smaller units, like words or sentences
- Normalization: Standardizes text forms, such as lowercasing, stemming, or lemmatization
- Stop word removal: Removes words like "the" or "and"
- Part-of-speech tagging: Assigns grammatical categories
- Named entity recognition: Identifies entities like people or places
- Sentiment analysis: Detects emotions
- Topic modeling: Uncovers main themes

Common Text Data Wrangling challenges

- Extract only numeric values from text
- Remove escaping characters and extra spaces / Tabs from text
- Detect (or find) specific characters from text
- Replace specific characters with something else
- Recode or Map to pre-defined values
- Split text data into multiple values and count them
- Convert Character to Date data type
- Concatenate (or combine) text values from multiple columns

Text Wrangling and preprocessing for HTML content



Before we start our text preprocessing the first step is to load our data from its data source. Data source include CSV, HTML, XML, Database, JSON, NoSQL, PDF and so on. Data from the data source can be parsed using various python parsers like import CSV, import HTML parser, import json. Snippet below shows how you can read a file in CSV and JSON format.

```
import json
with open('example.json', 'r') as file:
    data = json.load(file)
    print(data['list'])
```

```
[1, 2, 3, 4]
```

```
import csv
with open('example.csv') as file:
    reader = csv.reader(file)
    for line in reader:
        print(line[0])
```

```
list
[1, 2, 3,4]
[5, 6,7,8]
```

•Text Wrangling

•Wrangling is a process where one transforms “raw” data for making it more suitable for analysis and it will improve the quality of your data.

•The process involves data munging, text cleansing, specific preprocessing, tokenization, stemming or lemmatization and stop word removal.

•Text wrangling converts and transforms information at different levels of granularity. For example, information in a list could be converted into a table, or vice versa. Text wrangling can restructure and renarrate information

• Let's start with a basic example of parsing a csv file:

```
>>>import csv
>>>with open('example.csv','rb') as f:
>>> reader = csv.reader(f,delimiter=',',quotechar='"')
>>> for line in reader :
>>> print line[1] # assuming the second field is the raw sting
```

For example, json looks like:

```
{
"array": [1,2,3,4],
"boolean": True,
"object": {"a": "b”},
"string": "Hello World"
}
```

Let's say we want to process the string. The parsing code will be:

```
>>>import json
>>>jsonfile = open('example.json')
>>>data = json.load(jsonfile)
>>>print data['string']
Output:"Hello World“
```

Python allows you to choose and process it to a raw string form.

•Text cleansing

- Text cleansing is loosely used for most of the cleaning to be done on text, depending on the data source, parsing performance, external noise and so on.

- Once we have parsed the text from our data sources, the challenge make sense of raw data. Our aim is to remove all the noise surrounding the text.

- In Natural Language Processing for cleaning the html using html_clean, can be labeled as text cleansing.

- In another case, where we are parsing a PDF, there could be unwanted noisy characters, non ASCII characters to be removed, and so on.

- Before going on to next steps we want to remove these to get a clean text to process further.

Sentence Splitting/Segmentation

- Sentence segmentation is concerned about splitting the given input text in to sentences.
- Characters like !, ?, . may indicate sentence boundaries
- However, there could be ambiguity, for instance:
  - The quarterly results of Yahoo! showed a promising trend
  - Microsoft announced updates to .NET framework
  - Indian rupee appreciated by 0.5% against USD during today's trade
  - Who moved my cheese? is a great read.
- The ambiguity may also arise due to spelling mistakes
- Possible ways to resolve:
  - Traditional rule based regular expressions
  - Decision trees encoding some rules
  - Classifiers

- Sentence splitter

  - Some of the NLP applications require splitting a large raw text into sentences to get more meaningful information out.
  - A sentence is an acceptable unit of conversation.
  - A typical sentence splitter can be something as simple as splitting the string on (.), to something as complex as a predictive classifier to identify sentence boundaries:

```
>>>inputstring = ' This is an example sent. The sentence splitter will split on sent markers. Ohh really !!'
>>>from nltk.tokenize import sent_tokenize
>>>all_sent = sent_tokenize(inputstring)
>>>print all_sent
[' This is an example sent', 'The sentence splitter will split on markers.','Ohh really !!']
```

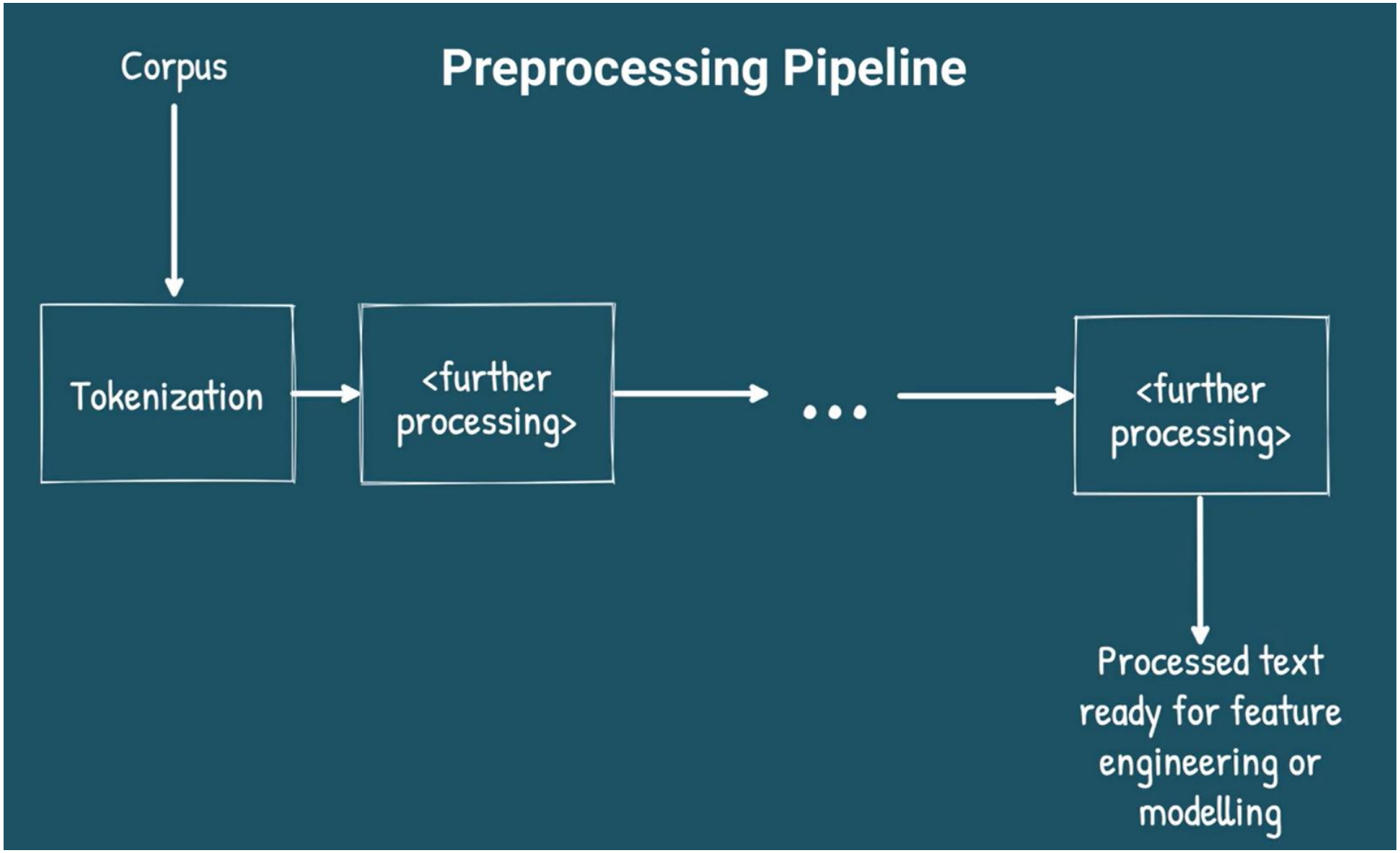
•We are trying to split the raw text string into a list of sentences.

•The function sent_tokenize, internally uses a sentence boundary detection algorithm that comes pre-built into NLTK.

