Foundations and Trends® in Accounting

Using Python for Text Analysis in Accounting Research *(forthcoming)*

Suggested Citation: Vic Anand, Khrystyna Bochkay, Roman Chychyla and Andrew Leone (2020 isbn), "Using Python for Text Analysis in Accounting Research *(forthcoming)*", Foundations and Trends® in Accounting: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

Vic Anand
University of Illinois at Urbana-Champaign
vanand@illinois.edu

Khrystyna Bochkay
University of Miami
kbochkay@bus.miami.edu

Roman Chychyla
University of Miami
rchychyla@bus.miami.edu

Andrew Leone
Northwestern University
andrew.leone@kellogg.northwestern.edu

now
the essence of knowledge
Boston — Delft

Contents

Using Python for Text Analysis in Accounting Research *(forthcoming)*

Vic Anand[1], Khrystyna Bochkay[2], Roman Chychyla[3] and Andrew Leone[4]

[1] *University of Illinois at Urbana-Champaign; vanand@illinois.edu*
[2] *University of Miami; kbochkay@bus.miami.edu*
[3] *University of Miami; rchychyla@bus.miami.edu*
[4] *Northwestern University; andrew.leone@kellogg.northwestern.edu*

ABSTRACT

The prominence of textual data in accounting research has increased dramatically. To assist researchers in understanding and using textual data, this monograph defines and describes common measures of textual data and then demonstrates the collection and processing of textual data using the Python programming language. The monograph is replete with sample code that replicates textual analysis tasks from recent research papers.

In the first part of the monograph, we provide guidance on getting started in Python. We first describe Anaconda, a distribution of Python that provides the requisite libraries for textual analysis, and its installation. We then introduce the Jupyter notebook, a programming environment that improves research workflows and promotes replicable research. Next, we teach the basics of Python programming and demonstrate the basics of working with tabular data in the Pandas package.

Vic Anand, Khrystyna Bochkay, Roman Chychyla and Andrew Leone (2020 isbn), "Using Python for Text Analysis in Accounting Research *(forthcoming)*", Foundations and Trends® in Accounting: Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

The second part of the monograph focuses on specific textual analysis methods and techniques commonly used in accounting research. We first introduce regular expressions, a sophisticated language for finding patterns in text. We then show how to use regular expressions to extract specific parts from text. Next, we introduce the idea of transforming text data (unstructured data) into numerical measures representing variables of interest (structured data). Specifically, we introduce dictionary-based methods of 1) measuring document sentiment, 2) computing text complexity, 3) identifying forward-looking sentences and risk disclosures, 4) collecting informative numbers in text, and 5) computing the similarity of different pieces of text. For each of these tasks, we cite relevant papers and provide code snippets to implement the relevant metrics from these papers.

Finally, the third part of the monograph focuses on automating the collection of textual data. We introduce web scraping and provide code for downloading filings from EDGAR.

1

Introduction

Analyzing the textual content of corporate disclosures, contracts, analyst reports, news articles, and social media posts has gained an increased popularity among accounting and finance researchers and the investment community in general. Unlike numbers, which are often the outcome of formal accounting rules, trading activities, deal negotiations, etc., texts bring with them an infinite number of possibilities. Even when thinking about a single concept or thought, the number of ways in which that thought might be expressed is seemingly boundless, and this is no less true in the domain of corporate communications than in interpersonal communications.

In this monograph, we provide an interactive step-by-step framework for analyzing spoken or written language for faculty and PhD students in social sciences. Our goal is to demonstrate how textual analysis can enhance research by automatically extracting new and previously unknown information from voluminous disclosures, news articles, and social media posts. We present all materials in a way that allows the reader to learn about a textual analysis concept or technique and also replicate it by doing. Specifically, for each concept / technique, we cite relevant papers and provide reader-friendly code snippets, allowing

readers to execute our code on their own machines. We do not provide a comprehensive review of the textual analysis literature and refer our readers to Li (2010a), Loughran and McDonald (2016), and Henry and Leone (2016) that provide excellent surveys of the literature on the topic.

We begin by showing how to install and use Python. Python is a general purpose programming language that has been consistently ranked in the top ten most popular programming languages in the world. It is very efficient and intuitive in the areas of pattern matching and text analysis. We review Python's basic programming syntax, operators, data types, functions, etc., allowing the readers to familiarize themselves with the programming environment first. We also discuss the Jupyter notebook which is an open-source web application that allows creating, running, and testing your Python code interactively. We introduce the Pandas package for working with tabular data; this will aid researchers as they convert unstructured textual data into structured, tabular data.

Next, we introduce regular expressions which represent patterns for matching different elements in texts (e.g., individual words, variants of words, numbers, symbols, etc.). Regular expressions are the foundation of being able to calculate different textual analysis metrics. We then proceed with the discussion and coding of different textual analysis methods used in accounting and finance studies. These methods include parsing texts into individual words and / or sentences, measuring tone / sentiment of a document, identifying specific words or phrases in text, measuring text complexity, classifying sentences into categories, identifying linguistic structure of a sentence, and measuring textual similarity. To facilitate the exposition of our code, we cite relevant research studies that demonstrate specific uses of textual metrics.

Finally, we provide an overview of web scraping and file processing features in Python. Specifically, we focus on downloading EDGAR filings and identifying specific sections in them.

Taken together, the first five chapters of this monograph will help readers get started with Python and prepare for writing their own code. The remaining chapters will help the reader to learn various textual analysis methods and implement the coding of the methods in Python.

We make all our code (in Jupyter Notebooks) and supplemental

materials available here. We kindly ask researchers who use our materials to cite this paper.

2

Installing Python on Your Computer

Despite Python's reputation as a beginner-friendly language, many newcomers find it difficult to get started. There are different versions of Python, different distributions of Python, and numerous code editors. Neophytes typically go to the main Python website (https://www.python.org/), download the Python interpreter, and quickly become frustrated that sample code they found on the internet does not work. Additionally, they find that the code editor that accompanies the base Python distribution is primitive and lacks features that they took for granted in SAS and Stata. We hope to help our readers avoid such difficulties by providing specific advice on how to get started.

We strongly recommend that readers download and install the Anaconda distribution of Python (https://www.anaconda.com/). Further, we strongly recommend that readers install version 3 of Python. In the remainder of this chapter, we discuss these recommendations and then provide instructions on how to install Anaconda.

2.1 The Role of Packages in Python

While the base language is very capable, much of Python's value derives from the rich ecosystem of available packages (aka libraries). Packages

are blocks of computer code that perform certain functions or tasks. Packages allow users to perform tasks without worrying about the implementation details. For example, a package might provide a function to read an Excel file into memory. Users only need to know how to call the function and do not need to worry about its inner workings. Newcomers typically do not know which packages to use for their desired tasks, even though such knowledge is common in the Python community. In this paper, we provide recommendations on packages to use for text analysis and data manipulation.

Managing packages can be tedious. Many packages depend on specific versions of other packages. For example, the `Pandas` package (for loading and manipulating data) depends on the `NumPy` package. Without package manager software, a user who wishes to install (or upgrade) `Pandas` would need to know about this dependency and first install `NumPy`. As the number of packages increases, package management becomes exponentially more tedious. Luckily, Anaconda includes an excellent package manager that frees Python users from these concerns.

2.2 The Anaconda Distribution of Python

Anaconda is a company that provides a free distribution of Python. Their download includes the Python interpreter, a package manager, hundreds of packages popular in the data science community, easy access to thousands more packages, and multiple code editors, such as Jupyter notebooks. Thus, Anaconda makes it very easy to get started and we strongly recommend it to researchers who wish to work with Python.

During the initial installation of Anaconda, the installer downloads and installs approximately 300 popular packages. Included among these are the `Pandas` package (for working with tabular data), `NLTK` (natural language toolkit, for working with text data), `Beautiful Soup` (web scraping), `scikit-learn` (machine learning), and numerous packages for data visualization. Anaconda automatically ensures that all dependencies are satisfied. For example, Pandas version 0.25.1 requires the `NumPy` package, version 1.14.6 or higher. When a user installs `Pandas` through Anaconda, the package manager ensures that a compatible version of `NumPy` is installed. Thus, Anaconda greatly simplifies the task

of upgrading packages as developers release new versions.[1]

By default, Anaconda includes Jupyter notebooks, a development environment that we will describe in Chapter 3. It also automatically installs Spyder, a powerful integrated development environment (IDE) that will feel familiar to those with past programming experience. Finally, Anaconda easily integrates with Microsoft Visual Studio Code, another extremely popular code editor with excellent Python support.

In sum, we strongly recommend that readers begin their Python journey by installing Anaconda. We describe the installation process in the next section.

2.3 Installing Anaconda

At the Anaconda home page (https://www.anaconda.com), hover over the *Products* menu in the upper left corner of the window, then choose *Individual Edition*. On the ensuing page, click on the Download button. Once the download page has loaded, scroll down until something similar to Figure 2.1 is visible. Click on the link for the 64-bit installer for your operating system.

Locate and run the installer package that was downloaded to your computer. Accept any warnings that may appear. We recommend that you accept most default/recommended settings. In particular, Windows users should install for "Just Me (recommended)". We note that Windows users may wish to change the default install location for Anaconda (see Figure 2.2). The installer *usually* defaults to `C:\users\<your username>\Anaconda3`, and this is what we prefer. However, if another location appears as default, we recommend changing it to our preference. Regardless of your preference, do not install to a system folder, e.g.

[1] Confusingly, the term *Anaconda* has multiple meanings. Anaconda is the name of a company. It is also the name of a software distribution that includes Python, a package manager (called *conda*), and Jupyter notebooks. Finally, just to make things extra confusing, Anaconda also refers to a "meta package," i.e., a set of specific versions of specific packages curated by the Anaconda company that work well together. At the time of this writing, the Anaconda meta-package is at version 2020.07; this meta-package includes specific versions of the aforementioned 300 or so packages for data science. When the Anaconda company releases the next version of the Anaconda meta-package, many of the bundled packages will be upgraded.

Figure 2.1: Anaconda download page

`C:\Windows\...` or `C:\Program Files\...` as that may cause problems with file permissions. Accept all other defaults and proceed with the installation.

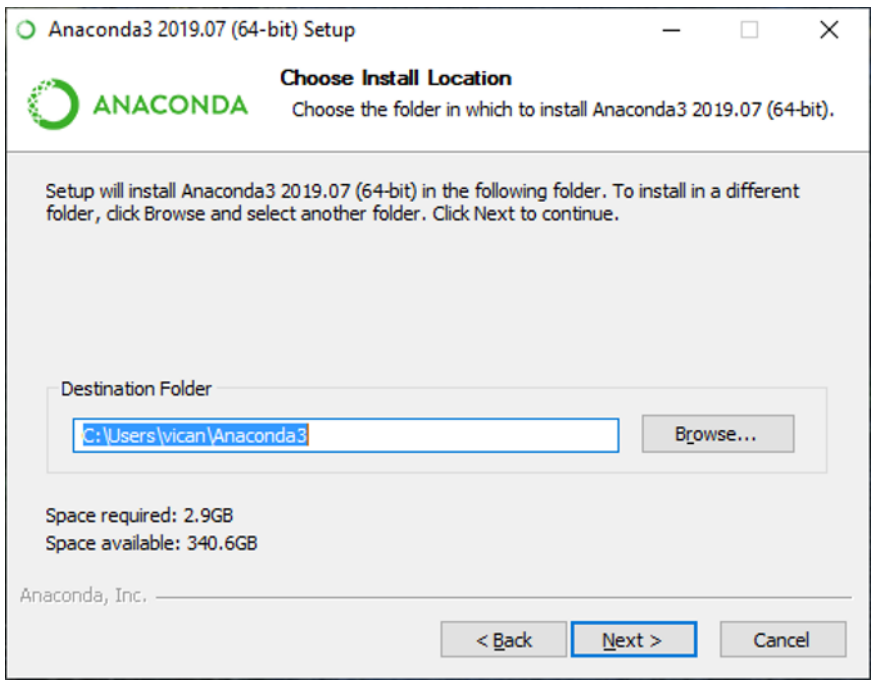


Figure 2.2: Anaconda installation location dialog box

The installation process may take 10-15 minutes. Anaconda will

download and install many packages and this takes some time.

2.4 Launching and Using Anaconda

Anaconda provides a graphical interface called *Anaconda Navigator* for those who prefer it (see Figure 2.3). From Anaconda Navigator, users can launch code editors such as Jupyter Notebook and Spyder. Users can also install other code editors such as Visual Studio Code. Finally, Anaconda Navigator allows users to view installed and available packages, as well as install or upgrade packages. To accomplish this, click on the Environments tab on the left side of the Navigator window.

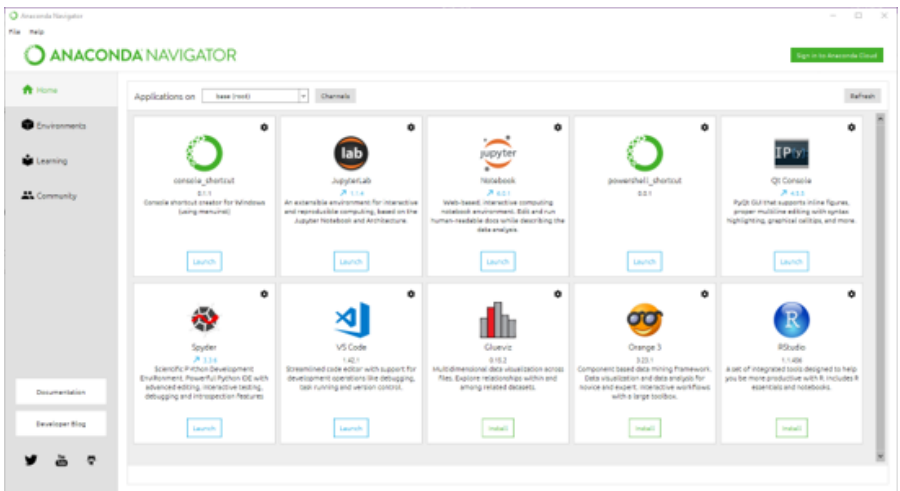


Figure 2.3: Anaconda Navigator window

While Anaconda Navigator provides a slick, graphical interface, we find that it is very slow to launch. We therefore prefer to use a terminal window. To do this, Windows users should launch `Anaconda Prompt (Anaconda 3)` from their start menu; Mac users should launch a `Terminal` window. In the next chapter, we provide the command for launching a Jupyter notebook from a terminal window. Since most users will find the default installation of Anaconda adequate, we do not discuss package management from the command line in this paper. Users who wish to learn how to manage packages from the terminal should visit

the conda command reference webpage (https://docs.conda.io/projects/conda/en/latest/commands.html). We find the `conda install`, `conda update`, and `conda list` commands particularly useful.

3

Jupyter Notebooks

There are many ways to write and execute Python code. In this chapter, we introduce the Jupyter Notebook, a popular tool in the data science community, and recommend that the readers use it in their text analysis research projects. Jupyter Notebooks are live documents that contain code, outputs from the code, visualizations, equations, and formatted text. As we will show, because Jupyter Notebooks support all of these features and store them in one place, they can greatly simplify and improve research workflows.

3.1 Motivating Example

Consider the following research task: load a data set, create some new columns, make plots, compute summary statistics, and share the results with coauthors. With many popular statistics packages, one would need to copy the outputs (tables and figures) into a Word document or email, add some annotations or comments, and send these to coauthors. When it is time to revise the analysis, the Word document or email must be updated manually and shared again.

For illustrative purposes, let us assume we use Stata, a popular

statistics package in the research community.[1] Consider the following Stata code that uses a sample dataset that accompanies this paper, *FTA_Python_Ch3.csv*. This dataset contains data from the Compustat fundamentals annual dataset for Apple, Microsoft, and Amazon in fiscal years 2010-2019.

```
log using "Ch3_log_file.smcl", replace smcl

import delimited "FTA_Python_Ch3.csv"

keep gvkey fyear tic conm at ceq csho dt ib oancf prcc_c
sort gvkey fyear
xtset gvkey fyear

gen ROE = ib / (0.5 * (ceq + L.ceq))
gen ROA = ib / (0.5 * (at + L.at))
gen AnnualStockReturn = (prcc_c / L.prcc_c) - 1.0

drop if missing(ROE, ROA, AnnualStockReturn )

foreach measure in ROE ROA AnnualStockReturn {
    xtline ‘measure’, i(conm) t(fyear) ///
    ytitle("‘measure’") ttitle(Fiscal Year) ///
    byopts(title("‘measure’")) ///
    name("‘measure’", replace)
    graph export "‘measure’.png", name("‘measure’")
}

table fyear, contents(mean AnnualStockReturn)
log close
```

This code loads the sample data file, *FTA_Python_Ch3.csv*, keeps the variables of interest, and computes return on equity (*ROE*), return on assets (*ROA*), and the annual stock return (*AnnualStockReturn*) for each firm. The code then generates line plots of these variables by firm

[1] This example workflow is similar in SAS.

and by year; the plots are displayed on the screen and also saved as PNG files. Finally, the code computes the annual return by year of an equally-weighted portfolio of these three stocks.

Consider the problem of saving these results and sharing them with coauthors. A researcher needs to save the code in a Stata Do file, save the log file, and save the plot files. To share these results with coauthors, the researcher would need to copy the summary table from the Stata results window (or from the log file) and the plots into another program (e.g. email, Word, OneNote), annotate the findings, and send the results to the coauthors.

A Jupyter Notebook is a single file that provides functionality for all these tasks. It is a container for the code and output, and allows the user to easily annotate the output. Consider the sample Jupyter Notebook that accompanies this chapter; this sample notebook implements in Python the same functionality as in the Stata code above. We have reproduced the sample notebook in Figures 3.1, 3.2, and 3.3. Notice the notebook contains results in tabular form, figures, headings, and commentary on the results. Since it is a single file, this notebook can easily be shared with coauthors; if the coauthors do not have Python, the notebook can be exported to HTML or PDF format, or hosted on a website. Additionally, if the source data or the code changes, the cells in the notebook can simply be run again and the notebook re-shared. In sum, a Jupyter Notebook is a container for the entire research workflow and can easily be shared.

3.2 JupyterLab: A Development Environment for Jupyter Notebooks

Project Jupyter is an open-source project to support interactive computing. Jupyter Notebooks are designed to be language agnostic and currently support dozens of programming languages. In fact, the name Jupyter is an amalgamation of Julia, Python, and R, three programming languages originally supported in Jupyter Notebooks (Project Jupyter, 2020).

The original app for creating, editing, and running Jupyter Notebooks is (confusingly) named *Jupyter Notebook*. This app offers all of

Chapter 3: Sample Notebook

1. Imports

The following is a "code cell". When you run it (by clicking in it or on it and pressing CTRL+ENTER), the cell will execute.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Import and prepare data

[2]:
```
# Import the data
df = pd.read_csv('FTA_Python_Ch3.csv',
                 usecols=['gvkey','fyear','conm','tic','at','ceq','ib','prcc_c'])
# Ensure the data is properly sorted so that the LAG commands will work correctly.
df.sort_values(['gvkey','fyear'], inplace=True)
```

The following cell shows a preview of the data. Its functionality is similar to that of the `browse` command in Stata.

[3]:
```
df.head()
```

[3]:

	gvkey	fyear	tic	conm	at	ceq	ib	prcc_c
0	1690	2010	AAPL	APPLE INC	75183	47791	14013	46.080000
1	1690	2011	AAPL	APPLE INC	116371	76615	25922	57.857143
2	1690	2012	AAPL	APPLE INC	176064	118210	41733	76.024700
3	1690	2013	AAPL	APPLE INC	207000	123549	37037	80.145714
4	1690	2014	AAPL	APPLE INC	231839	111547	39510	110.380000

[4]:
```
# Compute ROE using average total equity
df['ROE'] = df['ib'] / (0.5 * (df['ceq'] + df.groupby('gvkey')['ceq'].shift()))
# Compute ROA using average total assets
df['ROA'] = df['ib'] / (0.5 * (df['at'] + df.groupby('gvkey')['at'].shift()))
# Compute annual stock return for each stock
df['AnnualStockReturn'] = (df['prcc_c'] / df.groupby('gvkey')['prcc_c'].shift()) - 1.0
```

[5]:
```
# Drop the rows with missing values generated from the lags
df.dropna(inplace=True)
```

Figure 3.1: Sample Jupyter notebook, snippet 1 of 3

the features we described previously, but through a primitive interface. Project Jupyter improved on the original app by creating *Jupyter-*

3. Make graphs of ROE

The following command is similar to the `reshape` command in Stata. It transforms the data into the form needed for the plots.

[6]:
```
df_ROE = df[['fyear','tic','ROE']].pivot(index='fyear', columns='tic', values='ROE')
df_ROE.head(2)
```

[6]:

tic	AAPL	AMZN	MSFT
fyear			
2011	0.416732	0.086314	0.448391
2012	0.428415	-0.004891	0.275068

[7]:
```
df_ROE.plot(subplots=True, layout=(1,3), figsize=(10,3),
            marker='o', sharex=True, sharey=True)
plt.show()
```

Notice how Apple's ROE is nuch higher than the other two companies. Now I'm curious about the average ROE of each company during the sample period.

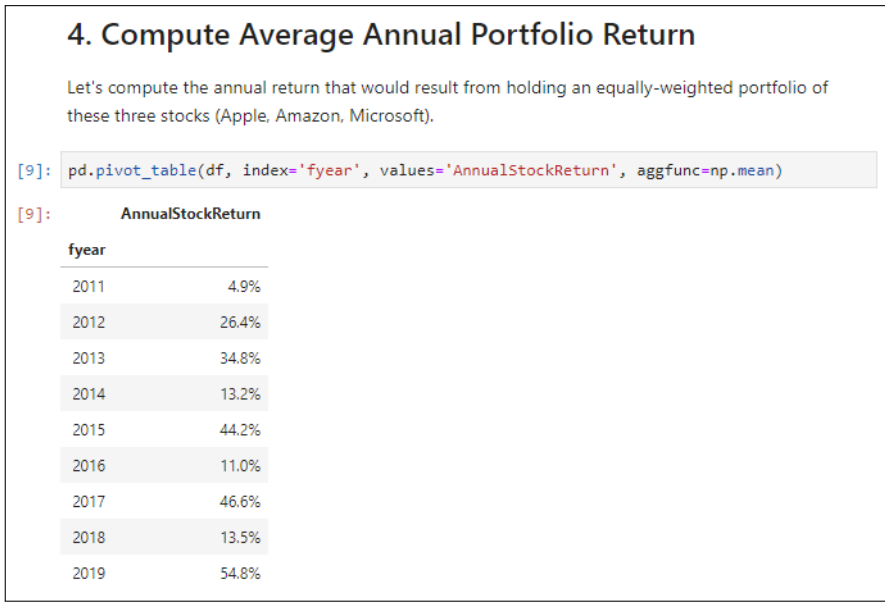
[8]:
```
# This sets a display option so percentages are formatted neatly.
pd.set_option("display.float_format", lambda x: f'{x*100.0:.1f}%')
# This computes mean ROE by ticker across the sample period.
df[['tic','ROE']].groupby('tic').mean()
```

[8]:

tic	ROE
AAPL	41.6%
AMZN	10.2%
MSFT	28.7%

Figure 3.2: Sample Jupyter notebook, snippet 2 of 3

Lab, "the next generation web-based user interface for Project Jupyter" (Project Jupyter, 2018).



4. Compute Average Annual Portfolio Return

Let's compute the annual return that would result from holding an equally-weighted portfolio of these three stocks (Apple, Amazon, Microsoft).

```
[9]: pd.pivot_table(df, index='fyear', values='AnnualStockReturn', aggfunc=np.mean)
```

[9]:

fyear	AnnualStockReturn
2011	4.9%
2012	26.4%
2013	34.8%
2014	13.2%
2015	44.2%
2016	11.0%
2017	46.6%
2018	13.5%
2019	54.8%

Figure 3.3: Sample Jupyter notebook, snippet 3 of 3

We recommend *JupyterLab* for most users.[2] JupyterLab is more user-friendly and offers features that researchers will likely find useful. The only reason to use the traditional Jupyter Notebook app is if you wish to use an extension that is not available for JupyterLab; this is an unlikely scenario for most researchers.

3.3 How to Launch JupyterLab

There are two ways to launch JupyterLab on your computer.

1. Launch Anaconda Navigator. In the Home tab of the Navigator window, the reader will be able to see tiles like those shown in Figure 2.3. Click the Launch button in the JupyterLab tile. JupyterLab will open in a new tab in your default web browser.
2. Open Anaconda Prompt (Windows PC) or Terminal (Mac). At the command line, type `jupyter lab` and press Enter. JupyterLab

[2]Both the traditional Jupyter Notebook app and JupyterLab are installed by default with Anaconda (see Figure 2.3).

will open in a new tab in your default web browser.

We recommend the second method since the Anaconda Navigator window loads slowly.

3.4 Working in JupyterLab

Upon launching JupyterLab, you will see a browser tab like that shown in Figure 3.4. JupyterLab runs entirely within the browser. The left sidebar contains a file explorer. You can navigate through the folder tree on your computer and find existing notebooks. Jupyter Notebook files, which have the file extension *.ipynb*, have an orange icon (notice the orange icon in Figure 3.4 next to the notebook `Ch3_Sample_notebook`). Launch a notebook by double-clicking on the filename next to its orange icon. Alternatively, you can create a new notebook by clicking on the `Python 3` tile.

3.4.1 Opening Multiple Notebooks and Files in JupyterLab

JupyterLab allows users to open and view multiple notebooks, or copies of the same notebook, simultaneously. We show an example of this in Figure 3.5. To create a new view of a notebook, right-click on the notebook tab and select `New Console for Notebook`. Then drag the tab for the new window to the desired location. Note that we hid the file explorer. You can toggle its visibility by clicking on the file explorer icon (circled in red).

JupyterLab can open many file types, such as CSV and images. This ability, combined with the window layouts feature, allows users to customize their layout and view multiple items simultaneously. We shown an example of a more sophisticated layout in Figure 3.6.

3.4.2 Creating Jupyter Notebooks

To create a new, empty notebook, click the `Python 3` tile that is shown after launching JupyterLab (see Figure 3.4). Sometimes, upon launch, JupyterLab will open the last edited notebook. To create a new notebook, click on the File menu, then New, then Notebook. Alternatively, click

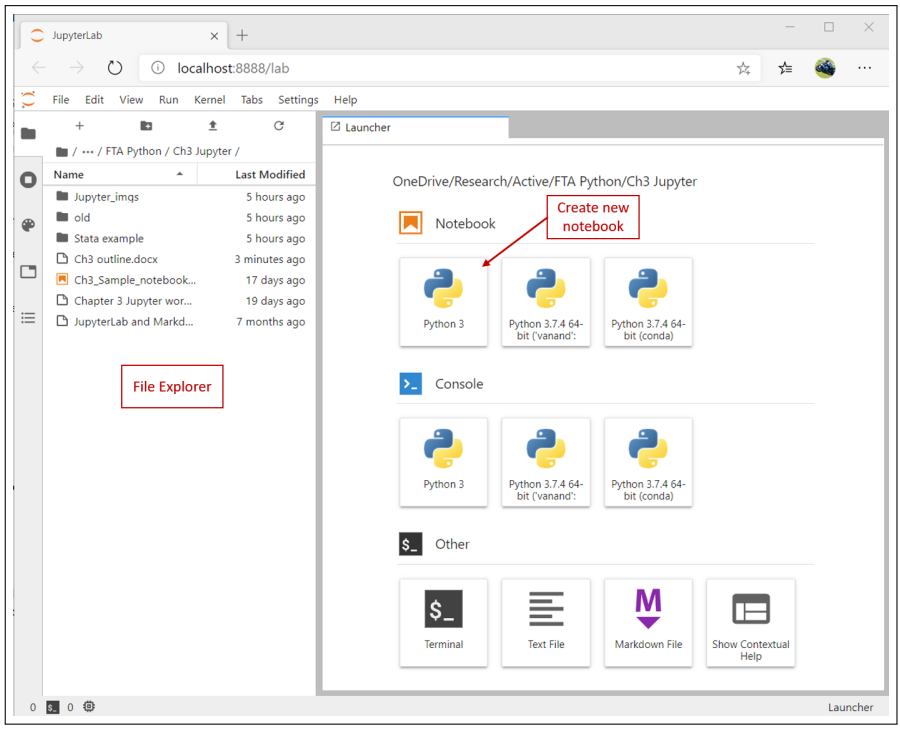


Figure 3.4: JupyterLab after launch

File, then New Launcher and a window such as that shown in (see Figure 3.4) will appear.

We shown an example of a new, empty notebook in Figure 3.7. The new notebook contains a single, empty cell.

3.4.3 Writing and Executing Code

To enter code in a cell, click inside the cell with the mouse. Alternatively, if the cell is selected (as shown in Figure 3.8), press Enter and a cursor will be shown inside the cell. At this point, we can type in our code. To run the code in the active cell, press `CTRL + ENTER`. Python will execute the code and show the output, if any, beneath the cell. Additionally, a number will appear in the brackets to the left of the cell. This number indicates the order in which cells were executed. Cells need not be

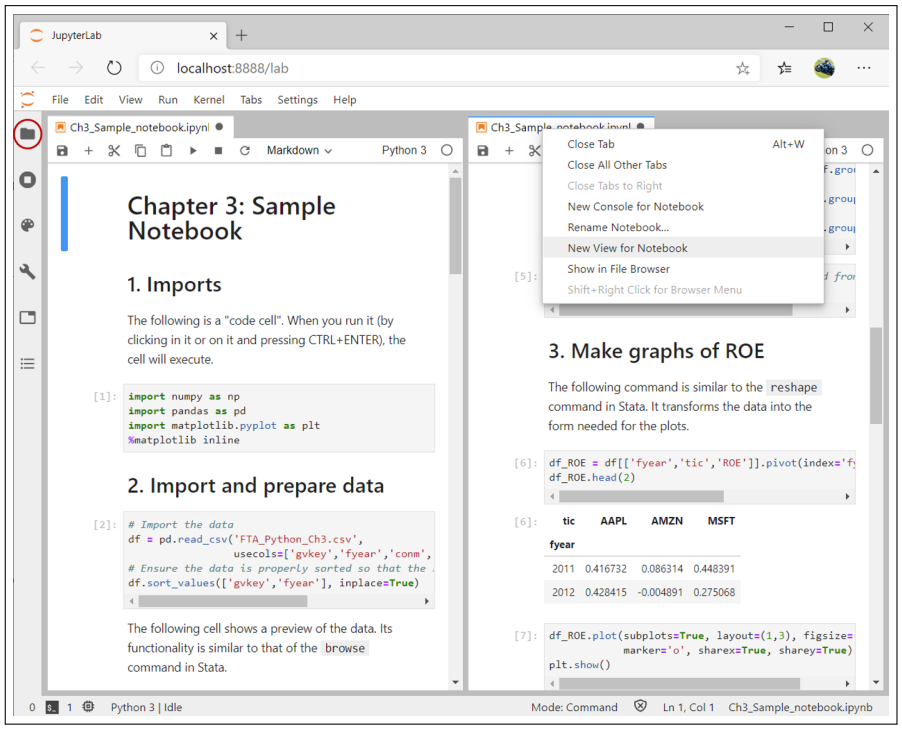


Figure 3.5: JupyterLab with two views of the same notebook. The red circle highlights the file explorer icon; clicking on this icon hides or displays the file explorer in JupyterLab.

executed in order, so these numbers allow the users to keep track of which cells have been executed and when. We show an example of some code execution in Figure 3.8.

Executing a Single Cell

`CTRL+ENTER` executes the currently selected cell and does not advance the cell selection. `SHIFT+ENTER` executes code and selects the next cell. If there is no next cell, one is created. `ALT+ENTER` executes the current cell and creates a new empty cell beneath the current cell.

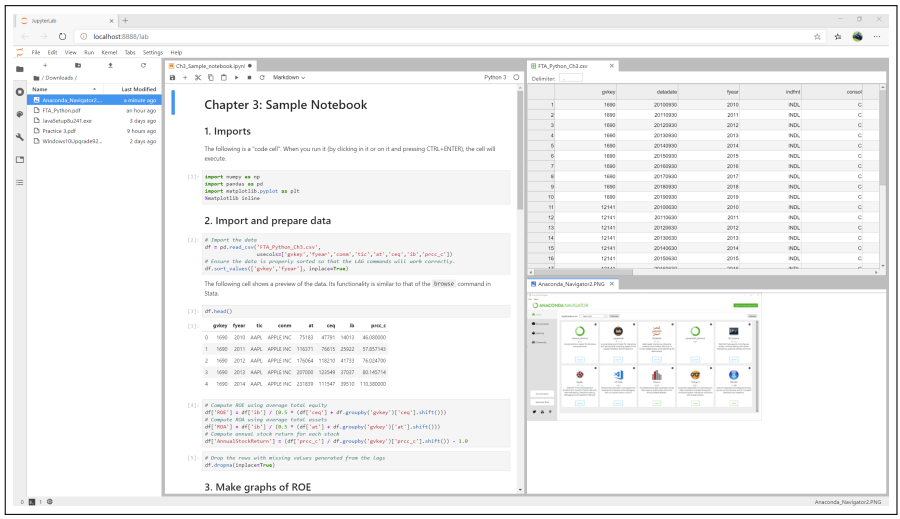


Figure 3.6: JupyterLab permits users to open CSV and image files. It also allows for sophisticated window layouts; these are useful when users need to view their data and code simultaneously.

Executing Multiple Cells

Sometimes, we may wish to execute all cells in a notebook. Another common scenario is to execute all cells before or after the selected cell. JupyterLab provides these functions in the `Run` menu.

