Session 3: Working with Text Data

2021 July 19

Dr. Richard M. Crowley
rcrowley@smu.edu.sg
http://rmc.link/

Main application

Application 1: Analyze 1 annual report

- Citigroup in 2014

> We will expand to the full year of annual reports in Session 4

- Import text
- Pattern matching
- Counts
- Sentiment
- Grammar parsing
  - Grammar
  - Named entities

Application 2: Scraping data from FASB

- Idea: Scrape FASB speeches

- Unlike SEC filings, FASB speeches are not nicely indexed

- Use python + Beautiful Soup to determine where speeches are located on the FASB webpage

Getting started with text

Special characters in python

- `\t` is tab
- `\r` is newline (files from Macs)
- `\r\n` is newline (files from Windows)
- `\n` is newline (files from *nix-based systems)
    - This is the usual convention used in data sets
- `\'` is an explicit single quote – it always works
    - E.g., `'\'Single\''` works, though so would `"'Single'"`
- `\"` is an explicit double quote – it always works
    - E.g., `"\"Double\""` works, though so would `'"Double"'`
- `\\` is a backslash
    - Since `\` is used to denote special characters, it would be ambiguous to allow a single backslash

> In some contexts, the following are also special: `. ^ $ * + ? {} [] | ()`

Defining a string

1. Use single quotes

```python
print('This is a string')
```

```
## This is a string
```

2. Use double quotes

```python
print("This is also a string")
```

```
## This is also a string
```

Defining a string

3. Multi-line strings: Triple quoting with either `'''` or `"""`

```
print("""This is a multi-line
string since it has triple quotes""")
```

```
## This is a multi-line
## string since it has triple quotes
```

4. Multi-line strings: use a `\n` instead

```
print('This is also two lines\nsince it has a newline')
```

```
## This is also two lines
## since it has a newline
```

Importing a single text file

- Two ways to read a file:

#1: Open the file, read it, close it

```python
f = open('../Data/0001104659-14-015152.txt', 'rt')
text = f.read()
f.close()
```

- A more proper variant:

```python
f = open('../Data/0001104659-14-015152.txt', 'rt')
try:
    text = f.read()
finally:
    f.close()
```

- The `finally:` part ensures the file is closed even if there is an error reading it

Importing a single text file

#2: Using a context manager (e.g., a `with` statement) [Better approach!]

```
with open('../Data/0001104659-14-015152.txt', 'rt') as f:
    text = f.read()
```

- Guarantees the file gets closed properly
- A bit more readable as well
- This is the preferred approach when possible

Finding text by location

```
print(text[9895:9929])
```

```
## Citis net income was $13.5 billion
```

```
print(wrap.fill(text[28899:29052]))
```

```
## Net income decreased 14%, mainly driven by lower revenues and lower loan loss
## reserve releases, partially offset by lower net credit losses and expenses.
```

- What is the `wrap.fill()` above? It is from python's built-in `textwrap` library

```
import textwrap
wrap = textwrap.Textwrap(width=80)
```

Commonly used text functions

1. Convert anything to a string with `str( )`

```
x = 72
x_string = str(x)
x_string
```

```
## '72'
```

2. Combining text with +

```
'Hello' + ' ' + 'world'
```

```
## 'Hello world'
```

Commonly used text functions

3. Casing text with `.lower()`, `.upper()`, and `.title()`

```python
print('soon TO be UPPERCASE'.upper())
```

```
## SOON TO BE UPPERCASE
```

```python
print('SOON tO be lowercase'.lower())
```

```
## soon to be lowercase
```

```python
print('soon to be titlecase'.title())
```

```
## Soon To Be Titlecase
```

Commonly used text functions

4. Checking if text contains something particular

```python
x = 'What is in this string?'

[x.startswith('What'), x.startswith('this')]
```

```
## [True, False]
```

```python
[x.endswith('string?'), x.endswith('string')]
```

```
## [True, False]
```

```python
['this' in x, 'ing' in x, 'zzz' in x]
```

```
## [True, True, False]
```

In python, `in` is an operator much like $>$ or $<$. It indicates if the LHS is contained in the RHS, working on strings or lists!