If your application requires a custom sentence splitter, there are ways that we can train a sentence splitter of our own:

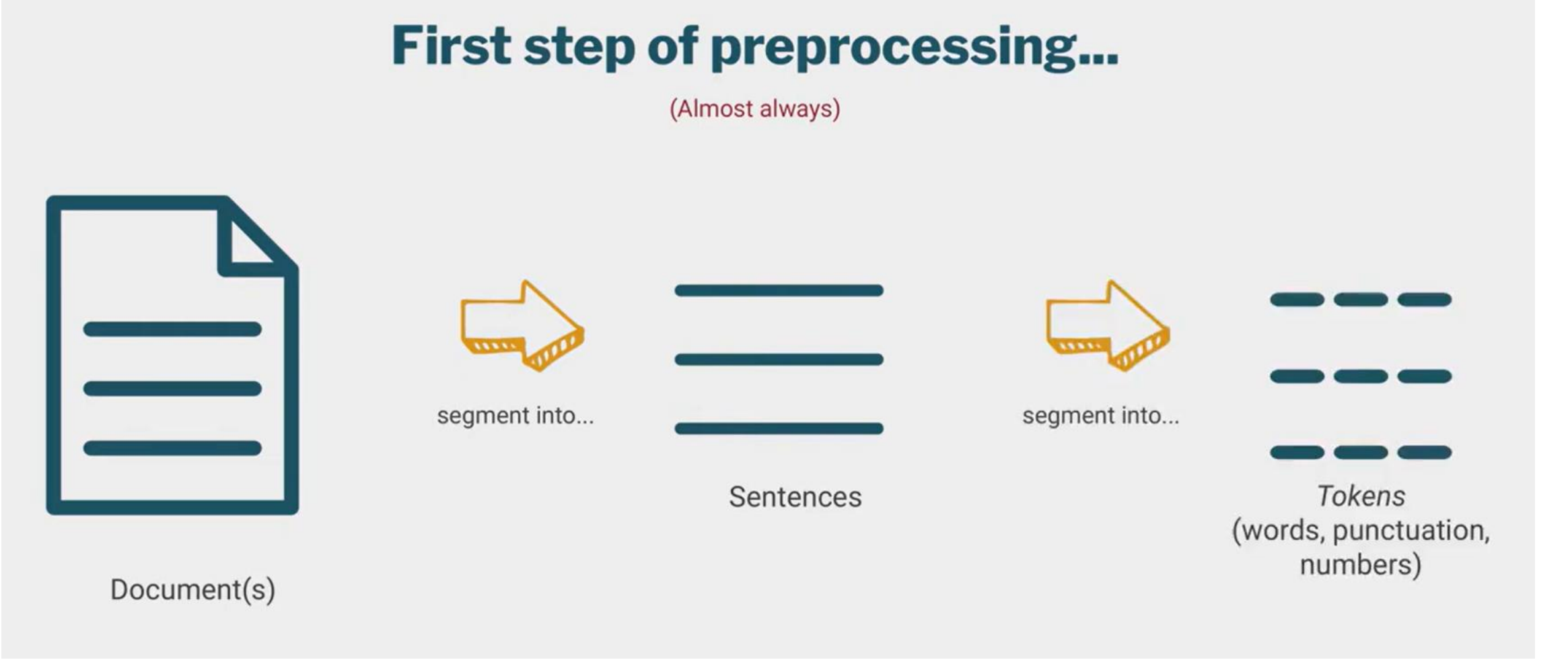
```
>>>import nltk.tokenize.punkt
>>>tokenizer = nltk.tokenize.punkt.PunktSentenceTokenizer()
```

Text Preprocessing Pipeline

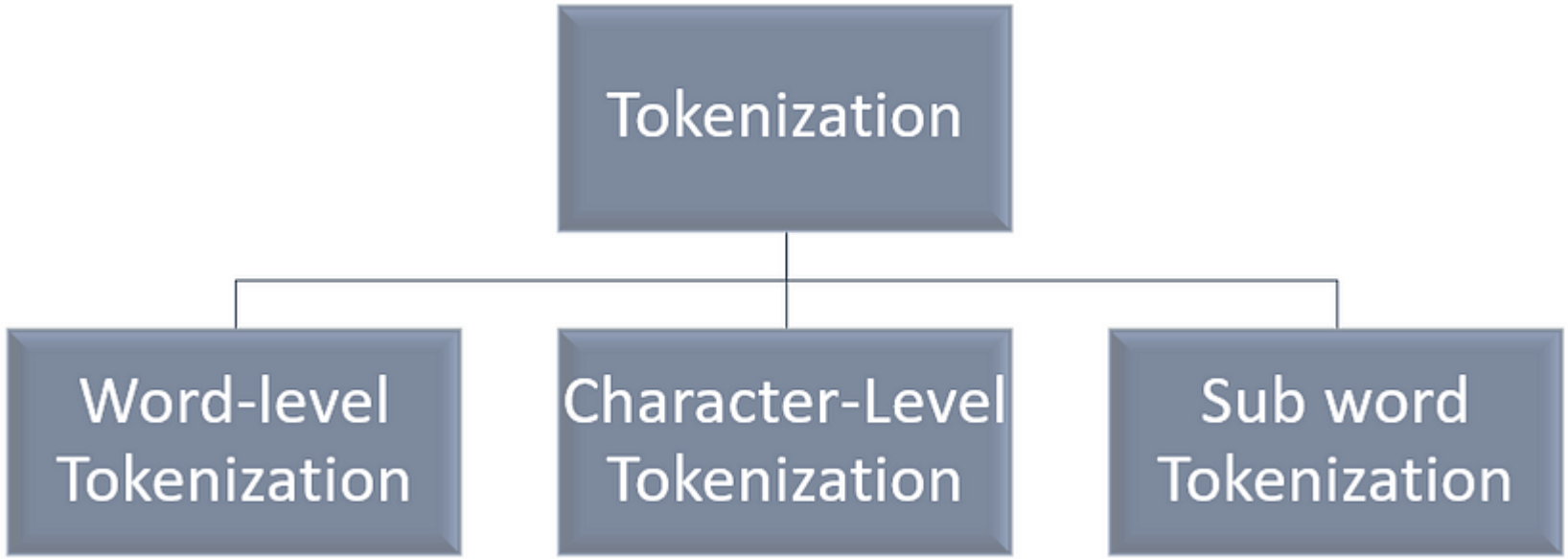


Tokenization

Tokenization



Tokenization



Tokenization Techniques

https://towardsdatascience.com/tokenization-for-natural-language-processing-a179a891bad4

Word Level Tokenization

- Splits the given sentence into words based on a certain delimiter

"She is smarter" becomes "she", "is", "smarter". Here the delimiter is space.

- Disadvantages :
  - When there are "Out of Vocabulary (OOV)" words. (One solution is to replace the rare words with an unknown token (UNK). In this case, vocabulary will only contain top k frequently occurring words, but the information about the new word will be lost)
  - The vocabulary size created will be huge, leading to memory and performance issues (one solution - switch to character level tokenization)
  - When splitting sentences based on whitespace and punctuation, there are problems handing words that are regarded as a single token but are separated by space or punctuation like don't, New York, etc

Word Tokenization

- How do we represent an input text?
- So far in this class… we chop it up into *words*

Input text: students opened their books

Input token IDs: ***11*** ***298*** ***34*** ***567***

This tokenization step requires an external *tokenizer* to detect word boundaries!

Word Tokenization

- Not as simple as split on whitespace and punctuation…

Mr. **O'Neill** thinks that the boys' stories about San Francisco **aren't** amusing.

- Word tokenizers require lots of specialized rules about how to handle specific inputs
  - Check out spaCy's tokenizers! (https://spacy.io/)

Handling unknown words

- What happens when we encounter a word at test time that we've never seen in our training data?
  - With word level tokenization, we have no way of assigning an index to an unseen word!
  - This means we don't have a word embedding for that word and thus cannot process the input sequence
- Solution: replace low-frequency words in training data with a special <UNK> token, use this token to handle unseen words at test time too
  - Why use <UNK> tokens during training?