Amount of Code Per Cell

Notice that the cell in Figure 3.8 contains two lines of code. A cell can contain one or more lines of code. It is entirely up to the user. We could write all of our code in a single cell. Or, we could only have one line of code per cell. Common practice is to put all code to perform an action within in each cell. For example, a cell might open a file and display the results. Another cell might contain multiple lines of code that create a graph. Notice that some cells in Figure 3.1 contain multiple lines of code and others contain a single line of code. The first cell contains all import statements. The second opens a data file and sorts it. The third cell displays the first 5 rows of the data. The fourth computes return measures, and the fifth drops rows with missing values. We could have

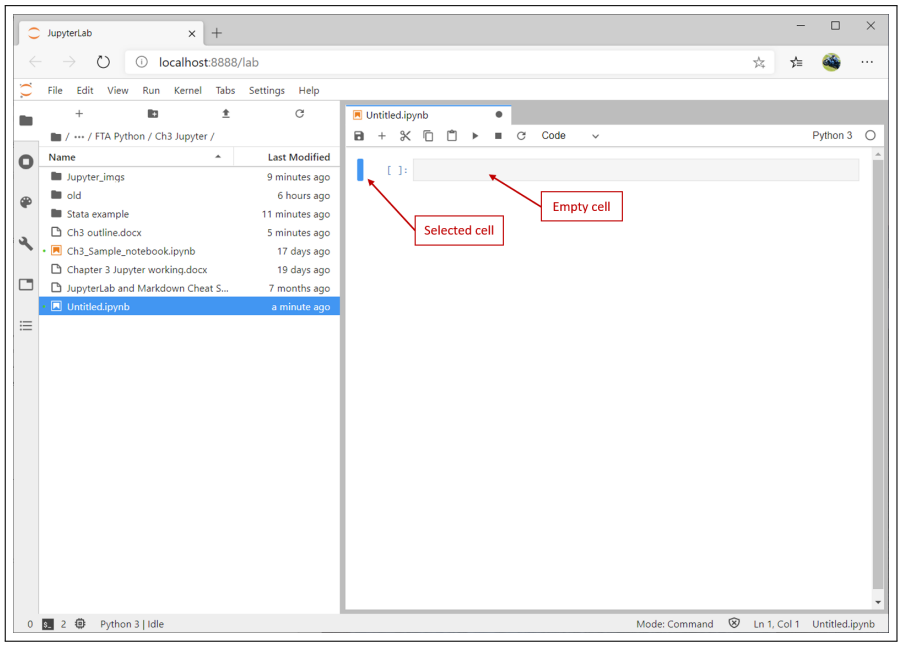


Figure 3.7: New notebook in JupyterLab.

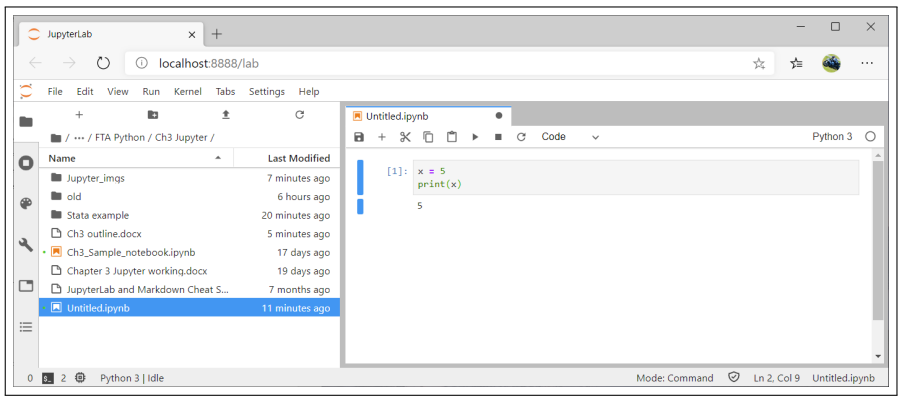


Figure 3.8: Example of code execution. We typed code into the cell and pressed CTRL + ENTER. The output is shown beneath the cell, and the cell counter shows 1 since this was the first cell executed since the notebook was opened.

used more or fewer cells; this is ultimately a style choice.

3.4.4 Working with Cells in JupyterLab

Adding and Deleting Cells

To add a new cell, select an existing cell and press `a` to add a new cell above the selected cell. Press `b` to add a new cell beneath the selected cell. Alternatively, press the `+` button on the toolbar for the active tab in JupyterLab (to the right of the save button). Finally, to delete the currently selected cell, press the `d` key twice in quick succession. Alternatively, right-click on the cell and select delete, or choose `Delete Cells` from the Edit menu.

Keystroke	Action
`a`	Insert new cell above currently selected cell
`b`	Insert new cell below currently selected cell
`dd`	Delete currently selected cell

Table 3.1: Commands to add and delete cells in JupyterLab

Copying, Cutting, and Pasting Cells

To copy a cell, select it and press the `c` key. Then select a cell and press `v`. A copy of the copied cell will appear underneath the selected cell. If you wish to cut a cell, use the `x` key in lieu of the `c` key. Note that all of these actions can be performed through the Edit menu, or through right-clicking.

Keystroke	Action
`c`	Copy currently selected cell
`x`	Cut currently selected cell
`v`	Paste below currently selected cell

Table 3.2: Commands to copy, cut, and paste cells in JupyterLab

Moving Cells by Dragging

JupyterLab allows users to drag cells to a new location in the notebook. To drag a cell, move the mouse to the left of the brackets with the cell counter. The mouse pointer will change to a crosshair. Left click and

hold down the mouse button, then drag the cell up or down to the desired location.

Selecting Multiple Cells

All of the commands above can work on multiple cells. To select more than one cell, hold down the SHIFT key and then click on the up or down arrow to select multiple cells.

Other Editing Features

JupyterLab contains many other useful editing features. We encourage the readers to discover them by exploring the menus and the command palette (button that looks like a painter's palette; located under the file explorer button shown in Figure 3.5).

3.5 The Markdown Language and Formatted Text Cells

Jupyter Notebooks permit sophisticated text formatting. This capability is very useful when structuring a notebook. For example, the notebook can be organized into sections, each with a heading; text can be formatted in boldface and italic. In this section, we demonstrate some common formatting tasks. Underlying these is the Markdown language. Markdown is a language that provides shortcuts to HTML. While typing HTML code can be cumbersome, Markdown is easy. In the following sections, we will demonstrate how to change a cell in a notebook from a code cell to a markdown cell (and vice versa). We will then show some common Markdown commands.

3.5.1 Changing the Type of a Notebook Cell

By default, cells in Jupyter Notebooks are code cells. If we wish to enter text (formatted or unformatted) into a cell, we must change the type of that cell to markdown. To do this, select the cell and press the `m` key. Alternatively, in the toolbar underneath the tab, there is a dropdown menu with the word `Code`. Click on this dropdown menu and select

`Markdown`. To revert to a code cell, press the `y` key or use the dropdown menu.

Keystroke	Action
`m`	Change cell type to markdown
`y`	Change cell type to code

Table 3.3: Commands to change cell type in JupyterLab

Within a Markdown cell, you can type text. However, you still need to use `CTRL+ENTER` or an alternative keystroke to render the text in the cell.

3.5.2 The Markdown language

Markdown provides shortcuts to HTML. It is possible to enter HTML directly into a Markdown cell in a Jupyter Notebook, and we occasionally do so when we wish to use an HTML feature that is not supported in Markdown (e.g., colored text). However, for many notebooks, Markdown is sufficient.

Emphasis

To italicize text, place the text between asterisks. For example:

```
*sample italicized text*
```

will appear as *sample italicized text*. To see this, create a new cell in a notebook, type an asterisk, some text, and another asterisk. Then press `CTRL+ENTER`.

To render text in boldface, place the text between pairs of asterisks. For example:

```
**sample boldface text**
```

will appear as **sample boldface text**.

Headings

Markdown supports six levels of headings. Heading 1 is the largest. To render text as heading 1, place a single **#** before the text. To render text as heading 2, place two **#** characters before the text. And so on. For example, entering the following code into a markdown cell:

```
# Sample heading 1
## Sample heading 2
### Sample heading 3
#### Sample heading 4
##### Sample heading 5
###### Sample heading 6
```

might appear as:

Sample heading 1

Sample heading 2

Sample heading 3

Sample heading 4

Sample heading 5

Sample heading 6

Lists

Markdown supports unordered (i.e. bulleted) and ordered (i.e. numbered) lists. To create an unordered list, place one list item on each line. Before each list item, type a single asterisk at the beginning of a line and a space.[3] For example, the following markdown would create a

[3]Markdown also permits pluses or minuses instead of asterisks to denote list items.

bulleted list:

```
* Python
* R
* SAS
* Stata
```

To create an ordered list, place one list item on each line. Before each list item, type a number and a period at the beginning of a line. The numbers must be sequential. For example, the following markdown would create a numbered list.

```
1. Python
2. R
3. SAS
4. Stata
```

Markdown cheat sheets

Markdown offers many features and we have only demonstrated a few here. For example, the users can include quotes, code, hyperlinks, and images in Markdown cells in their Jupyter notebook. To learn more about Markdown and its features, we suggest visiting the website Markdown Guide (https://www.markdownguide.org/). That website provides a nice "cheat sheet" that summarizes common Markdown formats.

4

A Brief Introduction to the Python Programming Language

This chapter provides a brief introduction to the Python programming language. We assume the reader is proficient with statistical software (e.g., SAS or Stata) but has little or no experience with a programming language like Python. We will not undertake the Herculean task of teaching programming in a single chapter. Instead, we introduce the basics of the Python programming language with the goal of providing the reader with enough proficiency to be able to understand the code in this monograph. Thus, we only demonstrate a small subset of Python's features and capabilities.

4.1 Fundamentals

Researchers who are new to Python or to programming must be aware of three fundamental concepts.

1. Python executes code one line at a time.
2. Python is case-sensitive.
3. Python uses indentation to group statements.

4.1.1 Python Executes Code One Line at a Time

A computer program is a sequence of steps that the computer will perform when the program is executed. Python code is executed one line at a time by a program known as the *Python interpreter*. Each running Jupyter Notebook is given its own Python interpreter. The Python interpreter does not look ahead or look behind. It simply executes each line of code in order. If an error results, the interpreter stops execution and does not execute any remaining code. This contrasts with SAS data steps and procedures. SAS will execute one row (observation) at a time. Stata is a hybrid. A Stata Do file will execute line-by-line, like Python, but each line of Stata code can potentially operate on all rows of a dataset. In sum, the Python interpreter executes one line of code at a time and stops after the last line of code, or upon encountering an error, whichever occurs first.

4.1.2 Python is Case-Sensitive

Python is *case-sensitive*. The variables `ROA` and `roa` are distinct. If you try to call the built-in absolute value function **`abs`** using capital letters (e.g., `ABS`), an error will result.

4.1.3 Python Uses Indentation to Group Statements

Consider the following code:

In:

```
div_yield_firm1 = 0.06
div_yield_firm2 = 0.04

if div_yield_firm1 > 0.05:
    print('Invest in firm 1.')
else:
    if div_yield_firm2 > 0.05:
        print('Invest in firm 2.')
    else:
        print('Do not invest at this time.')
```

Out:

```
Invest in firm 1.
```

The meaning of the code should be self-evident. Notice that Python uses indentation to group statements. The indentation indicates which statements will be performed and when. The logic of this code is to first check the dividend yield of firm 1. If its yield exceeds 5% then invest in firm 1 and stop execution of the code. However, if firm 1's dividend yield is less than or equal to 5% then check the dividend yield of firm 2. If firm 2's yield exceeds 5%, invest in firm 2. If neither firm's dividend yield exceeds 5%, do not invest. Notice that the statements that correspond to each condition are indented, as are the `if` statements and `else` clauses.

Indentation is *mandatory* in Python. Use a single tab character or four white spaces for each level of indentation.

4.1.4 Comments

To create a comment in Python code, use the `#` symbol. The remainder of the line following a `#` symbol is ignored. For example, the first line of the following is ignored by Python:

```
# Some sample code
x = 3
```

4.2 Variables and Data Types

Variables are containers for data. Variables are commonly used to store the result of a computation or an entire dataset.

4.2.1 Creating Variables

How to Create a New Variable

Python makes it very easy to create a variable. Simply type the desired variable name, an equals sign, and a valid value. The following code illustrates this by creating a new variable named `z` and storing the value 14 in it.

```
z = 14
```

Changing an Existing Variable

When a line of code assigns a value to a variable, the Python interpreter first checks whether that variable already exists in the environment. If it does, the variable's value is updated. If the variable does not exist, the interpreter creates the variable in the environment and sets its value. Thus, if `z` exists in the environment and contains the value 14 and we run the code `z = -2`, then the value of `z` will be updated to `-2`.

Rules for Naming Variables

Python is very flexible about variable names and we encourage you to exploit this flexibility by using meaningful names for variables. Researchers using Python should be aware of the following rules for valid Python variable names:

- Variable names can contain:
  - Uppercase letters (A-Z)
  - Lowercase letters (a-z)
  - Digits (0-9)
  - The underscore character (_). Note: this is different than the hyphen, or dash (-), which is not allowed in a variable name.
- Variable names cannot contain spaces.
- A variable name cannot begin with a digit.

Some example variable names are `accruals_2017`, `Accruals_2017`, `ACCRUALS_2017`, and `AcCruAls_2017`. However, recall that Python is case-sensitive so these four variables are distinct.

4.2.2 Environment and State

One of the fundamental ideas in programming is that code executes inside an *environment*. An environment is a container for all variables. The *state* of environment is simply a list of all variables and their values. When we first open a Jupyter Notebook, there will be no user-defined variables in its environment (i.e., it has an "empty" environment). As we execute code, Python will create variables and save them in the environment. These variables, and the entire environment, will be deleted

when we close our notebook. If we want to recreate those variables, we will need to reopen the notebook and execute the cells.

4.2.3 Data Types

Variables are containers for data. Since data comes in many forms, different pieces of data have different storage requirements. Python handles this by assigning a *data type* to each variable. Once it infers a variable's data type, the Python interpreter sets aside the appropriate amount of memory.

Python has many built-in data types but in this section, we will introduce only a subset. Later, we will demonstrate how these basic data types are grouped together to form more complex data structures like lists.

Some built-in data types in Python are:

Data Type	Description
`int`	`int` stands for *integer*.
`float`	`float` stands for "floating point number". These are real numbers.
`str`	`str` stands for "string". These are textual values.
`bool`	`bool` stands for *boolean*. These can have a value of `True` or `False`.

Table 4.1: Basic data types in Python

Integers (`int` Data Type)

An integer is a positive or negative whole number, or zero. To create an integer value, simply enter a whole number into a cell or assign it to a variable.

In:

```
k = -7352
print(k)
```

Out:

```
-7352
```

Researchers should note the following when entering numerical values into Python:

- Do not use commas when entering numbers larger than `1,000`.
- Do not use parentheses to enter negative numbers.
- Do not use dollar signs.

Notice that the code above creates a large, negative number, `-7352`. Had we used parentheses and commas, e.g. `k = (7,352)`, Python would have created a tuple containing the numbers 7 and 352. We describe tuples in Section 4.7.2.

Floating Point Numbers (`float` Data Type)

A float stores a real number. To create a float, enter a number with a decimal. The following code creates three variables, each holding a floating point number.

```
myFloat1 = -5.7
myFloat2 = 5.0
myFloat3 = 5.
```

Notice that we can force Python to treat 5 as a floating point number by adding a decimal or a .0 after the 5. Without the trailing decimal point, Python would store that 5 as an integer.

As with integers, do not use commas, parentheses, or dollar signs when entering floating point values.

Strings (`str` Data Type)

In Python, textual data is stored as a string (type **`str`**). To create a new string, enter text inside quotes. You can use single quotes (') or double quotes ("). For example:

```
s1 = 'The ticker symbol for Harley-Davidson is HOG.'
s2 = "157"
```

Python permits single- and double-quotes in string literals so that users can include quotes inside strings. For example, consider the following:

In:

```
myString = 'The CEO said, "Earnings growth will exceed
    5% next quarter."'
print(myString)
```

Out:

```
The CEO said, "Earnings growth will exceed 5% next
    quarter."
```

By using single-quotes to delimit the string value, we were able to include double-quotes in the string. We could instead use double-quotes to delineate the string and include single quotes inside the string. For example:

In:

```
myString = "Harley's ticker symbol is HOG."
print(myString)
```

Out:

```
Harley's ticker symbol is HOG.
```

When Python sees a single quote, it looks for the next single quote. Everything in between the two single quotes is treated as a string. The same rule applies for double quotes.

It is possible to include a single quote inside a single-quoted string, or to include a double quote inside a double-quoted string. To do so, prefix the quote inside the string with a backslash character (\). When Python sees a backslash character, it treats the next character as a literal, meaning it does not assign a special meaning to it. Here are some examples:

In:

```
myString = 'Harley\'s ticker symbol is HOG.'
print(myString)

myString = "The CEO said, \"Earnings growth will exceed
     5% next quarter.\""
print(myString)
```

Out:

```
Harley's ticker symbol is HOG.
The CEO said, "Earnings growth will exceed 5% next
    quarter."
```

Empty Strings

Python allows "empty strings," i.e., zero-length strings. The empty string is entered as `''` (two single quotes in succession). The empty string is often used to represent missing values in columns of text data.

Booleans (`bool` Data Type)

A Boolean (**`bool`**) can only assume the values `True` or `False`. Boolean values are extremely important because they are used in every conditional expression in Python. All **`if`** statements and **`while`** loops use Boolean values. For example, consider the following code.

In:

```
x = 0

if (x > 3):
    print('Something')
else:
    print('Something else')
```

Out:

```
Something else
```

When this code is executed, Python checks whether `x` is greater than 3. If it is, then the value `True` is substituted for `(x > 3)`. If `x` is less than or equal to 3, then the value `False` is substituted for `(x > 3)`. In other words, Python converts the expression `(x > 3)` into a Boolean value and uses that to execute the code. In sum, every condition given to an **`if`** or to a **`while`** is reduced and converted into a Boolean.

4.2.4 Checking the Type of a Variable, Value, or Expression

Use the built-in function **`type()`** to check the type of a variable, value, or expression. For example:

In:

```
type(7)
```

Out:

```
int
```

The **`type()`** function can also be used to check the type of data stored in a variable. For example:

In:

```
y = 'Hello there.'
type(y)
```

Out:

```
str
```

4.2.5 Converting Between Types

Python provides built-in functions to convert a value from one type to another. The functions are shown in Table 4.2.

Function	Notes
`int()`	Attempts to convert its argument to an integer. If its argument is a float, it will *truncate* the fractional portion.
`float()`	Attempts to convert its argument to a floating point number.
`str()`	Attempts to convert its argument to a string.
`bool()`	Attempts to convert its argument to a Boolean.

Table 4.2: Functions to convert values from one data type to another

Some conversions will fail. For example, if you attempt to convert a string to an `int` or to a `float`, Python will throw an error. For example:

In:

```
int('EBITDA is 1.03 million.')
```

Out:

```
ValueError                                Traceback (
    most recent call last)
<ipython-input-4-7a251f5038b9> in <module>
----> 1 int('EBITDA is 1.03 million.')

ValueError: invalid literal for int() with base 10: '
    EBITDA is 1.03 million.'
```

Additional notes on conversions:

- When converting a Boolean to a numeric type, `True` will convert to 1 or 1.0, and `False` to 0 or 0.0.
- When converting a numeric value to Boolean, 0 or 0.0 will convert to `False` and non-zero values to `True`.
- When converting a string to a Boolean, the empty string is converted to `False`, and any other string is converted to `True`. This feature can be used to find empty strings.

4.3 Operators

Operators are symbols that act on values. Python provides the following types of operators. Knowledge of these operators will be helpful when

performing text analysis:

- Arithmetic operators
- Comparison operators
- Logical operators (for working with Boolean values)
- Membership operators (for checking whether something is part of a collection)

4.3.1 Arithmetic Operators

The following table shows Python's arithmetic operators. Since many of these operators do not require explanation, we will only describe operators that may be unfamiliar. We will also explain some subtleties that arise when working with mixed data types.

Operator	What It Does	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`4.0 - 19.3`	`-15.3`
`*`	Multiplication	`2 * 3`	`6`
`/`	Division	`3.1514 / 2`	`1.57075`
`//`	Floor division	`7 // 2`	`3`
`%`	Mod (remainder)	`14 % 3`	`2`
`**`	Exponentiation	`3.0 ** 4`	`81`

Table 4.3: Python's arithmetic operators.

Floor division (`//`) performs division and then truncates any fractional portion of the result. The mod operator (`%`) performs division and returns the remainder.

Working with Mixed Data Types

Generally, if both operands are integers, the result will be an integer and if either value is a float, the result will be a float. The only exception is the division operator which *always* returns a float. The division operator behaves this way to guarantee that it does not throw away information.

4.3.2 Comparison Operators

Comparison operators are used to compare two values. All comparison operators evaluate to a Boolean type (`True` or `False`).

Operator	What It Does	Example
`>`	Greater than	`x > y`
`>=`	Greater than or equal to	`x >= y`
`<`	Less than	`x < y`
`<=`	Less than or equal to	`x <= y`
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`

Table 4.4: Python's comparison operators.

The Equal To (==) and Not Equal To (!=) Operators

Python uses the double equal sign for comparing values in order to distinguish between assignment (assigning a value to a variable) and comparison (comparing two values). For example:

In:

```
x = 5  # Set the value of x to 5.
if x == 5:
    print('x equals 5.')
```

Out:

```
x equals 5.
```

Also, Python followed the convention of the C programming language, which uses these comparison operators.

4.3.3 Logical Operators

Logical operators operate on Boolean values. They are typically used to combine multiple conditions. For example, say we have a dataset of company financial statements and we want to filter it to years after 2010 and to companies with revenue in excess of $1 billion. We can do this by creating two conditions and joining them with the **`and`** operator as follows:

```
(year > 2010) and (revenue > 1000)
```

where the `revenue` value is in millions.

Python has three logical operators: **`and`**, **`or`**, and **`not`**. We summarize them in the following table.

Operator	What It Does	Example
`and`	Returns `True` if both operands are true	`x and y`
`or`	Returns `True` if either or both operands are true	`x or y`
`not`	Returns the opposite of its operand	`not x`

Table 4.5: Python's logical operators

4.3.4 Membership Operators

Membership operators test whether a value is part of a collection, such as a list.[1] Two useful membership operators are **`in`** and **`not in`**. The **`in`** operator returns `True` if its first operand belongs to its second operand, while **`not in`** returns the opposite.

Operator	What It Does	Example
`in`	Returns `True` if `x` is in the collection `y`	`x in y`
`not in`	Returns `True` if `x` is not in the collection `y`	`x not in y`

Table 4.6: Python's membership operators

For example:

In:

```
myList = [1,2,3] # list of integers 1, 2, and 3

if 5 in myList:
    print('5 is in the list.')
else:
    print('5 is not in the list.')
```

Out:

```
5 is not in the list.
```

Another common use of the **`in`** operator is to iterate over a collection. For example:

In:

[1] We introduce collections in Section 4.7.

```
myList = [1,2,3]

for x in myList:
    print(x)
```

Out:

```
1
2
3
```

4.3.5 Operator Precedence / Order of Operations

Precedence	Operator	Description
Highest	`**`	Exponentiation
	`-`	Negative sign, e.g., `-3`
	`*`, `/`, `//`, `%`	Multiplication, division, remainder
	`+`, `-`	Addition and subtraction
	`<`, `<=`, `>`, `>=`, `!=`, `==`, `in`, `not in`	Comparison and membership operators
	`not`	Boolean NOT
	`and`	Boolean AND
Lowest	`or`	Boolean OR

Table 4.7: Operator precedence table. When Python evaluates an expression, it first evaluates the operator with the highest precedence, then the second-highest, and so on. If two or more operators have equal precedence, Python evaluates them from left to right.

Python has rules for operator precedence. When Python encounters an expression, it first evaluates the operator with the highest precedence and substitutes the result. It then evaluates the operator with the second highest precedence, and so on until the expression reduces to a single value. The precedence of the operators introduced in this chapter are summarized in Table 4.7.[2] If Python encounters two or more operators with identical precedence, it evaluates the leftmost operator first.

Consider the following example. According to Table 4.7, the operator with the highest precedence in the expression is `*`. Since there are two

[2]See here for the complete, official rules of operator precedence.

multiplication operators, the leftmost one is given higher precedence. Thus, the 3 and 4 are multiplied and substituted with 12. The next operator evaluated is the second *. This is applied to the operands 12 and 2 and the value 24 is substituted. This continues until all operators have been evaluated and the entire expression reduces to a single value. The steps that Python takes internally to reduce the expression are shown in the sample code below.

In:

```
not 4 + 3 * 4 * 2 == 28
```

Out:

```
not 4 + 3 * 4 * 2 == 28 # Starting expression
not 4 + 12 * 2 == 28    # 3*4 --> 12
not 4 + 24 == 28        # 12*2 --> 24
not 28 == 28            # 4+24 --> 28
not True                # 28==28 --> True
False                   # not True --> False
```

Finally, note that *parentheses override* the operator precedence rules and can be used to specify any desired order of evaluation. Consider a modified version of the example above.

In:

```
not (4 + 3) * 4 * 2 == 28
```

Out:

```
True
```

By adding parentheses, we forced Python to evaluate the + operator first. The expression reduces to `not 56 == 28`, which reduces to `True`.

4.4 The print Function

Python has a built-in **print** function. When performing text analysis, it is useful to include the **print** function for testing and debugging. For example, as we write code, we may wish to print the value of a variable or expression. In this section, we will demonstrate the basics of printing. We introduce the **print** function here because we use it extensively in the remainder of this chapter. We only demonstrate a few features of **print**.

We caution the reader that the syntax for **print** has changed substantially in recent years. In this paper, we demonstrate the newest syntax, *f-strings*. However, many online code samples use older styles. For a thorough overview of all printing styles, we refer the reader to this article.

4.4.1 Printing a Single Value or Variable

To print a value, simply pass the value as an argument to **print**:

In:

```
pi = 3.14
print(pi)
```

Out:

```
3.14
```

Note that when we try to print a value or variable whose type is not string, **print** automatically uses the **str()** function to convert its argument to a string.

4.4.2 Printing Multiple Values

We can pass more than one argument to print. For example:

In:

```
print(5, -3, 'EDGAR')
```

Out:

```
5 -3 EDGAR
```

4.4.3 Printing More Complicated Expressions

A common scenario is to substitute the values of variables into a string and then print that string. For example, say we want to print a sentence that includes the name, fiscal year, and net income of a company, all of which are stored in variables. The simplest and most elegant way to perform this is through *f-strings*. Consider the following example:

In:

```
CompanyName = 'General Motors'
FiscalYear = 2019
NetIncome = 6581000000
```

```
print(f'In FY{FiscalYear}, {CompanyName.upper()} had
    net income of {NetIncome / 1e9} billion.')
```

Out:

```
In fiscal year 2019, GENERAL MOTORS had net income of
    6.581 billion.
```

When Python sees the letter *f* before a string, it treats the string differently. Inside the string, Python looks for expressions inside curly braces. It evaluates those expressions and replaces them with the string equivalents of their values. Notice that the expression `{FiscalYear}` was replaced with the value `2019`, the value of the variable `FiscalYear`. Also notice that the expressions inside f-strings need not contain variables; they can contain any valid Python expression. Notice the second expression, `{CompanyName.upper()}`. The `upper()` method converts a string to all upper-case. Also notice that the third expression divides the variable `NetIncome` by one billion.[3]

4.5 Control Flow

Control flow is the order in which statements are executed in a program. In this section, we will discuss basic control flow in Python, specifically branching (`if` statements) and loops (`while` and `for`).

4.5.1 `if` Statements

The syntax for Python `if` statement is:

```
if condition_A:
    A statements
elif condition_B:
    B statements
elif condition_C:
    C statements
else:
    else_statements
```

An `if` statement begins with the keyword `if`. This keyword is followed by a condition, and then a colon (:). The condition must be an

[3] Python accepts numbers in scientific notation. `1e9` is equivalent to 1.0×10^9.

expression that evaluates to, or can be converted to, a Boolean value. Logical operators can be used to join multiple conditions and create complex logic. All statements to be executed if the condition is true must begin on the next line and must be indented. Thus, in the above example, if `condition_A` evaluates to `True`, all of the `A statements` will be executed. Note that the `A statements` can contain another **`if`** statement. This is called "nesting" and an example is shown below.

```
if x > y:
    print(x)
    if (y > 3):
        print(y)
```

In this example, the value of the variable `x` is printed if `x` is greater than `y`. The nested if statement then checks whether `y` is greater than 3. If so, the value of `y` is printed. If x is less than or equal to y, nothing is printed and neither the condition, `(y < 3)`, nor the **`print`** statement in the nested **`if`** is executed.

Python allows optional **`elif`**[4] and **`else`** clauses. Any number of **`elif`** clauses is allowed, but only one **`else`** clause is allowed. These rules allow for arbitrarily complex logic. Also note that Python checks the clauses in order and stops execution after one clause evaluates to `True`. Consider the following example, which assumes that a variable `age` exists in the environment and contains a numeric value.

In:

```
age = 25

if (age < 13):
    print('child')
elif (age < 20):
    print('teenager')
elif (age < 30):
    print('young adult')
elif (age < 50):
    print('middle aged')
else:
    print('old')
```

Out:

```
young adult
```

[4] `elif` means "else if".

If the variable `age` equals 25, Python will check the first condition `(age < 13)`. This will evaluate to `False`. Python will then check the first `elif`, which contains the condition `(age < 20)`. This will evaluate to `False`, so Python will check the second `elif` condition, `(age < 30)`. This will evaluate to `True`, so Python will execute the corresponding statement and print `'young adult'`. Python will then stop execution.

4.5.2 Loops

A loop executes an action while a condition is met, then terminates.

`while` Loops

As its name suggests, a `while` loop executes while a condition is true. In Python, the `while` loop first checks whether its condition is true. If it is, the loop takes some actions and then rechecks the condition. If the condition is still true, the loop executes the actions again. This process repeats until the condition is false.[5]

The syntax of a `while` loop is shown below. The colon and the indentation are mandatory. Note that the statements can include `if` statements and other `while` loops.

```
while condition:
    statement 1
    statement 2
    ...
```

`for` Loops

A `for` loop is very similar to a `while` loop. It will test a condition and execute the body of the loop if the condition is satisfied. The difference is that a `for` loop will *iterate*, or loop over, something automatically. In other words, a `for` loop is a `while` loop with some built-in conveniences.

The syntax of a `for` loop is:

[5] A common programming mistake is to accidentally run an "infinite loop". This results when the actions do not update the condition and the condition always stays true.

```
for variable in something_iterable:
    statement 1
    statement 2
    ...
```

A common scenario is to iterate over a collection of data, such as a list, and do something to every item in the collection. Consider the following example that prints every item in a list.

In:

```
for x in [1,2,3,7]:
    print(x)
```

Out:

```
1
2
3
7
```

The code above creates a list with the values, 1, 2, 3, and 7. The **for** loop iterates over the list. It does so by automatically creating a new variable that we named **x**. The loop sets **x** to the first item in the list, which in this case is 1. It then executes the body of the loop, which prints **x**. The loop then sets **x** to the next thing on the list, 2, and executes the body of the loop. This continues until the loop has iterated over every value in the list.[6,7]

Another common scenario is to execute an action a given number of times. This is typically accomplished using the built-in **range()** function. In the example below, the **range** function returns the sequence 0, 1, 2, 3, 4, 5, 6. The **for** loop will iterate over this sequence and print every value in the sequence.

In:

[6] It is possible to implement the same functionality through a **while** loop. However, doing so requires more machinery. The code would need to keep track of the index of each item in the list and ensure that the number of iterations exactly equals the list length. Additionally, the code would need to use indexing to retrieve the list element that corresponds to the iteration number. The **for** loop is much simpler and less error-prone.

[7] When iterating over a list, you may need the list indexes as well as the elements. Python provides the **enumerate** function for this purpose. We provide an example of this function in section 7.3. We do not discuss this function here since it requires knowledge of lists (section 4.7.1) and tuples (section 4.7.2).

```
for x in range(7):
    print(x)
```

Out:

```
0
1
2
3
4
5
6
```

Note that by default the `range` function starts at zero. Thus, `range(7)` returns a sequence of length 7 that begins at zero and ends at 6. This is common in many programming languages.[8] To start the `range` function at an arbitrary integer `k`, instead of the default 0, use `range(k, n)`. This will return the sequence $k, k+1, \ldots, n-1$. Note that if $k \geq n$, the `range()` function will return an empty sequence and the loop will not execute.

4.6 Functions

A function is a reusable block of code. In this section, we will first show how to create functions in Python. We will then show how to use functions written by others. Such functions are typically organized into modules;[9] we will show how to reference functions stored in modules.

4.6.1 Writing Functions

A Simple Function

Consider the following code. It defines a function, `MyAverage`, that computes the average of its two inputs.

[8] Under the hood, programming languages keep track of the memory address of the first element of a collection. The index is then used as an offset to that memory address. That is why the index 0, which implies an offset of 0, returns the first element.

[9] The terms *package* and *library* are synonyms for module and we will use these terms interchangeably.

```
def MyAverage(x, y):
    return (x + y) / 2
```

A function definition begins with the keyword **def**, which stands for *define*. Following the **def** keyword is a name for the function. Next is an argument list; this list must be enclosed in parentheses and separated by commas. Our `MyAverage` function takes two arguments, `x` and `y`. These arguments will create variables, but these variables will only "live" inside the function. The first line of the function definition must end with a colon. The next lines are the body of the function and they must be indented. Python uses the indentation to determine which statements constitute the "body" of the function. Python will execute the body of the function, line by line, until it reaches a **return** statement, or until it executes the last indented statement. A **return** statement, if provided, "returns" a value to whatever line of code called the function.

Syntax for Defining Functions

To define a function, use the following syntax:

```
def FunctionName(arg1, arg2, ...):
    statement1
    statement2
    ...
    return return_value
```

It is possible to define a function with zero arguments. To do so, place open and close parentheses after the function name (e.g., `def FunctionName():`).

Functions need not return anything. If you do not wish to return anything from your function, you may omit the return statement or use a **return** statement with no return value.

Rules for Naming Functions

The rules for naming functions are the same as the rules for naming variables. Function names can contain upper- and lowercase letters, digits, and the underscore character. Function names cannot contain spaces and cannot begin with a digit.

Arguments are Local Variables

In Section 4.2.2, we introduced the concepts of environment and state. When a function is called, Python creates a new environment for that function. The new environment inherits everything from its calling environment. Any arguments to the function are created as variables in the function's environment. Those variables, and the function's environment, are deleted once the function returns a value or finishes execution.