Commonly used text functions

5. Finding *where* the content is
   - '.find()' returns `-1` if your query isn't found
   - '.index()' works the same as `.find()`, except it gives an error if your query isn't found

```
x = 'What is in this string?'
[x.find('this'), x.find('ing'), x.find('zzz')]
```

```
## [11, 19, -1]
```

```
for y in ['this', 'ing', 'zzz']:
    try:
        print(x.index(y))
    except:
        print('Error!')
```

```
## 11
## 19
## Error!
```

Commonly used text functions

6. Counting the number of occurrences of a word or phrase
   - Can only check 1 phrase at a time
   - There are more efficient ways to check this for a list of words

```python
print('Mentions of SEC: ' + str(text.count('SEC')))
```

```
## Mentions of SEC: 33
```

```python
print('Mentions of FASB: ' + str(text.count('FASB')))
```

```
## Mentions of FASB: 33
```

```python
print('Mentions of Citi: ' + str(text.count('Citi')))
```

```
## Mentions of Citi: 2248
```

Commonly used text functions

7. Splitting strings

```
x = '1,2,3,4,5'.split(',')
print(x)
```

```
## ['1', '2', '3', '4', '5']
```

8. Joining strings together

```
print(' & '.join(x))
```

```
## 1 & 2 & 3 & 4 & 5
```

Joining strings is very useful when working with a list of data

Commonly used text functions

9. Replacing string content

```python
x = 'I like mee goreng with mutton and mee goreng with chicken'
print(x.replace('mee', 'nasi'))
```

```
## I like nasi goreng with mutton and nasi goreng with chicken
```

```python
print(x.replace('mee', 'nasi', 1))
```

```
## I like nasi goreng with mutton and mee goreng with chicken
```

> `.replace()` has two required arguments (what to replace, replacement), and an optional argument (how many times to replace, default: infinite)

Commonly used text functions

10. Removing blank content
    - Nice functions for keeping text clean

```python
x = '    this is awkwardly padded        '
print([x.strip(), x.lstrip(), x.rstrip()])
```

```
## ['this is awkwardly padded', 'this is awkwardly padded        ', '    this is awkwardly padded']
```

11. Padding strings
    - This is particularly useful when working with databases that zero-pad keys

```python
gvkey = 1024
gvkey = str(gvkey).zfill(6)
print(gvkey)
```

```
## 001024
```

Commonly used text functions

12. Checking if strings are a certain type

```python
output = '\t'.join(['input', 'alnum', 'alpha', 'decimal', 'digit', 'numeric', 'ascii'])
for x in ['ABC123', 'AAABBB', '12345', '12345²', '12345½', '123.1', '£12.0']:
  output += '\n' + '\t'.join(map(str,[x, x.isalnum(), x.isalpha(), x.isdecimal(), x.isdigit(), x.isnumeric(), x.isascii()]))
print(output)
```

```
## input    alnum   alpha   decimal digit   numeric ascii
## ABC123   True    False   False   False   False   True
## AAABBB   True    True    False   False   False   True
## 12345    True    False   True    True    True    True
## 12345²   True    False   False   True    True    False
## 12345½   True    False   False   False   True    False
## 123.1    False   False   False   False   False   True
## £12.0    False   False   False   False   False   False
```

- If you want a match on an explicit set of characters, using a regular expression is likely more intuitive

Addendum: Using R

- The `read_file()` function from `tidyverse`'s `readr` package works well.

```r
library(tidyverse)
# Read text from a .txt file using read_file()
doc <- read_file("../../Data/0001104659-14-015152.txt")
# str_wrap is from stringr from tidyverse
cat(str_wrap(substring(doc,1,500), 80))
```

```
## UNITED STATES SECURITIES AND EXCHANGE COMMISSION WASHINGTON, D.C. 20549 FORM
## 10-K ANNUAL REPORT PURSUANT TO SECTION 13 OR 15(d) OF THE SECURITIES EXCHANGE
## ACT OF 1934 For the fiscal year ended December 31, 2013 Commission file number
## 1-9924 Citigroup Inc. (Exact name of registrant as specified in its charter)
## Securities registered pursuant to Section 12(b) of the Act: See Exhibit 99.01
## Securities registered pursuant to Section 12(g) of the Act: none Indicate by
## check mark if the registrant is a
```