Limitations of <UNK>

- We lose lots of information about texts with a lot of rare words / entities

The chapel is sometimes referred to as "Hen Gapel Lligwy" ("*hen*" being the Welsh word for "old" and "*capel*" meaning "chapel").

The chapel is sometimes referred to as " Hen <unk> <unk> " (" hen " being the Welsh word for " old " and " <unk> " meaning " chapel ").

Other limitations

- Word-level tokenization treats different forms of the same word (e.g., “open”, “opened”, “opens”, “opening”, etc) as separate types —> separate embeddings for each

This can be problematic especially when training over smaller datasets, why?

Character-level tokenization

❖Splits the given sentence into a sequence of characters.

“Smarter” becomes “s”, “m”, “a”, “r”, “t”, “e”, “r”.

❖Advantages :

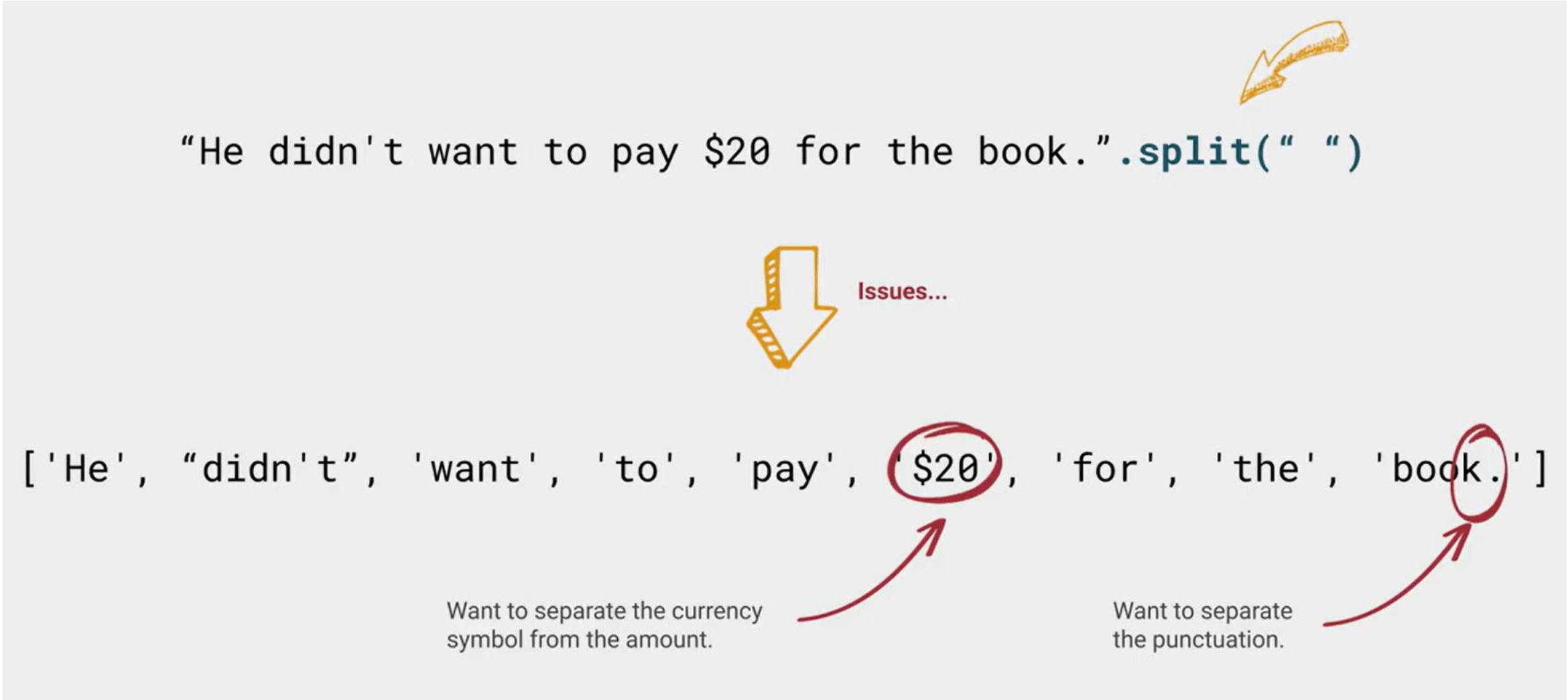
- ➢Smaller vocabulary size (26 alphabets + special characters etc)
- ➢Misspellings are handled

Subword tokenization:

- Splits the words into smaller pieces.

“Smarter” becomes “Smart”, “er”

Tokenizing English



Tokenizing English: Challenges

Tokenizing English: Challenges

What is a “word” anyway?

- Is “full moon” one word or two words?
- Does a word always map to one meaning? e.g. should contractions such as “don't” be split into two?

Word: the smallest unit of speech that carries some meaning on its own.

Related concepts:

- *Morpheme*: the smallest unit of speech which has a meaning, but doesn't necessarily stand on its own. Examples: -ing, re-, pre-, un-.
- *Grapheme*: the smallest functional unit of a writing system. In English, that's letters.

Tokenizing English: Libraries to use

You could roll your own tokenizer, but better to use a library instead.

(You can still customize as needed.)

• Tokenization

A word (*Token) is the minimal unit that a machine can understand and process.*
Tokenization is the process of splitting the raw string into meaningful tokens. A text could be tokenized into sentences and words. In sentence tokenization, each sentence is seen as a unit likewise in word tokenization.
Tokenization can come in different ways but the most common one is the word tokenization. In word tokenization texts are broken than into words which serves a the minimal unit. Tokenization modules in nltk include:

-word_tokenizer
- sent_tokenizer
-punkt_tokenizer
- Regexp_tokenizer
-TreebankWord_Tokenizer

Customized to tokenization could be done using regex_tokenize, Tokenization could also be done using split from regex
re.split('/W+', string).

In [7]:
```
from nltk.tokenize import word_tokenize
train["clean"] = train["Text"].apply(word_tokenize)
```

In [9]:
```
train['clean'].head(1)
```

Out[9]:
```
0    [centers, of, biomedical, research, excellence, cobre, phase, iii, tra
nsitional, centers, funding, opportunity, description, the, institutional,
development, award, idea, program, endeavors, to, stimulate, research, at,
institutions, in, states, that, have, not, traditionally, received, signifi
cant, levels, of, research, funding, from, the, nih, created, through, cong
ressional, mandate, the, idea, program, broadens, the, geographic, distribu
tion, of, nih, funding, for, competitive, biomedical, and, behavioral, rese
arch, by, enhancing, the, research, capabilities, of, institutions, in, eli
gible, states, the, idea, program, aims, to, achieve, this, goal, through,
two, major, initiatives, 1, the, idea, networks, of, biomedical, research,
excellence, inbre, and, 2, the, centers, of, biomedical, research, excellen
ce, cobre, ...]
Name: clean, dtype: object
```

In [ ]:

•Example Tokenization

This is the process of splitting large raw text into many pieces. A token is a minimal unit that a machine can understand.

```
>>>s = "Hi Everyone ! hola gr8" # simplest tokenizer
>>>print s.split()
['Hi', 'Everyone', '!', 'hola', 'gr8']
>>>from nltk.tokenize import word_tokenize
>>>word_tokenize(s)
['Hi', 'Everyone', '!', 'hola', 'gr8']
>>>from nltk.tokenize import regexp_tokenize, wordpunct_tokenize,
blankline_tokenize
>>>regexp_tokenize(s, pattern='\\w+')
['Hi', 'Everyone', 'hola', 'gr8']
>>>regexp_tokenize(s, pattern='\\d+')
['8']
>>>wordpunct_tokenize(s)
['Hi', 'Everyone', '!!', 'hola', 'gr8']
>>>blankline_tokenize(s)
['Hi, Everyone !! hola gr8']
```

Tweet Data: Pre-processing and Text Normalisation

A motivating example: Pre-processing and Text Normalization challenges from tweet data