Functions may reference variables in their calling environment. However, if a function attempts to alter a variable in its calling environment, Python will create a new local variable with the same name. Any changes made to the local variable do not affect the calling environment. This subtlety is best illustrated with an example:

In:

```
def SampleFunction():
    x = 0
    print(f'Inside SampleFunction, x is {x}')

x = 15
SampleFunction()
print(f'Outside SampleFunction, x is {x}')
```

Out:

```
Inside SampleFunction, x is 0
Outside SampleFunction, x is 15
```

In the above code, the variable `x` is created in some environment and set to 15. When `SampleFunction()` is called, it tries to assign the value 0 to `x`. When Python sees this, it creates a new local variable `x` and sets its value to 0. This new `x` will live only within the environment of `SampleFunction`. Additionally, the new symbol `x` *masks* the variable `x` in the top-level environment. Thus, when the function exits, the local `x` is destroyed and the top-level `x` is unmasked.

It is possible for functions to alter variables in calling environments, but we strongly discourage this practice. In good programming practice, functions act as "black boxes." What happens inside the function should stay inside the function, and a function should not directly alter its calling environment.

Anonymous Functions / Lambda Expressions

Python allows programmers to create anonymous functions "on the fly." A common use case occurs when performing an operation on each row of a dataset; sometimes, there is no available, preexisting function for the desired operation. A programmer could create the function with a `def` statement, but that would create unnecessary clutter. It is simpler and easier to use an anonymous function.

Use the `lambda` keyword to create an anonymous function. The following example demonstrates the creation and use of an anonymous function:

In:

```
# Create the list [0,1,...9]
L = list(range(10))

# Extract all elements of L greater than 7
# Save to new list named filteredList
filteredList = list(filter(lambda x: x > 7, L))

print(filteredList)
```

Out:

```
[8, 9]
```

The above example creates a list `L`. The second line of code extracts all elements of `L` greater than 7 and saves them to a new list. We used a lambda expression to perform the filtering. The expression, `lambda x: x > 7`, begins with the `lambda` keyword and is followed by an argument `x` and a colon.[10] The body of the function follows the colon. In this case, the function body is simply `x > 7`, which evaluates to a Boolean. The `filter()` function applies the anonymous function to every element of the list `L` and keeps list elements for which the anonymous function evaluates to True. Notice that the anonymous function is created inline, used once, and discarded.

[10] Lambda expressions allow multiple arguments. Simply separate the arguments with commas.

4.6.2 Using Functions Written By Others

Researchers performing text analysis in Python will mostly work with functions written by others. In this section, we demonstrate how to work with such functions. We discuss how functions are organized into *modules* and show how to call a function from a module.

In programming, functions behave like "black boxes"; a function receives inputs (arguments) and emits an output (the return value). Python supports two types of arguments, positional arguments and keyword arguments. We will demonstrate how to use both types.

Modules and `import` Statements

Since Python is so popular, people have written thousands of Python functions that researchers can download and use in their own work. Without organizing those functions, many problems would result. For example, say we want to write a function to filter some data. A natural name for such a function is **`filter`**, but that function already exists in Python's standard library. Additionally, many programmers find it useful to bundle a set of related functions. The solution to these problems is to package related functions into a *module* (sometimes called a *package* or a *library*). A module is simply a group of functions that reside in the same file.

When we open Python, it loads some built-in functions into the environment (see here). Python also provides a "standard library" that contains many modules with useful functions. If we want to use one of those modules, we have to tell Python by using an **`import`** statement. For example, to use the functions in Python's math library, we would type:

```
import math
```

That statement tells Python to load all of the functions from its math module into the environment. After we execute this **`import`** statement, we can use any of the functions in the module. For example, the following code computes the factorial of a number using a function from the math

library.

In:

```
import math
print(math.factorial(5))
```

Out:

```
120
```

Note that the name of the function is `math.factorial`, *not* `factorial`. Readers might wonder why Python requires them to prefix the function name with "`math.`." The reason is that, without such a prefix, an import statement might hide, or mask, other functions in the environment with the same name.

Readers who do not wish to type a prefix have two options. They can either import the function directly from a module into their environment, or define an alias for the module. Both options are commonly used by Python programmers.

Option 1: Import the functions directly

If we do not want to type `math.` before every function name, we can use something like the following. However, we discourage this practice as it clutters the environment with unneeded function names and increases the chance of a "name collision."

```
from math import factorial, log, sin
```

This code imports specific functions from the math library. Additionally, such functions do not require the prefix "`math.`." If someone wishes to import all functions from a library, they can use something like the following:

```
from math import *
```

Option 2: Define an alias for the module

Some libraries have long names, such as the `statistics` module. Typing `statistics.` before every function would be tedious. To prevent this, Python allows users to create an alias for a library name. For example:

In:

```
import statistics as st
```

```
mydata = [1,3,5,7,9]
my_median = st.median(mydata)

print(f'The median of my data is {my_median}')
```

Out:

```
The median of my data is 5
```

The above code tells Python to use `st` as an alias for the statistics module. An alias can be any valid variable name.

4.6.3 Calling Functions

Consider the following code. It creates a variable, `text`, and then calls a function `re.sub`[11] to substitute all occurrences of the string `'OI'` with the more informative string `'Operating Income'`. Notice that we passed three arguments to `re.sub`. The first argument is the search text, the second argument is the replacement text, and the third argument is the text to search.

In:

```
import re
text = "OI for FY 2019 was 12.4 billion, up more than
    eight percent from OI in FY 2018."
pretty_text = re.sub("OI", "Operating Income", text)
print(pretty_text)
```

Out:

```
Operating Income for FY 2019 was 12.4 billion, up more
    than eight percent from Operating Income in FY
    2018.
```

Positional Arguments

If we do not tell it otherwise, `re.sub` assumes that the first argument is the search text, the second argument is the replacement text, and the third argument is the text to search. Functions called in this manner rely on *positional arguments*: the position of the argument in the function call has meaning. When using positional arguments, we must read the

[11] In the remainder of this section, we will work with the function `re.sub` from Python's built-in regular expression library. Regular expressions are a powerful tool for finding patterns in text. We introduce regular expressions in Chapter 6.

function documentation to learn the correct order of the arguments. The argument order is specified in the function's signature. For example, the documentation for `re.sub` provides this function's signature:

```
re.sub(pattern, repl, string, count=0, flags=0)
```

The function signature specifies the arguments that a function accepts. If an argument is followed by an equals sign and a value, that argument is optional. If the argument is omitted, a default value (0 in this case) is used.

Many Python programmers use positional arguments when calling frequently-used functions that accept few arguments. However, many functions accept dozens of arguments, many of them optional, making it difficult to remember which argument is which. In this situation, we recommend the reader to consider *keyword arguments*, which we discuss next.

Keyword Arguments

Consider the call to `re.sub` in the previous example. An alternative method of calling this function is:

```
pretty_text = re.sub(pattern="OI",
                     repl="Operating Income",
                     string=text)
```

In the modified code, we explicitly told Python that the value of the `pattern` argument is `"OI"`, the value of `repl` is `"Operating Income"`, and the value of `string` is `text`. A function called this way relies on *keyword arguments*. Keyword arguments provide many advantages over positional arguments. Code readability is increased since the arguments are clearly specified. Additionally, arguments can be passed *in any order*. Thus, this function call would yield an identical result:

```
pretty_text = re.sub(string=text,
                     repl="Operating Income",
                     pattern="OI")
```

Mixing Positional and Keyword Arguments

Python allows function calls with positional and keyword arguments. The only stipulation is that once a keyword argument is used, all remaining arguments must be keyword arguments. Thus, the following is allowed:

In:

```
re.sub("OI", "Operating Income", string=text)
```

Out:

```
'Operating Income for FY 2019 was 12.4 billion, up more
    than eight percent from Operating Income in FY
    2018.'
```

However, the following would raise an error since a positional argument follows a keyword argument:

In:

```
re.sub(pattern="OI", "Operating Income", text)
```

Out:

```
  File "<ipython-input-37-1a96811d74d8>", line 1
    re.sub(pattern="OI", "Operating Income", text)
                         ^
SyntaxError: positional argument follows keyword
    argument
```

4.7 Collections - Lists, Tuples, and Dictionaries

Python provides many ways of storing collections of data. In this section, we introduce three data structures for storing data collections: lists, tuples, and dictionaries. In the next chapter, we will introduce another data structure for storing collections of data, the Pandas DataFrame. Much of the syntax for lists and dictionaries applies to Pandas DataFrames.

4.7.1 Lists

A list is a mutable collection of objects. Python lists can contain data of different data types. List items can be added, modified, and deleted.

Creating a List

Create a list by enclosing data inside square brackets and separating each data item with a comma. Spaces between commas are optional. The following example creates a list and saves the list into a variable, `L`. The list contains three elements, a string, a float, and a list. Note that the type of the variable is `list`. Also note that it is possible to nest lists within lists.

In:

```
L = ['GM', -3.14, [1, 2, 3]]
type(L)
```

Out:

```
list
```

Creating, Transforming, and Filtering Lists Using List Comprehensions

Python programmers often need to transform, filter, or create lists from a pattern. Consider the task of transforming every element of a list and returning a new list. For example, double every element in a list. One way to do this is to create a new empty list and then use a `for` loop to iterate over a source list. In the `for` loop, each element of the source list is doubled and added to the new list. This "brute force" approach will work, but is complicated and error-prone. To simplify this task, Python provides "syntactic sugar" called a *list comprehension*.

The following code creates a list and then uses a list comprehension to double every element and return a new list:

In:

```
L = [1, 2, 3, 4]
doubles = [x*2 for x in L]
print(doubles)
```

Out:

```
[2, 4, 6, 8]
```

Notice that the new list, `doubles`, is created in one line of code. The code inside the square brackets, `[x*2 for x in L]`, is a list comprehension. The general form of a list comprehension is:

```
[expression for var in list]
```

where `expression` is a valid expression containing the variable `var`. In the above example, `expression` is `x*2`, `var` is `x`, and `list` is `L`. The list comprehension embeds a for loop that iterates over the source list `L`. Each element of the source list is transformed using the expression and stored in a new list. Note that a list comprehension does not modify its source list.

Consider the task of filtering a list. The brute force method would iterate over a source list and append elements that meet the filtering criteria to a new list that must be created beforehand. This task is more easily accomplished with a list comprehension. To illustrate this, we borrow an example from section 10.4. A list, `words`, needs filtering; any "stop words" in the list must be removed. The stop words are stored in another list `stop_words`. The following code demonstrates filtering with a list comprehension.

```
filtered_words = [w for w in words if w not in stop_words]
```

This list comprehension uses a more general form of list comprehension:

```
[expression for var in list if condition]
```

In this more general form of list comprehension, `expression` is only evaluated and kept only for items that meet the condition. If the condition is not met, `expression` is not evaluated and not stored in the new list. Note that in the above code, the expression `w` does not transform the list items. It simply copies list items that meet the condition. Thus, this list comprehension implements a pure filter.

Retrieving List Elements

In Python, each list element is assigned an index. List indexes begin at 0 and end at $n-1$, where n is the number of elements in the list. Thus, in the previous example, the list element `'GM'` has an index of 0, `-3.14` has an index of 1, and `[1,2,3]` has an index of 2.

To access a single element of a list, type the name of the list and then enter the list index in square brackets. For example:

In:

```
L[1]
```

Out:

```
-3.14
```

Python provides a shortcut to retrieve elements from the end of the list. The list index -1, refers to the last element of the list, -2 to the element that precedes the last, and so on. For example:

In:

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

Out:

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

Slicing

Python makes it very easy to retrieve more than one element of a list. This process is called *slicing*. In our opinion, slicing is one of the most useful features of Python.

The syntax for slicing a list is:

```
list_name[start:end:step]
```

where `start` is the index of the first element we wish to retrieve (inclusive lower bound), `end` is one more than the index of the last element we wish to retrieve (exclusive upper bound), and `step` is the step size, or gap between indexes. Note that `start`, `end`, and `step` are optional. If omitted, Python assumes 0 for `start`, the list length for `end`, and 1 for `step`. Slicing is best illustrated with examples:

In:

```
L = ['a', 'b', 'c', 'd', 'e', 'f']

print(f'Example 1: {L[0:2]}')
print(f'Example 2: {L[2:2]}')
print(f'Example 3: {L[4:]}')
print(f'Example 4: {L[:2]}')
print(f'Example 5: {L[0:6:2]}')
print(f'Example 6: {L[5:2:-1]}')
```

Out:

```
Example 1: ['a', 'b']
Example 2: []
Example 3: ['e', 'f']
Example 4: ['a', 'b']
Example 5: ['a', 'c', 'e']
Example 6: ['f', 'e', 'd']
```

Example 1 tells Python to retrieve all elements beginning with index 0 (the first element) and ending with one minus the `end` parameter of 2. This yields the first two elements of the list. Example 2 tells Python to start at index 2 and end at index 1. No list indices meet these criteria so the empty list is returned. Examples 3 and 4 illustrate the optional nature of the parameters; example 3 retrieves the list element with index 4 and all remaining list elements; example 4 retrieves all elements with indices less than 2. Example 5 retrieves elements with indices 0, 2, and 4; it begins with index 0 and increments the index in steps of 2. Finally, the last example demonstrates how to retrieve list elements in reverse order. It begins with index 5 and retrieves all elements with indexes strictly greater than 2 (recall that the `end` parameter is an exclusive bound).

Modifying List Elements

To modify a single element of a list, simply assign a value to that list element using the indexing method demonstrated above. Thus, to modify the element `'b'` in the previous example:

In:

```
L[1] = 6
print(L)
```

Out:

```
['a', 6, 'c', 'd', 'e', 'f']
```

To modify multiple elements simultaneously, simply assign to a list slice:

In:

```
L[4:6] = [-999, 0]
print(L)
```

Out:

```
['a', 6, 'c', 'd', -999, 0]
```

Other List Operations

Python provides many useful functions for manipulating lists. In this section, we will demonstrate how to concatenate (join) lists, duplicate lists, copy lists, retrieve list length, and add and remove elements from lists.

Concatenating Lists

Use the + operator to concatenate, or join, lists.

In:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
list1 + list2
```

Out:

```
['a', 'b', 'c', 1, 2, 3]
```

Duplicating Lists

The * operator makes copies of a list and concatenates them into a new list.

In:

```
L = [1, 2, 3]
L * 3
```

Out:

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

Copying Lists

To demonstrate list copying, we must first introduce the concepts of shallow and deep copies. Say that a list is stored in the variable `L`. The statement `L2 = L` creates a shallow copy. It creates a new symbol in the environment, `L2`, but that symbol points to the same data as `L`. Thus, a change made to `L2` will affect `L`. To see this, consider the following code:

In:

```
L = [1, 2, 3]

L2 = L
```

```
L2[0] = 'text'

print(f'L = {L}')
print(f'L2 = {L2}')
```

Out:

```
L = ['text', 2, 3]
L2 = ['text', 2, 3]
```

To make a deep copy of a list, you must use a list's `copy` method. This method duplicates the list in memory and prevents behavior like that shown in the previous example. The following code demonstrates the `copy` method of list.

In:

```
L = [1, 2, 3]

L2 = L.copy()
L2[0] = 'text'

print(f'L = {L}')
print(f'L2 = {L2}')
```

Out:

```
L = [1, 2, 3]
L2 = ['text', 2, 3]
```

Getting the Length of a List

The `len` function retrieves the length of a list. Thus, `len([1,2,3])` returns 3.

Adding and Removing Elements from Lists

Use the `append` method of list to add a new element to the end of a list.

In:

```
L = [1, 2, 3]
L.append('cat')
print(L)
```

Out:

```
[1, 2, 3, 'cat']
```

Python also provides the `insert` and `remove` methods to insert and remove list elements. `L.insert(i, x)` inserts `x` at the index given by `i`. `L.remove(x)` removes the first item from `L` where `L[i]` is equal to `x`.

4.7.2 Tuples

A tuple is an immutable list - it cannot be changed after it is created. The syntax for tuples is nearly identical to that for lists. The main difference is that tuples use parentheses `( )` whereas lists use square brackets `[ ]`. The process for retrieving an element from a tuple is identical to that for lists. We mention tuples because many Python functions either require tuples as arguments or return tuples. For example, the Pandas DataFrame, which we introduce in the next chapter, stores a table of data. Pandas provides a function that retrieves the dimensions of a DataFrame and this function returns a tuple containing the number of rows and columns.

4.7.3 Dictionaries

A dictionary is a list of key-value pairs. Dictionaries are very useful data structures since they allow users to assign data items (values) to keys of their choice. This makes it easier to store and retrieve data. For example, consider the following code that stores an income statement as a dictionary.

```
income_stmt = {'Revenue': 100,
               'COGS': 52,
               'Gross margin': 48,
               'SG&A': 40,
               'Net Income': 8}
```

To create a dictionary, use curly braces `{ }`. Within the curly braces, enter a key, followed by a colon, followed by the value. Separate each key-value pair with a comma. Note that Python permits newlines inside a dictionary to increase code readability. Keys can be numbers, strings, or tuples, and *the keys must be unique* (i.e., no repeated keys). The values can be any valid data type.

To retrieve a value from a dictionary, use the same syntax for lists. However, instead of using a numerical index, use a key.

In:

```
income_stmt['Revenue']
```

Out:

```
100
```

To add a key-value pair to a dictionary, simply assign the value to a new key and Python will add the key-value pair to the dictionary. Thus,

```
income_stmt['Fiscal Year'] = 2018
```

will create a new key `'Fiscal Year'` in the dictionary `income_stmt` and assign it the value `2018`.

To modify a value in the dictionary, simply set the new value using a key. Say, for example, we wish to change the fiscal year that we just added. We would type:

```
income_stmt['Fiscal Year'] = 1998
```

Python provides support for many other built-in dictionary operations. You can get the length of the dictionary (number of key-value pairs), delete dictionary keys (and their values), clear all keys and values, copy the dictionary, retrieve only the keys, retrieve only the values, etc. We do not demonstrate those here.

4.8 Working with Strings

In many ways, Python strings behave like lists. Strings can be sliced, joined using the + operator, duplicated using the * operator, just like lists. That is why this section appears after the section on lists.

4.8.1 Strings as Lists of Characters

Conceptually, a string is just a list of characters. To see this, consider the following code and its output. The code uses Python's built-in `list` function to split a string into a list of single-character strings. This illustrates how strings can be perceived as lists of characters.

In:

```
s = 'Hello world.'
list(s)
```

Out:

```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd',
    '.']
```

4.8.2 Extracting Pieces of Strings

To extract a substring, use list indexing or slicing notation.

In:

```
s = 'Hello world.'
print(s[-1])
print(s[0:5])
```

Out:

```
.
Hello
```

4.8.3 String Operations

Joining Strings

Use the + operator to join (concatenate) two strings.

In:

```
s1 = 'Hello'
s2 = 'World'
s1 + ' ' + s2 + '.'
```

Out:

```
'Hello World.'
```

Repeating Strings

Use the * operator to repeat a string. **In:**

```
s1 = 'Hello'
s1 * 3
```

Out:

```
'HelloHelloHello'
```

Checking Whether a String Contains a Substring

Use the `in` operator to check whether a string contains a substring. The expression `substr in s` evaluates to `True` if substr appears anywhere in `s`, and `False` otherwise. In the following example, note that the second expression evaluates to `False` because Python strings are case-sensitive.

In:

```
s = 'Foundations and Trends in Accounting'
print('Accounting' in s)
print('AND' in s)
```

Out:

```
True
False
```

4.8.4 Useful String Functions and Methods

Following are some useful functions for working with strings. We only demonstrate a subset of the features of these functions. Many of these functions have optional arguments whose use we do not demonstrate. To learn more about these functions, see the official documentation on Python's built-in string methods.

We distinguish between functions and methods. A function is a reusable block of code. A method is a specific type of function. Python (and many other programming languages) support a construct known as objects and implement "object-oriented programming." An object is a container for data and may contain its own functions. For example, lists and strings are implemented as objects in Python. Functions that belong to an object are called *methods*. Methods are called differently than ordinary functions. Recall the `append` method of the list object. We called it using a different syntax than we use for ordinary functions, e.g., `L.append(x)`. Methods are called by typing an object's name, followed by a dot (`.`), followed by the method name, then any arguments in parentheses.

Getting the Length of a String

Use the `len` function, e.g., `len('abcd')` returns 4.

`lower` and `upper`

`lower` and `upper` convert a string to all lower case or all upper case, respectively. These functions return a new string and do not modify the original string.

In:

```
s = 'Hello world.'
print(s.upper())
print(s.lower())
```

Out:

```
HELLO WORLD.
hello world.
```

`strip`, `lstrip`, and `rstrip`

A common need when cleaning text data is to remove white space from strings. Python provides the `strip` method for this purpose. `strip` removes all white space characters from the beginning and end of a string. White space includes space, tab, and newline characters.

In:

```
s = '  Dirty string with unnecessary spaces at
    beginning and end.  '
s.strip()
```

Out:

```
'Dirty string with unnecessary spaces at beginning and
    end.'
```

Notice that `strip` removed the spaces from the beginning *and* the end of the string. If we wish to remove white space from only the beginning (end) of a string, use `lstrip` (`rstrip`).

By default, these methods remove white space, but we can pass an optional argument to tell these methods which characters to strip.

In:

```
s = 'Hello.'
s.rstrip('.')
```

Out:

```
'Hello'
```

These functions return a new string and do not modify the original string.

`startswith` and `endswith`

Use the `startswith` and `endswith` to determine whether a string starts or ends with a given substring. These methods are similar to the `in` operator, but they only test the beginning or end of a string. These methods return `True` or `False`.

In:

```
s = '111 33'
print(s.startswith('11'))
print(s.endswith('x'))
```

Out:

```
True
False
```

`find`

`find` searches for a substring within a string. If the substring is found, `find` returns the *index* of the first occurrence. If the substring is not found, `find` returns -1.

In:

```
s = 'text analysis'
s.find('xt a')
```

Out:

```
2
```

`replace`

Use `replace` to replace one substring with another. By default, it replaces all occurrences of a substring. An optional *count* argument allows to specify the number of replacements.

In:

```
s = 'text analysis'
s.replace('text', 'TEXT')
```

Out:

```
'TEXT analysis'
```

This function returns a new string and does not modify the original string.

Count Occurrences of a Substring

The `count` method returns the number of times a substring occurs in a string.

In:

```
s = "At FedEx Ground, we have the market leading e-
    commerce portfolio. We continue to see strong
    demand across all customer segments with our new
    seven-day service. We will increase our speed
    advantage during the New Year. Our Sunday roll-out
    will speed up some lanes by one and two full
    transit days. This will increase our advantage
    significantly. And as you know, we are already
    faster by at least one day when compared to UPS's
    ground service in 25% of lanes. It is also really
    important to note our speed advantage and seven-day
     service is also very valuable for the premium B2B
    sectors, including healthcare and perishables
    shippers. Now, turning to Q2, I'm not pleased with
    our financial results."
s.lower().count('we')
```

Out:

```
4
```

split

The `split` method splits a string and returns a list of substrings. By default, it splits using spaces but we can tell Python which delimiter we wish to use.

In:

```
s = 'text analysis is fun.'
s.split()
```

Out:

```
['text', 'analysis', 'is', 'fun.']
```

5

Working with Tabular Data: The Pandas Package

Text analysis typically involves unstructured and structured data. Unstructured data is data that is not organized in a pre-defined manner, such as a 10-K or an image. A 10-K is unstructured since each company organizes its 10-K differently. Much of this paper deals with unstructured text data. However, researchers performing text analysis will also work with structured data such as SAS and Stata datasets, Excel files, and CSV files corresponding to Compustat, IBES, and CRSP data. Structured data is organized into pre-defined columns, each of which contains a specific type of data, and rows, each of which contains an observation.

This chapter introduces the popular Pandas package for working with structured data. Pandas is an acronym that stands for "panel data." Pandas was originally developed for financial data and has since grown in size and scope to be an excellent, general-purpose library for manipulating structured data. If you decide to do your data munging in Python, then Pandas will almost certainly be your "go to" library. Even if you continue to use SAS or Stata for data munging, you will eventually import your data into Python for use in text analysis and will use Pandas.

5.1 The Main Objects in Pandas

The Pandas library contains two main objects, the *DataFrame* and the *Series*.

5.1.1 The Pandas DataFrame

A DataFrame is a table of data. It contains cells that are organized into rows and columns. Each cell contains a single data value. The image below contains a snapshot of a DataFrame containing selected Compustat data for General Motors Corporation.

	GVKEY	FYEAR	TIC	IB	PRCC_F
0	005073	2013	GM	5346.0	40.87
1	005073	2014	GM	3949.0	34.91
2	005073	2015	GM	9687.0	34.01
3	005073	2016	GM	9427.0	34.84
4	005073	2017	GM	348.0	40.99

Figure 5.1: Sample DataFrame containing selected Compustat data for General Motors Corporation. GVKEY is the unique Compustat identifier. FYEAR is the fiscal year. TIC is the ticker symbol. IB is income before extraordinary items. PRCC_F is closing stock price at the end of the fiscal year.

A DataFrame is analogous to an Excel worksheet but with a few key differences. Where an Excel worksheet can have multiple blocks of data, charts, pivot tables, etc., a DataFrame contains a *contiguous block of data*. Like SAS and Stata datasets, DataFrames have named columns and numbered rows (i.e., an index). Pandas allows for different indexes, such as timestamps for time series data. We do not discuss those in this chapter.

5.1.2 The Pandas Series

The Series object represents a column or a row of data. We can think of it as a vector or a list if that helps. We can also think of a DataFrame as a collection of columns, each of which are Series objects. Many

common Pandas tasks require users to manipulate Series objects. We will elaborate on this point throughout this chapter.

Following is a snapshot of a Series object. Specifically, it's a snapshot of the fourth (IB) column of the DataFrame shown above. Notice that the series has a data type ("dtype"), in this case `float64`. Pandas checks whether all values in a Series have a common type. If they do, the Series is given that type. If the values have different types, then the Series is classified as having the more generic "object" type.

```
0    5346.0
1    3949.0
2    9687.0
3    9427.0
4     348.0
Name: IB, dtype: float64
```

Figure 5.2: Sample Series containing selected Compustat data for General Motors Corporation. This Series is the fourth (IB) column of the DataFrame depicted in figure 5.1 above.

5.2 Required `import` Statements

The most common method of importing the Pandas library is with this call to the **`import`** statement. It is common practice to use the alias `pd` for Pandas.

```
import pandas as pd
```

The Pandas library is built on top of a numerical analysis library called NumPy, and most Pandas users frequently call NumPy functions to manipulate data. Therefore, most users include the following import statements at the top of their notebooks.

```
import numpy as np
import pandas as pd
```

5.3 Importing and Exporting Data

In this section, we introduce Pandas' capabilities for importing data from Excel, CSV, Stata, and SAS formats. We provide sample code and explanation for common use cases. Pandas provides powerful and flexible import capabilities and supports many other data formats. Consult the official documentation for use cases not covered here.[1]

All Pandas functions for importing data begin with `read_`, e.g. `pd.read_excel`, `pd.read_sas`, etc. The options for these functions are fairly standardized so the learning curve flattens after the user gains proficiency with one import function.

5.3.1 Importing Data from Excel

Use the function `pd.read_excel` to import data from an Excel file. If the Excel file contains only one worksheet with well-formatted data (i.e., one contiguous block of data with header row and no blank lines at the top of the worksheet) then `pd.read_excel` requires only one argument, the path to the file. We illustrate this by loading one of the sample data files that accompanies this chapter.

```
import numpy as np
import pandas as pd

# Assume the Excel file is located in the same folder
# as the code file.
df = pd.read_excel('Ch5_Data_Example1.xlsx')
```

This code loads the Excel file `Ch5_Data_Example1.xlsx`, which is assumed to reside in the same folder as the code file, and saves the resulting DataFrame into a variable `df`. To view the DataFrame, simply type `df` into a cell and run the cell. Doing so yields the DataFrame shown in Figure 5.3 which contains stock prices for selected stocks.

Notice that the `pd.read_excel` function assumed that the first row contained column names and automatically used them as column labels

[1] Wharton Research Data Services (WRDS) provides the `wrds` package for Python users. This library allows users to download WRDS data directly into Python by connecting to a PostgreSQL database. We do not discuss the `wrds` library in this paper, nor do we discuss Python integration with databases.

	Company Name	Ticker	Closing Price
0	Microsoft	MSFT	138.43
1	Apple	AAPL	240.51
2	Google	GOOG	1246.15
3	Oracle	ORCL	55.13
4	SAP	SAP	132.71

Figure 5.3: Sample DataFrame containing selected stock data.

in the DataFrame. Also notice that, even though rightmost column of the Excel file contains dollar signs, `pd.read_excel` automatically converted the values to float.

One word of caution: attempting to use `pd.read_excel` on a file that is open in Excel will throw an error.

Using `pd.read_excel` with an Excel Workbook Containing Multiple Worksheets

Excel workbooks often contain multiple worksheets. By default, the function `pd.read_excel` will read data from the first worksheet in the workbook. To read a different worksheet, use the optional keyword argument `sheet_name`. This argument accepts the following values:

- A case-sensitive string containing the name of the worksheet.
- A zero-based integer index of the worksheet. 0 means the first worksheet, 1 means the second worksheet, and so on.

The following code reads data from the second worksheet of a data file that accompanies this chapter.

```
# Assume pandas has already been imported
# Assume the Excel file is located in the same folder
# as the code file.
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name='Other Stocks')
```

Using `pd.read_excel` When the Header Row is Missing

By default, `read_excel` assumes that the first row of data contains column names. If these column names are missing, pass the keyword argument `header=None`. Provide column names by passing a list to the optional keyword argument `names`. We illustrate this with the sample data file that accompanies this notebook.

```
# Assume pandas has already been imported, and the Excel
# file is located in the same folder as the code file.
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name='MSFT No Header',
                   header=None,
                   names=['DATADATE','FYEAR','AT','CEQ',
                          'IB','PRCC_F_ADJ','ADD1','ZIP',
                          'CITY','STATE']))
df.head()
```

This code imports data from the worksheet `MSFT No Header` in the Excel file `Ch5_Data.xlsx`. The output is shown in Figure 5.4.

	DATADATE	FYEAR	AT	CEQ	IB	PRCC_F_ADJ	ADD1	ZIP	CITY	STATE
0	1986-06-30	1986	170.739	139.332	39.254	0.068926	One Microsoft Way	98052	Redmond	wa
1	1987-06-30	1987	287.754	239.105	71.878	0.228631	One Microsoft Way	98052	Redmond	wa
2	1988-06-30	1988	493.019	375.498	123.908	0.300359	One Microsoft Way	98052	Redmond	wa
3	1989-06-30	1989	720.598	561.780	170.538	0.237597	One Microsoft Way	98052	Redmond	wa
4	1990-06-30	1990	1105.349	918.563	279.186	0.681410	One Microsoft Way	98052	Redmond	wa

Figure 5.4: Sample DataFrame containing selected financial data for Microsoft Corporation. The raw data lacked a header row. The keyword argument `header=None` informed `pd.read_excel` that the data lacked a header row. Column names were supplied using the `names` keyword argument.

Skipping Rows and Blank Lines

Many Excel users place blank lines and text above the data in a worksheet. To handle this use case, use the optional keyword argument `skiprows` when calling `pd.read_excel`. To skip the first `n` lines of the file, use `skiprows=n`. Alternatively, to skip specific rows, pass a list of row numbers; unlike Excel, `skiprows` assumes row numbers begin at zero.

The following sample code skips the first five lines from the appropriate worksheet of the data file that accompanies this chapter. An alternative to `skiprows=5` is `skiprows=[0,1,2,3,4]`.

```
# Assume pandas has already been imported, and the Excel
# file is located in the same folder as the code file.
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name='MSFT Extraneous Lines',
                   skiprows=5)
```

Getting Help for `pd.read_excel`

The official documentation page for `pd.read_excel` is located here. Alternatively, type `help(pd.read_excel)` into a Jupyter notebook cell and execute the cell.

5.3.2 Importing Data from a CSV File

CSV stands for "comma separated value." A CSV file is a *text file* that uses commas to separate, or delimit, values. CSV format is commonly used to store small-to-medium sized data files because it can be used to transfer data between different software (e.g., SAS and Stata), and because all major operating systems support it.

By convention, the first row of a CSV file contains the column names. Thus, the DataFrame shown in Figure 5.3 in CSV format would be:

```
Company Name,Ticker,Closing Price
Microsoft,MSFT,138.43
Apple,AAPL,240.51
Google,GOOG,1246.15
Oracle,ORCL,55.13
SAP,SAP,132.71
```

The Pandas function `pd.read_csv` is similar to `pd.read_excel`. These two functions accept many of the same arguments such as `skiprows` and `names`. However, since CSV files are text files, some issues may arise when importing financial data in CSV format. In the remainder of this section, we demonstrate how to handle two of these issues, converting financial strings to floating point numbers and parsing date columns.

Converting Financial Strings to `float`

When a CSV file contains numbers in accounting format, such as $136.72, Pandas cannot extract the numeric value. Thus, when Pandas encounters a number formatted as an accounting string, it will import that number as text. The researcher must clean it and then manually convert it.

We will use the dataset `Ch5_Acctg_Format.csv` that accompanies this chapter. The contents of this file are:

```
Year,Income
2016," $(16,798.00)"
2017," $21,204.00 "
2018," $(16,571.00)"
2019," $39,240.00 "
```

The first net income value is " $(16,798.00)". To convert this to a string that can subsequently be converted to type **`float`**, we must remove the dollar sign, comma, and right parenthesis. We must also replace the left parenthesis with a negative sign. The resulting string, "-16798.00" can then be converted to **`float`**.

The following code converts the entire `Net Income` column from accounting format to **`float`**. After reading the CSV file as a DataFrame, the code uses a regular expression to replace all commas, right parentheses, and dollar signs with the empty string. The next line of code replaces any left parentheses with negative signs. Note that the replacements occur on all values in the `Income` column. The final line of code converts the entire `Income` column to type **`float`**.[2]

In:

```
import numpy as np
import pandas as pd

# Assume the file is in the same folder
df = pd.read_csv('Ch5_Acctg_Format.csv')

# Remove comma, right parenthesis, dollar sign
df['Income'] = df['Income'].str.replace('[,\)\$]', '',
    regex=True)
# Replace left parenthesis with negative sign
```

[2]Regular expressions are discussed in Chapter 6. We explain the other lines of code in more detail later in this chapter.