- For string manipulation, I recommend using the `stringr` library
  - The functions have more readable syntax and are `dplyr`-friendly

Exercise

> Do Set 1 in the `Session 3-Exercises` file

- Focuses on useful functions applied to to:
  - A paragraph from a JPMorgan 10-K
  - Some illustrative text

Regular expressions

A motivating example

- Suppose you want to find all emails mentions in the 10-K
- Emails follow a consistent pattern:
  1. A local name
  2. An `@` sign
  3. A domain, which will have at least 1 `.` in it
- Local names can have almost any character in them, except whitespace
  - In python regular expressions, we match this with `\S+`
- Domain names should usually be alphanumeric with 1 or more `.` in them
  - In python regular expressions, we match this with `[\w-]++\.[.\w-]+`

```python
import re
```

```python
re.findall('\S+@[\w-]+\.[.\w-]+', text)
```

```
## ['shareholder@computershare.com', 'shareholder@computershare.com', 'docserve@citi.com', 'shareholderrelations@citi.com']
```

Breaking down the example

- `@` was itself – it isn't a special character
- `\.` is a literal period
- `\S` is a special character
  - It matches any character that is not whitespace
- `+` is used to indicate that we want *at least* 1 of the pattern immediately preceding the `+`
  - Regular expressions are *greedy* by default, meaning they will choose the longest matching text
- Square brackets, `[ ]`, ask for any of the included elements
  - You don't need to escape most special characters in these
    - Exception: `^` if it is the first character
    - Optional for now: `--`, `&&`, `~~`, `||` – these may need escaping in a future version of python and will raise a warning (`FutureWarning`) if not escaped

Breaking down the example

- Let's examine the output `shareholder@computershare.com`
- Our regex was `\S+\@[\w-]+\.[.\w-]+`
- Matching regex components to output:
  - `\S+` ⇒ `shareholder`
  - `@` ⇒ `@`
  - `[\w-]+` ⇒ `computershare`
  - `\\.` ⇒ `.`
  - `[.\w-]+` ⇒ `com`

Calling regexes

- 3 most useful functions to call regexes
  1. `re.findall()`
     - Finds all occurences of your pattern and provides them back in a list
     - If you just want the count, apply `len()` to the list
  2. `re.sub()`
     - Use this for complex substitutions that are too much for `.replace()`
  3. `re.split()`
     - Use this for complex splits that are too much for `.split()`

Useful components

- `.` matches anything
- `\w` matches all characters that could be in a word
  - Except – and including _
- `\S` matches any non-whitespace characters
- `\s` matches any whitespace characters
- `\b` matches the start or end of a word
  - It is the boundary between `\S` and `\s`
  - Useful for matching whole words
- `\B` matches anything except the end of a word
- `^` or `\A` match the beginning of a string
  - Note: in *multiline* mode, `^` become the beginning of a line
- `$` or `\Z` match the end of a string
  - Note: in *multiline* mode, `$` become the end of a line

Useful patterns

- `[ ]` matches anything inside of it, like an "or" for regex
- `[^ ]` matches anything *except* for what is inside it
- Quantity specification (they always try to get the most text possible)
  - `x?` looks for 0 or 1 of x
  - `x*` looks for 0 or more of `x`
  - `x+` looks for 1 or more of `x`
  - `x{n}` looks for `n` (a number) of `x`
  - `x{n, }` looks for at least `n` of `x`
  - `x{n,m}` looks for at least `n` and at most `m` of `x`
  - To make any of the above non-greedy, append a `?` to them, like `x+?`

Complex patterns: Groups

- `( )` can be used to make groups
  - You can call for explicit matches of groups using a slash number:
    - `([0-9]).+\\1` Will match a number, followed by anything up until it hits that number again
  - By default, groups are capturing, meaning that the regex will only return the group text
  - There are two solutions:
    1. Put a group around the whole regex
    2. If you don't need to reference the group, use a non-capturing group with `(?: )`

```
re.findall('([0-9]).+\\1', '12asda2asd')
```

```
## ['2']
```

```
re.findall('(([0-9]).+\\2)', '12asda2asd')
```

```
## [('2asda2', '2')]
```

```
re.findall('(?:12|sd)a', '12asda2asd')
```

```
## ['12a', 'sda']
```