- Consider the following tweets:
  - GOD's been very kind as HE didn't create FRIENDS with price tags.If He did, I really couldn't afford U **#HappyFriendshipDay** @KimdeLeon
  - Why are wishing friendship day today? Aren't we sppsd to be friends for all the 365 days?**#HappyFriendshipDay**
  - eVrYthin ChnGZ & nthin StaYs D SMe bt aS v grOw uP 1 thnG vl reMain i ws wT u b4& vL b tiLl D end **#HappyFriendshipDay** pic.twitter.com/Ym3ZAnFiFn
- How do we tokenize, normalize the above text?
  - Often the answer to the above is application dependent

Challenges with Social Data

1	i	17,071,657
2	-i	4,254
3	ijust	1,446
4	dontcha	872
5	ireally	705
6	d'you	527
7	whaddya	511
8	istill	477
9	//i	438
10	ijus	424
11	i-i	393
12	inever	362
13	#uever	347

1	haven't	170,431
2	havent	38,303
3	shoulda	14,114
4	would've	13,043
5	should've	12,997
6	hadn't	11,899
7	woulda	10,061
8	could've	7,435
9	coulda	6,009
10	havnt	5,227
11	shouldve	3,972
12	wouldve	3,752
13	must've	3,164
14	musta	2,159
15	couldve	2,142
16	haven't	1,282

1	love	1,908,093
2	luv	58,486
3	lovee	9,927
4	envy	9,668
5	salute	5,629
6	loveee	5,107
7	lov	3,625
8	dread	3,188
9	looove	3,109
10	cba	2,975
11	loveeee	2,932
12	loooove	2,740
13	loove	1,664
14	looooove	1,584
15	loveeeee	1,435
16	lurve	868
17	looooooove	849
18	l0ve	712
19	loveeeeee	667
20	luff	626
21	l.o.v.e	585
22	lovelovelove	583
23	luvv	558

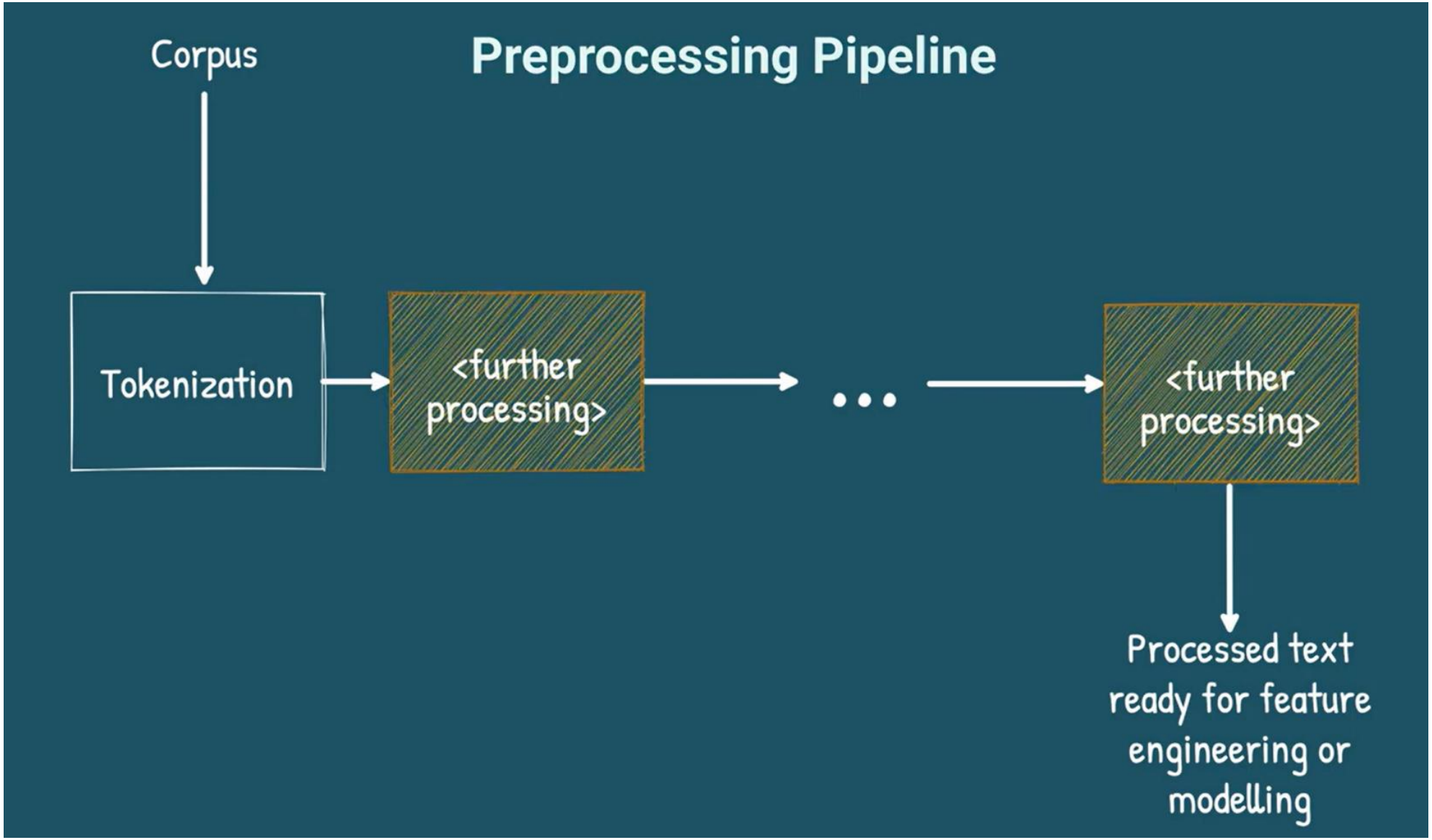
Some key challenges are:

- How do we tokenize words like: ireally, l0ve, #unever etc.
- How to handle tagging problems like: POS tagging, NE tagging and so on
- More broadly: How to address the traditional NLP core tasks and applications?

Tweet Data: Words

- How do we identify word boundaries and tokenize the words?
  - Don't, would've, b4 (if we have a rule/logic where we consider valid English words to consist only of alphabets [a-zA-Z], then the word boundary would break at the letter 4)
- Normalizing lowercase, uppercase
  - Some news headlines may be capitalized: "PM Modi Announces US $1 Billion Concessional Line of Credit to Nepal"
  - Some tweets may be in all capital letters
- Normalization to same word expressed with differing syntax
  - This is needed for applications like information retrieval so that the query string can be more appropriately matched with the content being searched - E.g, U.K and UK may be resolved to one term, similarly Mr and Mr., Ph.D, PhD etc
- How do we handle words that are distorted versions of a valid English word?
  - eVrYthin, ChnGZ, D, etc – if we have a vocabulary of a given corpus that has the terms everything, changes, the would we count the abbreviated terms same as the valid ones or would we consider them different?
  - "I am tooooooooo happppppppppppy" – If we correct these in to: "I am too happy" we may lose the implied and hidden emotion of the strength of happiness expressed in the phrase.
- How do we handle new words not part of the language?
  - Tweeple, selfie, bestie etc – consider a spell checker application that we would like to build. Would we correct these? Are these candidate correction words of some one's misspells, eg, Selfy instead of selfie?

Text Processing Pipeline



Some steps may also be there before Tokenization. E.g., for HTML content, removing HTML Tags can be one such step

Text Preprocessing: Case Folding

Lower- or upper-casing all tokens.

Vocabulary: The set of all **unique** tokens in a corpus.

"Mr. Cook went into the kitchen to cook dinner."

Without c.f.

{ cook, dinner, into, kitchen, mr., the, to, went, **Cook** }

With c.f.

{ cook, dinner, into, kitchen, mr., the, to, went }

Smaller vocabulary can be more efficient in space and computation, and lead to higher number of search hits ("recall") in information retrieval.

Information loss; "Cook" as a person is different from "cook" as an activity, which can lead to poorer quality search results ("precision").

Text Preprocessing: Stop Word Removal

Stop Word Removal

Removing words which occur frequently but carry little information.

{ the, a, of, an, this, that, ... }

Like case folding, this could lead to potential efficiency gains...

But whether we should do it depends on the task.

Text Preprocessing: Stop Word Removal

Removing stop words when topic modelling...

Python was created in the late 1980s, and first released in 1991, by Guido van Rossum as a successor to the ABC programming language. Python 2.0, released in 2000, introduced new features, such as list comprehensions, and a garbage collection system with reference counting, and was discontinued with version 2.7 in 2020.[29] Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows).