```
df['Income'] = df['Income'].str.replace('(', '-')
# Convert column to float
df['Income'] = df['Income'].astype(float)
```

Parsing Dates

The `pd.read_csv` function can parse dates. Simply tell the function which columns contain dates through the `parse_dates` keyword argument, and Pandas usually imports the dates correctly. To see this, consider the following sample data file, `Ch5_Dates.xlsx`, that contains stock information for General Motors Corporation:

```
Date1,Date2,Date3,Company Name,Ticker,Closing Price
4-8-2019,8/4/2019,08 Apr 19,General Motors,GM,39.06
4-9-2019,9/4/2019,09 Apr 19,General Motors,GM,38.86
4-10-2019,10/4/2019,10 Apr 19,General Motors,GM,39.25
```

To import the `Date1` and `Date3` columns as date values, and not text strings, use the following code.

```
dfDates = pd.read_csv('Ch5_Dates.csv',
                      parse_dates=['Date1', 'Date3'])
```

Pandas is able to infer many common date formats automatically. However, when a date value is ambiguous, Pandas will assume month-day-year format. If this is not correct, we can pass the optional argument `dayfirst=True`. However, this will apply to *all* date columns specified in `parse_dates`. If the data contains dates in mixed formats, it may be best to import some dates as strings and convert them later. To convert the `Date2` column after it has been imported as a string, we can use the following code:

```
dfDates['Date2'] = pd.to_datetime(dfDates['Date2'],
                                  format='%d/%m/%Y')
```

This code uses the function `pd.to_datetime`, along with a format string that explicitly specifies the format in the column. For a complete list of format strings, see this website.

Getting Help for `pd.read_csv`

The official documentation page for `pd.read_csv` is located here. Alternatively, type `help(pd.read_csv)` into a Jupyter notebook cell and execute the cell.

5.3.3 Importing Data from Stata

Use the function `pd.read_stata` to read Stata `.dta` files. At the time of this writing, the latest version of Pandas (version 1.1.2), supports Stata files up to and including version 16. If, for some reason, you have difficulty reading a Stata file into Pandas, simply export it from Stata to Excel or CSV format and then read it into Pandas.

5.3.4 Importing Data from SAS

Use the function `pd.read_sas` to read SAS files. This function can read SAS xport (.XPT) files and SAS7BDAT files. In our experience, this function can be finicky. If you have a valid SAS license and SAS installation on your computer and plan to regularly pass data between SAS and Python, we highly recommend the package SASPy. This package was written by *The SAS Institute* and is officially supported.

5.3.5 Manually Creating a DataFrame

Occasionally, it is useful to manually create a DataFrame or Series. To manually create a series, pass a list to the function `pd.Series`. To manually create a DataFrame, pass a *dictionary* to the function `pd.DataFrame`. The dictionary keys should be the column names and the dictionary values should either be lists or Pandas Series objects. Following are examples of manual creation of Pandas objects. The first line of code creates a Series with the values 1-4. The second line of code creates a DataFrame with two columns, the first containing the numbers 1-4, and the second containing their squares.

In:

```
s = pd.Series([1,2,3,4])
print(f'Series s:\n{s}\n')
```

```
df = pd.DataFrame({'X': [1,2,3,4], 'X_sq': [1,4,9,16]})
print(f'DataFrame df:\n{df}')
```

Out:

```
Series s:
0    1
1    2
2    3
3    4
dtype: int64

DataFrame df:
   X  X_sq
0  1     1
1  2     4
2  3     9
3  4    16
```

5.3.6 Handling Missing Data

Pandas provides excellent and intricate functionality for handling missing data. Pandas provides functions for detecting, counting, and filling missing values. The behavior of these functions differs by data types; numeric, datetime, and string columns are handled differently. We refer the reader to the Pandas documentation for a detailed treatment of this topic.

5.3.7 Exporting Data

Every `read_` function discussed in this paper has a corresponding `write_` function. It is therefore straightforward to save data to a desired format. To save datasets larger than a few hundred megabytes, we suggest the Apache Parquet format. This format stores data by column and therefore allows for high compression. Pandas is able to read Parquet files very fast using the `read_parquet` function. The only caveat is that you must install the `pyarrow` package before using this format. To do so, use Anaconda Navigator or type `conda install pyarrow` at the Anaconda Prompt (Windows) or Terminal (Mac).

We note that Pandas provides functionality for reading from and writing to databases, e.g. the `pd.to_sql` function. However, we do not

discuss database integration in this paper.

5.4 Viewing Data in Pandas

Researchers working with Pandas in Jupyter Notebooks have several options for viewing data. To view an entire DataFrame (or Series), simply type the name of the variable into a cell and run the cell. The data will be shown underneath the cell. If the DataFrame has many rows, Jupyter will show the top and bottom rows. Pandas users typically only need to view a few rows at a time. This is typically accomplished with the `head` and `tail` commands. For example, if the variable `df` stores a DataFrame, `df.head()` will display the first 5 rows of that DataFrame, and `df.tail()` will display the last 5 rows. To specify the number of rows to display, pass an optional argument to `head` or `tail`. For example, `df.head(3)` will display the first 3 rows of `df`.

Use DataFrame's `sample` method to view randomly selected rows. `df.sample()` will display one randomly chosen row and `df.sample(n)` will display `n` randomly chosen rows.

Researchers wishing to view an entire DataFrame (as they would in SAS, Stata, or Excel) can install an extension to JupyterLab called Variable Inspector. Note that JupyterLab extensions require the `nodejs` package to be installed in Anaconda. The `nodejs` package is not installed by default with Anaconda. To install the `nodejs` package, run the command `conda install nodejs` at the Anaconda Prompt (Windows) or Terminal (Mac). Also, note that some development environments, such as Spyder and Visual Studio Code, provide built-in variable inspectors.

5.4.1 Descriptive Statistics

Use the `pd.describe` function to compute simple descriptive statistics of a Series or DataFrame. `pd.describe` computes the mean, standard deviation, maximum, minimum, and commonly used percentiles. It also computes counts. Pandas also provides individual functions to compute the mean, median, and other descriptive statistics. We refer the reader to the documentation for `pd.describe` for examples and links to the other functions.

5.5 Selecting and Filtering Data

This chapter demonstrates the basics of selecting and filtering data in Pandas DataFrames. By selecting, we mean choosing a subset (usually columns) of a DataFrame. By filtering, we mean choosing a subset of rows with values that meet certain criteria. Pandas provides many features for selecting and filtering data and there are often multiple ways to do the same thing. We will only scratch the surface of Pandas' capabilities in this chapter.

5.5.1 Selecting Columns of a DataFrame

Retrieving One or More Columns as a DataFrame

Following is the syntax to retrieve one or more columns from a DataFrame named `df`. Notice the double square brackets following the variable `df`.

```
df[['col1', 'col2', ...]]
```

The following code operates on one of the example files that accompanies this chapter. It loads a data file containing selected financial data for Microsoft Corporation and then selects a subset of the columns. Specifically, the code `df[['Fiscal Year', 'Net Income']]` selects the `Fiscal Year` and `Net Income` columns and returns a new DataFrame containing those columns. We then show the last three rows (using the `tail` method).

In:

```
import numpy as np
import pandas as pd

df = pd.read_excel('Ch5_Data.xlsx', sheet_name='MSFT
    Clean')

# Get the fiscal year and net income columns
# Save the results to a new DataFrame
df_Selected = df[['Fiscal Year', 'Net Income']]

print(df_Selected.tail(3))
```

Out:

```
    Fiscal Year  Net Income
```

```
31          2017      21204.0
32          2018      16571.0
33          2019      39240.0
```

When selecting columns from a DataFrame, the columns can be in any order. Also, repeating columns is allowed.

Retrieving a Single Column as a Series

When retrieving a single column of a DataFrame, it is possible to retrieve that column as either a DataFrame or as a Series.[3] If a single column of a DataFrame is selected using double brackets, Pandas returns a DataFrame. However, if a single column is selected using single brackets, Pandas returns a Series.

In:

```
import pandas as pd
df = pd.DataFrame({'x': [1,2,3], 'xsq': [1,4,9]})

# Double brackets returns a DataFrame
print(type(df[['x']]))
# Single brackets returns a Series
print(type(df['x']))
```

Out:

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
```

5.5.2 Testing the Values in a Series

In the following section, we demonstrate how to filter data by retrieving rows that meet some criteria. To explain filtering, we first need to explain how to work with Pandas Series objects.

Comparing a Series to a Numeric Constant or String

Often it is necessary to test every value in a column. For example, consider the `Fiscal Year` column of the Microsoft financial data above. Say we wish to filter our data so we only have firm-year observations after 2014. In other words, we only want rows where the value in the

[3] This matters when filtering rows. We elaborate on this point later in this chapter.

`Fiscal Year` column is greater than 2014. To filter the data this way, we need to take the column `Fiscal Year` column and compare every value to 2014. Let us see what happens when we do that:

In:

```
import numpy as np
import pandas as pd

df = pd.read_excel('Ch5_Data.xlsx', sheet_name='MSFT
    Clean')
df['Fiscal Year'] > 2014
```

Out:

```
0     False
1     False
...
32     True
33     True
Name: Fiscal Year, dtype: bool
```

In the above code, Pandas took the column `Fiscal Year` (which is a Series) and compared every value in the column to 2014. It returned a new Series of Boolean values. Each value corresponds to one fiscal year in the data. If the fiscal year is greater than (less than) 2014, the corresponding value in the returned Series is `True` (`False`). The new Series has exactly the same length as the original DataFrame.

Comparing a Series to another Series

When testing a Series, it is possible to compare values to numbers, strings, or to other Series. When comparing to a scalar or to a string, Pandas compares each value in the Series to that single value. When comparing one Series to another Series, Pandas compares row-by-row. Thus, when comparing two Series, they must have the same length. To illustrate this, consider the following code. It checks whether Microsoft's total assets exceeds its equity in each fiscal year. It then uses the method `all` to confirm that the comparison is true in every fiscal year.

In:

```
import numpy as np
import pandas as pd
```

```
df = pd.read_excel('Ch5_Data.xlsx', sheet_name='MSFT
    Clean')
# Compare assets to equity. Save results in new_Series
new_Series = df['Assets - Total'] > df['Common Equity']
# Check whether all values in new_Series are True
new_Series.all()
```

Out:

```
True
```

5.5.3 Filtering a DataFrame

Filtering is the process of extracting rows of a dataset that meet certain criteria. Filtering in Pandas relies heavily on the the ideas in section 5.5.2 about working with Series. The main idea behind filtering in Pandas is to pass a Boolean series to a DataFrame; only rows for which the Boolean Series is true are kept in the filtered DataFrame. To see this, consider the following example:

In:

```
import pandas as pd

# Create a simple DataFrame
df = pd.DataFrame({'col1': ['a','b','c'],
                   'col2': [1,2,3]})
print(f'df, unfiltered:\n{df}\n')

# Create a Boolean Series
sBool = pd.Series([True, False, True])

# Filter df using sBool
df_filt = df[sBool]
print(f'df filtered using sBool:\n{df_filt}')
```

Out:

```
df, unfiltered:
  col1  col2
0    a     1
1    b     2
2    c     3

df filtered using sBool:
  col1  col2
0    a     1
2    c     3
```

In the example above, the line of code on which filtering occurs is `df_filt = df[sBool]`. The square brackets indicate that a Series is being passed to `df`. The Series `sBool` contains the values `True`, `False`, and `True`. Since the first and third values are `True`, the first and third rows of `df` are kept in `df_filt`.

Filtering with a Single Condition

To filter with a single condition, simply write a condition using the format shown in section 5.5.2 and place it in square brackets after a DataFrame. For example, the following code loads a sample DataFrame containing stock data and filters it to certain ticker symbols. The code `df['Ticker'] == 'MSFT'` compares all values in the `Ticker` column to the string `'MSFT'`. This code evaluates to a Boolean Series. When the Series is enclosed inside sqaure brackets and follows `df`, it tells Pandas to show only rows for which the value in the `'Ticker'` column equals `'MSFT'`.

In:

```
import pandas as pd
# Assume the data file is in the same folder
df = pd.read_excel('Ch5_Data_Example1.xlsx')

df[df['Ticker'] == 'MSFT']
# The following two lines are equivalent
# to the previous line.
# s = df['Ticker'] == 'MSFT'
# df[s]
```

Out:

```
  Company Name Ticker  Closing Price
0    Microsoft   MSFT         138.43
```

In sum, to filter a DataFrame, create a Boolean Series using the method shown in section 5.5.2. Place the code to create the Boolean Series inside square brackets. The square brackets should follow a DataFrame.

Filtering with Multiple Conditions

To filter a DataFrame using multiple conditions:

- Use the logical operators `&`, `|`, and `~` to separate the conditions.
- Enclose each condition in parentheses.

Logical Operators

We introduced Python's logical operators, **`and`**, **`or`**, and **`not`** in Section 4.3.3. The same ideas apply here, but when working with Pandas Series, we must use symbols for these operators. The following table shows the symbols.

Python Operator	Pandas Equivalent
`and`	`&`
`or`	`|`
`not`	`~`

Table 5.1: Pandas logical operators for joining multiple conditions and their Python equivalents.

Joining Conditions

When filtering a Pandas DataFrame with multiple conditions, each condition must be enclosed *in parentheses*. Briefly, the reason for this is the order of operations. If the parentheses are omitted, Pandas will attempt to perform operations in the wrong order. This will cause an error or yield incorrect results.

The following examples work with the Microsoft financial data that accompanies this chapter. The first example filters the data, so it only includes firm-year observations for which fiscal year is greater than 2013 and common equity exceeds $80 billion. The second example filters the data so it includes fiscal years before 1990 and after 2015.

In:

```
import pandas as pd
# Assumes the Excel file is in the same folder
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name='MSFT Clean')

# Filter 1: FY > 2013 and CEQ > 80 billion
df1 = df[(df['Fiscal Year'] > 2013) &
         (df['Common Equity'] > 80000)]

# Filter2: FY < 1990 or FY > 2015
df2 = df[(df['Fiscal Year'] < 1990) |
```

```
            (df['Fiscal Year'] > 2015)]
```

Filtering rows and columns simultaneously with `.loc`

Pandas provides the `.loc` function that allows users to filter rows and columns simultaneously. Consider the following code that filters the Microsoft financial data. This code restricts years 2011 and up and retrieves only the assets and equity columns.

```
import pandas as pd
# Assumes the Excel file is in the same folder
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name='MSFT Clean')

# Filter 1: FY > 2013 and CEQ > 80 billion
df1 = df.loc[df['Fiscal Year'] > 2010,
             ['Assets - Total', 'Common Equity']]
```

Notice that `.loc` has the syntax of an indexer (i.e., square brackets `[]`), not that of a function. The first argument is the indexes of the desired rows and the second argument is the names of the desired columns. If the second argument is a list, `.loc` returns a DataFrame. However, if the second argument is the name of a single column, not in a list, then `.loc` returns a Series.

Pandas provides many other related functions. `.iloc` behaves similarly but accepts zero-based index numbers for rows and columns. `.at` and `.iat` allow users to retrieve a single value from a DataFrame. We do not describe these accessors in detail and leave it to the reader to consult the Pandas documentation.

5.6 Creating New Columns

Each column of a DataFrame is a Series. Therefore, creating a new column implies creating a Series. Typically, new columns are created as transformations of existing columns. However, sometimes new columns contain single values. We will demonstrate both procedures in this section.

5.6.1 Creating a Column from a Scalar Value

Pandas makes it very easy to create a new column that contains a single, repeated value. Simply assign that value to a column of the DataFrame that does not already exist. For example, say we wish to create a new column named `'newcol'` in DataFrame `df`, and we want this column to contain the value 5. We could do that as follows:

In:

```
df = pd.DataFrame({'col1': [1,2,3]})
df['newcol'] = 5
print(df)
```

Out:

```
   col1  newcol
0     1       5
1     2       5
2     3       5
```

This code tells Pandas to first create a DataFrame containing a single column. The second line of code references a column named `'newcol'` in DataFrame `df`.[4] Since that column does not exist, Pandas will create it. When Pandas sees a single, scalar value assigned to a Series, it automatically adds that value to every row of the DataFrame.[5]

To summarize, to create a new column in a DataFrame containing a single, repeated value:

- Type the DataFrame name followed by square brackets.
- Inside the square brackets, type the name of the new column as a string.
- After the brackets, type an equals sign and then the desired value.

5.6.2 Creating a Column from a List

To create a new column containing data from a list, follow a procedure nearly identical to that in the previous section. However, instead of assigning a single value, assign the list. The length of the list *must equal* the number of rows in the DataFrame. For example:

[4] Recall that single-bracket notation implies a Series. The code `df['newcol']` attempts to extract a Series from DataFrame `df`.

[5] In the language R, this feature is known as "recycling" since a single value is recycled throughout the column.

In:

```
df = pd.DataFrame({'col1': [1,2,3]})
df['newcol'] = ['a','b','c']
print(df)
```

Out:

```
   col1  newcol
0     1       a
1     2       b
2     3       c
```

5.6.3 Creating a Column as a Transformation of an Existing Column

Column transformations are very common. Examples of column transformation are to (1) strip whitespace from all values in a column, (2) convert a string column to uppercase, and (3) perform a mathematical operation on every value in a column. In this section, we provide sample code that implements these examples. Often, when cleaning data like this, it is desirable to do so "in place" (i.e., instead of creating a new column, replace data in an existing column with transformed data). The first two examples clean data in place, while the third creates new columns.

The examples in this section rely on the data file `Ch5_Data.xlsx` that accompanies this chapter. We assume the following code has been run before running any of the examples.

```
import numpy as np
import pandas as pd
# Assume the file resides in the same
# folder as the code.
df = pd.read_excel('Ch5_Data.xlsx',
                   sheet_name="MSFT Clean")
```

Example: Strip Whitespace from a String Column in Place

In the sample DataFrame, the values in the `City` column have extraneous whitespace characters at the beginning and end of the string "Redmond". The following code cell strips whitespace from every value in the `City` column and saves the result in the same column:

```
df['City'] = df['City'].str.strip()
```

Let us analyze the code on the right-hand side of the equals sign, `df['City'].str.strip()`. That tells Pandas to take the `City` column as a Series, run the string method `strip` on every value in the column, and return a new Series.[6] We then save that new Series in the existing column `df['City']`.

Example: Convert a String Column to Uppercase in Place

The `State` column in the sample data contains the string `'wa'`. To convert every value in the column to uppercase, use this code:

```
df['State'] = df['State'].str.upper()
```

The explanation for this code is identical to that provided in the previous example.

Example: Convert a Numeric Column from Millions to Actual

The following code converts the total assets, common equity, and net income columns from millions to actual. However, unlike in the previous examples, this code creates new columns.

```
for col in ['Assets - Total',
            'Common Equity',
            'Net Income']:
    df[f'{col} ACTUAL'] = df[col] * 1000000
```

Note that we used a **`for`** loop to iterate over the columns, and an f-string to create a new column name from an existing column name.

5.6.4 Creating a Column as a Combination of Multiple Columns

Often the need arises to create a column from multiple columns. For example, to compute return on assets (ROA), it is necessary to divide current year net income by current year assets. We provide two examples of this below. In both examples, Pandas operates row by row, just like

[6] Pandas' string methods operate on Series objects. All string methods begin with the prefix `str`. For a complete list of Pandas string methods, see here.

a SAS data step. Note that the second example combines values from four columns, as well as three strings. It also uses the Pandas `astype` method to convert a Series of integers to a Series of strings.

```
# Compute ROA
df['ROA'] = df['Net Income'] / df['Assets - Total']

# Compile full address
df['Full Address'] = df['Address'] + '\n' + df['City'] + ',
    ' + df['State'] + ' ' + df['ZIP Code'].astype(str)
```

5.7 Dropping and Renaming Columns

5.7.1 Dropping Columns

Use the aptly named `drop` method to drop columns from a Pandas DataFrame. Pass a list of unwanted column names to the `columns` keyword argument. For example, to drop columns named `'x'`, `'y'`, and `'z'` from DataFrame `df`, use the following code.

```
df.drop(columns=['x','y','z'], inplace=True)
```

Notice the keyword argument `inplace`. By default, the `drop` method returns a new DataFrame. To force `drop` to operate in place, this argument is needed. An alternative method for dropping columns in place is to reassign the result to the existing DataFrame. For example:

```
df = df.drop(columns=['x','y','z'])
```

5.7.2 Renaming Columns

The `rename` method works similarly to the `drop` method demonstrated above. However, instead of passing a list to the `columns` keyword argument, `rename` requires a *dictionary*. Each key in the dictionary must be the name of an existing column; the corresponding value is the new name for that column. This should be clear from the following example.

In:

```
# Create a simple DataFrame
df = pd.DataFrame({'col1': [1,2,3], 'col2': [1,4,9]})
```

```
# Rename col1 to X and col2 to X_sq
df = df.rename(columns={'col1': 'X', 'col2': 'Xsq'})
print(df)
```

Out:

```
   X  Xsq
0  1    1
1  2    4
2  3    9
```

Alternatively, the following code, which relies on the `inplace=True` keyword argument, would have an identical result to that in the previous example.

In:

```
# Create a simple DataFrame
df = pd.DataFrame({'col1': [1,2,3], 'col2': [1,4,9]})
# Rename col1 to X and col2 to X_sq
df.rename(columns={'col1': 'X', 'col2': 'Xsq'},
          inplace=True)
```

5.8 Sorting Data

To sort data in a DataFrame, use the `sort_values` method.[7] Use the following keyword arguments to achieve the desired result:

- `by`: pass a list of column names by which to sort.
- `ascending`: Use `True` (`False`) to sort all columns in ascending (descending) order. Alternatively, pass a list of Booleans to specify the sort order for each column named in the `by` argument.
- `inplace`: Use `True` to sort in-place. If this argument is omitted or set to `False`, then `sort_values` will return a new, sorted DataFrame.

As an example, consider the following code:

In:

```
# Create a simple DataFrame
df = pd.DataFrame({'Y': ['b','a','a','b'],
                   'X': [2,4,3,1]})
# Sort by column Y descending, then by column X
    ascending
```

[7] Pandas also provides a `sort_index` method. We do not discuss that in this chapter. See here for details.

```
df.sort_values(by=['Y', 'X'], ascending=[False,True],
    inplace=True)
print(df)
```

Out:

```
   Y  X
3  b  1
0  b  2
2  a  3
1  a  4
```

5.9 Merging Data

Pandas provides a convenient function for merging two DataFrames. This function, `merge`, exposes SQL-like arguments to the user and implements inner, left, right, and outer joins. We assume the reader is familiar with SQL joins.

The syntax of the Pandas `merge` function is shown below.

```
DataFrame.merge(right,
                how='inner',
                on=None, left_on=None, right_on=None,
                left_index=False, right_index=False,
                sort=False,
                suffixes='_x', '_y', copy=True,
                indicator=False, validate=None)
```

We focus on the commonly used arguments. The first argument, `right`, is required and is a DataFrame that is treated as the "right" table in the join. The calling DataFrame is treated as the "left" table. The `how` argument must be one of `'inner'`, `'left'`, `'right'`, or `'outer'` and indicates the desired type of join. The argument `on` specifies the column(s) on to be used in the merge. The argument `on` can only be used if the joining columns in the left and right DataFrames have identical names. If not, the user must specify the joining columns through the `left_on` and `right_on` arguments.

The following example creates two DataFrames and merges them.

In:

```
import pandas as pd

dfLeft = pd.DataFrame({'myIndex': [1,2,3,4],
```

```
                          'LeftVals': [10, 20, 30, 40]})
dfRight = pd.DataFrame({'myIndex': [1,3,5],
                        'RightVals': [100, 300, 500]})

mergedDF = dfLeft.merge(dfRight, how='inner', on='
    myIndex')
print(mergedDF)
```

Out:

```
   myIndex  LeftVals  RightVals
0        1        10        100
1        3        30        300
```

Note that the `merge` function creates and returns a new DataFrame.

6

Introduction to Regular Expressions

6.1 Looking for Patterns in Text

The heart of text analysis is finding patterns in text. These patterns may be individual words, numbers, symbols, and many more. In this chapter, we introduce *Regular Expressions*, a powerful tool for finding patterns in text.

A *Regular Expression*, or *Regex*, is a sequence of characters that define a text search pattern. Regular expressions are useful in a wide variety of text processing tasks including identifying specific words, phrases, entity names, numbers, url links in text, splitting text into sentences, grouping sentences into categories, etc. For example, we can use regular expressions to identify positive and negative words in a document to subsequently determine its tone, to classify sentences in a document as forward-looking or risk-related, or to classify documents into categories depending if certain conditions are met. This chapter introduces basic Regex concepts and provides numerous illustrative examples for different textual analysis needs.

In Python, the `re` module provides regular expression support. Some commonly used functions from this module include:

- `re.search(pattern, string)` scans through a string, looking

for the first location where the regex `pattern` matches; output is a `Match` object if there is a match, or `None` otherwise;

- `re.findall(pattern, string)` finds all substrings where the regex `pattern` matches and returns them as a list;
- `re.split(pattern, string)` splits a string at every match of the regex `pattern` and returns a list of strings. For example, one can retrieve the individual words in a sentence by splitting at the spaces;
- `re.sub(pattern, repl, string)` finds all matches of `pattern` in `string` and replaces them with `repl`.

Regular expression patterns can be specified in either single and double quotation marks, `''` and `""`. It is a good practice to insert the letter "r" before Regex patterns in Python's `re` operations (e.g., `re.search(r'pattern', string)`). Prefixing with "r" indicates that backslashes \ should be treated literally and not as escape characters in Python. In other words, the "r" prefix indicates that the string is a "raw string." For example, the following code that demonstrates how to match a basic regex `r"OI"` in a sentence.

In:

```
# loads Python's regular expressions module
import re
text = "OI for FY 2019 was 12.4 billion, up more than
    eight percent from OI in FY 2018."
# returns a Match object of the first match,
# if it exists
x1 = re.search(r"OI", text)

# finds all matches of "OI"
x2 = re.findall(r"OI", text)

# splits text at ","
x3 = re.split(r",", text)

# replaces "OI" with "Operating Income"
x4 = re.sub(r"OI", "Operating Income", text)

print(f'Result of re.search:\n{x1}\n')
print(f'Result of re.findall:\n{x2}\n')
print(f'Result of re.split:\n{x3}\n')
print(f'Result of re.sub:\n{x4}')
```

Out:

```
<re.Match object; span=(0, 2), match='OI'>
['OI', 'OI']
['OI for FY 2019 was 12.4 billion', ' up more than
    eight percent from OI in FY 2018.']
Operating Income for FY 2019 was 12.4 billion, up more
    than eight percent from Operating Income in FY
    2018.
```

To perform case-insensitive Regex matching, we can either specify IGNORECASE flag in a `re` function or convert the input string to lower case using string's `lower()` method:

In:

```
x1 = re.findall(r'MD&A', "This year's MD&a Section is
    located... Please refer to our md&A section on page
    ...", re.IGNORECASE)
x2 = re.findall(r'md&a', "This year's MD&a Section is
    located... Please refer to our md&A section on page
    ...".lower())
print(x1)
print(x2)
```

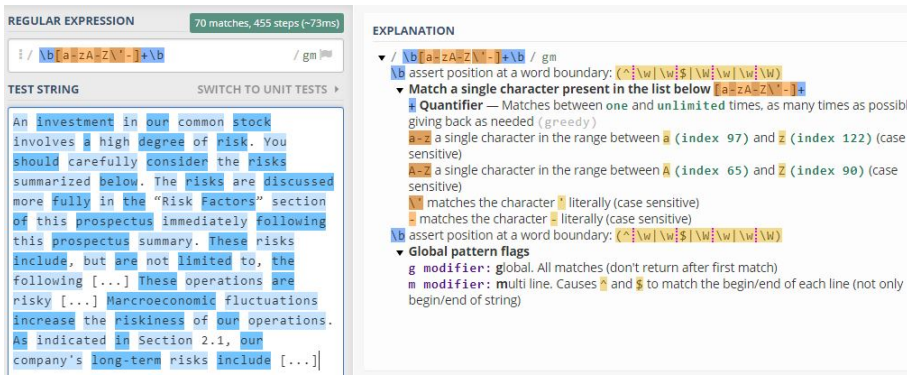
Out:

```
['MD&a', 'md&A']
['md&a', 'md&a']
```

Additional information on Regex in Python is available on the following websites:

http://www.regular-expressions.info/,
https://docs.python.org/3/howto/regex.html,
https://docs.python.org/3.4/library/re.html,
https://www.w3schools.com/python/python_regex.asp.

Also, there are many interactive websites online that allow users to test their regular expressions for correctness. For instance, https://regex101.com/ allows to perform regex testing on sample texts; it also provides useful explanations for what each regex element captures as illustrated in the figure below.



We suggest performing multiple Regex tests before running a large scale textual analysis task as the content and format of textual documents are extremely unstructured.

6.2 Characters and Character Sets

6.2.1 Special Characters

There are several special purpose characters in regular expressions:
. ^ $ * + ? { } [ ] \ | ( )
For a special character to be treated literally, we have to add a backslash (\) before that character. For example, \$ indicates the dollar sign and \\ indicates the backslash. For example:

- Regex `r"\$4.99"` will match $4.99 in "Apple's Earnings per Share for the three months ended in December 2019 was $4.99," whereas Regex `r"$4.99"` will return no matches.
- Regex `r"S-1\\A"` will match S-1\A in "Form S-1\A," whereas Regex `r"S-1\A"` will return no matches.

6.2.2 Character Sets in Regex

Character Sets provide means to match any single character in a given set of characters. To specify a character set, we use square brackets, `[` and `]`. For example, regex `'10-[KQ]'` has a character set `'[KQ]'` and will match both '10-K' and '10-Q' in a string.

A hyphen can be used inside character sets to indicate a range of characters:

- `[a-z]` matches any lowercase letter between 'a' and 'z';
- `[A-Z]` matches any uppercase letter between 'A' and 'Z';
- `[a-zA-Z]` matches any letter in English alphabet (both upper- and lowercase);
- `[0-9]` matches a single digit between 0 and 9.

Negated Character Sets are used to match any single character except a character from a specified character set. Negated character sets are enclosed in [^]. For example, `[^a-z]` matches anything as long as it is not a lowercase letter in the a-z range.

Common Character Sets have shorthand notations. For instance:

- `.` matches any character except newline (i.e., `[^\n]`);
- `\d` matches a digit character (i.e., `[0-9]`);
- `\D` matches a non-digit character (i.e., `[^0-9]`);
- `\w` matches a word character (including digits and underscores) (i.e., `[a-zA-Z0-9_]`);
- `\W` matches a non-word character (i.e., `[^a-zA-Z0-9_]`);
- `\s` matches a whitespace, new line, tab, carriage return, etc. (i.e., `[ \f\n\r\t\v]`);
- `\S` matches everything, but a whitespace (i.e., `[^ \f\n\r\t\v]`).

For example:

In:

```
text = "This project has increased our revenues by more
    than 70% in FY 2019."
# returns all single digit matches
x1 = re.findall(r'[0-9]', text)
# returns all non-word characters, also excludes
# spaces, periods, and commas
x2 = re.findall(r'[^a-zA-Z \.,]', text)
# returns all two-digit numbers followed by "%"
x3 = re.findall(r'\d\d%', text)
print(x1)
print(x2)
print(x3)
```

Out:

```
['7', '0', '2', '0', '1', '9']
['7', '0', '%', '2', '0', '1', '9']
['70%']
```

6.3 Anchors and Boundaries in Regex

In addition to matching word and non-word characters in text, Regex can match specific positions in text. This becomes very useful when we want to match single words or word phrases in text, identify sentences and further classify them, etc. In textual analysis tasks, we commonly use these Regex anchors and boundaries:

- `^` matches the beginning of a string or line (when it is not inside [ ]);
- `$` matches the end of a string or line;
- `\A` matches the beginning of a string;
- `\Z` matches the end of a string;
- `\b` matches a word boundary and allows "whole word only" searches in the form of `'\bword\b'`;
- `\B` matches anything but a word boundary.

For example:

- Regex `r"^inf"` will match 'inf' in "information retrieval," but not in "retrieval of information";
- Regex `r"\bhigh\b"` will match 'high' in "high income," but not in "higher income";
- Regex `r"\bhigh\B"` will match 'high' in "higher income," but not in "high income".

6.4 Quantifiers in Regex

Quantifiers allow to specify whether a Regex element can be matched zero, one, or many times. Quantifiers are always placed *after* Regex elements. To process text documents, we commonly use the following quantifiers:

- `+` matches an item one or more times;
- `*` matches an item zero or more times;
- `{n}` matches an item exactly n times;
- `{k,n}` matches an item at least k times and at most n times;
- `?` matches an item zero or one time.

For example:

- Regex `r"^\w+\b"` will match 'high' in "high income" and 'low' in "low income";
- Regex `r"\b19\d{2}\b"` will match any integer number between 1900 and 1999;
- Regex `r"\b\d{2,3}\b"` will match all two- and three-digit numbers in text.

6.5 Groups in Regex

Elements of Regex can be grouped together by enclosing them into parentheses, ( ). By default, groups are also captured in addition to the whole regular expression. That is, a regex `Match` object will store the textual value of the complete match as well as textual values of patterns specified in groups. To turn off group capturing, we need to specify ?: after the left parenthesis, i.e., (?:). Groups are useful for recording specific values in the match and not the whole match. In other words, groups can be thought of as regex sub-patterns. For example:

- Regex `r"Total Assets = (\$[\d,\.]+)\b"` will match the text "Total Assets = ", followed by a group capturing the dollar sign `\$`, one or more numbers `\d`, allowing for commas `,` and periods `\.`, and a word boundary `\b`. Applying this regex pattern to the string "Total Assets = $10,000,000" will result in two matches:
  - Full match: 'Total Assets = $10,000,000'
  - Group match : '$10,000,000'

- Regex `r"Email: ([\w\.-]+@[\w\.-]+.edu)"` will match the text "Email: ", followed by a group capturing either an alphanumeric, '.' , or '-' character `[\w\.-]` (matched one or many times as indicated by '+'), followed by the at symbol `@`, followed again by alphanumeric, '.' , or '-' characters (matched one or many times), and ending with '.edu'. Applying this regex to string "Email: abc@ABC.edu" will result in two matches:
  - Full match: 'Email: abc@ABC.edu'
  - Group match : 'abc@ABC.edu'

It is possible to specify *alternative patterns* in Regex using the "or" operator, denoted by |. For example, regex `'Form (10-K|10-Q)'` will

match both 'Form 10-K' and 'Form 10-Q' in text.