Complex patterns: Looking assertions

- Sometimes you want text that was preceded or followed by something, but don't want that something in the output
- `(?=...)` provides a lookahead where the `...` must be next in the string, but won't output
- `(?!...)` provides a negative lookahead; if the `...` is next in the string, the match won't count
- `(?<=...)` provides a lookbehind, while `(?<!...)` provides a negative lookbehind

```
re.findall('(?<=\.)[0-9]+', '1 2.3 4. 5 6.78')
```

```
## ['3', '78']
```

Pros and cons of regexes

Positives

- Very flexible, can match almost any pattern
  - E.g., finding the MD&A of a 10-K
- Allows us to find text directly rather than just indices
- Built in to python already

Negatives

- Regexes can be quite slow to run
- Complex regexes are hard to read

Extra info

- Regexes can run in other modes rather than just the default
  - These can be passed using the `re flags` parameter, or by using shorthand in your regex itself
- Ignore case with `re.IGNORECASE` or `(?i)`
- Convert UTF to ASCII for matching with `re.ASCII` or `(?a)`
- Run regexes across multiple lines using `re.MULTILINE` or `(?m)`
- Make `.` match newlines using `re.DOTALL` or `(?s)`
- Write better documented regular expressions using `re.VERBOSE` or `(?x)`

Full documentation here

Addendum: Using R

- The same `stringr` library from earlier handles these well as well
- Note that while the overall pattern structure is the same in R…
  - The special characters are often different
- There's a nice cheat sheet here
  - More detailed documentation here

Exercise

> Do Set 2 in the `Session 3-Exercises` file

- Focuses on extracting content using regexes
  - Data: The same paragraph from a JPMorgan 10-K

Using NLP parsers

NLTK

- `NLTK` stands for Natural Language Toolkit
- It provides a bunch of handy things for text analytics
  1. Corpora that are used in research and algorithm development
     - Tagged corpora are particularly valuable
  2. Models for things like dependency parsing
  3. Useful functions for working with text

Setting up NLTK

- When using a resource from `NLTK`, we will often have install needed datasets
- For instance, to run the word tokenizer on the next slide, we will need to install `punkt`
- We can install this using:

```python
nltk.download('punkt')
```

- We will also need the following:

```python
nltk.download('stopwords')
```

Tokenizing

```python
tokens = nltk.tokenize.word_tokenize(text)
print(tokens[0:50])
```

```
## ['UNITED', 'STATES', 'SECURITIES', 'AND', 'EXCHANGE', 'COMMISSION', 'WASHINGTON', ',', 'D.C.', '20549', 'FORM', '10-K', 'AN
```

```
## ['UNITED', 'STATES', 'SECURITIES', 'AND', 'EXCHANGE', 'COMMISSION', 'WASHINGTON', ',', 'D.C.', '20549']
##  ['FORM', '10-K', 'ANNUAL', 'REPORT', 'PURSUANT', 'TO', 'SECTION', '13', 'OR', '15']
##  ['(', 'd', ')', 'OF', 'THE', 'SECURITIES', 'EXCHANGE', 'ACT', 'OF', '1934']
##  ['For', 'the', 'fiscal', 'year', 'ended', 'December', '31', ',', '2013', 'Commission']
##  ['file', 'number', '1-9924', 'Citigroup', 'Inc.', '(', 'Exact', 'name', 'of', 'registrant', 'as', 'specified', 'in', 'its'
```

Stop words

- There are words in text that are grammatically needed, but often provide little information
  - E.g.: the, of, an
- We will manually remove ‘no’ and ‘not’ from the stopword list, however
  - These are potentially useful in our context

```
# If you get an error that you are missing 'stopwords', run: nltk.download('stopwords')
stop_words = set(nltk.corpus.stopwords.words("english"))
stop_words.remove('no')
stop_words.remove('not')

filtered_tokens = [t.lower() for t in tokens if t.lower() not in stop_words]
print(stop_words)
```

```
## {'until', 'shouldn', 'have', 'few', 're', 'ma', 'does', 'their', 'am', 'these', "that'll", 'hasn', 'them', "wouldn't", 'the
```