Remove stop words

Python created late 1980s, released 1991, Guido van Rossum successor ABC programming language. Python 2.0, released 2000, introduced features, list comprehensions, garbage collection system reference counting, discontinued version 2.7 2020.[29] Python 3.0, released 2008, major revision language completely backward-compatible Python 2 code unmodified Python 3. With Python 2's --, Python 3.6.x[30] supported, older versions supporting e.g. Windows 7 (old installers restricted 64- Windows).

Probably still can tell what this is about.

Text Preprocessing: Stop Word Removal

But removing stop words can affect context in other tasks...

e.g. Sentiment Analysis

"I saw the movie last night. I was not amused."

Remove stop words

"Saw movie night. amused."*

This is going to throw off a classifier.

Different NLP libraries have their own stop word lists. There is no Universal list of stop words.
People create their own based on the need.

Text Processing: Stemming

Text Preprocessing: Stemming

Mostly Hard Rules

Stemming

Removing word suffixes (and sometimes prefixes).

{ ing, s, y, ed, ... }

Goal is to reduce a word to some base form.

Typically done through an algorithm, the most famous of which is *Porter's algorithm*.

Stemming Examples

```
connection
connections
connective          --->    connect
connected
connecting
```

```
likes
liked
likely              --->    like
liking
```

```
Original words: ['running', 'jumps', 'happily', 'running', 'happily']

Stemmed words: ['run', 'jump', 'happili', 'run', 'happili']
```

Text Processing: Stemming

A *stemming* algorithm converts words to their root forms ("stems") in order to focus on their underlying meaning.

- It works well in English or in Arabic, with many nouns and verbs deriving from a common root
- It works worse in German or Turkish, which compose very long words with complex meanings.

It's common to add the root to the query's term vector (while leaving the unstemmed form present).

kitab	*a book*
kitabi	*my book*
al**kitab**	*the book*
kitabuki	*your book (f)*
kitabuka	*your book (m)*
kitabuhu	*his book*
kataba	*to write*
ma**ktaba**	*library, bookstore*
ma**ktab**	*office*

Arabic words that stem to **ktb**

Stemming Algorithms

Two major families of stemming algorithms exist:

- *Dictionary-based stemmers* use lists of related words.
- *Algorithmic stemmers* use some algorithm to derive related words.

A simple algorithmic stemmer for English may remove the suffix -s:

- cats → cat, lakes → lake, plays → play
- But many false negatives: supplies → supplie
- And some false positives: ups → up

Producing high quality rules is very challenging.

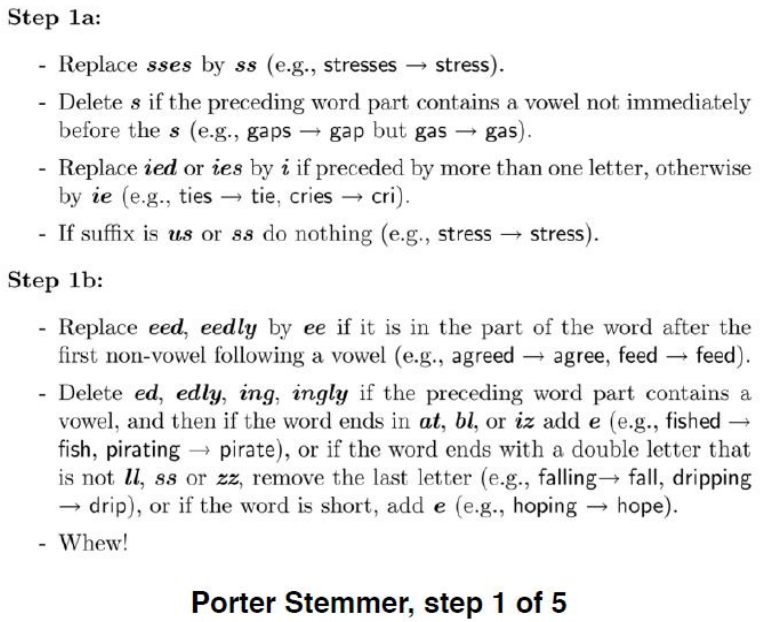
Multiple Stemming Algorithms: https://www.geeksforgeeks.org/introduction-to-stemming/

Porter Stemmer

The *Porter Stemmer* was developed in the 70's, and consists of a large series of rules to repeatedly apply until only the stem is left.

It is fairly effective, though makes many categorical errors. Its complexity makes it hard to modify, though the porter2 stemmer fixes some of its problems.

It outputs stems, not recognizable words.



Step 1a:

- Replace ***sses*** by ***ss*** (e.g., stresses → stress).
- Delete ***s*** if the preceding word part contains a vowel not immediately before the ***s*** (e.g., gaps → gap but gas → gas).
- Replace ***ied*** or ***ies*** by ***i*** if preceded by more than one letter, otherwise by ***ie*** (e.g., ties → tie, cries → cri).
- If suffix is ***us*** or ***ss*** do nothing (e.g., stress → stress).

Step 1b:

- Replace ***eed***, ***eedly*** by ***ee*** if it is in the part of the word after the first non-vowel following a vowel (e.g., agreed → agree, feed → feed).
- Delete ***ed***, ***edly***, ***ing***, ***ingly*** if the preceding word part contains a vowel, and then if the word ends in ***at***, ***bl***, or ***iz*** add ***e*** (e.g., fished → fish, pirating → pirate), or if the word ends with a double letter that is not ***ll***, ***ss*** or ***zz***, remove the last letter (e.g., falling→ fall, dripping → drip), or if the word is short, add ***e*** (e.g., hoping → hope).
- Whew!

Porter Stemmer, step 1 of 5

Original words: ['running', 'jumps', 'happily', 'running', 'happily']
Stemmed words: ['run', 'jump', 'happili', 'run', 'happili']

- **Advantage:** It produces the best output as compared to other stemmers and it has less error rate.
- **Limitation:** Morphological variants produced are not always real words.

Refer to slides: 68-79

Krovetz Stemmer

The *Krovetz Stemmer* is a hybrid of dictionary and algorithmic methods.

It first checks the dictionary. If not found, it tries to remove suffixes and then checks the dictionary again.

It produces recognizable words, unlike the Porter stemmer.

Its effectiveness is comparable to the Porter stemmer. It has a lower false positive rate, but somewhat higher false negative.

It was proposed in 1993 by Robert Krovetz. Following are the steps:
1) Convert the plural form of a word to its singular form.
2) Convert the past tense of a word to its present tense and remove the suffix 'ing'.
Example: 'children' -> 'child'

- **Advantage:** It is light in nature and can be used as pre-stemmer for other stemmers.

- **Limitation:** It is inefficient in case of large documents.

Snowball Stemmer

- Snowball Stemmer: It is a stemming algorithm which is also known as the Porter2 stemming algorithm as it is a better version of the Porter Stemmer since some issues of it were fixed in this stemmer.
- Some few common rules of Snowball stemming are:
- Few Rules:
  - ILY -----> ILI
  - LY -----> Nil
  - SS -----> SS
  - S -----> Nil
  - ED -----> E,Nil

- Nil means the suffix is replaced with nothing and is just removed. There may be cases where these rules vary depending on the words.
- As in the case of the suffix ‘ed’ if the words are ‘cared’ and ‘bumped’ they will be stemmed as ‘care‘ and ‘bump‘. Hence, here in cared the suffix is considered as ‘d’ only and not ‘ed’.
- One more interesting thing is in the word ‘stemmed‘ it is replaced with the word ‘stem‘ and not ‘stemmed‘. Therefore, the suffix depends on the word.

Examples

Word	Stem
cared	care
university	univers
fairly	fair
easily	easili
singing	sing
sings	sing
sung	sung
singer	singer
sportingly	sport

Text Preprocessing: Stemming

Should you stem?

Similar tradeoffs to other (simple) techniques which reduce vocabulary.

Reduces vocabulary size and generalizes your model to behave the same for words with the same stem. May also help with new words outside of the current vocabulary.