It is also possible to reuse previous group matches in the same regex. This is called *backreferencing*. To reference the text matched by the n^{th} group, we use `\n` notation in a regex expression. For instance, regex `r"\b(\w+)\s\1\b"` would capture repeated words in text (e.g., "we we", "in in", "above above", etc.); element `\b(\w+)\s` captures a word followed by space `\s`. The pattern `\1` followed by the boundary `\b` captures the result of the first (and only) group, making sure that the second match is not a prefix or beginning of a longer word.

Finally, we can give names to Regex groups for better readability and referencing. To give a name to a group, use syntax `(?P<name>group)`; to reference this group, use `(?P=name)`. For example, a named group `word` in `\b(?P<word>\w+)\s(?P=word)\b` will capture instances where the same word is repeated two times as in "*we we* expect" and "let *us us* begin." Then, if we want to replace all double words in text with a single word, we would reference the named group using `\g<word>` syntax:

In:

```
import re
regex = r"\b(?P<word>\w+)\s(?P=word)\b"
text = "we we expect this this trend to continue"
# prints an output where all double words are replaced
    with a single word
print(re.sub(regex,r"\g<word>", text))
```

Out:

```
we expect this trend to continue
```

When processing a match object `m`, we can reference any named group using `m.group('name')` syntax as follows:

In:

```
import re
date = "09/14/2020"
# specifies three named groups, namely 'Month', 'Day',
    and 'Year'
regex = r"(?P<Month>\d{1,2})/(?P<Day>\d{1,2})/(?P<Year
    >\d{2,4})"
# identifies regex matches in date
date_matches= re.search(regex, date)
```

```
#prints matches for each of the named groups
print("Month: ", date_matches.group('Month'))
print("Day: ", date_matches.group('Day'))
print("Year: ", date_matches.group('Year'))
```

Out:

```
Month:  09
Day:  14
Year:  2020
```

6.6 Lookahead and Lookbehind in Regex

It is possible to check whether a regex item is followed by a certain pattern without including that pattern in the resulting match. *Positive lookahead*, denoted by `(?=pattern)`, checks whether a regex item is followed by a given pattern. *Negative lookahead*, denoted by `(?!pattern)`, checks whether a regex item is not followed by a given pattern.

Lookbehind checks whether a regex item is preceded (or not) by a certain pattern without including that pattern in the output match. *Positive lookbehind*, denoted by `(?<=pattern)`, checks whether a regex item is preceded by a given pattern. *Negative lookbehind*, denoted by `(?<!pattern)`, checks whether a regex item is not preceded by a given pattern.

For example:

- Regex `r"filename(?=\.txt)"` matches 'filename' (not filename.txt) in "filename.txt," but not in "filename.csv"
- Regex `r"filename(?!\.txt)"` matches 'filename' (not filename.csv) in "filename.csv," but not in "filename.txt"
- Regex `r"(?<=year\s)\d{4}"` matches '2020' (not year 2020) in "year 2020," but not in "series 2020"
- Regex `r"(?<!year\s)\d{4}"` matches '2020' (not series 2020) in "series 2020," but not in "year 2020"

6.7 Examples of Regex for Different Textual Analysis Tasks

The following Python code snippets provide more examples of using Regex for various textual analysis tasks.

Example 1: Character Sets
Assume we want to identify all numbers in text followed by % or the word percent. To do so, we create a character set of digits (allowing for period '.' in numbers); we also specify the "% vs. percent" option using the regex "or" symbol, |.[1]

In:

```
text = "This project has resulted in over 70% of our
    2019 revenues to date. As a result, our operating
    income increased by 9%, while our operating
    expenses increased by 12%. We had a 12.5 percent
    increase in regional sales."
# recall that ?: after the left parenthesis
# specifies a non-capturing group
x = re.findall(r'[\d\.]+(?:\%|\s\bpercent\b)', text)
print(x)
```

Out:

```
['70%', '9%', '12%', '12.5 percent']
```

Example 2: Character Sets, Quantifiers, Groups, Lookbehinds
Assume we want to identify basic company information – CIK number, company name, filing date, and SIC industry code – from the company filing's SEC header. To do so, we need to create separate regular expressions for all items that we want to capture.

In:

```
# an example of a standard input header used in SEC
# filings
header = """<SEC-HEADER>0000080424-18-000100.hdr.sgml :
     20181019
<ACCEPTANCE-DATETIME>20181019161731
ACCESSION NUMBER:    0000080424-18-000100
CONFORMED SUBMISSION TYPE:  10-Q
PUBLIC DOCUMENT COUNT:    68
CONFORMED PERIOD OF REPORT: 20180930
FILED AS OF DATE:    20181019
DATE AS OF CHANGE:    20181019
FILER:
  COMPANY DATA:
```

[1] This code can be adopted to calculate the number of informative numbers in texts as in Dyer *et al.* (2017) and Blankespoor (2019).

```
    COMPANY CONFORMED NAME:     PROCTER & GAMBLE Co
    CENTRAL INDEX KEY:      0000080424
    STANDARD INDUSTRIAL CLASSIFICATION: SOAP, DETERGENT
    , CLEANING PREPARATIONS, PERFUMES, COSMETICS [2840]
    IRS NUMBER:       310411980
    STATE OF INCORPORATION:     OH
    FISCAL YEAR END:      0630
[...]
</SEC-HEADER>"""

# CIK is the 10-digit number, so we use the quantifier
# {10} to consider only 10-digit numbers in the match
# this Regex specifies text "CENTRAL INDEX KEY:",
# followed by space \s (matched zero or many times
# as indicated by *), followed by a group capturing
# 10-digit numbers
# Also, note that re.findall with a group Regex only
# returns the group match, and not the full match
cik = re.findall(r"CENTRAL INDEX KEY:\s*(\d{10})",
                 header)

# This Regex specifies text "COMPANY CONFORMED NAME:",
# followed by space \s (matched zero or many times),
# followed by a group capturing any character one or
# many times
# flag MULTILINE makes ^ and $ characters capture
# beginning and end positions of text lines instead
# of only text files
company_name = re.findall(r"COMPANY CONFORMED NAME:\s
    *(.+)$",
                          header,
                          re.MULTILINE)

# This Regex specifies text "FILED AS OF DATE:",
# followed by space \s (matched zero or many times),
# followed by a group capturing 8-digit numbers as
# all dates in the SEC headers are in the YYYYMMDD
# format
filing_date=re.findall(r'FILED AS OF DATE:\s*(\d{8})',
                       header)

# This Regex uses a positive lookbehind to check if a
# 4-digit number is preceded by text "STANDARD
# INDUSTRIAL CLASSIFICATION:"
sic = re.findall(r'(?<=STANDARD INDUSTRIAL
    CLASSIFICATION:).+(\d{4})', header)
```

```
print(cik)
print(company_name)
print(filing_date)
print(sic)
```

Out:

```
['0000080424']
['PROCTER & GAMBLE Co']
['20181019']
['2840']
```

Example 3: Word Boundaries and Quantifiers

Assume we want to identify all word matches that start with "risk", but can have different endings (e.g., risks, risky, risking, etc.). We also want to calculate the percentage of such risk words relative to all words in text. To do so, we use regex word boundaries to perform single word matches.

In:

```
text = "An investment in our common stock involves a
    high degree of risk. You should carefully consider
    the risks summarized below. The risks are discussed
     more fully in the Risk Factors section of this
    prospectus immediately following this prospectus
    summary. These risks include, but are not limited
    to, the following [...] These operations are risky
    [...] Macroeconomic fluctuations increase the
    riskiness of our operations. As indicated in
    Section 2.1, our company's long-term risks include
    [...]"

# this Regex matches a word boundary followed by a
# text string 'risk', followed by an alphanumeric
# character (repeated zero or many times), followed
# by a word boundary; re.IGNORECASE specifies a
# case-insensitive matching
risk_words = re.findall(r"\brisk\w*\b",
                        text, re.IGNORECASE)

# matches all single words (allowing for '-'
# between two words and apostrophe) in text
all_words = re.findall(r"\b[a-zA-Z\'\-]+\b", text)
```

```
# function len() here returns the number of words
# that start with string 'risk', i.e., the number
# of matches in risk_words list
risk_words_freq =  len(risk_words)

# the number of all words in text
all_words_freq = len(all_words)

# percentage of risk-related words in text
text_riskiness = 100 * risk_words_freq/all_words_freq

print(risk_words)
print(text_riskiness)
```

Out:

```
['risk', 'risks', 'risks', 'Risk', 'risks', 'risky', '
    riskiness', 'risks']
11.428571428571429
```

To summarize, this chapter introduces basic syntax for regular expressions and provides different examples of how to use regular expressions to identify specific patterns in text. In the following chapters, we discuss different techniques for textual analysis of corporate disclosures, illustrating specific uses of regular expressions.

7

Dictionary-Based Textual Analysis

7.1 Advantages of Dictionary-based Textual Analysis

Generally speaking, there are two ways to identify meaningful information content from text documents: (1) using machine learning and (2) using pre-determined dictionaries of word classifications. In the former case, an algorithm first looks for differentiating patters in previously labeled textual data, and then it applies learned patterns to label the remaining unlabeled data. In the latter case, textual analysis relies on an input dictionary (provided by the researcher), where each textual document is represented by its relative frequencies of words from the input dictionary.

For example, if we want to automatically classify a large sample of sentences as positive or negative, we can run a supervised machine learning algorithm for this classification. First, we need to ask human annotators to label a random set of sentences as positive, negative, or neutral. Then, we can use labeled sentences as the training data for the selected classification algorithm. The algorithm then 'learns' what words appear most frequently in neutral vs. positive vs. negative sentences and can apply this rule to label previously unseen sentences. Once all document sentences are labeled, we can average their scores

to calculate the overall document positivity / negativity. Li (2010b) is a great example of a research study that applies a machine learning algorithm to determine the tone, positive vs. negative, of forward-looking sentences in periodic SEC reports.

Alternatively, to determine document tone, we can use existing dictionaries of positive and negative words and just count the proportion of such words in text. The dictionary-based approach is less costly and less complex, and, as a result, has become popular among accounting and finance researchers. Moreover, a recent review paper on different methodologies to measure disclosure tone by Henry and Leone (2016) concludes that simple dictionary-based approaches often perform as well as (or better than) more complex machine-learning algorithms. Therefore, in this chapter, we focus the use of dictionaries to analyze text.

There are many different dictionaries for content analysis, such as:

1. General dictionaries for tone / sentiment analysis:
   http://www.wjh.harvard.edu/~inquirer/homecat.htm
   Tetlock (2007), Tetlock *et al.* (2008), and Price *et al.* (2012) are examples of studies that use general dictionaries of positive and negative words to measure tone / sentiment of financial texts.
2. Business domain-specific dictionaries for tone / sentiment analysis:
   https://sraf.nd.edu/textual-analysis/resources/#LM%20Sentiment%20Word%20Lists
   Loughran and McDonald (2011), Loughran and McDonald (2013), Huang *et al.* (2014), Bochkay and Levine (2019), among others, are examples of studies that rely on positive and negative dictionaries in the business domain.[1]
3. Dictionaries of forward-looking terms:
   Muslu *et al.* (2015) and Bozanic *et al.* (2018) develop dictionaries of forward-looking terms to identify forward-looking sentences in annual and quarterly SEC filings.
4. Dictionary of linguistic extremity:
   https://inkwellanalytics.com/textart/extreme.html

[1] A recent study by Heitmann *et al.* (2020) evaluates the accuracy of different sentiment analysis methods. We refer our readers to that research to learn more about popular sentiment analyses methods.

For example, Bochkay *et al.* (2020) develop a dictionary of linguistic extremity to measure the extent of exaggeration in corporate disclosures.

5. Dictionary of legal terms:
   http://www.learnaboutlaw.com/General/glossary.htm
   For example, Hanley and Hoberg (2010) use a dictionary with legal terms to capture legal texts in IPO prospectuses.
6. Dictionary of corporate governance terms:
   www.corp-gov.org/glossary.php3
   For example, Hanley and Hoberg (2010) use a dictionary of corporate governance terms to measure the relevance of corporate governance disclosures in IPO pricing.

Depending on your research question, you can always construct your own dictionary (e.g., dictionary of specific accounting terms that identify non-GAAP reporting as in Bentley *et al.* (2018), dictionary of words identifying risk disclosures as in Campbell *et al.* (2014), dictionary of macroeconomic terms as in Brochet *et al.* (2018), dictionary of words identifying shareholder value and value creation disclosures as in Larcker and Zakolyukina (2012)).

7.2 Understanding Dictionaries

Before performing a large scale word count based on a specific dictionary, it is important to understand the structure of the input dictionary. Do all words in the dictionary begin with a lowercase or uppercase letter, or both cases are possible? Are all words in the dictionary in their *base* form? The *base* form is the word's *infinitive* (e.g., to deliver, to develop, to earn) but without 'to'. Or, does the dictionary include both base form words and their conjugations (e.g., deliver, delivered, delivering, delivers)?

To mitigate the problem of not matching dictionary words due to inconsistencies in word capitalization, we highly recommend to convert all words in the input dictionary as well as the main text to lowercase. In fact, this should be the first preparatory step for a large-scale word

count.[2,3]

If an input dictionary includes base form words only, then counting the frequencies of such words in a text will result in significant understatement of the 'true' word count in the document. This will happen because a simple regular expression in the form `r'\b' + word + r'\b'` will find matches of base words only, and all words that have different endings will be ignored. For example, if an input dictionary contains 'damage' (and no other words with the same beginning) as one of its negative words, then regex `r'\bdamage\b'` will return zero matches in a sentence "Our business could be damaged" because the ending 'ed' is not specified in the regex pattern.

There are several ways to deal with this problem. First, we can develop a more complex regex that will allow different endings in regex word matching. This approach works relatively well; however, we have to be careful with matching words that may have different meanings depending on the word ending (e.g., care*ful* vs. care*less*). Second, we can perform word stemming or lemmatization of input text documents, so that all input document words are in their base form.[4] This approach also works well; however, stemming or lemmatization are not always 100% accurate, introducing some noise in the subsequent word count. Finally, we can modify the original input dictionary by manually adding derivative words to the dictionary, i.e., including 'damage', 'damages', 'damaging', 'damaged' as negative words in the dictionary. This approach is feasible if working with relatively short lists of words, but may become increasingly costly if working with thousands of words in the dictionary.

We have to be even more careful when input dictionaries contain both single words and multi-word phrases. Then, in addition to different word endings, we should consider whether other words may be present in the middle of a given phrase. For example, if an input dictionary contains 'economic environment' as one of its entries, then

[2]Python's `string.lower()` method converts all characters in an input string into lowercase characters (e.g., `"Higher returns".lower()` returns "higher returns").

[3]Pandas' `.str.lower()` method converts all characters in a series/column of data to lowercase (e.g., `df['colname'] = df['colname'].str.lower()` replaces text in column `colname` of DataFrame `df` with its lowercase equivalent).

[4]We cover *stemming* and *lemmatization* methods in Section 7.4.

regex `r'\beconomic environment\b'` will not yield a match in sentence "Our performance greatly depends on economic and competitive environment." This happens because 'and competitive' is not a part of the regex pattern and we are not allowing for other words to be part of this pattern search. To allow for up to two words in the middle of 'economic environment' phrase, we can use regular expression `r'\beconomic\W+(?:\w+\W+){0,2}?environment\b'`. The quantifier `{0,2}?` allows zero to two words between "economic" and "environment."[5]

7.3 Identifying Words and Sentences in Text

A large portion of textual analysis involves identifying words and sentences in texts. In this chapter, we provide several examples on how to split textual documents into single words and / or sentences. For example, to calculate the number of all words in text, we can use the following Python code with regex:

In:

```
text = "We invested in six areas of the business that
    account for nearly 40% of total Macy's sales.
    Dresses, fine jewelry, big ticket, men's tailored,
    women's shoes and beauty, these investments were
    aimed at driving growth through great products, top
    -performing colleagues, improved environment and
    enhanced marketing. All six areas continued to
    outperform the balance of the business on market
    share, return on investment and profitability. And
    we capture approximately 9% of the market in these
    categories."
x = re.findall(r"\b[a-zA-Z\'\-]+\b", text)
# Regex "\b[a-zA-Z\'\-]+\b" searches for all words in
# text, allowing apostrophes and hyphens in words,
# e.g., company's, state-of-the-art
print(x)
print(len(x))
```

Out:

[5] Again, we highly recommend to test all regex expressions prior to performing large scale counts. https://regex101.com/ is a useful website for that.

```
['We', 'invested', 'in', 'six', 'areas', 'of', 'the', '
    business', 'that', 'account', 'for', 'nearly', 'of'
    , 'total', "Macy's", 'sales', 'Dresses', 'fine', '
    jewelry', 'big', 'ticket', "men's", 'tailored', "
    women's", 'shoes', 'and', 'beauty', 'these', '
    investments', 'were', 'aimed', 'at', 'driving', '
    growth', 'through', 'great', 'products', 'top-
    performing', 'colleagues', 'improved', 'environment
    ', 'and', 'enhanced', 'marketing', 'All', 'six', '
    areas', 'continued', 'to', 'outperform', 'the', '
    balance', 'of', 'the', 'business', 'on', 'market',
    'share', 'return', 'on', 'investment', 'and', '
    profitability', 'And', 'we', 'capture', '
    approximately', 'of', 'the', 'market', 'in', 'these
    ', 'categories']
73
```

Regex for identifying sentences is more complex as sentences themselves are more complex textual units compared single words. In linguistics, a sentence is a textual unit delimited by graphological features such as UPPER case letters and markers such as periods, question marks, and exclamation marks (“.”, “!”, “?”). Therefore, to identify a sentence, we need a regex that will search for these features. The following regex elements can help us to identify a sentence:

- `\b` - a sentence should be preceded by a word boundary;
- `[A-Z]` - a sentence should start with a capital letter;
- `[.!?]` - a sentence should should end with either “.”, “!”, or “?”;
- `(?:[^\.!?]|\.\d)*` - allowing (1) any character that is not “.”, “!”, or “?”, and (2) “.” followed by a digit (to allow decimal numbers, such as 3.14, to be a part of a sentence).

The following example identifies sentences using the regex elements described above. To facilitate the code exposition, we continue using the input text from the previous example.

In:

```
# Regex pattern that identifies a sentence
# re.compile compiles a regular expression pattern
# into a regular expression object in Python
sentence_regex = re.compile(r"\b[A-Z](?:[^\.!?]|\.\d)
    *[\.!?]")


def identify_sentences (input_text:str):
```

```
    # finds all matches of sentence_regex in input_text
    sentences = re.findall(sentence_regex, input_text)
    return sentences

sentences = identify_sentences(text)

# enumerate is a Python function that when applied to
# a list, returns list elements along with their
# indexes (counter); 1 indicates that the counter
# should start from 1 instead of default 0
for counter, sentence in enumerate(sentences, 1):
    print(counter, sentence)
```

Out:

```
1 "We invested in six areas of the business that
   account for nearly 40% of total Macy's sales."
2 "Dresses, fine jewelry, big ticket, men's tailored,
   women's shoes and beauty, these investments were
   aimed at driving growth through great products, top
   -performing colleagues, improved environment and
   enhanced marketing."
3 "All six areas continued to outperform the balance of
    the business on market share, return on investment
    and profitability."
4 "And we capture approximately 9% of the market in
   these categories."
```

Some Python libraries offer alternative ways to identify words and sentences in texts. Namely, `spacy` is an open-source natural language processing library for Python that identifies various linguistic structures. `spacy` can be installed using either `conda` (if using anaconda distribution) or `pip` package managers:

```
conda install -c conda-forge spacy
pip install spacy
```

Prior to using `spacy`, we need to download its English (or other language) model. This has to be done one time only.

```
python -m spacy download en
```

The following example shows how to use `spacy` to identify (word) tokens and sentences in a text:

In:

```
import spacy
# load the English language model in spacy
nlp = spacy.load('en_core_web_sm')
# create an "nlp" object that parses a textual document
a_text = nlp(text)

# create a list of word tokens; note, this list will
# include punctuation marks and other symbols
token_list = []
for token in a_text:
    token_list.append(token.text)
print(token_list)

sentences = list(a_text.sents)

# print all sentences
for counter, sentence in enumerate(sentences, 1):
    print(counter, sentence)
```

Out:

```
['We', 'invested', 'in', 'six', 'areas', 'of', 'the', '
    business', 'that', 'account', 'for', 'nearly', '40'
    , '%', 'of', 'total', 'Macy', "'s", 'sales', '.', '
    Dresses', ',', 'fine', 'jewelry', ',', 'big', '
    ticket', ',', 'men', "'s", 'tailored', ',', 'women'
    , "'s", 'shoes', 'and', 'beauty', ',', 'these', '
    investments', 'were', 'aimed', 'at', 'driving', '
    growth', 'through', 'great', 'products', ',', 'top'
    , '-', 'performing', 'colleagues', ',', 'improved',
     'environment', 'and', 'enhanced', 'marketing', '.'
    , 'All', 'six', 'areas', 'continued', 'to', '
    outperform', 'the', 'balance', 'of', 'the', '
    business', 'on', 'market', 'share', ',', 'return',
    'on', 'investment', 'and', 'profitability', '.', '
    And', 'we', 'capture', 'approximately', '9', '%', '
    of', 'the', 'market', 'in', 'these', 'categories',
    '.']

1 "We invested in six areas of the business that
    account for nearly 40% of total Macy's sales."
2 "Dresses, fine jewelry, big ticket, men's tailored,
    women's shoes and beauty, these investments were
    aimed at driving growth through great products, top
    -performing colleagues, improved environment and
    enhanced marketing."
3 "All six areas continued to outperform the balance of
```

```
    the business on market share, return on investment
    and profitability."
4 "And we capture approximately 9% of the market in
   these categories.
```

7.4 Stemming and Lemmatization

In Chapter 7.2, we discussed that having a dictionary with base words only may result in significant understatement of how many dictionary words occur in text. *Stemming* and *lemmatization* are two popular ways to mitigate this problem. Stemming is the process of reducing inflected words (i.e., derivative words) to their base form called *word stem*. *Lemmatization* is a more complex approach to determine a word's stem; it requires determining the part of the speech of a word and its meaning in text. Intuitively, stemming and lemmatization are closely related. The main difference is that stemming is performed on a single word basis without taking into account the word's context, and lemmatization requires the knowledge of the context. Cecchini *et al.* (2010), Lang and Stice-Lawrence (2015), and Bochkay and Levine (2019) are examples of research studies that apply these techniques to corporate disclosures.

In this chapter, we demonstrate how to perform stemming and lemmatization of texts using Python. Additional information on both algorithms can be found at https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html.

In Python, there is a `stem` module in the Natural Language Toolkit (`NLTK`) which includes a collection of libraries and programs for natural language processing (NLP). We first need to import `NLTK`'s stemming and lemmatization modules and then apply `stem` and `lemmatize` commands to the words of interest. For example:

In:

```
# import Porter stemmer Module
from nltk.stem import PorterStemmer
# import WordNet lemmatization Module
from nltk.stem import WordNetLemmatizer

# object for Porter stemmer
stemmer = PorterStemmer()
# object for WordNet lemmatizer
```

```
lemmatizer = WordNetLemmatizer()

# Then, performing stemming on single words is as
# simple as:
print(f"Stemming for 'increasing' is {stemmer.stem('
    increasing')}")
print(f"Stemming for 'increases' is {stemmer.stem('
    increases')}")
print(f"Stemming for 'increased' is {stemmer.stem('
    increased')}")

# To improve the accuracy of lemmatization, we need to
# provide each word's part of the speech (POS)
# specifying POS as verb "v"
print(f"Lemmatization for 'increasing' is {lemmatizer.
    lemmatize('increasing', pos='v')}")
print(f"Lemmatization for 'increases' is {lemmatizer.
    lemmatize('increases', pos='v')}")
print(f"Lemmatization for 'increased' is {lemmatizer.
    lemmatize('increased', pos='v')}")
```

Out:

```
Stemming for 'increasing' is increas
Stemming for 'increases' is increas
Stemming for 'increased' is increas
Lemmatization for 'increasing' is increase
Lemmatization for 'increases' is increase
Lemmatization for 'increased' is increase
```

Performing lemmatization or stemming on a sentence level requires more work as we need to split sentences into single words and identify each word's part of the speech. The following code demonstrates this in detail:

In:

```
# WordNet is just another NLTK corpus reader
from nltk.corpus import wordnet
# uncomment the following line if
# 'averaged_perceptron_tagger' has not been yet
# downloaded
# nltk.download('averaged_perceptron_tagger')

# import NLTK tokenizer and (part of speech) POS tagger
from nltk import word_tokenize, pos_tag
# import Porter stemmer class
from nltk.stem import PorterStemmer
```

```
# import WordNet lemmatizer class
from nltk.stem import WordNetLemmatizer
# default dictionary is similar to Python's regular
# dictionary, but allows the dictionary to return a
# default value if a requested key does not exist in
# the dictionary from collections import defaultdict

# object for Porter stemmer
stemmer = PorterStemmer()
# object for WordNet lemmatizer
lemmatizer = WordNetLemmatizer()

# create a dictionary where single-letter keys are
# mapped to part of speech (noun, adjective, etc.)
# WordNet identifiers; by default, if a key does not
# exists the dictionary, return noun (wordnet.NOUN)
tag_map = defaultdict(lambda: wordnet.NOUN)
# add key 'J' to the dictionary indicating adjective
tag_map['J'] = wordnet.ADJ
# add key 'V' to the dictionary indicating verb
tag_map['V'] = wordnet.VERB
# add key 'R' to the dictionary indicating adverb
tag_map['R'] = wordnet.ADV

text = "We delivered adjusted earnings per share of $2
    .12. For the year, comparable sales were down 0.7%
    on an owned plus licensed basis, and we delivered
    adjusted earnings per share of $2.91."

# function that stems text
def stem_text(text:str):
    # split text into (word) tokens
    tokens = word_tokenize(text)
    stemmed_text = []
    for token in tokens:
        stem = stemmer.stem(token)
        stemmed_text.append(stem)
    # concatenate stemmed tokens elements with
    # space (" ") in-between
    return " ".join(stemmed_text)

# function that to lemmatizes text
def lemmatize_text(text:str):
    # splits text into tokens
    tokens = word_tokenize(text)
    lemmatized_text = []
```

```
        for token, tag in pos_tag(tokens):
            # lemmatize word tokens, tag[0] returns POS
            # letter identifier
            lemma = lemmatizer.lemmatize(token, tag_map[tag
        [0]])
            lemmatized_text.append(lemma)
        # concatenate lemmatized tokens elements with
        # space in-between
        return " ".join(lemmatized_text)

    # print stemmed version of text
    print(stem_text(text))
    # print lemmatized version of text
    print(lemmatize_text(text))
```

Out:

```
"We deliv adjust earn per share of $ 2.12 . for the
    year , compar sale were down 0.7 % on an own plu
    licens basi , and we deliv adjust earn per share of
     $ 2.91 ."

"We deliver adjusted earnings per share of $ 2.12 . For
     the year , comparable sale be down 0.7 % on an
    owned plus licensed basis , and we deliver adjusted
     earnings per share of $ 2.91 ."
```

7.5 Word Weighting

An important decision to make when performing dictionary-based word counts is how to weight individual word counts. Are all words equally relevant to the concept we are trying to measure? Or, do some words carry more weight (i.e., are more important in some sense) than others?

The simplest approach is to assign equal weights to individual words, so that the total count is equal to:

$$ProportionOfDictionaryWords_j = \frac{\sum_i Count_{i,j}}{TotalWords_j}, \qquad (7.1)$$

where $Count_{i,j}$ is the number of times a dictionary term i occurs in a document j, and $TotalWords_j$ is the total number of words in the document.

A popular alternative to equal-weighting of all word counts is to weight each word count by its document frequency. Specifically, *inverse*

document frequency (IDF) of a word i,

$$IDF_i = Log(\frac{\text{Number of documents in the sample}}{\text{Number of documents in the sample containing a word } i})$$

penalizes commonly-used words and assigns greater weights to less common words. For example, if "increase" occurs in every document in the sample, then its IDF weight is zero (i.e., log(1)). Stated formally, we apply the following formula to calculate the weighted proportion of dictionary words in the document:

$$WeightedProportionOfDictionaryWords_j = \frac{\sum_i Count_{i,j} \times idf_i}{TotalWords_j}. \tag{7.2}$$

In Chapter 10, we include an example of how to calculate *idf* weights in Python. More information about word weightings can be found at: https://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html.
Also, in the context of corporate disclosures, Loughran and McDonald (2011) and Jegadeesh and Wu (2013) provide great examples of different word weighting methods when counting dictionary words.

7.6 Dictionary-Based Word-Count Functions

Sections 7.1-7.5 establish building blocks for performing a large-scale dictionary-based word count of textual documents. In this chapter, we provide illustrative examples of how to apply these building blocks to calculate a document's tone. To facilitate the interpretation of Python code, we include explanatory commentaries throughout the code.

We begin by uploading dictionaries of words that we are interested in counting. We recommend having these dictionary files in plain text (.txt) format, either tab delimited or comma separated, with every dictionary word / phrase in a separate line. In the example below, both "positive.txt" and "negative.txt" dictionary files contain base-form words as well as inflected words (e.g., increase, increases, increasing, increased), so we do not need to perform word stemming or lemmatization.

In:

```
import re
```

```
# Let us start with a simple tone analysis, where each
# word is equally-weighted and we do not account for
# negators.

# First, we need to specify the locations of
# our dictionary files.
# file path (location) to a text file with positive
# words; every word is in a separate line in the file
positive_words_dict = r"./dictionaries/positive.txt"
# file path to a text file with negative words
negative_words_dict = r"./dictionaries/negative.txt"

# To be able to match all positive and negative words
# from the dictionaries, we need to create a list of
# regular expressions corresponding to these words

# The following function reads all dictionary terms
# to a Python list, and converts the terms regular
# expressions
def create_dict_regex_list(dict_file:str):
    """Creates a list of regex expressions of
    dictionary terms."""
    # opens the specified dict_file in "r" (read) mode
    with open(dict_file,"r") as file:
        # reads the content of the file
        # line-by-line and creates a list of
        # dictionary phrases
        dict_terms = file.read().splitlines()
    # re.compile(pattern) in Python compiles a regular
    # expression pattern, which can be used for
    # matching using its re.search, re.findall, etc.
    # by adding "\b" (i.e., word boundary) on each
    # side of a dictionary term in Regex, we force
    # an exact match that dictionary term
    dict_terms_regex = [re.compile(r'\b' + term + r'\b'
    )
                        for term in dict_terms]

    # specifies the output of the function - in our
    # case, a list of Regex expressions that
    # correspond to the input dictionary file
    return dict_terms_regex

# Now we can apply our function to create Regex lists
# for positive and negative dictionary terms
positive_dict_regex = create_dict_regex_list(
```

```
    positive_words_dict)
negative_dict_regex = create_dict_regex_list(
    negative_words_dict)

# print the first three entries of each Regex
# dictionary
print(positive_dict_regex[0:3])
print(negative_dict_regex[0:3])
```

Out:

```
[re.compile('\\bable\\b'), re.compile('\\babundance\\b'
    ), re.compile('\\babundant\\b')]
[re.compile('\\babandon\\b'), re.compile('\\babandoned
    \\b'), re.compile('\\babandoning\\b')]
```

Next, we need to write a function that will count positive, negative, and all words in a given text, so we can calculate document *Tone* as follows:

$$Tone(\%) = 100 \times \frac{(PositiveWordCount - NegativeWordCount)}{TotalWordCount}$$

As before, to facilitate exposition of our code, we add comments in every line of the code.

In:

```
def get_tone (input_text:str):
    """Counts All and Specific Words in Text"""

    ### Positive Words ###

    # finds all regex matches and returns them as a
    # list of lists so, the output of this search
    # will be of the following format:
    # [['able'], [], ['abundant','abundant'], [], ... ]
    positive_words_matches = [re.findall(regex,
    input_text) for regex in positive_dict_regex]

    # len() measures the length of each list match
    # so, the output of this list transformation
    # will be of the following format: [1,0,2,0,...]
    positive_words_counts = [len(match) for match in
    positive_words_matches]

    positive_words_sum = sum(positive_words_counts)

    ### Negative Words ###
```

```
    # in similar manner, we can get word counts for
    # negative words
    # finds all matches of negative words'
    # regular expressions
    negative_words_matches = [re.findall(regex,
    input_text) for regex in negative_dict_regex]
    # calculates the number of matches for each
    # dictionary term regex
    negative_words_counts = [len(match) for match in
    negative_words_matches]
    negative_words_sum = sum(negative_words_counts)

    ### Total Words ###
    # searches for all words in text, allowing
    # apostrophes and hyphens in words, e.g.,
    # "company's", "state-of-the-art"
    total_words = re.findall(r"\b[a-zA-Z\'\-]+\b",
    input_text)
    # calculates the number of all words in text
    total_words_count = len(total_words)

    # Finally, we can calculate Tone
    # (expressed in % terms) as:
    tone = 100 * (positive_words_sum -
    negative_words_sum)/total_words_count
    return (total_words_count, positive_words_sum,
    negative_words_sum, tone)

# Applying our count_words function to an input text:
counts = get_tone("At FedEx Ground, we have the market
    leading e-commerce portfolio. We continue to see
    strong demand across all customer segments with our
     new seven-day service. We will increase our speed
    advantage during the New Year. Our Sunday roll-out
    will speed up some lanes by one and two full
    transit days. This will increase our advantage
    significantly. And as you know, we are already
    faster by at least one day when compared to UPS's
    ground service in 25% of lanes. It is also really
    important to note our speed advantage and seven-day
     service is also very valuable for the premium B2B
    sectors, including healthcare and perishables
    shippers. Now, turning to Q2, I'm not pleased with
    our financial results.")
# output the results as (Total Word Count,
```

```
# Number of Positive Words, Number of Negative Words,
# Tone)
print(counts)
```

Out:

```
(114, 7, 0, 6.140350877192983)
```

When calculating document tone, we often encounter the problem of positive and negative words being negated in texts (e.g., not bad, not good). There are several ways to deal with this problem. We can simply reverse the sentiment of the word next to the negator. For example:

- "Our performance was not *bad*" with -1 original score → +1 shifted score, indicating that "not bad" has a positive sentiment.
- "Our performance was not *good*" with +1 original score → -1 shifted score, indicating that "not good" has a negative sentiment.