Stemming and lemmatization

- Stemming and lemmatization are ways to cut down on the amount of unique words in a data set
- Stemming just normalizes words by removing suffixes
  - Often this works correctly
  - Sometimes words behave a bit oddly in English and this doesn't work
    - E.g., 'are' is the 3rd person plural form of 'is'
- Stemming can be done using `nltk.stem.PorterStemmer()`
- Lemmatization is the same concept, but based on grammar parsing
  - It is more accurate than stemming, but slower
- Lemmatization can be done using `nltk.stem.wordnet.WordNetLemmatizer()`

> We will look at an example of this in Session 4

How does dictionary sentiment work

1. Sentiment dictionaries provide a list of words
2. You add up the number of times the words in your text are in the dictionary
3. Sometimes there is some normalization applied
   - Divide by the number of words
   - Optional: apply TF-IDF weighting

We need a reliable way to count a lot of words

Counting words

- Next, we'll do a quick sentiment measure
- First, we need to import the dictionary

```
with open('../../Data/S3_LM_Neg.csv', 'rt') as f:
    neg = [x.strip().lower() for x in f.readlines()]
print(neg[0:5])
```

```
## ['abandon', 'abandoned', 'abandoning', 'abandonment', 'abandonments']
```

- We will also take this time to sum up the total number of words we have

```
n_tokens = len(filtered_tokens)
```

Counters

- To count our document and make it easier to calculate score on, we will use `Counter()`
  - This is like a dictionary, except it automatically counts the input list and it returns 0 for missing items.
- Counters are a great general-purpose tool for counting things

```
from collections import Counter

token_count = Counter(filtered_tokens)
print(token_count.most_common(10))
```

```
## [(',', 7263), ('.', 4454), ('(', 1224), (')', 1224), ('citi', 798), ('$', 770), ('2013', 737), ('credit', 660), ('citis', 6
```

- To determine the amount of sentiment we can just add up the counts

```
neg_sentiment = 0
for w in neg:
    neg_sentiment += token_count[w]
print(neg_sentiment / n_tokens)
```

```
## 0.026602464533499015
```

SpaCy

- SpaCy provides a machine-learning based approach to many of the things NLTK does
- SpaCy is also perhaps a bit more user-friendly

```python
import spacy

# python -m spacy download en_core_web_sm
nlp = spacy.load("en_core_web_sm")
# pipes enabled by default: tok2vec, tagger, parser, ner, attribute_ruler, lemmatizer]

doc = nlp(text)
```

Parse trees in SpaCy

- SpaCy has a visualization module called displaCy
- With this, we can quickly see how a sentence is structured
- To run it in a Jupyter notebook, use the below code:

```
sent = nlp("""Citi intends to release a revised Quarterly Financial Data
Supplement reflecting this realignment prior to the release of first quarter of
2014 earnings information.""")
spacy.displacy.render(sent, style="dep", jupyter=True, options={'compact':True})
```

Take a look at the code file to see the output

NER: Named Entity Recognition

- During the `nlp()` call earlier, spaCy automatically did named entity recognition'
- Using an ML algorithm + the dependency tree, it tries to determine any proper nouns in the document
  - It also tries to label them
- You can visualize these as well with displayCy

```python
spacy.displacy.render(sent, style="ent", jupyter=True)
```

Citi ORG intends to release a revised Quarterly Financial Data Supplement ORG reflecting this realignment prior to the release of first quarter DATE of 2014 DATE earnings information.