A stemmer can *overstem* ("university and "universe" both stem to "univers") and *understem* ("alumnus" and "alumni" stem to "alumnu" and "alumni") which can lead to poor results.

Search engines will search on both unstemmed and stemmed words and blend the results based on other relevancy metrics.

There's a better alternative to stemming...

•Stemming

- Stemming, in literal terms, is the process of cutting down the branches of a tree to its stem.
- With the use of some basic rules, any token can be cut down to its stem.
- Stemming is more of a crude rule-based process by which we want to club together different variations of the token.
- For example, the word *eat will have* variations like eating, eaten, eats, and so on.

http://www.nltk.org/api/nltk.stem.html

Stemming is the process of cutting down the branche of a tree to its stem. So basically stemming is the breaking down of a token to its basic form. A typical example is the use of eat in a sentence which has other variations like eating, eaten, eats, and so on.

In nltk we can use LancasterStemmer, SnowballStemmer, PorterStemmer, the major difference between the trio is that Lancaster algorithm is very aggressive so sometimes it over stem certain words while PorterStemmer is less aggressive. SnowballStemmer is language designed, it can be used in several languages. To import each of these stemmers you type from nltk.stem import PorterStemmer, LancasterStemmer, SnowballStemmer.

In the following snippet, we show a few stemmers:

```
>>>from nltk.stem import PorterStemmer # import Porter stemmer
>>>from nltk.stem.lancaster import LancasterStemmer
>>>from nltk.stem.Snowball import SnowballStemmer
>>>pst = PorterStemmer() # create obj of the PorterStemmer
>>>lst = LancasterStemmer() # create obj of LancasterStemmer
>>>lst.stem("eating")
eat
>>>pst.stem("shopping")
shop
```

In [18]:
```
train['Text'].head(1)
```

Out[18]:
```
0    centers of biomedical research excellence cobre phase iii transition
al centers funding opportunity description the institutional development
award idea program endeavors to stimulate research at institutions in sta
tes that have not traditionally received significant levels of research f
unding from the nih created through congressional mandate the idea progra
m broadens the geographic distribution of nih funding for competitive bio
medical and behavioral research by enhancing the research capabilities of
institutions in eligible states the idea program aims to achieve this goa
l through two major initiatives 1 the idea networks of biomedical researc
h excellence inbre and 2 the centers of biomedical research excellence co
bre the cobre initiative seeks to develop unique innovative multidiscipli
nary and collaborative state of the art biomedical and behavioral researc
h centers focused on scientific theme that is nascent or only minimally d
eveloped at applicant institutions this is accomplished by nurturing and
expanding critical mass of competitive biomedical research investigators
```

using SnowballStemmer:

```
from nltk.stem import SnowballStemmer
snstem = SnowballStemmer('english') #create object of SnowbllStemmer
train['stem']=train['clean'].apply(lambda x : [snstem.stem(y) for y in x])
```

```
train['stem'].head(1)
```

Out[14]:
```
0    [center, of, biomed, research, excel, cobr, phase, iii, transit, cente
r, fund, opportun, descript, the, institut, develop, award, idea, program,
endeavor, to, stimul, research, at, institut, in, state, that, have, not, t
radit, receiv, signific, level, of, research, fund, from, the, nih, creat,
through, congression, mandat, the, idea, program, broaden, the, geograph, d
istribut, of, nih, fund, for, competit, biomed, and, behavior, research, b
y, enhanc, the, research, capabl, of, institut, in, elig, state, the, idea,
program, aim, to, achiev, this, goal, through, two, major, initi, 1, the, i
dea, network, of, biomed, research, excel, inbr, and, 2, the, center, of, b
iomed, research, excel, cobr, ...]
Name: stem, dtype: object
```

Text Preprocessing: Lemmatization

Lemmatization

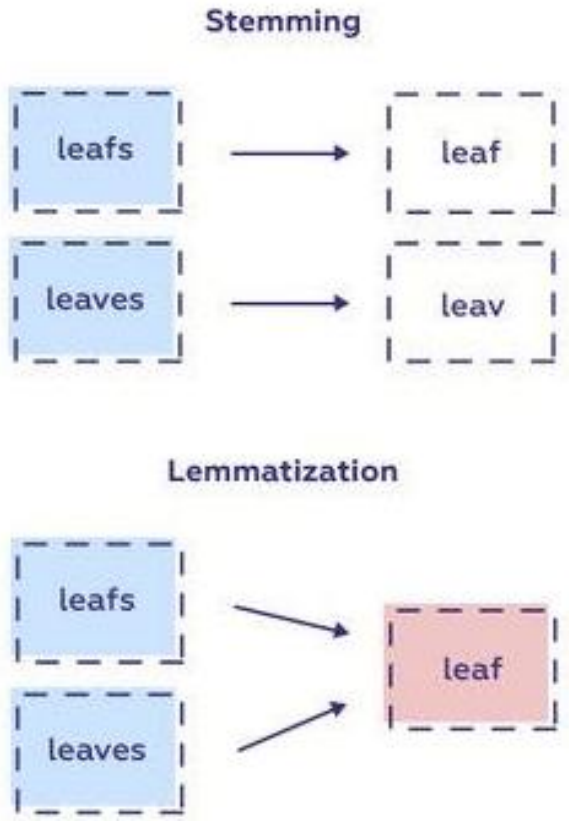
Reduce a word down to its *lemma*, or dictionary form.

(more sophisticated than stemming)

Also takes into account whether a word is a noun, verb, etc.

"Talkative" → lemmatizer → "Talkative"

"Talkative" → stemmer → "Talk"



The aim of lemmatization, like stemming, is to reduce inflectional forms to a common base form. As opposed to stemming, lemmatization does not simply chop off inflections. It uses lexical knowledge bases to get the correct base forms of words.

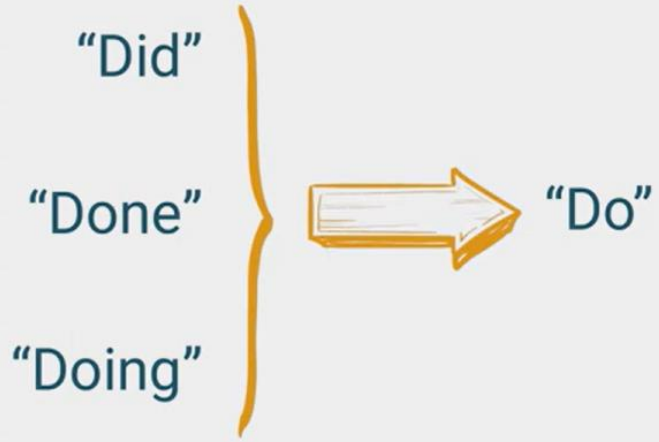
Text Preprocessing: Lemmatization

Lemmatization

Reduce a word down to its *lemma*, or dictionary form.

(more sophisticated than stemming)

Maps different forms (e.g. inflections) of a word (and some synonyms) to its base form.



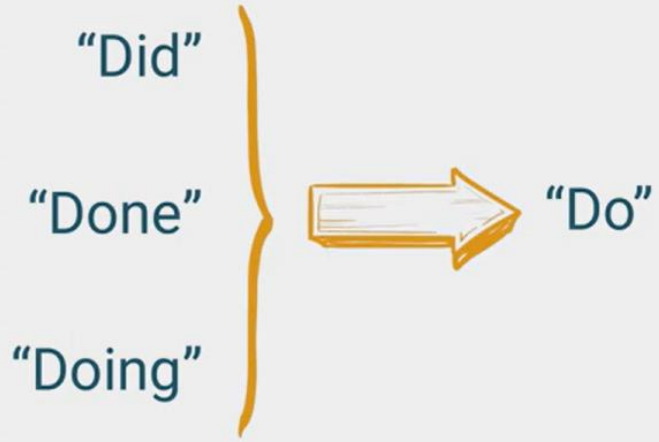
Text Preprocessing: Lemmatization

Lemmatization

Reduce a word down to its *lemma*, or dictionary form.

(more sophisticated than stemming)

Maps different forms (e.g. inflections) of a word (and some synonyms) to its base form.



Text Preprocessing: Lemmatization

Lemmatization

Reduce a word down to its *lemma*, or dictionary form.