Alternatively, we can assign a new weight to a negated word in some systematic fashion. For example, we can assign 0.5 × original sentiment score to all negated words.

Here is a list of negators we might want to consider in our regular expressions:
not, never, no, none, nobody, nothing, don't, doesn't, won't, shan't, didn't, shouldn't, wouldn't, couldn't, can't, cannot, neither, nor

To account for negators in our previous code, we need to rewrite our regular expressions for word counts as follows:

In:

```
# First, we update our function that compiles regular
# expressions
def create_dict_regex_list_with_negators(dict_file:str)
    :
    """Creates a list of regex expressions of
    dictionary terms."""
    with open(dict_file,"r") as file:
        # reads dictionary lines one-by-one
        dict_terms = file.read().splitlines()

    # the first capturing group in this Regex
    # captures all possible negators, allowing for
    # zero or one match as indicated by ? after the
    # group; the second group captures dictionary terms
    dict_terms_regex =[re.compile(r"(not|never|no|none|
    nobody|nothing|don\'t|doesn\'t|won\'t|shan\'t|didn
```

```
    \'t|shouldn\'t|wouldn\'t|couldn\'t|can\'t|cannot|
    neither|nor)?\s(" + term + r")\b") for term in
    dict_terms]

    # returns a list of Regex expressions that
    # correspond to the input dictionary file,
    # allowing for negators
    return dict_terms_regex


# Now we can apply our function to create Regex lists
# for positive and negative dictionary terms
positive_dict_regex =
    create_dict_regex_list_with_negators(
    positive_words_dict)
negative_dict_regex =
    create_dict_regex_list_with_negators(
    negative_words_dict)

# prints the first entries of each Regex dictionary
print(positive_dict_regex[0])
print(negative_dict_regex[0])
```

Out:

```
re.compile("(not|never|no|none|nobody|nothing|don\\'t|
    doesn\\'t|won\\'t|shan\\'t|didn\\'t|shouldn\\'t|
    wouldn\\'t|couldn\\'t|can\\'t|cannot|neither|nor)
    ?\\s(able)\\b")
re.compile("(not|never|no|none|nobody|nothing|don\\'t|
    doesn\\'t|won\\'t|shan\\'t|didn\\'t|shouldn\\'t|
    wouldn\\'t|couldn\\'t|can\\'t|cannot|neither|nor)
    ?\\s(abandon)\\b")
```

Then, the updated version of our function to calculate document tone is as follows:

In:

```
# calculates tone with negators
def get_tone2 (input_text:str):
    """Counts All and Specific Words in Text, and
    checks for the presence of negators"""

    # find all words in text
    total_words = re.findall(r"\b[a-zA-Z\'\-]+\b",
    input_text)
    total_words_count = len(total_words)
```

```
# Positive Words #
# To account for negators, we can separately count
# positive and negated positive words
positive_word_count = 0
negated_positive_word_count = 0

for regex in positive_dict_regex:
    # searches for all occurences of Regex
    matches = re.findall(regex, input_text)
    for match in matches:
        # if match is not empty
        if len(match)>0:
            # prints the match output; this
            # is for illustration purposes
            # (i.e., optional)
            print(match)
        # if the first element of the match
        # is empty, no negator is present
        if match[0] == '':
            # so, increase the count of
            # positive words by 1
            positive_word_count += 1
        else:
            # otherwise, a negator is present,
            # so increase the count of negated
            # positive words by 1
            negated_positive_word_count += 1

# If we are simply shifting the sentiment of negated
# positive words (from +1 to -1), then the final
# positive word count is just:
positive_words_sum = positive_word_count

# Repeat the same for Negative Words:
negative_word_count = 0
negated_negative_word_count = 0

for regex in negative_dict_regex:
    # search for all occurences of Regex
    matches = re.findall(regex, input_text)
    for match in matches:
        # if match is not empty
        if len(match)>0:
            print(match)
        # if the first element of the match
        # is empty, no negator is present
```

```
                if match[0] == '':
                    # so, increase the count of
                    # negative words by 1
                    negative_word_count += 1
                else:
                    # otherwise, a negator is present, so
                    # increase the count of negated
                    # negative words by 1
                    negated_negative_word_count += 1

        # If we are simply shifting the sentiment of negated
        # negative words (from -1 to +1), then the final
        # negative word count is just:
         negative_words_sum = negative_word_count

         # Then, Tone is:
         tone = 100 * (positive_words_sum -
        negative_words_sum)/total_words_count
         return (total_words_count, positive_words_sum,
        negative_words_sum, tone)

    # Applying function get_tone2 function to an
    # example text:
    counts = get_tone2("At FedEx Ground, we have the market
         leading e-commerce portfolio. We continue to see
        strong demand across all customer segments with our
         new seven-day service. We will increase our speed
        advantage during the New Year. Our Sunday roll-out
        will speed up some lanes by one and two full
        transit days. This will increase our advantage
        significantly. And as you know, we are already
        faster by at least one day when compared to UPS's
        ground service in 25% of lanes. It is also really
        important to note our speed advantage and seven-day
         service is also very valuable for the premium B2B
        sectors, including healthcare and perishables
        shippers. Now, turning to Q2, I'm not pleased with
        our financial results.")
    # output results
    print(counts)
```

Out:

```
    ('', 'advantage')
    ('', 'advantage')
    ('', 'advantage')
    ('', 'leading')
    ('not', 'pleased')
```

```
('', 'strong')
('', 'valuable')
(114, 6, 0, 5.2631578947368425)
```

Finally, it is also possible that there are additional words between a given negator and a dictionary term (e.g., "not very encouraging" or "never went well"). In this case, we can modify the regular expressions above (in the `dict_terms_regex` list) and allow them to match phrases (N-grams) that contain dictionary terms (as opposed to individual dictionary terms). In other words, we can modify the regular expressions to allow for extra word(s) between the negators and dictionary terms:

```
# the first capturing group in this Regex
# captures all possible negators, allowing for
# zero or one match as indicated by ? after the
# group; the second group captures dictionary terms;
# the non-capturing group , s(:?\w+\s){0,2}, matches
# either none, one, or two words between a negator
# and a dictionary term.
dict_terms_regex =[re.compile(r"(not|never|no|none|nobody|
    nothing|don\'t|doesn\'t|won\'t|shan\'t|didn\'t|shouldn
    \'t|wouldn\'t|couldn\'t|can\'t|cannot|neither|nor)?\s
    (:?\w+\s){0,2}(" + term + r")\b") for term in
    dict_terms]
```

To summarize, performing dictionary-based counts is a relatively simple way of capturing information content in a text. However, we have to be mindful of potential issues that may arise because of formats of input dictionaries and different text semantic structures.

8

Quantifying Text Complexity

8.1 Understanding Text Complexity

Text complexity refers to a relative difficulty of reading, writing, or understanding a document, or a combination of these activities. Accounting and finance researchers have been interested, in particular, in complexity of corporate disclosures. Complex disclosures are difficult to read and comprehend and thus increase processing costs for users of such disclosures. Researchers proposed several reasons of why some corporate textual disclosures are more complex than others. For instance, complex reporting may be an attempt by managers to obfuscate information or conceal a firm's poor performance (Li, 2008; Lo *et al.*, 2017). On the other hand, complex disclosures may be the result of complex business operations and regulatory reporting requirements, and not intentional obfuscation by firm's management (Guay *et al.*, 2016; Dyer *et al.*, 2017; Chychyla *et al.*, 2019). Regardless of the reason, complex disclosures are associated with negative externalities such as less transparent information environments (You and Zhang, 2009; Lehavy *et al.*, 2011) and increased financial misstatement risk (Filzen and Peterson, 2015; Hoitash and Hoitash, 2018).

Quantifying text complexity is not a straightforward task as methods

used to quantify text complexity depend on the research question and underlying text source. For example, if a researcher is interested in measuring the relative difficulty of *reading* a piece of text, the researcher can utilize general text readability measures such as the fog index or Flesch-Kincaid readability score (Gunning *et al.*, 1952; Kincaid *et al.*, 1975). For a given text, these measures output readability scores based on the average number of words per sentence and the average number of syllables per word; higher scores indicate higher text complexity.

However, if a researcher is interested in a more sophisticated measure of readability that, for example, captures writing style complexity, BOG index would be a better choice (see Bonsall *et al.*, 2017). Moreover, if the research objective is to quantify the difficulty of *preparing* financial reports (as opposed to reading them), a more appropriate complexity measure can be the one that quantifies the complexity of accounting standards applicable to these financial reports (see Chychyla *et al.*, 2019).

In this Chapter, we introduce several commonly-used text complexity measures and demonstrate how to calculate them in Python.

8.2 Calculating Text Length

Text length is the simplest, yet informative, measure of text complexity. It is equal to the total word count in a given text. Since lengthier texts take more time to read (and write), they are more difficult to process (and prepare). In Chapter 7, in order to calculate linguistic tone, we wrote a Python code that counts the total number of words in a given text. We now present that code as a function, called `count_words`, that calculates word count for an input text:

```
import re

def identify_words(input_text:str):
    """Extracts all words from a given text."""
    words = re.findall(r"\b[a-zA-Z\'\-]+\b",input_text)
    return words

def count_words (input_text:str):
    """Counts the number of words in a given text."""
    words = identify_words(input_text)
```

```
    # calculates the number of words in a given text
    word_count = len(words)
    return word_count
```

In the code above, function `identify_words` uses a regular expression to find and output all words in a given text. Function `count_words` first applies `identify_words` to a given text to get a list of all words from that text (stored in variable `words`). It then outputs the length of that list, which is equivalent to the total number of words in text.

Applying `count_words` to a short text would yield:

In:

```
# excerpt from Microsoft Corporation's 2016 10-K.
text = """We acquire other companies and intangible
    assets and may not realize all the economic benefit
     from those acquisitions, which could cause an
    impairment of goodwill or intangibles. We review
    our amortizable intangible assets for impairment
    when events or changes in circumstances indicate
    the carrying value may not be recoverable. We test
    goodwill for impairment at least annually. Factors
    that may be a change in circumstances, indicating
    that the carrying value of our goodwill or
    amortizable intangible assets may not be
    recoverable, include a decline in our stock price
    and market capitalization, reduced future cash flow
     estimates, and slower growth rates in industry
    segments in which we participate. We may be
    required to record a significant charge on our
    consolidated financial statements during the period
     in which any impairment of our goodwill or
    amortizable intangible assets is determined,
    negatively affecting our results of operations."""

text_length = count_words(text)
print(f"Number of words in text: {text_length}")
```

Out:

```
Number of words in text: 143
```

In a similar manner, we can write a function that counts the number of sentences in an input text by modifying the regular expression in the `identify_words` to match sentences as discussed in Chapter 7.

In:

```
def identify_sentences(input_text:str):
    """Extracts all sentences from a given text."""
    sentences = re.findall(r"\b[A-Z](?:[^\.!?]|\.\d)
    *[\.!?]",input_text)
    return sentences

def count_sentences (input_text:str):
    """Counts the number of sentences in input_text."""
    sentences = identify_sentences(input_text)
    sentence_count = len(sentences)
    return sentence_count

num_sentences = count_sentences(text)
print(f"Number of sentences in text: {num_sentences}")
```

Out:

```
Number of sentences in text: 5
```

8.3 Measuring Text Readability Using the Fog Index

The fog index (or Gunning fog index) is a numerical score assigned to an input text where larger values indicate greater difficulty of reading the text. Reading levels by grade approximately correspond to the following fog index scores:

Grade	Fog index	Grade	Fog index
Sixth grade	6	High school senior	12
Seventh grade	7	College freshman	13
Eighth grade	8	College sophomore	14
High school freshman	9	College junior	15
High school sophomore	10	College senior	16
High school junior	11	College graduate	17

For a given piece of text, the score is calculated as the weighted sum of the average number of words per sentence and the average number of complex words (i.e., words with three or more syllables) per word:

$$\text{Fog index} = 0.4 \times \left[\frac{\text{all words}}{\text{all sentences}} + 100 \times \frac{\text{complex words}}{\text{all words}}\right] \tag{8.1}$$

Several Python packages provide functions that compute fog index. We will show how to use one of these packages, `py-readability-metrics`, later in this Chapter. However, it is not very difficult to write a function to calculate the fog index that relies only on regular expressions. Below, we demonstrate how to write such function. We encourage the reader to practice writing Python code from scratch (rather then rely on available libraries) when learning Python and textual analysis techniques.

8.3.1 Writing a Function to Calculate the Fog Index

As per Eq. (8.1), we need three pieces of numerical information to calculate the fog index for a given text: number of sentences, number of words, and number of complex words (i.e., words with three or more syllables). We already know how to calculate the number of words and sentences for a given text. Yet, we still need a method to identify complex words. A simple, albeit not 100% accurate, approach to count syllables in a word is to count the number of vowels that are either at the beginning of the word or follow a consonant, excluding the vowel *e* from the count if it is the last character in the word.

The following regular expression can be used to "capture" all such vowels when applied to a single word: `'(^|[^aeuoiy])(?!e$)[aeouiy]'`, where:

- The first part, `(^|[^aeuoiy])`, is a group that either matches the beginning of a word, `^`, or a non-vowel character (consonant), `[^aeuoiy]`.
- The last part, `[aeuoiy]`, is a character set that matches a vowel. The last part together with the first part identifies any vowel characters that are either at the beginning of the word or are preceded by a consonant.
- Finally, the middle part `(?!e$)` is a *negative lookahead* group. Negative lookahead groups have the following syntax `(?!`*pattern*`)`. A regular expression that has a negative lookahead group will fail to match an input text if the pattern included in that negative lookahead group is present in text. In our case, the pattern inside the negative lookahead group is `e$`; that is, the pattern matches the vowel *e* if it is the last character in the input word. As such, the

middle part ensures that our regular expression will not capture the vowel *e* if it is the last character in the given word.

We can use the regular expression above to write a function `count_syllables` that counts the number of syllables in a given word, and function `is_complex_word` that for a given word returns `True` if the word has more than three syllables in it, and `False` otherwise.

```
# regex pattern that matches vowels in a word
# (case-insensitive); used for syllable count
re_syllables = re.compile(r'(^|[^aeuoiy])(?!e$)[aeouiy]',
                          re.IGNORECASE)

def count_syllables(word:str):
    """Counts the number of syllables in a word."""
    # gets all syllable regex pattern matches
    # in the input word
    syllables_matches = re_syllables.findall(word)
    return len(syllables_matches)

def is_complex_word(word:str):
    """Checks whether word has three or more syllables."""
    return count_syllables(word) >= 3
```

Consider the following example with `count_syllables` and `is_complex_word` functions:

In:

```
print("Number of syllables in word \"Text\":",
      count_syllables("Text"))
print("Is word \"Text\" complex:",
      is_complex_word("Text"))
print("Number of syllables in word \"analysis\":",
      count_syllables("analysis"))
print("Is word \"analysis\" complex?:",
      is_complex_word("analysis"))
print("Number of syllables in word \"procedure\":",
      count_syllables("procedure"))
print("Is word \"procedure\" complex?:",
      is_complex_word("procedure"))
```

Out:

```
Number of syllables in word "Text": 1
Is word "Text" complex: False
Number of syllables in word "analysis": 4
Is word "analysis" complex?: True
```

```
Number of syllables in word "procedure": 3
Is word "procedure" complex?: True
```

We finally have all the necessary tools to write a function that computes the fog index score as in Eq. (8.1):

```
def calculate_fog(text:str):
    """Calculates the fog index for a given text."""
    # extracts all sentences from the input text
    sentences = identify_sentences(text)

    # extracts all words from the input text
    words = identify_words(text)

    # creates a list of complex words by using
    # is_complex_word function as a filter
    complex_words = list(filter(is_complex_word, words))

    # calculates and returns the fog index
    return 0.4*(float(len(words))/float(len(sentences)) +
    100*float(len(complex_words))/float(len(words)) )
```

For a given input text, `calculate_fog` identifies all sentences and words and stores them in `sentences` and `words` list variables, respectively. Next, the function creates a list called `complex_words` that includes all words with three or more syllables. Finally, the function calculates and returns an expression as defined in Eq. (8.1).[1]

We can now apply the `calculate_fog` function to any text. For example, applying `calculate_fog` to the text we used in the text length example (see previous section) would yield:

In:

```
fog_score = calculate_fog(text)
print("The fog index score is ", fog_score)
```

Out:

[1]Note that in the last line of `calculate_fog` definition we use `float` function to convert the *integer* (i.e., whole digit number) output of the list length function, `len`, to a *continuous* value (i.e., number with fraction). While in Python 3, dividing integer numbers yields continuous results (see section 4.3.1), in Python 2 the division operator, `/`, yields continuous number only if either the numerator or denominator is a continuous number. For example, if 5.0 and 2.0 are continuous numbers, 5.0 / 2.0 yields 2.5 in Python 2, as expected. If 5 and 2 are integer numbers, 5 / 2 yields 2 in Python 2; that is, Python 2 only yields the integer (whole number) part of the division if both the numerator and denominator are integer numbers.

```
the fog index score is 21.78965034965035
```

Interestingly, the fog index score of the text in the example is around 21.8, significantly higher than the expected fog index score of the graduate college reading level of 17. In accounting literature, Li (2008) finds that the average the fog index score of corporate annual (10-K) filings is also high – 19.39 on average.

8.3.2 Using Python Packages to Calculate the Fog Index

There are numerous Python packages that implement textual analysis techniques. In the previous chapters, we introduced `NTLK` and `Spacy` Python libraries. In this section, we consider a relatively small Python package called `py-readability-metrics` that facilitates calculation of various text readability metrics. This package relies on `NTLK` package and can be installed using `pip` package manager as follows:

```
pip install py-readability-metrics
python -m nltk.downloader punkt
```

Let us look at an example of how to calculate the fog index using this package:

In:

```
# Readability class provides methods to compute various
# readability metrics
from readability import Readability

# create a new Readability object with the example text
# as an input
r = Readability(text)

# calculate and output the fog index
fog_score = r.gunning_fog()
print(fog_score)
```

Out:

```
score: 21.78965034965035,
grade_level: 'college_graduate'
```

In the code above, we import a class called `Readability` from the `readability` library which is part of `py-readability-metrics` package. We then create object `r` by initializing `Readability` class with our

example text as an input. Finally, executing method `r.gunning_fog()` returns the text's fog index score and suggested grade reading level.

8.4 Measuring Text Readability Using BOG Index

In a recent study, Bonsall *et al.* (2017) propose an alternative measure of complexity, the BoG index.[2] They argue that the Bog index better reflects the spirit of the SEC's 1998 Plain English Mandate [SEC Rule 421(d)]. The Plain English Mandate requires registrants to prepare IPO prospectuses using "plain English". The SEC advocates avoiding certain writing constructs such as passive voice, weak or hidden verbs, superfluous words, legal jargon, numerous defined terms, abstract words, unnecessary details, lengthy sentences, and unreadable design and layout in their financial disclosures (Securities and Commission, 1998).

A shortcoming of the fog index is that it only captures one the nine attributes of plain English outlined by the SEC, which is lengthy sentences. It also captures abstract words to only to the extent that abstract words and the number of syllables in words are related. The Bog index measures abstract words using a proprietary list of over 200,000 words scored on familiarity. Words that occur less frequently on the Internet are considered to be more abstract.

One practical shortcoming of the Bog index has been that the only way to compute the Bog index is to run documents through the StyleWriter program, one document at a time, which can be time intensive. However, StyleWriter recently released a version of its software that can load approximately 100 files at once for batch processing with summary statistics being printed to a single output file. In addition, a Python wrapper for the software is being developed to allow batch processing for a much larger number of files. Bog Index scores for 10-Ks are available at http://textart.us.

[2]The Bog index was was developed for the StyleWriter software package, designed to help people improve their writing. See: http://www.stylewriter-usa.com for more details.

9

Sentence Structure and Classification

In Chapters 7 and 8, we discussed different textual analysis measures at the word level (e.g., tone, readability). However, it is often useful to analyze textual documents at the sentence level, especially when we want to preserve sentence structure, word dependencies, and time dependencies. In this chapter, we discuss basic techniques for working with sentences. We begin by explaining how to classify sentences as forward-looking. We then discuss dictionary-based techniques of determining sentence topics. Finally, we review techniques for identifying sentence subject and objects, and determining sentence named entities.

9.1 Identifying forward-looking sentences

Forward-looking statements (FLS) in corporate disclosures convey information about expected events, trends, and management's strategies and plans that can affect firms' operating and economic environments. Because forward-looking disclosures are important to investors, The Private Securities Litigation Reform Act of 1995 introduced safe harbor provision for such disclosures, facilitating disclosures of forward-looking information. To reduce potential litigation costs associated with FLS, all companies accompany their FLS with legal disclaimers stating that

FLS are not guarantees of future performance and different risks and uncertainties may cause a company's actual results to differ significantly from management's expectations.

To identify forward-looking statements, we will use the methodology developed in Muslu *et al.* (2015).[1] Specifically, Muslu *et al.* (2015) classify a sentence as a forward-looking statement if it includes one of the following:

- references to the future: *will, future, next fiscal, next month, next period, next quarter, next year, incoming fiscal, incoming month, incoming period, incoming quarter, incoming year, coming fiscal, coming month, coming period, coming quarter, coming year, upcoming fiscal, upcoming month, upcoming period, upcoming quarter, upcoming year, subsequent fiscal, subsequent month, subsequent period, subsequent quarter, subsequent year, following fiscal, following month, following period, following quarter,* and *following year*;
- future-oriented verbs and their conjugations: *aim, anticipate, assume, commit, estimate, expect, forecast, foresee, hope, intend, plan, project, seek, target*, etc.;
- reference to a year that comes after the year of the filing. For example, 2022 if the filing's fiscal year is 2020.

Therefore, to automatically capture a forward-looking statement, we need to write a code that tests whether at least one of the three FLS conditions is true. We will start with generating regular expressions that correspond to future-oriented terms as per Appendix "Identifying Forward-Looking Disclosures" in Muslu *et al.* (2015). To facilitate exposition of the code, we include explanatory comments in every line.

In:

```
import re

# To identify FLS, we need a dictionary file that
# includes future-oriented verbs and their
# conjugations as well as terms that identify
# references to the future. In our case, this
```

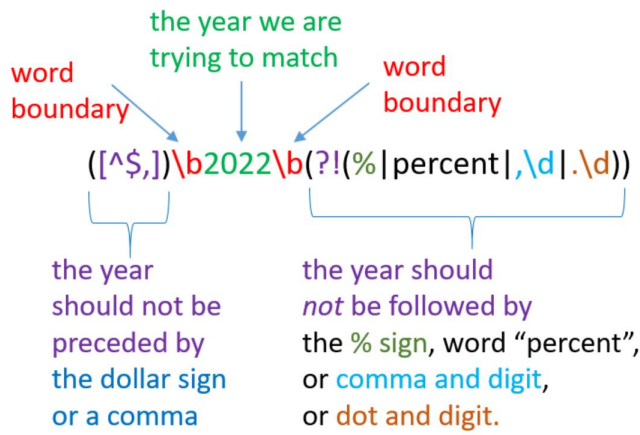
[1] Bozanic *et al.* (2018) is another excellent example of FLS classification. The methodology we review here can be used to replicate FLS measures as per Bozanic *et al.* (2018).

```
# file is "fls_terms.txt."

# file path (location) to a text file with FLS
# terms (dictionary structure: one term per line)
fls_terms_file = r".\dictionaries\fls_terms.txt"

# next, create a list of regex expressions that
# match FLS terms
def create_fls_regex_list(fls_terms_file:str):
    """Creates a list of regex expressions of
    FLS terms"""

    # opens the specified dict_file in "r" (read) mode
    with open(fls_terms_file,"r") as file:
        # reads the content of the file line-by-line
        # and creates a list of FLS terms
        fls_terms = file.read().splitlines()

    # creates a list of FLS regex expressions by adding
    # word boundary (\b) anchors to the beginning and
    # the ending of each FLS term
    fls_terms_regex = [re.compile(r'\b' + term + r'\b')
     for term in fls_terms]
    return fls_terms_regex

# creates a list of FLS regex expressions
fls_terms_regex = create_fls_regex_list(fls_terms_file)
print(fls_terms_regex[0:3])
```

Out:

```
[re.compile('\\bwill\\b'), re.compile('\\bfuture\\b'),
    re.compile('\\bnext fiscal\\b')]
```

In addition to checking whether a sentence contains one of the forward-looking terms, we need to check whether it contains a reference to future years. Note that to identify a year in text, we have to specify that a year cannot be a part of a (larger) number (e.g., 2020123), or preceded by a dollar sign ($) (e.g, $2020), or followed by a percentage sign (%) or word "percent". We can use the following regex expression to search for a specific year reference:



Once we have regular expressions for FLS terms and references to future years, we can write a function that checks if a sentence is forward-looking or not.

In:

```
def is_forward_looking(sentence:str, year:int):
    """Returns whether sentence is forward-looking."""

    #creates a list of regex expression that match up
    # to 10 years into the future
    future_year_terms=[re.compile(r"[^$,]\b" +
                                  str(y) +
                                  r"\b(?!(%|,\d|.\d))")
                       for y in range(year+1,year+10)]

    # combines FLS regex expressions, i.e., regular
    # expressions for FLS terms and future years
    fls_terms_with_future_years = fls_terms_regex +
    future_year_terms

    for fls_term in fls_terms_with_future_years:
        # fls_term.search(sentence) returns a match
        # object if there is a match, and "None"
        # if there is no FLS term match in the
        # sentence
        if fls_term.search(sentence):
            return True
    return False

# Input text - excerpt from Apple's Q4 2018
# Earnings Conference Call Transcript
```

```
text = """Finally, we launched a completely new website
experience for Atlanta. The new online experience
provides a modern and fresh brand look and includes
enhanced simplicity and flexibility for shopping and
buying that easily transitions to a home delivery or
in-store experience. We are excited to put the customer
in the driver seat. This experience is a unique and
powerful integration of our own in-store and online
capabilities. Keep in mind, we will continue to improve
both the customer and associate experience in Atlanta
and use these earnings to inform how we roll out into
other markets. As we previously announced, we
anticipate having the omni channel experience available
to the majority of our customers by February 2020. To
expand omni channel, we anticipate opening additional
customer experience centers. We're currently in the
process of planning the next locations while taking
state regulations into consideration."""

sentence_regex = re.compile(r"\b[A-Z](?:[^\.!?]|\.\d)
    *[\.!?]")
def identify_sentences(input_text:str):
    sentences = re.findall(sentence_regex, input_text)
    return sentences

sentences = identify_sentences(text)
for sentence in sentences:
    print(is_forward_looking(sentence,2018),":",
    sentence)
```

Out:

```
False : Finally, we launched a completely new website
    experience for Atlanta.
False : The new online experience provides a modern and
     fresh brand look and includes enhanced simplicity
    and flexibility for shopping and buying that easily
     transitions to a home delivery or in-store
    experience.
False : We are excited to put the customer in the
    driver seat.
False : This experience is a unique and powerful
    integration of our own in-store and online
    capabilities.
True : Keep in mind, we will continue to improve both
    the customer and associate experience in Atlanta
    and use these learnings to inform how we roll out
    into other markets.
```

```
True : As we previously announced, we anticipate having
     the omnichannel experience available to the
    majority of our customers by February 2020.
True : To expand omnichannel, we anticipate opening
    additional customer experience centers.
False : We're currently in the process of planning the
    next locations while taking state regulations into
    consideration.
```

9.2 Dictionary Approach to Sentence Classification

It is a common practice in corporate disclosure research to classify sentences by topic. For example, Muslu *et al.* (2015) further classify forward-looking statements (FLS) as operations-, financing-, and accounting-related; Bozanic *et al.* (2018) further classify FLS as earnings-related and/or quantitative in nature; Kravet and Muslu (2013) and Campbell *et al.* (2014) classify sentences as risk-oriented or not.

Performing a dictionary-based regex matching is an easy way to group sentences into categories. A challenge with this approach is to construct a comprehensive dictionary of words that can be used for sentence classification. Therefore, it is important to test reliability of the classification algorithm and check the construct validity of resulting text-based measures.

As an example of dictionary-based sentence classification, the Python code below illustrates how to classify a sentence as either earnings-oriented or not, and either quantitative or not as in Bozanic *et al.* (2018).

In:

```
# This code implements is a simplified version of
# sentence classification as earnings-oriented or
# not and quantitative or not as in Bozanic et
# al.(2018)

# regex for identifying sentences
sentence_regex = re.compile(r"\b[A-Z](?:[^\.!?]|\.\d)
    *[\.!?]")

def identify_sentences(input_text:str):
    """Returns all sentences in the input text"""
    sentences = re.findall(sentence_regex, input_text)
```

```
    return sentences

earn_terms = ["earnings", "EPS", "income", "loss",
              "losses", "profit", "profits"]
quant_terms = ["thousand", "thousands", "million",
               "millions", "billion", "billions",
               "percent", "%", "dollar", "dollars",
               "$"]

# creates a list of earnings regex expressions
earn_terms_regex = [re.compile(r'\b' + term + r'\b')
                    for term in earn_terms]
# creates a list of regexes for quantitative terms
quant_terms_regex = [re.compile(r'\b' + term + r'\b')
                     for term in quant_terms]

# checks if there is a match for at least one earnings
# term in the input sentence
def is_earn_oriented(sentence:str):
    """Checks whether a sentence is earnings-oriented.
    """
    for term in earn_terms_regex:
        if term.search(sentence, re.IGNORECASE):
            return True
    return False

# checks if there is a match for at least one
# qualitative term in the input sentence
def is_quantitative(sentence:str):
    """Checks whether a sentence is quantitative
    in nature."""
    for term in quant_terms_regex:
        if term.search(sentence, re.IGNORECASE):
            return True
    return False

# input text
text = """Operating income margins, excluding the
restructuring charges, are projected to be in the
range of 4.5% to 4.8%, and interest expense and
other income are forecasted to be approximately
$18 million and $6 million, respectively. While
operating performance is expected to remain
strong, Agribusiness profits are expected to be
lower in the third and fourth quarters as pricing
for subsequent sales will not match the high level
```

```
of the June delivery. The Company expects its
capital expenditures in 2008 to be approximately
$300 million, an 8% reduction from 2007 capital
expenditures of $326 million. During the third
quarter, the company made further progress
implementing the strategic cost reductions that
will support the targeted growth investments
announced in July 2005."""

sentences = identify_sentences(text)

# next, we classify each sentence as earnings-
# oriented or not, quantitative or not
for sentence in sentences:
    print("***Earnings-oriented:",
          is_earn_oriented(sentence),
          "***Quantitative:",
          is_quantitative(sentence),
          "---", sentence)
```

Out:

```
***Earnings-oriented: True ***Quantitative: True ---
    Operating income margins, excluding the
    restructuring charges, are projected to be in the
    range of 4.5% to 4.8%, and interest expense and
    other income are forecasted to be approximately $18
     million and $6 million, respectively.
***Earnings-oriented: True ***Quantitative: False ---
    While operating performance is expected to remain
    strong, Agribusiness profits are expected to be
    lower in the third and fourth quarters as pricing
    for subsequent sales will not match the high level
    of the June delivery.
***Earnings-oriented: False ***Quantitative: True ---
    The Company expects its capital expenditures in
    2008 to be approximately $300 million, an 8%
    reduction from 2007 capital expenditures of $326
    million.
***Earnings-oriented: False ***Quantitative: False ---
    During the third quarter, the company made further
    progress implementing the strategic cost reductions
     that will support the targeted growth investments
    announced in July 2005.
```

9.3 Identifying Sentence Subjects and Objects

In addition to identifying a sentence orientation using dictionaries, it is possible to parse full sentence structure and linguistic dependency using `spacy` library in Python. This kind of sentence tagging can be useful if we want to precisely identify an object and subject of a conversation. For example, we may want to identify most common objects in analysts' questions at earnings conference calls. In addition to sentence tagging support for different languages, `spacy` can identify stop words, named entities and monetary amounts in text, lemmatize text, and many other textual analysis techniques. We strongly recommend exploring this library for processing raw texts.[2]

First, we demonstrate how to extract sentences from text using `spacy`:

In:

```
import spacy

# load spacy's English language model
nlp = spacy.load("en_core_web_sm")

# a sample text
text = """Q1 revenue reached $12.7 billion. We are
thrilled with the continued growth of Apple Card.
We experienced some product shortages due to very
strong customer demand for both Apple Watch and
AirPod during the quarter. Apple is looking at
buying U.K. startup for $1 billion."""

# parses the input text using spacy's nlp class
parsed_text = nlp(text)

# gets a list of sentences identified by spacy
# property "sents" yields identified sentences
sentences = list(parsed_text.sents)

# recall that function enumerate() when applied
# to a list, returns its elements along with their
# indexes
for num,sentence in enumerate(sentences,1):
    print("Sentence", str(num), ":", sentence)
```

[2] See Chapter 7 for instructions on how to install `spacy` library.

Out:

```
Sentence 1 : Q1 revenue reached $12.7 billion.
Sentence 2 : We are thrilled with the continued growth
    of Apple Card.
Sentence 3 : We experienced some product shortages due
    to very strong customer demand for both Apple Watch
     and AirPod during the quarter.
Sentence 4 : Apple is looking at buying U.K. startup
    for $1 billion.
```

Next, we can apply `spacy`'s tagging method to identify subjects and objects in sentences. To do so, we continue our previous code by creating a function that extracts all (word) tokens from a sentence and their dependencies, then it filters them to keep only those tokens that are either "subj" (subjects) or "obj" (objects) in the sentence.