Parsing HTML

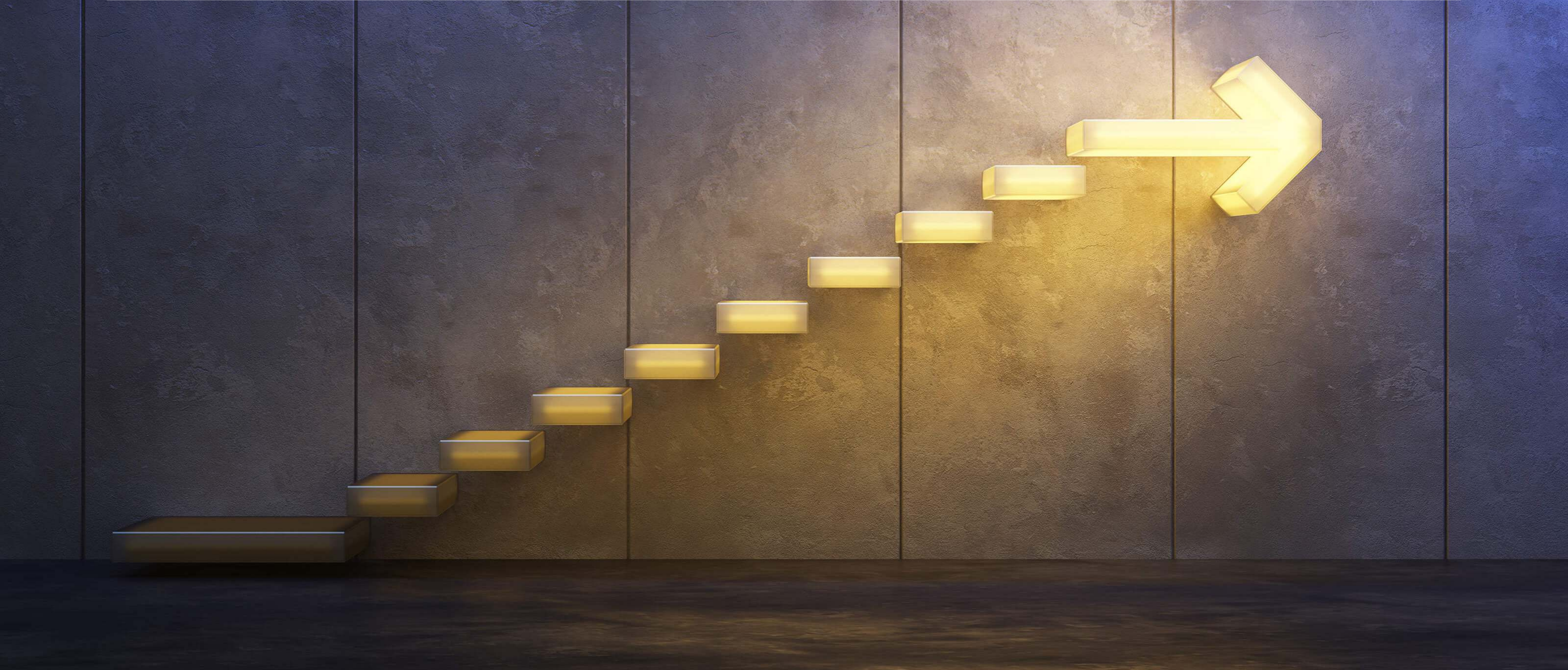
Overview

- As this part is code-heavy, we will do it in Jupyter
- The main idea is:
  1. Grab the main page of the website using `requests`
  2. Structure it with `beautifulsoup4` so we can traverse the page
  3. Grab the links to and names of standards, along with the publication years
  4. Traverse the links
  5. Extract the pdf locations from the traversed pages
  6. Grab the pdf files

Addendum: Using R

- HTML files
  - You can load from a URL using `httr` or `RCurl`
  - You can use `XML` or `rvest` to parse out specific pieces of html files
- JSON files
  - You can process JSON data using `jsonlite`
- PDF files
  - Use `pdftools` to extract text into a vector of pages of text
  - Use `tabulizer` to extract tables straight from PDF files!
    - This is very painful to code by hand without this package
    - The package itself is a bit difficult to install, requiring Java and `rJava`, though

Conclusion



Wrap-up

Text functions

- Python has good support for text built in

Pattern matching with regular expression

- Strong, versatile, and built in, but rather slow
- A good solution to simpler problems

Libraries: NLTK, spaCy, Beautiful Soup

- These help us to easily process more complicated text

Packages used for these slides

Python

- beautifulsoup4
- nltk
- numpy
- requests
- spacy

R

- kableExtra
- knitr
- reticulate
- revealjs