(more sophisticated than stemming)

Tradeoffs...

Similar benefits to stemming but more accurate and is generally preferred. spaCy offers lemmatization but not stemming.

Lemmatization can be detrimental in cases where tenses matter (lemmatizer will normalize them away) or when subtle word usage matters (e.g. author identification). Also more resource-intensive than stemming.

•Lemmatization

•Lemmatization is a more methodical way of converting all the grammatical/inflected forms of the root of the word.
•Lemmatization uses context and part of speech to determine the inflected form of the word and applies different normalization rules for each part of speech to get the root word (*lemma)*

```
>>>from nltk.stem import WordNetLemmatizer
>>>wlem = WordNetLemmatizer()
>>>wlem.lemmatize("ate")
Eat
```

Here, WordNetLemmatizer is using wordnet, which takes a word and searches wordnet, a semantic dictionary. It also uses a morph analysis to cut to the root and search for the specific lemma (variation of the word). Hence, in our example it is possible to get *eat for the given variation ate, which was never possible with stemming.*

Lemmatization Example:

In [24]:
```
from nltk.stem import WordNetLemmatizer
lemma = WordNetLemmatizer()
train['stem']=train['clean'].apply(lambda x : [lemma.lemmatize(y) for y in
```

In [25]:
```
train['stem'].head(1)
```

Out[25]:
```
0    [center, of, biomedical, research, excellence, cobre, phase, iii, tran
sitional, center, funding, opportunity, description, the, institutional, de
velopment, award, idea, program, endeavor, to, stimulate, research, at, ins
titution, in, state, that, have, not, traditionally, received, significant,
level, of, research, funding, from, the, nih, created, through, congression
al, mandate, the, idea, program, broadens, the, geographic, distribution, o
f, nih, funding, for, competitive, biomedical, and, behavioral, research, b
y, enhancing, the, research, capability, of, institution, in, eligible, sta
te, the, idea, program, aim, to, achieve, this, goal, through, two, major,
initiative, 1, the, idea, network, of, biomedical, research, excellence, in
bre, and, 2, the, center, of, biomedical, research, excellence, cobre, ...]
Name: stem, dtype: object
```

•Stop word removal

•Stop word removal is one of the most commonly used preprocessing steps across different NLP applications.

•The idea is simply removing the words that occur commonly across all the documents in the corpus.

•Typically, articles and pronouns are generally classified as stop words. These words have no significance in some of the NLP tasks like information retrieval and classification, which means these words are not very discriminative.

•To implement the process of stop word removal, below is code that uses NLTK stop word.

```
>>>from nltk.corpus import stopwords
>>>stoplist = stopwords.words('english') # config the language name
# NLTK supports 22 languages for removing the stop words
>>>text = "This is just a test"
>>>cleanwordlist = [word for word in text.split() if word not in stoplist]
# apart from just and test others are stopwords
 [‘test']
```

Python commands to stop word removal:

```
>>> tokens = ['hello','java','python','the']


>>> from nltk.corpus import stopwords


>>> stop = set(stopwords.words('english'))


>>> clean_tokens = [tok for tok in tokens if len(tok.lower())>1 and (tok.lower() i


>>> clean_tokens
['hello', 'java', 'python']
```

•Rare word removal

•This is very intuitive, as some of the words that are very unique in nature like names, brands, product names, and some of the noise characters, such as html leftouts, also need to be removed for different NLP tasks.
•For example, it would be really bad to use names as a predictor for a text classification problem, even if they come out as a significant predictor.

We also use length of the words as a criteria for removing words with very a short length or a very long length:

```
>>># tokens is a list of all tokens in corpus
>>>import nltk
>>>tokens=
['hi','i','am','am','whatever','this','is','just','a','test','test','java','python','java']
>>>freq_dist = nltk.FreqDist(token)
>>>rarewords = freq_dist.keys()[-50:]
>>>after_rare_words = [ word for word in token not in rarewords]
```

•Spell correction

It is not a necessary to use a spellchecker for all NLP applications, but some use cases require you to use a basic spell check. We can create a very basic spellchecker by just using a dictionary lookup. There are some enhanced string algorithms that have been developed for fuzzy string matching. One of the most commonly used is edit-distance. NLTK also provides you with a variety of metrics module that has edit_distance.

```
>>>from nltk.metrics import edit_distance
>>>edit_distance("rain","shine")
3
```

rain → sain (substitution of “s” for “r”)
sain → shin (substitution of “h” for “a”)
shin → shine (insertion of “e” at the end)

https://michael-fuchs-python.netlify.app/2021/06/16/nlp-text-pre-processing-vi-word-removal/

https://medium.com/machine-intelligence-team/text-wrangling-and-cleansing-with-nltk-8e55fa25c28b

https://likegeeks.com/nlp-tutorial-using-python-nltk/#Parse_trees

References

- NLP Regular Expressions Basics: by Data Camp (Maria Eugenia Inzaugarat, Data Scientist
  - https://s3.amazonaws.com/assets.datacamp.com/production/course_17118/slides/chapter1.pdf
  - https://s3.amazonaws.com/assets.datacamp.com/production/course_17118/slides/chapter2.pdf
  - https://s3.amazonaws.com/assets.datacamp.com/production/course_17118/slides/chapter3.pdf
  - https://s3.amazonaws.com/assets.datacamp.com/production/course_17118/slides/chapter4.pdf
- https://course.khoury.northeastern.edu/cs6200sp15/slides/m02.s10%20-%20stemming.pdf
- https://www.geeksforgeeks.org/introduction-to-stemming/
- https://www.slideshare.net/ananth/natural-language-processing-l02-words

Regular Expressions

Really clever “wild card” expressions for matching and parsing strings

http://en.wikipedia.org/wiki/Regular_expression

Really smart “Find” or “Search”

Regular Expression Quick Guide

^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace
*	Repeats a character zero or more times
*?	Repeats a character zero or more times (non-greedy)
+	Repeats a character one or more times
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(	Indicates where string extraction
)	Indicates where string extraction

String Manipulation

- **String manipulation**
  - e.g. replace and find specific substrings
- **String formatting**
  - e.g. interpolating a string in a template
- **Basic and advanced regular expressions**
  - e.g. finding complex patterns in a string

Strings

- Sequence of characters
- Quotes

```
my_string = "This is a string"
my_string2 = 'This is also a string'
```

```
my_string = 'And this? It's the wrong string'
```

```
my_string = "And this? It's the correct string"
```

String Manipulation

More strings

- Length

```
my_string = "Awesome day"
len(my_string)
```

```
11
```

  - Convert to string

```
str(123)
```

```
'123'
```

Concatenation

- Concatenate: `+` operator

```
my_string1 = "Awesome day"
my_string2 = "for biking"
```

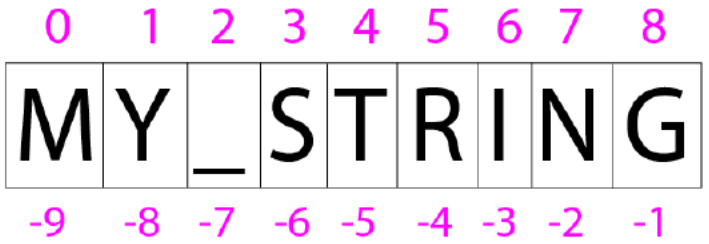
```
print(my_string1+" "+my_string2)
```

```
Awesome day for biking
```

String Manipulation: Indexing

Indexing

- Bracket notation




```
my_string = "Awesome day"
```

```
print(my_string[3])
```

```
s
```

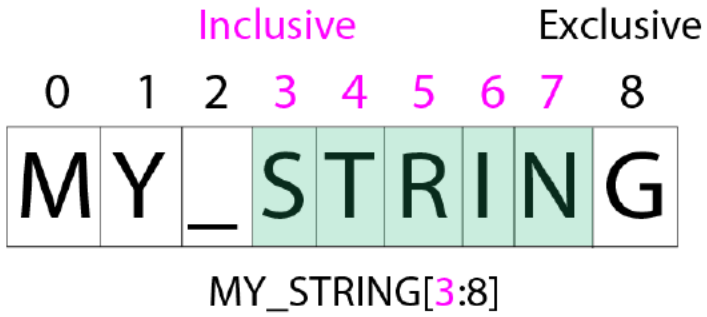
```
print(my_string[-1])
```

```
y
```

String Manipulation: Slicing

Slicing

- Bracket notation




```
my_string = "Awesome day"
print(my_string[0:3])
```

```
Awe
```

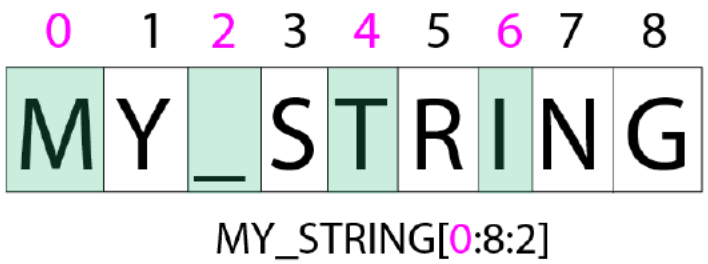
```
print(my_string[:5])
print(my_string[5:])
```

```
Aweso
me day
```

String Manipulation: Stride

Stride

- Specifying stride




```
my_string = "Awesome day"
print(my_string[0:6:2])
```

```
Aeo
```

```
print(my_string[::-1])
```

```
yad emosewA
```

String Manipulation: Adjusting Cases

Adjusting cases

```
my_string = "tHis Is a niCe StriNg"
```

- Converting to lowercase

```
print(my_string.lower())
```

```
this is a nice string
```

- Converting to uppercase

```
print(my_string.upper())
```

```
THIS IS A NICE STRING
```

```
my_string = "tHis Is a niCe StriNg"
```

- Capitalizing the first character

```
print(my_string.capitalize())
```

```
This is a nice string
```

String Manipulation: Splitting

Splitting

```
my_string = "This string will be split"
```

- Splitting a string into a list of substrings

```
my_string.split(sep=" ", maxsplit=2)
```

```
['This', 'string', 'will be split']
```

```
my_string.rsplit(sep=" ", maxsplit=2)
```

```
['This string will', 'be', 'split']
```

The rsplit() method splits a string into a list, starting from the right.
If no "max" is specified, this method will return the same as the split() method.
Note: When maxsplit is specified, the list will contain the specified number of elements plus one.

```
my_string = "This string will be split\nin two"
print(my_string)
```

```
This string will be split
in two
```

Escape Sequence	Character
\n	Newline
\r	Carriage return

"This string will be split\nin two"

- Breaking at line boundaries

```
my_string = "This string will be split\nin two"
```

```
my_string.splitlines()
```

```
['This string will be split', 'in two']
```

Porter Stemming Algorithm

The Porter algorithm

- The Porter algorithm consists of a set of condition/action rules.

- The condition fall into three classes
  - Conditions on the stem
  - Conditions on the suffix
  - Conditions on rules

Conditions on the stem

1.The measure , denoted m ,of a stem is based on its alternate vowel-consonant sequences.

$$[C](VC)^{m}[V]$$

Measure	Example
M=0	TR,EE,TREE,Y,BY
M=1	TROUBLE,OATS,TREES,IVY
M=2	TROUBLES,PRIVATE,OATEN

Conditions on the stem (con't)

2.*<X> ---the stem ends with a given letter X

3.*v*---the stem contains a vowel

4.*d ---the stem ends in double consonant

5.*o ---the stem ends with a consonant-vowel-consonant,sequence ,where the final consonant is not w, x or y

Suffix conditions take the form: (current_suffix == pattern)

Conditions on the rules

- The rules are divided into steps. The rules in a step are examined in sequence , and only one rule from a step can apply

```
{       step1a(word);
        step1b(stem);
                if (the second or third rule of step 1b was used)
                        step1b1(stem);
        step1c(stem);
        step2(stem);
        step3(stem);
        step4(stem);
        step5a(stem);
        step5b(stem);
}
```

Step 1a Rules

Conditions	Suffix	Replacement	Examples
NULL	sses	ss	caresses -> caress
NULL	ies	i	ponies -> poni ties -> tie
NULL	ss	ss	caress -> caress
NULL	s	NULL	cats -> cat

Step 1b Rules

Conditions	Suffix	Replacement	Examples
(m>0)	eed	ee	feed -> feed agreed -> agree
(*v*)	ed	NULL	plastered -> plaster bled -> bled
(*v*)	ing	NULL	motoring -> motor sing -> sing

Step 1b1 Rules

Conditions	Suffix	Replacement	Examples
NULL	at	ate	conflat(ed) -> conflate
NULL	bl	ble	troubl(ing) -> trouble
NULL	iz	ize	siz(ed) -> size
(*d and not (*<L> or *<S> or *<Z>))	NULL	single letter	hopp(ing) -> hop tann(ed) -> tan fall(ing) -> fall hiss(ing) -> hiss fizz(ed) -> fizz
(m=1 and *o)	NULL	e	fail(ing) -> fail fil(ing) -> file

Step 1c Rules

Conditions	Suffix	Replacement	Examples
(*v*)	y	i	happy -> happi sky -> sky

Step 2 Rules

Conditions	Suffix	Replacement	Examples
(m>0)	ational	ate	relational -> relate
(m>0)	tional	tion	conditional -> condition rational -> rational
(m>0)	enci	ence	valenci -> valence
(m>0)	anci	ance	hesitanci -> hesitance
(m>0)	izer	ize	digitizer -> digitize
(m>0)	abli	able	conformabli -> conformable
(m>0)	alli	al	radicalli -> radical
(m>0)	entli	ent	differentli -> different
(m>0)	eli	e	vileli -> vile
(m>0)	ousli	ous	analogousli -> analogous
(m>0)	ization	ize	vietnamization -> vietnamize
(m>0)	ation	ate	predication -> predicate
(m>0)	ator	ate	operator -> operate
(m>0)	alism	al	feudalism -> feudal
(m>0)	iveness	ive	decisiveness -> decisive
(m>0)	fulness	ful	hopefulness -> hopeful
(m>0)	ousness	ous	callousness -> callous
(m>0)	aliti	al	formaliti -> formal
(m>0)	iviti	ive	sensitiviti -> sensitive
(m>0)	biliti	ble	sensibiliti -> sensible

Step 3 Rules

Conditions	Suffix	Replacement	Examples
(m>0)	icate	ic	triplicate -> triplic
(m>0)	ative	NULL	formative -> form
(m>0)	alize	al	formalize -> formal
(m>0)	iciti	ic	electriciti -> electric
(m>0)	ical	ic	electrical -> electric
(m>0)	ful	NULL	hopeful -> hope
(m>0)	ness	NULL	goodness -> good

Step 4 Rules

Conditions	Suffix	Replacement	Examples
(m>1)	al	NULL	revival -> reviv
(m>1)	ance	NULL	allowance -> allow
(m>1)	ence	NULL	inference -> infer
(m>1)	er	NULL	airliner -> airlin
(m>1)	ic .	NULL	gyroscopic -> gyroscop
(m>1)	able	NULL	adjustable -> adjust
(m>1)	ible	NULL	defensible -> defens
(m>1)	ant	NULL	irritant -> irrit
(m>1)	ement	NULL	replacement -> replac
(m>1)	ment	NULL	adjustment -> adjust
(m>1)	ent	NULL	dependent -> depend
(m>1 and (*<S> or *<T>))	ion	NULL	adoption->adopt
(m>1)	ou	NULL	homologou->homolog
(m>1)	ism	NULL	communism->commun
(m>1)	ate	NULL	activate->activ
(m>1)	iti	NULL	angulariti->angular
(m>1)	ous	NULL	homologous ->homolog
(m>1)	ive	NULL	effective->effect
(m>1)	ize	NULL	bowdlerize->bowdler

Step 5a Rules

Conditions	Suffix	Replacement	Examples
(m>1)	e	NULL	probate -> probat rate -> rate
(m=1 and not *o)	e	NULL	cease -> ceas

Step 5b Rules

Conditions	Suffix	Replacement	Examples
(m>1 and *d and *<L>)	NULL	single letter	controll -> control roll -> roll