In:

```
def sentence_subj_obj(sentence):
    """Identifies subjects and objects in a sentence"""
    results = []
    for token in sentence:
         # records the token's text and its dependency
        entry = {"Token": token.text,
                 "Dependency": token.dep_}
        results.append(entry)

    # spacy parses token dependencies and assigns a
    # dependency code for each token; tokens that are
    # either objects or subjects will include "obj" or
    # "subj" in their dependency codes; for a full list
    # of spacy's dependencies and their codes, visit
    # spacy.io

    # creates a new list of tokens and their
    # dependencies based on results list by keeping
    # only tokens with "obj" and "subj" dependencies
    filtered_results=[entry for entry in results
                      if ('obj' in entry['Dependency'])
                      or
                      ('subj' in entry['Dependency'])]
    return filtered_results

# recall that function enumerate() when applied to a
# list, returns its elements along with their indexes
for num,sentence in enumerate(sentences,1):
    print("Sentence", str(num), ":",
```

```
            sentence_subj_obj(sentence))
```

Out:

```
Sentence 1 : [{'Token': 'revenue', 'Dependency': 'nsubj
    '}, {'Token': 'billion', 'Dependency': 'dobj'}]
Sentence 2 : [{'Token': 'We', 'Dependency': 'nsubjpass'
    }, {'Token': 'growth', 'Dependency': 'pobj'}, {'
    Token': 'Card', 'Dependency': 'pobj'}]
Sentence 3 : [{'Token': 'We', 'Dependency': 'nsubj'}, {
    'Token': 'shortages', 'Dependency': 'dobj'}, {'
    Token': 'demand', 'Dependency': 'pobj'}, {'Token':
    'Watch', 'Dependency': 'pobj'}, {'Token': 'quarter'
    , 'Dependency': 'pobj'}]
Sentence 4 : [{'Token': 'Apple', 'Dependency': 'nsubj'
    }, {'Token': 'startup', 'Dependency': 'dobj'}, {'
    Token': 'billion', 'Dependency': 'pobj'}]
```

Finally, spacy allows us to easily output and visualize complete sentence structure with all word dependencies:

In:

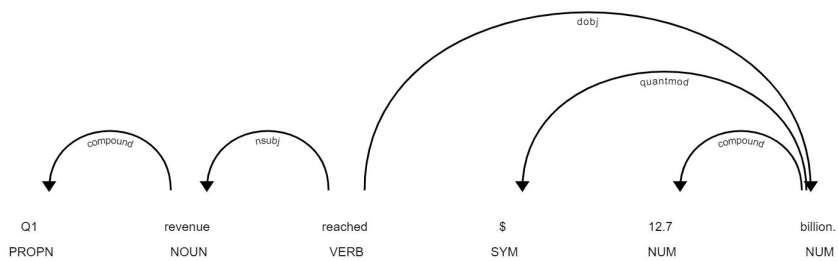
```
# displacy allows to visualize a sentence structure
from spacy import displacy

# tags all (word) tokens in an input sentence
def sentence_tagging(sentence):
    results = []
    for token in sentence:
        # gets a token, its lemmatized version, POS,
        # dependency, and checks whether it is a stop
        # word or not
        entry = {"Token": token.text,
                 "Lemma_Token": token.lemma_,
                 "POS": token.pos_,
                 "Dependency": token.dep_,
                 "Stop_word": token.is_stop}
        results.append(entry)
    return results

# applies sentence_tagging to all sentences
tagged_sentences = [sentence_tagging(s) for s in
    sentences]

# prints the output for the first sentence
print(tagged_sentences[0])

# visualizes sentence dependency
```

```
displacy.render(parsed_text, style="dep")
```

Out:

```
[{'Token': 'Q1', 'Lemma_Token': 'Q1', 'POS': 'PROPN', '
    Dependency': 'compound', 'Stop_word': False}, {'
    Token': 'revenue', 'Lemma_Token': 'revenue', 'POS':
     'NOUN', 'Dependency': 'nsubj', 'Stop_word': False
    }, {'Token': 'reached', 'Lemma_Token': 'reach', '
    POS': 'VERB', 'Dependency': 'ROOT', 'Stop_word':
    False}, {'Token': '$', 'Lemma_Token': '$', 'POS': '
    SYM', 'Dependency': 'quantmod', 'Stop_word': False
    }, {'Token': '12.7', 'Lemma_Token': '12.7', 'POS':
    'NUM', 'Dependency': 'compound', 'Stop_word': False
    }, {'Token': 'billion', 'Lemma_Token': 'billion', '
    POS': 'NUM', 'Dependency': 'dobj', 'Stop_word':
    False}, {'Token': '.', 'Lemma_Token': '.', 'POS': '
    PUNCT', 'Dependency': 'punct', 'Stop_word': False}]
```

Then, a `spacy`'s visual representation of "Q1 revenue reached $12.7 billion" is:



9.4 Identifying Named Entities

One of the major problems in textual disclosure research is to assess the level of information usefulness. The SEC specifically discourages use of "boilerplate" language in corporate disclosures. One way to quantify the degree of boilerplate information in disclosures is to calculate the level of similarity between various disclosures. For example, disclosure text that does not change over years (e.g., MD&A section or risk factors), or disclosure text that is very similar to disclosures of other firms in the industry may not be particularly useful. We discuss text similarity

measures in Chapter 10. A different approach is to quantify the level of *specificity* as in Hope *et al.* (2016).

Hope *et al.* (2016) measure text specificity as the number of specific *named entities* mentioned in the text divided by all words in the text. Named entities include names of organizations, people, places, numbers, dates, times, etc. The argument is that specific texts that explicitly mention named entities are less likely to be boilerplate than texts that use more general words. Dyer *et al.* (2017) and Gow *et al.* (2019) measure specificity in a similar manner.

To calculate text specificity measure as in Hope *et al.* (2016), we can use Named Entity Recognition (NER) feature in `spacy`. In the example below, we will use the same sample text as in the previous section, and its parsed version `parsed_text`. First, we demonstrate how to identify and extract named entities from text:

In:

```
# create a dictionary with descriptions for spacy's
# entity type codes; the list is available on spacy.io
entity_type_descriptions = {
    'PERSON':'People, including fictional.',
    'NORP':'Nationalities or religious or political
    groups.',
    'FAC':'Buildings, airports, highways, bridges, etc.
    ',
    'ORG':'Companies, agencies, institutions, etc.',
    'GPE':'Countries, cities, states.',
    'LOC':'Non-GPE locations, mountain ranges, bodies
    of water.',
    'PRODUCT':'Objects, vehicles, foods, etc. (Not
    services.)',
    'EVENT':'Named hurricanes, battles, wars, sports
    events, etc.',
    'WORK':'OF_ART  Titles of books, songs, etc.',
    'LAW':'Named documents made into laws.',
    'LANGUAGE':'Any named language.',
    'DATE':'Absolute or relative dates or periods.',
    'TIME':'Times smaller than a day.',
    'PERCENT':'Percentage, including "%".',
    'MONEY':'Monetary values, including unit.',
    'QUANTITY':'Measurements, as of weight or distance.
    ',
    'ORDINAL':'"first", "second", etc.',
    'CARDINAL':'Numerals that do not fall under another
```

```
        type.'}

    # gets a list of all named entities identified
    # by spacy, and output them
    # property "ents" returns all identified named
    # entities in the text
    named_entities = parsed_text.ents
    for ent in named_entities:
        # gets the named entity (ent.text)
        entity = ent.text
        # gets the named entity type code
        # (e.g., PERSON, ORG, etc.)
        entity_type = ent.label_
        # gets the named entity description from
        # entity_type_descriptions dictionary using
        # its type code
        entity_desc = entity_type_descriptions[entity_type]

        print(f'{entity:<15}{entity_type:<10}{entity_desc}'
        )
```

Out:

```
Q1              CARDINAL  Numerals that do not fall
    under another type.
$12.7 billion   MONEY     Monetary values, including
    unit.
Apple Card      ORG       Companies, agencies,
    institutions, etc.
Apple Watch     ORG       Companies, agencies,
    institutions, etc.
AirPod          ORG       Companies, agencies,
    institutions, etc.
the quarter     DATE      Absolute or relative dates or
    periods.
Apple           ORG       Companies, agencies,
    institutions, etc.
U.K.            GPE       Countries, cities, states.
$1 billion      MONEY     Monetary values, including
    unit.
```

Now, we can calculate the specificity measure by dividing the number of named entities by the number of words in text:

In:

```
# counts the number of all words
# we assume that every token in a sentence is a word
# unless it is punctuation.
```

```
num_words = len([token
                 for token in parsed_text
                 if not token.is_punct])

num_entities = len(named_entities)
specificity_score = num_words / num_entities

print('Number of named entities:', num_entities)
print('Number of words:', num_words)
print('Specificity score:', specificity_score)
```

Out:

```
Number of named entities: 9
Number of words: 52
Specificity score: 5.777777777777778
```

9.5 Using Stanford NLP for part-of-speech and named entity recognition tasks

In the code above, we show how to use `spacy` library to tokenize text and identify named entities. Another popular set of tools for natural language analysis is Stanford NLP. For example, Hope *et al.* (2016) use Stanford NLP to calculate their specificity measure. Stanford NLP's official Python library is called `Stanza`. It includes tools for sentence and word recognition, multi-word token expansion, lemmatization, part-of-speech dependency parsing, and name entity recognition parsing. Below, we demonstrate how to use Stanford NLP in Python for part-of-speech and NER applications.

`Stanza` can be installed using either `conda` or `pip` as follows:

```
conda install -c stanfordnlp stanza
pip install stanza
```

Before processing text, we need to download a `Stanza` language module and create a `Pipeline` object. Pipeline object specifies the type of processing that will be applied to a given text (e.g., tokenization, lemmatization, dependency parsing, etc.).

```
import stanza
# downloads the English module. The size of the
# downloaded module is about 400 MB. The module
```

```
# has to be download only once
stanza.download('en')

# creates a (text processing) Pipeline object using
# the English language module with tokenizer, part
# of speech and named entity recognition
nlp = stanza.Pipeline(lang = 'en', processors = 'tokenize,
    pos,ner')
```

Next, we create a `Stanza` document object by providing an input text. `Stanza` will immediately parse the input text using previously specified text processors at this step. We can retrieve parsed sentences and words through document object properties:

In:

```
# sample text (same as in the previous example)
text = """Q1 revenue reached $12.7 billion. We are
    thrilled with the continued growth of Apple Card.
    We experienced some product shortages due to very
    strong customer demand for both Apple Watch and
    AirPod during the quarter. Apple is looking at
    buying U.K. startup for $1 billion."""

# creates Stanza document object
doc = nlp(text)

# extracts sentences
sentences = doc.sentences

print('Sentences:')
# prints the first 20 characters of each sentence
for sentence in sentences:
    print(sentence.text[0:20] + '...')

print('\nWords:')
# prints all the words in the first sentence
for word in sentences[0].words:
    print(word.text)
```

Out:

```
Sentences:
Q1 revenue reached $...
We are thrilled with...
We experienced some ...
Apple is looking at ...
```

```
Words:
Q1
revenue
reached
$
12.7
billion
.
```

For each word, we can output its part-of-speech tag by accessing the value of its `.pos` property:

In:

```
# outputs POS information for each word in the second
    sentence
for word in sentences[1].words:
        print(f'{word.text:<10} {word.pos}')
```

Out:

```
We          PRON
are         AUX
thrilled    ADJ
with        ADP
the         DET
continued   VERB
growth      NOUN
of          ADP
Apple       PROPN
Card        PROPN
.           PUNCT
```

Similarly, we can output all entities identified by `Stanza`'s NER processor for a given text (or individual sentence) by accessing `.ents` property:

In:

```
# outputs all entities identified in the input text
for ent in doc.ents:
    print(f'{ent.text:<15} {ent.type}')
```

Out:

```
$12.7 billion   MONEY
Apple Card      ORG
Apple Watch     ORG
AirPod          ORG
the quarter     DATE
Apple           ORG
```

```
U.K.              GPE
$1 billion        MONEY
```

Note that the output of the code above is very similar to the output of `spacy`'s NER tool, except for one entity (`spacy` also recognized "Q1" as a cardinal numeral).

10

Measuring Text Similarity

10.1 Comparing Text using Similarity Measures

A common problem in textual analysis is to assess a degree of similarity between two (or more) pieces of text. Several studies in the accounting and finance literatures examine similarity of disclosures. For example, Brown and Tucker (2011) find that firms with larger economic changes have less similar MD&A disclosures across years. Kravet and Muslu (2013) find that greater yearly changes in risk disclosures are associated with greater stock volatility and trading volume around the disclosure filing dates. Hoberg and Phillips (2016) use text similarity of firm product descriptions in 10-K filings to generate sets of industry competitors.

Text similarity can be defined on *lexical* and *semantic* levels. Lexical similarity refers to similarity stemming from usage of common words or characters (e.g., "good" and "goodness" share similar characters). Semantic similarity refers to similarity based on word/phrase meaning similarity (e.g., "good" and "nice" have similar meanings). Measuring semantic similarity is a significantly more difficult problem and most measures of text similarity in the accounting and finance fields are lexical. In this chapter, we will demonstrate both lexical and semantic

similarity measures.

Finally, the length of text is an important factor when considering which similarity metric to use. If an input text is relatively long (five words or more), it is more appropriate to choose a similarity measure that operates on a word level. Conversely, if the text is rather short, it is more appropriate to work with measures that operate on a character level. In this chapter, we demonstrate how to use both long- and short-text similarity measures.

10.2 Text Similarity Measure for Long Text: Cosine Similarity

There are various similarity measures for relatively long pieces of text such as the Euclidean distance, cosine similarity, and the Jaccard similarity index. Most accounting and finance studies use the cosine similarity measure to compare texts. Therefore, in this chapter, we show how to compute textual cosine similarity in Python.

10.2.1 What is Cosine Similarity?

Before we define cosine similarity, we first need to introduce the *bag-of-words* model used to represent text. Under this approach, each piece of text is represented as a vector of words and their counts. For example, phrases "cash flows from operations" and "cash flows from investing" can be represented as vectors u and v, respectively, as follows:

	cash	flows	from	investing	operations
u	1	1	1	0	1
v	1	1	1	1	0

In this example, vector $u = (1, 1, 1, 0, 1)$ is defined on a feature space of all words present in two phrases above, and includes 1 for each word occurrence in phrase "cash flows from operations", and 0 for each word absence. If the word "cash" appeared two times in that phrase (e.g., "cash flows from cash operations"), vector u would include 2 for "cash" word component: $u = (2, 1, 1, 0, 1)$. Vector v represents word counts for "cash flows form investing" and is defined in a similar manner. Note

that the order of words, their part of speech, sentence structure, and other linguistic information is not recorded in bag-of-words vectors.

Cosine similarity between two vectors u and v is defined as the cosine of angle between these two vectors. It can be calculated as follows:

$$\cos(u, v) = \frac{u \cdot v}{|u||v|} = \frac{\sum_{i=1}^{N} u_i v_i}{\sqrt{\left(\sum_{i=1}^{N} u_i^2\right)\left(\sum_{i=1}^{N} v_i^2\right)}},$$

where u_i and v_i are vectors components, and $|u|$ and $|v|$ are lengths of vectors u and v, respectively. Simply put, cosine similarity is a measure of distance between two vectors. Values close to 1 indicate high degree of similarity between two vectors, and values close to 0 indicate little similarity. Since we can represent text as vectors using the bag-of-words approach, we can calculate distance (similarity) between pieces of text.

10.2.2 Representing Text as a Vector in Python

To calculate cosine similarity between several pieces of text in Python, we first need to convert these pieces of text to bag-of-words vectors. This process involves two steps: (1) extracting words from text and (2) representing individual word counts as numerical vectors. We use `NLTK` library to extract words from text and `Scikit-learn` library to convert lists of words to vectors. Both of these libraries are included in `Anaconda` distribution, but can also be installed via `pip` package installer if needed.

Although we could use simple regular expressions to extract words from text, `NTLK` library provides additional text processing capabilities to improve text similarity comparison. When using the bag-of-words approach for cosine similarity, researchers often perform *word stemming* and remove *stop words* (e.g., Lang and Stice-Lawrence, 2015). As already mentioned in Chapter 7, word stemming is a text analysis technique that converts words to their base forms (e.g., "reporting" to "report"). In addition, stop words are common words in language such as "a", "the", "on", "her", etc. Therefore, stop words are often filtered out when using the bag-of-words approach to make the vector representation and comparison more meaningful.

To prepare texts for cosine similarity comparisons, we first write a function that for a given text extracts all words from that text, removes stop words, and performs stemming on all the remaining words. We use `word_tokenize` from `NLTK` library (see Chapter 7) as a starting point to write a custom word tokenizer. To access `NLTK`'s word tokenizer and the list of stop words we need to download two `NLTK` modules as follows (this has to be done only once):

```
import nltk
#download NLTK's stopwords module
nltk.download('stopwords')
#downlod NLTK's punkt module
nltk.download('punkt')
```

`NLTK`'s word tokenizer extracts words from a given text and outputs them as a list of tokens. However, if the input text includes punctuation or apostrophe characters (e.g. ,, !, or ’), `NLTK`'s word tokenizer yields these characters as separate tokens (in addition to words). When calculating text similarity, we should exclude these punctuation character tokens as they introduce noise to bag-of-words vectors. Conveniently, Python includes a list of punctuation characters; we only need to add apostrophe to that list.

In:

```
# Python includes a collection of all punctuation
# characters
from string import punctuation

# add apostrophe to the punctuation character list
punctuation_w_apostrophe = punctuation + "’"

# print all characters
print(punctuation_w_apostrophe)
```

Out:

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~’
```

Now, we can write a custom word tokenizer using `NTLK`'s list of stop words and the Porter stemmer:

```
# imports word tokenizer from NLTK
from nltk import word_tokenize
# imports list of stop words from NLTK
from nltk.corpus import stopwords
```

```
# imports Porter Stemmer module from NLTK
from nltk.stem import PorterStemmer

# creates a list of English stop words
set_stopwords = set(stopwords.words('english'))
# creates a Porter stemmer object
stemmer = PorterStemmer()

# creates a custom tokenizer that removes stop words,
# punctuation, and stems the remaining words
def custom_tokenizer(text:str):
    # gets all tokens (words) from the lower-cased
    # input text
    tokens = word_tokenize(text.lower())
    # filters out stop words
    no_sw_tokens = [t for t in tokens
                    if t not in set_stopwords]
    # filters out punctuation character tokens
    no_sw_punct_tokens = [t for t in no_sw_tokens
                          if t not in
                          punctuation_w_apostrophe]
    # stems the remaining words
    stem_tokens = [stemmer.stem(t) for t in
                   no_sw_punct_tokens]
    # returns stemmed tokens (words)
    return stem_tokens
```

Let us demonstrate how this tokenizer works using text excerpts from business description sections of 10-K filings of three telecommunication companies:

In:

```
# excerpt from Verizon Communications Inc. 2018 10-K
doc_verizon = """Verizon Communications Inc. (Verizon
    or the Company) is a holding company that, acting
    through its subsidiaries, is one of the world's
    leading providers of communications, information
    and entertainment products and services to
    consumers, businesses and governmental agencies."""
# excerpt from AT&T Inc. 2018 10-K
doc_att = """We are a leading provider of
    communications and digital entertainment services
    in the United States and the world. We offer our
    services and products to consumers in the U.S.,
    Mexico and Latin America and to businesses and
    other providers of telecommunications services
```

```
    worldwide."""
# excerpt from Sprint Corporation 2018 10-K
doc_sprint = """Sprint Corporation, including its
    consolidated subsidiaries, is a communications
    company offering a comprehensive range of wireless
    and wireline communications products and services
    that are designed to meet the needs of individual
    consumers, businesses, government subscribers and
    resellers."""

tokens_verizon = custom_tokenizer(doc_verizon)
print(tokens_verizon)

tokens_att = custom_tokenizer(doc_att)
print(tokens_att)

tokens_sprint= custom_tokenizer(doc_sprint)
print(tokens_sprint)
```

Out:

```
['verizon', 'commun', 'inc.', 'verizon', 'compani', '
    hold', 'compani', 'act', 'subsidiari', 'one', '
    world', 'lead', 'provid', 'commun', 'inform', '
    entertain', 'product', 'servic', 'consum', 'busi',
    'government', 'agenc']
['lead', 'provid', 'commun', 'digit', 'entertain', '
    servic', 'unit', 'state', 'world', 'offer', 'servic
    ', 'product', 'consum', 'u.s.', 'mexico', 'latin',
    'america', 'busi', 'provid', 'telecommun', 'servic'
    , 'worldwid']
['sprint', 'corpor', 'includ', 'consolid', 'subsidiari'
    , 'commun', 'compani', 'offer', 'comprehens', 'rang
    ', 'wireless', 'wirelin', 'commun', 'product', '
    servic', 'design', 'meet', 'need', 'individu', '
    consum', 'busi', 'govern', 'subscrib', 'resel']
```

Note that words in the output lists are stemmed and do not include stop words. Finally, we can use `Scikit-learn`'s `CountVectorizer` class to convert text documents to bag-of-words vectors:

In:

```
# CountVectorizer converts text to bag-of-words vectors
from sklearn.feature_extraction.text import
    CountVectorizer

# creates a list of three documents; one for each
# company
```

```
documents = [doc_verizon,doc_att,doc_sprint]

# creates a CountVectorizer object with the custom
# tokenizer
count_vectorizer = CountVectorizer(tokenizer=
    custom_tokenizer)

# converts text documents to bag-of-word vectors
count_vecs = count_vectorizer.fit_transform(documents)

# prints first ten bag-of-words features (words)
print(count_vectorizer.get_feature_names()[:10])

# prints first ten bag-of-words elements (counts) for
# each vector the output is a matrix where each row
# represents a document vector the element (count)
# order in each vector corresponds to the order of
# the bag-of-word features
print(count_vecs.toarray()[:,:10])
```

Out:

```
['act', 'agenc', 'america', 'busi', 'commun', 'compani'
    , 'comprehens', 'consolid', 'consum', 'corpor']
[[1 1 0 1 2 2 0 0 1 0]
 [0 0 1 1 1 0 0 0 1 0]
 [0 0 0 1 2 1 1 1 1 1]]
```

In the above code, we created a new object `count_vectorizer` of `CountVectorizer` class that will use our custom tokenizer (function `custom_tokenizer`) to extract and process words from text documents. We then used `count_vectorizer` to convert three text documents to bag-of-words numerical vectors. Note that `CountVectorizer` sorts all (stemmed) words alphabetically and then returns vectors of word counts for input text documents.

10.2.3 Calculating Cosine Similarity

Once we have bag-of-words vectors, it is relatively easy to calculate cosine similarity measure using `Scikit-learn` package:

In:

```
# cosine_similarity calculates cosine similarity
# between vectors
from sklearn.metrics.pairwise import cosine_similarity
```

```
# calculates text cosine similarity and stores results
# in a matrix. The matrix stores pairwise similarity
# scores for all documents, similarly to a covariance
# matrix
cosine_sim_matrix = cosine_similarity(count_vecs)

# outputs the similarity matrix
print(cosine_sim_matrix)
```

Out:

```
[[1.         0.44854261 0.40768712]
 [0.44854261 1.         0.32225169]
 [0.40768712 0.32225169 1.        ]]
```

According to the output above, text excerpts of Verizon and AT&T have similarity score of 0.4485 (most similar), Verizon and Sprint have 0.4077 similarity score, and AT&T and Sprint have 0.3226 score (least similar).

In addition to removing stop words, to further minimize the impact of common words when measuring text similarity, we can use term frequency – inverse document frequency (TF-IDF) technique. As discussed in Chapter 7, inverse document frequency (IDF) is a word weighting technique where commonly-used words (across the entire document corpus) are assigned smaller word weights. TF-IDF is simply a product of term (word) frequency (TF) and its inverse document frequency (IDF) weight.

We can slightly modify our previous code to create bag-of-words vectors with IDF weights by using `TfidfVectorizer` class instead of `CountVectorizer`. The former class automatically calculates and applies IDF weights for each documents in the list (corpus) of documents.

In:

```
# TfidfVectorizer converts text to TF-IDF bag-of-words
# vectors
from sklearn.feature_extraction.text import
    TfidfVectorizer

# creates a TfidfVectorizer object with the custom
# tokenizer
tfidf_vectorizer = TfidfVectorizer(tokenizer=
    custom_tokenizer)

# converts text documents to TF-IDF vectors
```

```
tfidf_vecs = tfidf_vectorizer.fit_transform(documents)

# prints first four bag-of-words features (words)
print(tfidf_vectorizer.get_feature_names()[:4])

# prints first four bag-of-words TF-IDF counts for each
# vector. The output is a matrix where each row
# represents a document vector
print(tfidf_vecs.toarray()[:,:4]) # prints first four
    elements of each vector
```

Out:

```
['act', 'agenc', 'america', 'busi']
[[0.22943859 0.22943859 0.         0.13551013]
 [0.         0.         0.23464902 0.13858749]
 [0.         0.         0.         0.13365976]]
```

Notice that the bag-of-words vectors in the output are comprised of real (continuous) numbers as opposed to integers. Finally, to compute the cosine similarity between TF-IDF vectors, we can use `NTLK`'s `cosine_similarity` function (as we did previously):

In:

```
# computes the cosine similarity matrix for TF-IDF
# vectors
tfidf_cosine_sim_matrix = cosine_similarity(tfidf_vecs)
# outputs the similarity matrix
print(tfidf_cosine_sim_matrix)
```

Out:

```
[[1.         0.30593809 0.23499515]
 [0.30593809 1.         0.17890296]
 [0.23499515 0.17890296 1.        ]]
```

The similarity scores calculated based on TF-IDF vectors are smaller as compared to regular bag-of-words vectors. This is likely the result of words unique to each text document (such as company names) having greater inverse document frequency weights.

10.3 Text Similarity Measure for Short Text: Levenshtein Distance

It is often useful to assess similarity between pieces of short texts. For example, we may have to merge observations in two data sets that come from different sources based on company or person names. In

such instances, a short-text similarity measure is more applicable than measures like cosine similarity. Unlike long-text similarity measures that define similarity on a word level, short-text similarity measures, such as longest common substring, the Levenshtein distance, and the Jaro distance, define text similarity on a character level. As an example of short-text similarity coding, we show how to compute a popular short-text similarity measure, the Levenshtein distance.

10.3.1 Introducing the Levenshtein Distance

The Levenshtein distance (sometimes referred to as edit distance) between two pieces of text, t_1 an t_2, calculates the smallest number of single-character edits (insertions, deletions, and substitutions) needed to make t_1 identical to t_2. For example, the Levenshtein distance between words "account" and "accounts" is one since we need one edit to change "account" to "accounts": insert **s** at the end of the word. The Levenshtein distance between "account" and "count" is two: two characters **ac** have to be removed from "account" to become "count". Finally, the Levenshtein distance between "account" and "access" is four: characters **out** in "account" have to be substituted to **ess**, and the last character **t** has to be removed.

`NLTK` library provides a function called `edit_distance` that calculates the Levenshtein distance between two pieces of text:

In:

```
# edit_distance computes Levenshtein distance between
# two pieces of text
from nltk import edit_distance

#example: account and accounts
print(edit_distance("account","accounts"))

#example: account and count
print(edit_distance("account","count"))

#example: account and access
print(edit_distance("account","access"))
```

Out:

```
1
2
```

```
4
```

10.3.2 Creating a Similarity Measure using the Levenshtein Distance

There is one important difference between the outputs of cosine similarity and the Levenshtein distance measures. Cosine similarity outputs a real number between 0 and 1 where higher values indicate more similar text; the Levenshtein distance outputs an integer (0, 1, 2, etc.) where higher values indicate more dissimilar text. Also, the Levenshtein distance is likely to mechanically increase with the length of the input text as lengthier text may need more edits to become equivalent to some other text.

We can define a similarity metric based on the Levenshtein distance that will behave similarly to the cosine similarity, i.e., values ranging from 0 to 1 with higher values indicating higher similarity. To do so, we first scale the Levenshtein distance by the length of the longer input text. After that we subtract the scaled distance from one to get a number between 0 and 1 where greater values indicate greater degree of similarity:

```
# similarity measure based on the Levenshtein distance
# greater values indicate more similar text
def edit_similarity(t1,t2):
    # lowercase the input strings
    (t1,t2) = (t1.lower(),t2.lower())
    # calculates the Levenshtein distance between the
    # input strings
    distance = edit_distance(t1,t2)
    # calculates length of the longest input string
    longest_text_len = max(len(t1),len(t2))
    # if both t1 and t2 are empty strings, they are
    # identical; thus return 1 as the output
    if longest_text_len == 0:
        return 1.0
    # else compute the similarity measure as
    # 1 - (levenshtein_distance / length of the longest
    input string)
    else:
        return (1.0 - float(distance) / float(
    longest_text_len))
```

Let us demonstrate how to apply this similarity measure on an example. Consider the problem of matching observations based on company names. The name of an S&P 500 firm, Fidelity National Information Services, is recorded in Capital IQ's Compustat database as "Fidelity National Info Svcs". That is, the last two words are shortened. Ideally, a similarity measure will yield a high degree of similarity between the shortened and original versions of a company name.

In:

```
# original company name
orig_name = "Fidelity National Information Services"
# shortened company name
comp_name = "Fidelity National Info Svcs"

# calculates and outputs the Levenshtein distance
levenshtein_distance = edit_distance(orig_name,
    comp_name)
print("Levenshtein distance:",levenshtein_distance)

# calculates and output the similarity score based on
# Levenshtein distance
levenshtein_similarity = edit_similarity(orig_name,
    comp_name)
print("Levenshtein similarity score:",
    levenshtein_similarity)
```

Out:

```
Levenshtein distance: 11
Levenshtein similarity score: 0.7105263157894737
```

The similarity score between the two versions of the company name is 0.71 indicating high level of similarity.

10.4 Measuring Semantic Similarity using Word2Vec Embedding Model

Word embedding refers to the representation of words as vectors of numbers. In Section 10.2.1, we introduced the simplest word embedding approach called the *bag-of-words* (BOW), where textual documents are represented by the frequencies of individual words they contain. One of the main advantages of BOW is simplicity. Yet, one of the main disadvantages is the loss of contextual information and the order in which individual words appear in sentences.

Word2Vec embedding approach developed by Mikolov *et al.* (2013a) and Mikolov *et al.* (2013b) is considered the state-of-the-art approach of representing texts numerically. It uses a neural network model to convert words into vectors in such a way that the meaning of the word is inferred by its context, i.e., by the words that occur in close proximity to the word of interest. In other words, if we have two words that are used in the same context, then these words are likely similar in meaning or related. For example, *pleased*, *glad*, and *happy* are often used in similar contexts. Word2Vec model has gained enormous popularity among researchers and some practical implications of the model include word clustering and semantic similarity, synonym detection, content categorization and recommendation.

In this section, we show basic examples of Word2Vec model in Python. To learn more about Word2Vec and its applications, we refer our readers to two papers that developed the model, namely Mikolov *et al.* (2013a) and Mikolov *et al.* (2013b), and also to numerous online resources on Word2Vec including:

Google Word2Vec
Deep Learning with Word2Vec

Assume we want to create a Word2Vec model for Apple's MD&A section in its 2018 10-K report. Once the MD&A section is extracted from the 10-K filing,[1] we need to preprocess its content for Word2Vec by removing stop words, special characters, numbers, and extra spaces

[1] Chapter 12 discusses how to extract specific sections in 10-K filings.

and by extracting individual words from the input text. The code below summarizes these data preprocessing steps.

In:

```
import re
# imports word tokenizer from NLTK
import nltk
#download NLTK's stopwords module
nltk.download('stopwords')
from nltk import word_tokenize
# imports list of stop words from NLTK
from nltk.corpus import stopwords

# creates a list of English stop words
set_stopwords = set(stopwords.words('english'))

# path to the input txt file with Apple's 2018 MD&A
input_file = r".../Apple_MDNA.txt"

# reads file content
file_content = open(input_file,"r").read()

# converts text to lowercase; removes all special
    characters, digits and extra spaces
processed_content = file_content.lower()
processed_content = re.sub(r'[^a-zA-Z]', ' ',
    processed_content )
processed_content = re.sub(r'\s+', ' ',
    processed_content)

# creates a list of lists of individual words - this is
     the input format to Word2Vec model
processed_content = [processed_content]
words = [nltk.word_tokenize(e) for e in
    processed_content]

# removes stop words from the list of words
for i in range(len(words)):
     words[i] = [w for w in words[i] if w not in
    set_stopwords]
```

Now, after identifying individual words in Apple's MD&A section and preserving their order and context, we can use `Gensim` library to build our Word2Vec model.[2] `Gensim` can be installed using either `conda`

[2]Python's `Gensim` library offers an excellent implementation of Latent Dirichlet

or `pip` as follows:

```
conda install -c anaconda gensim
pip install gensim
```

As noted above, the contextual information of the words in Apple's 2018 MD&A is not lost when using the Word2Vec approach. As such, we can find word clusters within the document. For instance, based on Apple's MD&A wording, we can find similar or related words to the word "sales":

In:

```
# imports Word2Vec from Gensim library
from gensim.models import Word2Vec

# creates a Word2Vec model, ignoring words that occur
    less than two times in the input text
word2vec = Word2Vec(words, min_count=2)

# identifies most related/similar words to 'sales'
    based on the input text provided
related_words = word2vec.wv.most_similar('sales')
related_words
```

Out:

```
[('rate', 0.9937257766723633),
 ('interest', 0.9924227595329285),
 ('assets', 0.992143988609314),
 ('risk', 0.99197918176651),
 ('million', 0.9914211630821228),
 ('capital', 0.9911070466041565),
 ('mortgage', 0.9908864498138428),
 ('securities', 0.9907544255256653),
 ('financial', 0.9906978011131287),
 ('december', 0.9902232885360718)]
```

Using Apple's MD&A as an input to the Word2Vec model, we find words related/similar to "sales" along with their similarity index. For example, in the input MD&A file, words such as "rate" and "interest"

Allocation (LDA) - a popular method for topic modelling from large volumes of text (see Topic Modelling with Gensim Python and Gensim LDA). It is important to note that the extraction of good quality topics greatly depends on the quality of the input texts and their preprocessing as well as the methods used for detecting an optimal number of topics.

often coexist with the word "sales" as indicated by their high similarity scores.

In the example above, we used only one textual document to train the Word2Vec model. However, the performance of the model in identifying word clusters and similarities will greatly improve when we increase the training corpus. A popular option for training Word2Vec is the Google News dataset model. It consists of 300-dimensional embeddings for around three million words and phrases (see https://code.google.com/archive/p/word2vec/ for details and to download 'GoogleNews-vectors-negative300.bin.gz' file (∼1.5GB)). With the pre-trained model we can access the word vectors and get the similarity scores as follows:

In:

```
from gensim.models import KeyedVectors
# load embeddings directly from the downloaded file
    called "GoogleNews-vectors-negative300.bin"
model = KeyedVectors.load_word2vec_format('GoogleNews-
    vectors-negative300.bin', binary=True)
# similarity between pairs of words
a = model.similarity('confident', 'uncertain')
b = model.similarity('recession', 'crisis')
# most similar words
c = model.most_similar('accounting')
# identifies a word that does not belong in the list
d = model.doesnt_match("good great amazing bad".split()
    )
print(a)
print(b)
print(c)
print(d)
```

Out:

```
0.38531393
0.59829676

[('Accounting', 0.6579887866973877),
 ('bookkeeping', 0.6002781391143799),
 ('auditing', 0.5503429174423218),
 ('Arthur_Andersen_Enron', 0.5320826768875122),
 ('restatement', 0.5319857597351074),
 ('accountancy', 0.5315808057785034),
 ('bookeeping', 0.5051406621932983),
 ('Generally_Accepted_Accounting_Principles',
    0.5034366846084595),
```

```
('accouting', 0.5023787021636963)]

bad
```

Using the Word2Vec model pre-trained on Google News, we were able to identify "bookkeeping" and "auditing" as well as "Generally Accepted Accounting Principles" as words and phrases most closely related to the word "accounting." Similarly, Word2Vec is able to detect that the word "bad" does not belong in the list consisting of words "good great amazing bad."

In summary, there are at least three different methods of representing texts numerically: bag-of-words (BOW) approach, weighted (by the inverse document frequency (idf)) BOW, and word embeddings with Word2Vec. Each of the methods offers its advantages and disadvantages (e.g., modelling of the word context vs. loss of the context) when working with texts. Therefore, we suggest our readers to carefully consider their research questions when selecting one method over the other as sometimes an increased complexity of a method may not necessarily lead to better results.

11

Identifying Specific Information in Text

11.1 Text Identification and Extraction Problem

Many studies in the literature examine textual/linguistic properties of specific parts of a text document rather than the whole document. For example, Li (2008), Loughran and McDonald (2011), and Brown and Tucker (2011) consider linguistic characteristics of the management discussion and analysis (MD&A) section of 10-K filings. Similarly, when studying CEO communication in earnings conference calls, Li *et al.* (2014) and Bochkay *et al.* (2019) extract CEO-specific speech (text) from earnings conference call transcripts.

Identifying and extracting specific parts in text can be a challenging task. Although it is relatively easy to extract information from a single document, in most cases we need to automatically extract textual data from a large number of documents (e.g., 10-K filings, proxy statements, transcripts of earnings calls). Often even regulated corporate filings use different text formatting, data labels, writing style and conventions. Consequently, it is important to be able to identify common structure of the information we need to extract across multiple documents.

For example, suppose we need to extract a footnote to financial statements describing accounting policies. Typically, this is the first

footnote that follows financial statements. One way to identify a section of a document (such as a footnote, MD&A, risk factors, etc.) is to search for the section heading (title). However, section headings can be labeled differently across different filers and / or time periods. For instance, Amazon.com, Inc. uses **"Note 1 — DESCRIPTION OF BUSINESS AND ACCOUNTING POLICIES"** as the footnote heading title, while Facebook, Inc. uses **"Note 1. Summary of Significant Accounting Policies"**. In both cases, we observe that phrases "note 1" and "accounting policies" are present in both headings. In addition, both headings use **bold** font. Therefore, to identify accounting policy footnotes in 10-K filings, we can write a *query* in Python that will search for bold headings with these two phrases. We use word *query* here to denote a method (e.g., a Python function) that requests and retrieves (textual) data from a document.

We can also spot differences between the two examples of accounting policy footnote headings above: Amazon uses capital letters and Facebook does not include "description of business" in its heading (even though such information is included in Facebook's footnote). As we consider more examples of accounting policy footnote headings, we will observe more differences. Often times we have to adjust our search query to accommodated these differences. For example, some companies may simply write "1" instead of "Note 1." when referring to the first footnote. The most difficult part of identifying specific parts of text documents is to design queries that best describe commonalities across different documents, while also accurately identifying text of interest. In most cases, this task requires undertaking a trial-and-error approach, where queries are tested on a random subset of documents.

Another important factor to consider is the data format of the document. Textual data is recorded using different formats such as plain text, HyperText Markup Language (HTML), Extensible Markup Language (XML), JavaScript Object Notation (JSON), PDF, etc. Plain text files are the easiest to understand and process in Python. However, they lack certain features such as font and table formatting present in other formats (e.g., HTML). Format of the text and font (e.g., centered text and bold font) can be useful when trying to identify important information or part (e.g., subsections) of the document.

Some formats such as XML and JSON are specifically designed to store machine-readable data. These formats greatly simplify automated data identification and extraction. Later in this chapter, we briefly discuss eXtensible Business Reporting Language (XBRL) – format that leverages XML format to store and communicate business data (including text). It is difficult to work with PDF files directly. Typically, we first convert PDF files to plain text files, and then attempt text extraction on these plain text files.

In this chapter, we discuss and demonstrate how to extract sections of documents for various document formats.

11.2 Example: Extracting Management Discussion & Analysis Section from a plain-text 10-K filing

Plain-text is the simplest format to store textual data; it includes only text characters with no images, special formatting, or meta data. In 1993, when the SEC created Electronic Data Gathering, Analysis, and Retrieval (EDGAR) system to collect, store, and distribute public corporate filings that are required to be filed with the SEC, corporate filings were reported in the plain-text format. This has been the dominant filing format until around 2001 when the HTML format became widely used.[1]

The advantage of the plain-text format is its simplicity and the lack of formatting styles and source code that is not visible to the reader. It is relatively easy to write queries in Python using regular expressions to find patterns of text that are unique to the type of information we attempt to identify. However, the format's simplicity is also its disadvantage: we cannot leverage text style and format to identify relevant text.

Consider an example where we need to identify the MD&A section from a 10-K filing document. A common approach is to identify the document's section heading that denotes the beginning of the MD&A section and the section heading that denotes the beginning of the next

[1] On June 28, 1999 the SEC adopted HTML filing format for 10-K and 10-Q filings. However, it has not been widely adopted by filers until approximately the end of 2001.

section. All the text in-between is the content of the MD&A section:

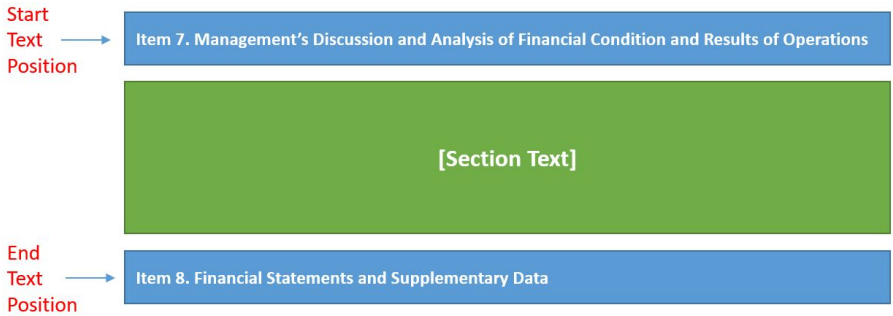


Figure 11.1: Identifying MD&A section in a 10-K document

In other words, our objective is to find the two positions (indexes) in a text file that denote the beginning and the end of the section. The challenge here is that the section heading titles differ across 10-K filings. For example, we may observe the following MD&A section headings:

- Item 7. Management's Discussion and Analysis of Financial Condition and Results;
- Item 7 - Management's Discussion and Analysis of Financial Condition and Results;
- ITEM 7. MANAGEMENT'S DISCUSSION AND ANALYSIS OF FINANCIAL CONDITION AND RESULTS OF OPERATIONS;
- 7. Management's Discussion and Analysis of Financial Condition and Results of Operations.

Let us write a regular expression that identifies a common pattern across these heading captions:

```
import re

# a regex to identify the location of Item 7 (MD&A) heading
# re.DOTALL flag allows . character to match new line
# characters; needed for case when heading titles span
# multiple lines
item7_regex = re.compile(r"(item.{1,5})?\b7\b.{1,5}
    management.{1,5}discussion.{1,5}analysis", re.
    IGNORECASE | re.DOTALL)
```

The regular expression defined in `item7_regex` variable consists of five keywords: `item` (this keyword is optional), `7`, `management`, `discussion`,

and `analysis` that are separated by sub-pattern `.{1,5}`. These keywords are common words across all four examples above. The sub-pattern `.{1,5}` indicates that there may be between one and five (regex `{1,5}`) instances of any character (regex `.`) between these keywords. We allow up to five characters between keywords to accommodate extra spaces, apostrophes or any other characters that separate the keywords. We also create this regular expression with two regex flags, `re.IGNORECASE` and `re.DOTALL`. The former flag enables case-insensitive pattern match, while the latter allows the regex dot character `.` to match new lines since long section headings are sometimes broken into multiple text lines.

The section that follows MD&A is typically called "Financial Statements and Supplementary Data". However, sometimes it is called "Summary of Selected Financial Data". The following regular expression attempts to capture the heading title of either of these subsections. Its pattern is constructed similarly to `item7_regex`.

```
# MD&A is typically followed by Item 8,
# "Financial Statements and Supplementary Data"
# However, sometimes it is followed by "Summary of Selected
# Financial Data" section
item8_regex = re.compile(r"(item.{1,5})?\b8\b.{1,5}(
    financial.{1,5}statements.{1,5}supplement.{1,5}data|
    summary.{1,5}selected.{1,5}financial.{1,5}data)", re.
    IGNORECASE | re.DOTALL)
```

Now we can write a function that extracts the MD&A section for a given 10-K filing text. Please note that it is not unusual for the MD&A section to be located in an exhibit (e.g., Exhibit 13) as opposed to the main 10-K file. Therefore, we recommend to search for the MD&A section in the "complete" submission filing, as opposed to the main 10-K file only. Moreover, it is important to check whether the length of MD&A section is sufficiently long as the MD&A section can be incorporated by reference in the main document.

```
def extract_mdna(plain_text:str, min_mdna_length = 500):
    """Attempts to extract MD&A section from a plain-text
       document. The extracted MD&A section should be of
       the minimum specified length (500 characters by
       default)."""
    # tries to find position of Item 7 heading
```

```
    section_start_match = item7_regex.search(plain_text)
    # if successful, continue
    if section_start_match:
        # save the text position of Item 7 heading to
        # a variable. Method start() returns the position
        # of the regex match in the text
        section_start_pos = section_start_match.start()

        # find the position of Item 8 heading;
        # starts search after Item 7 heading
        # position.

        # set temporarily section_end_pos to be equal to
        # section_start_pos
        section_end_pos = section_start_pos
        # try to find Item 7 heading that is at
        # least min_mdna_length characters further
        # from the start postion
        while (section_end_pos < section_start_pos +
min_mdna_length) and section_end_pos != -1:
            section_end_match = item8_regex.search(
plain_text,section_end_pos + 1)
            # if successful, set section_end_pos to
            # the position of the current match
            if section_end_match:
                section_end_pos = section_end_match.start()
            # if not successful, set section_end_pos
            # to -1 to indicate no regex match and
            # terminate the loop
            else:
                section_end_pos = -1

        # if success, extract the MD&A section
        if section_end_pos > 0:
            # text[a:b] allows to extract text (substring)
            # between a and b positions
            item7_text = plain_text[section_start_pos:
section_end_pos]
            # returns the content of the MD&A section
            return item7_text
    # if neither Item 7 nor Item 8 heading was
    # identified, returns None
    # note that the function will only reach the following
line of
    # code if the heading search failed
    return None
```

Function `extract_mdna` attempts to locate positions of Item 7 (MD&A) and Item 8 sections in text of an input 10-K filing text. If successful, it returns all text in-between. If unsuccessful, the function returns `None`. To apply this function to a 10-K filing, we first use Python's `requests` library to download a "complete" plain-text 10-K filing from the SEC's EDGAR system and save its text contents to a variable called `text_complete_10k`.

In:

```
# requests is a built-in Python library for
# HTTP (web) requests
import requests

# PepsiCo's 1997 10-K filings files are accessible
# through the URL link below:
# https://www.sec.gov/Archives/edgar/data
    /77476/0000077476-98-000014-index.html
# generates an HTTP request to download
# PepsiCo's 1997 10-K filing
response = requests.get('https://www.sec.gov/Archives/
    edgar/data/77476/0000077476-98-000014.txt')

# saves the response (filing) text to a variable
text_complete_10k = response.text

# checks if the 10-K file was downloaded correctly
# prints 300 characters of the text starting at
# 10000 character position
print(text_complete_10k[10000:10300])
```

Out:

```
 foods have been introduced to international  markets.
Principal  international  markets include Brazil,
    France,  Mexico,  Poland, the
Netherlands, South Africa, Spain and the United Kingdom
    .

COMPETITION

      Both of PepsiCo's  businesses are highly
    competitive.  PepsiCo's beverages
and snack fo
```

Next, applying `extract_mdna` function to the downloaded 10-K text yields:

In:

```
# extracts the MD&A section from the PepsiCo's 10-K
# filing text
text_mdna_only = extract_mdna(text_complete_10k)

# checks if the MD&A section extraction was successful
# prints the first 200 characters of the MD&A section
print(text_mdna_only[:200])
print('\n[...]\n')
# prints the last 200 characters of the MD&A section
print(text_mdna_only[-200:])
```

Out:

```
Item 7. Management's Discussion and Analysis of Results
     of Operations, Cash
Flows and Liquidity and Capital Resources

Management's Discussion and Analysis

All per share information is computed using

[...]

pital markets throughout the world.

ITEM 7A. QUANTITATIVE AND QUALITATIVE DISCLOSURES ABOUT
     MARKET RISK.

Included in Item 7, Management's Discussion and
    Analysis - Market Risk beginning
on page 9.
```

11.3 Example: Extracting Management Discussion & Analysis Section From an HTML 10-K filing

HTML (HyperText Markup Language) documents are web documents with (hyper)links, images, tables, special text formatting (e.g., bold and italics font styles), and other features. Most corporate financial reports filed with the SEC from 2001 until present are prepared using the HTML format. HTML documents are different from plain-text documents as they include HTML code that describes how to render text, tables, images, and other document objects.

We can employ several techniques to find and extract sections from HTML documents:

1. Search for section heading titles that are formatted using HTML styles commonly used for headings (e.g., text that is centered, bold, emphasized, etc.). Different filers use different HTML styles to render section headings in their filings.
2. Convert HTML to a different format such as Markdown or plain-text. Section extraction from the latter formats is typically easier, but some helpful data can be lost during the conversion process (e.g., bold or centered section headings).
3. Use embedded section hyperlinks from the document's table of contents that pinpoint to the locations of sections in the document (see example in Figure 11.2). This is a very reliable extraction method. However, not all HTML financial reports in EDGAR include the table of contents with hyperlinks.

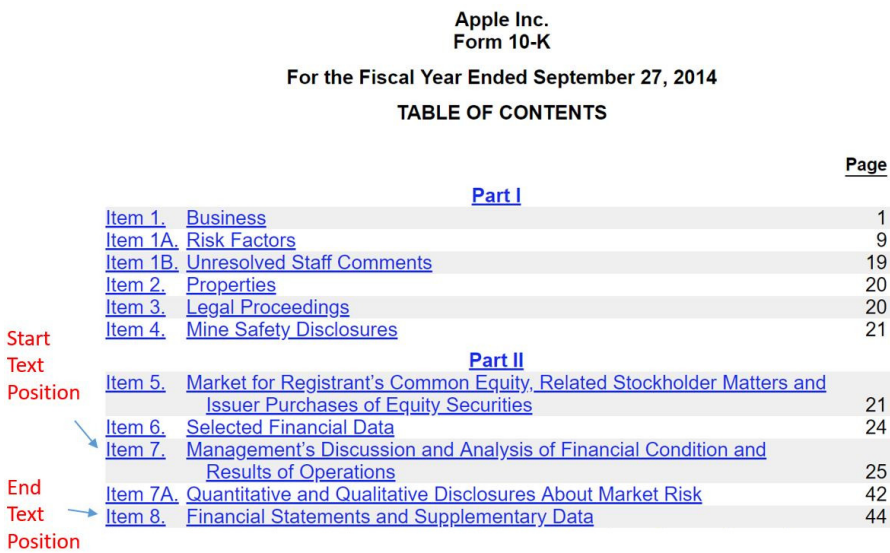


Figure 11.2: Example: Identifying the MD&A section in an HTML document using hyperlinks in table of contents.

In this section, we demonstrate how to implement the first approach in Python using a simple example.

11.3.1 HTML and Section Headings

First, we briefly discuss how HTML documents are created. An HTML document comprises of a set of *elements* that instruct a web browser how to render (output to the end-user) the document. Elements are represented using *tags* embedded in angle brackets, `<` and `>`. For example, consider the following element:

This is a paragraph

This element is used to render a text paragraph. It is represented by the tag `p`, where `<p>` and `</p>` indicate the beginning and the end of the tag's definition. The text between `<p>` and `</p>` is the *value* of the paragraph element, and is simply the paragraph's textual content.

Now, consider an example of the MD&A section heading title from Morgan Stanley's 2008 10-K filing displayed in Figure 11.3. The HTML code to represent this heading is the following:

Item 7. Management's Discussion and Analysis of Financial Condition and Results of Operations.

Introduction.

Morgan Stanley (the "Company") is a global financial services firm that maintains significant market positions in each of its business segments—Institutional Securities, Global Wealth

Figure 11.3: MD&A section heading in Morgan Stanley's 2008 10-K

Management's Discussion and Analysis of Financial Condition and Results of Operations.

Tag `<b>` indicates that the text of the heading should be rendered using **bold** font. Bold text is often used to represent section headings or other important text in corporate reports. Note that HTML documents can render bold text without using `<b>` tag. Consider the following example of an HTML element that renders bold MD&A section heading in Boeing's 2015 10-K filing:

Item 7. Management's Discussion and Analysis of Financial Condition and Results of Operations

In this example, inside the `font` tag there is a keyword `style`; `style` is an *attribute* of tag `font`. Attributes are used to define various properties of element tags. In this case, tag `font` is used to specify how to display the section's heading font, and its attribute `style` describes the properties of the font. Specifically, it states that the font used to display the heading should be of Arial family (`font-family:Arial`), its default size should be 10 points (`font-size:10pt`), and it should be rendered using bold font (`font-weight:bold`). That is, the property `font-weight:bold` here achieves the same effect as `<b>` tag in the previous example.

We can also observe that between words "Item" and "7" there is HTML code: " ". This code instructs a web browser to render in its place a non-breaking space character, a type of space that does not allow line breaks before or after its position.

11.3.2 Writing Code to Identify Section Titles in HTML Documents

Now that we know how HTML documents render bold font text, we can write a Python function that will search for section headings that are displayed in bold font. In fact, HTML documents render text that is centered, underlined, or otherwise emphasized in a similar manner they render bold text. For example, similarly to how the tag `<b>` and property `font-weight:bold` render **bold** text, the tag `<u>` and property `text-decoration:underline` render <u>underlined</u> text. Therefore, we can write a function that will search for section captions that are displayed using font styles commonly used for section headings. Consider the code below:

```
import re

# list of regex patterns that identify HTML text styles
# commonly used to display section headings
html_styles = [
          # b tag; bold text
          r"<b>(?P<value>.+?)</b>",
          # u tag; underlined text
          r"<u>(?P<value>.+?)</u>",
          # strong tag; important text
```

```
            r"<strong[^>]*>(?P<value>.+?)</strong>",
            # center tag; centered text
            r"<center[^>]*>(?P<value>.+?)</center>",
            # any tag that has an attribute ("style") with
            # 'font-weight: bold' value
            r"<(?P<tag>[\w-]+)\b[^>]*font-weight:\s*bold
    [^>]*>(?P<value>.+?)</(?P=tag)>",
            # any tag that has an attribute ("style") with
            # 'text-decoration: underline' value
            r"<(?P<tag>[\w-]+)\b[^>]*text-decoration:\s*
    underline[^>]*>(?P<value>.+?)</(?P=tag)>",
            # em tag; emphasized text
            r"<em>(?P<value>.+?)</em>"]

# function that for a given regex HTML style pattern and
# HTML source (document) returns all the (text) values of
# HTML elements that match that HTML style along with their
# positions (indexes) in the document's HTML source code
def get_html_style_values(html_style:str, html_source:str):
    # creates a regular expression from the input
    # HTML style pattern
    html_style_regex = re.compile(html_style, re.IGNORECASE
     | re.DOTALL)
    # finds all the matches for the above regular
    # expression in the HTML document
    style_matches = html_style_regex.finditer(html_source)
    # creates a dictionary list to store the value (text)
    # of all regex matches and their positions
    results = [{'text':m['value'],'position':m.start()} for
     m in style_matches]
    # outputs results
    return results
```

In the code above, `html_styles` includes regular expression patterns for HTML font styles that are likely to be used to display section headings. Function `get_html_style_values` outputs all matching text and its position for a given HTML font style. To demonstrate how to apply this code, we first download a sample HTML 10-K filing using `requests` library:

In:

```
import requests

# Boeing's 2015 10-K files are accessible through the
# URL link below:
```

```
# https://www.sec.gov/Archives/edgar/data
    /12927/000001292716000099/0000012927-16-000099-
    index.htm
# generates a HTTP request to download Boeing's
# 2015 HTML 10-K filing:
response = requests.get('https://www.sec.gov/Archives/
    edgar/data/12927/000001292716000099/a201512dec3110k
    .htm')

# saves the response (filing) source HTML to a variable
html_complete_10k = response.text

# checks if the 10-K file was downloaded correctly
# prints 300 characters of the HTML source code
# starting at 10000 character position
# the output should look like HTML source code
print(html_complete_10k[10000:10300])
```

Out:

```
dding-bottom:2px;padding-right:2px;"><div style="text-
    align:center;font-size:7pt;"><font style="font-
    family:Arial;font-size:7pt;font-weight:bold;">(Zip
    Code)</font></div></td></tr></table></div></div><
    div style="line-height:120%;padding-top:2px;text-
    align:center;font-size:9pt;"><font style="font-fam
```

Now, we search for all centered text in the HTML 10-K filing that is rendered using the `font-weight:bold` attribute, and print the first three instances of such text:

In:

```
# gets all text from the HTML 10-K filing defined using
# font-weight:bold attribute
style_values = get_html_style_values(html_styles[4],
    html_complete_10k)

# displays the first three instances of such text
for i in range(3):
    print(style_values[i])
```

Out:

```
{'text': 'UNITED STATES', 'position': 753}
{'text': 'SECURITIES AND EXCHANGE COMMISSION', '
    position': 906}
{'text': 'Washington, D.C. 20549', 'position': 1080}
```

Finally, we can write function `extract_mdna` that for a given HTML 10-K document attempts to extract its MD&A section:

```
# a regex to identify the location of Item 7 (MD&A) heading
item7_regex = re.compile(r"management.{1,20}discussion
    .{1,20}analysis", re.IGNORECASE | re.DOTALL)

# MD&A is typically followed by Item 8 heading
item8_regex = re.compile(r"financial.{1,20}statements
    .{1,20}supplement.{1,20}data|summary.{1,20}selected
    .{1,20}financial.{1,20}data", re.IGNORECASE | re.DOTALL
    )

def extract_mdna(html_source:str):
    """Extracts the MD&A section from an HTML filing"""
    # iterates over all possible HTML styles for headings
    # until we can identify MD&A section or until we
    # exhaust all the styles and fail to identify the
    # MD&A section
    for style in html_styles:
        # gets all text in the input HTML document that
        # matches the current style
        style_values = get_html_style_values(style,
                                             html_source)
        # attempts to identify the heading of Item 7
        # (MD&A) section
        section_start = next((v for v in style_values
                              if item7_regex.search(v['text
    '])),None)
        # if Item 7 heading position is identified,
        # proceeds with identifying Item 8 section
        if section_start:
            # Item 8 heading location should be after Item
    7
            section_end = next((v for v in style_values
                                if item8_regex.search(v['
    text'])
                                and v['position'] >
    section_start['position']), None)
            # if Item 8 heading position is identified,
    extracts the
            # HTML code of the MD&A section
            if section_end:
                item7_html = html_source[section_start['
    position']:section_end['position']]
                # outputs the HTML code of the MD&A section
                return item7_html
```

```
    # note that the function will reach the following line
    # of code only if the heading search failed
    return None
```

Function `extract_mdna` first identifies all text that matches a given HTML font style commonly used to display headings. It then searches for Item 7 and Item 8 section titles within this text. If successful, `extract_mdna` returns the content of Item 7 section. If not, `extract_mdna` repeats the process with a different HTML style. If all HTML styles have been attempted and MD&A section extraction is unsuccessful, `extract_mdna` returns `None`.

Applying `extract_mdna` to the Boeing's 10-K HTML file yields:

In:

```
html_mdna_only = extract_mdna(html_complete_10k)

# checks if the MD&A section extraction was successful
# prints the first 200 characters of the MD&A section
print(html_mdna_only[:200])
```

Out:

```
<font style="font-family:Arial;font-size:10pt;font-
    weight:bold;">Item 7. Management&#8217;s
    Discussion and Analysis of Financial Condition and
    Results of Operations</font></div><a name="
    sAE854BB7
```

Notice that although the extraction was successful, the output is the HTML source code of the MD&A section. We can convert this HTML code to plain-text using a popular Python library `lxml`, which is used to process XML and HTML files. This library is included by default in Anaconda, but can be installed via `conda` or `pip` as follows:

conda install -c anaconda lxml
pip install lxml

Below we include Python code of a useful function that converts HTML code to plain text while removing all the tables.

```
# lxml is a library that parses XML and HTML source code
import lxml.html

# converts HTML code to plain text
```

```
def get_text_from_html(html:str):
    # creates an lxml document object
    doc = lxml.html.fromstring(html)
    # optional: removes tables from the HTML source code
    for table in doc.xpath('.//table'):
        table.getparent().remove(table)
    # preserves line breaks
    # HTML tags in the list below should be followed by
    # new line character
    for tag in ["a", "p", "div", "br", "h1", "h2", "h3", "
    h4", "h5"]:
        # finds all elements for a given tag
        for element in doc.findall(tag):
            # if the text value is non-empty adds a
            # new line character (line break)
            if element.text:
                element.text = element.text + "\n"
            # else creates a text value with a
            # new line character
            else:
                element.text = "\n"
    # extracts and output text from the HTML source code
    return doc.text_content()
```

Finally, applying function `get_text_from_html` to the HTML code of the MD&A section we extracted:

In:

```
# extracts text from the HTML MD&A
mdna_text = get_text_from_html(html_mdna_only)
# prints the first 300 characters of the MD&A section
print(mdna_text[:300])
```

Out:

```
Item 7. Management's Discussion and Analysis of
    Financial Condition and Results of Operations

Consolidated Results of Operations and Financial
    Condition
Overview
We are a global market leader in design, development,
    manufacture, sale, service and support of
    commercial jetliners, military aircraft,
```

Using HTML font styles to identify text that is emphasized in some manner is a very useful technique. In fact, a recent study by Bentley

et al. (2019) leverages HTML tags to examine managers' emphasis techniques in earnings announcements.

11.4 Extracting text from XBRL financial reports

XBRL is a business information reporting technology that uses XML syntax to capture business data. XBRL documents are specifically designed to be machine-readable and facilitate automated data exchange. The SEC has mandated public companies to use XBRL reporting to file their 10-K and 10-Q filings. The largest filers (filers with public equity float greater than $5 billion) had to file their annual and quarterly reports using XBRL for filings with fiscal years that ended on or after June 15th, 2009. All firms (with a few exceptions) had to file their annual and quarterly reports with fiscal years that ended on or after June 15th, 2011, using XBRL.

In XBRL, information is communicated via *facts*. A fact consists of a *concept*, *value* and *dimensions*. For example, Home Depot uses the following XBRL fact to report its Total Assets in its 2013 10-K XBRL filing:

<us-gaap:Assets contextRef="FI2013Q4" decimals="-6"
id="Fact-D2708B01E0C51B648AFC01BD72BAF58C"
unitRef="usd">40518000000</us-gaap:Assets>

In the above example `us-gaap:Assets` is a concept with a value of 40,518 million. Also, this is the balance as of the end of fourth quarter of 2013 (`contextRef="FI2013Q4"`) and the amount is reported in U.S. dollars (`unitRef="usd"`). The syntax of the XBRL fact above is somewhat similar to the syntax of HTML elements we discussed in the previous section. In fact, XBRL concepts are often referred to as "tags".

Most XBRL concepts used for financial reporting are defined in the FASB's U.S. GAAP Financial Reporting *taxonomy*. In this taxonomy, FASB provides a list of concepts along with their descriptions and references to accounting standards. Filers have to conform to the taxonomy when creating their annual and quarterly reports using XBRL.

Nevertheless, filers are allowed to create their own XBRL concepts if such concepts are unavailable in the taxonomy. More information about the U.S. GAAP Financial Reporting taxonomy is available at:

https://xbrl.us/home/learn/us-taxonomies/,
http://xbrlview.fasb.org.

The SEC requires public firms to report all information in financial statements and footnotes using the XBRL format. This requirement applies to both textual and numerical data. While MD&A and risk factors sections are not part of financial statement footnotes, other important sections such as firm accounting policies, goodwill and intangible assets, income tax footnote are, and have to be reported using XBRL. Below we demonstrate how to extract a footnote section from an XBRL 10-K filing.

XBRL filings comprise of several files. For the purpose of text data extraction, we focus on the *instance* file where all the XBRL data facts must be reported. We first download a sample XBRL 10-K instance file in Python and save its content to a variable as follows:

In:

```
import requests

# Home Depot's 2013 10-K files are accessible through
# the URL link below:
# https://www.sec.gov/Archives/edgar/data
    /354950/000035495014000008/0000354950-14-000008-
    index.htm
# generates an HTTP request to download Home Depot's
# 2013 XBRL 10-K instance file
response = requests.get('https://www.sec.gov/Archives/
    edgar/data/354950/000035495014000008/hd-20140202.
    xml')

# saves the response (filing content) a variable
xbrl_10k = response.text

# checks if the 10-K file was downloaded correctly
# prints the first 400 characters of the XBRL 10-K
# instance document
print(xbrl_10k[:400])
```

Out:

```
<?xml version="1.0" encoding="US-ASCII"?>
<!--XBRL Document Created with WebFilings-->
<!--p:ee88abc9bc7d42cca8267c157db83ce8,x:1
    ee976a4cce447438165efc656d0aac8-->
<xbrli:xbrl xmlns:country="http://xbrl.sec.gov/country
    /2013-01-31" xmlns:dei="http://xbrl.sec.gov/dei
    /2013-01-31" xmlns:hd="http://www.homedepot.com
    /20140202" xmlns:invest="http://xbrl.sec.gov/invest
    /2013-01-31" xmlns:iso4217="http://ww
```

To extract a specific footnote from XBRL 10-K instance file, we need to know the name of the U.S. GAAP XBRL concept that is used to report that footnote. The concept name is defined in the U.S. GAAP Financial Reporting taxonomy. We can use a tool like XBRL viewer at http://xbrlview.fasb.org to search the taxonomy for XBRL concepts. For example, income tax footnote is typically reported using `us-gaap:IncomeTaxDisclosureTextBlock` concept.

We can now write a regular expression that will search and return the value (text) of `us-gaap:IncomeTaxDisclosureTextBlock` concept in an XBRL 10-K instance file. Once the footnote text is extracted, it has to be converted from the HTML (the default format of textual values in XBRL) to the plain-text format. For example,:

In:

```
import re

# html.unescape is a built-in Python function to decode
    HTML characters
from html import unescape

# regular expression that captures content of the
    income tax footnote
tax_footnote_regex = re.compile(r"<us-gaap:
    IncomeTaxDisclosureTextBlock[^>]*>(?P<value>.+?)</
    us-gaap:IncomeTaxDisclosureTextBlock>", re.
    IGNORECASE | re.DOTALL)

# XBRL documents report text values in HTML format.
    However, the HTML characters have to be decoded
    first.
tax_footnote_html = unescape(tax_footnote_regex.search(
    xbrl_10k)['value'])
```

```
# converts HTML income tax footnote to the plain-text
    format
tax_footnote_text = get_text_from_html(
    tax_footnote_html)

# outputs the first 300 characters
print(tax_footnote_text[:300])
```

Out:

```
INCOME TAXES
The components of Earnings before Provision for Income
    Taxes for fiscal 2013, 2012 and 2011 were as
    follows (amounts in millions):

The Provision for Income Taxes consisted of the
    following (amounts in millions):

The Company's combined federal, state and foreign
    effective tax rat
```

In similar manner, we can extract other financial statement footnotes from XBRL 10-K filings. Also note that the company CalcBench has partnered with the SEC to extract financial statement information from XBRL filings. Researchers who purchase a subscription to CalcBench can largely circumvent the extraction process described in this chapter.

12

Collecting Data from the Internet

12.1 Accessing Data on the Web

There is a massive amount of data out on the web that is potentially useful for accounting researchers. In this chapter we will apply the skills you developed in previous chapters to collect and organize data from the SEC's EDGAR site, as well as to do some basic web scraping.

12.2 EDGAR Data

The SEC's EDGAR site has served as a treasure trove of data for accounting researchers for over 15 years. One of the first studies to extract and analyze this data with a scripting language (as opposed to hand-collecting) was Butler *et al.* (2004). In this study, the authors extracted audit opinions from 10-Ks and classified modified audit opinions by type (e.g., going concern opinion). Later studies, such as Li (2008) began to apply methods from computer science (e.g., computational linguistics and machine learning) to analyze the readability and information content of various disclosures. In this section we cover the steps to obtain data directly from the EDGAR system.

12.2.1 EDGAR Index Files

All EDGAR submissions are recorded in index files, and there are separate index files for every qtr-year since the system began on experimental basis in 1993. The index file (master.idx) for the current calendar quarter is updated every evening and the file is "frozen" at the end of the quarter. For every submission, the index file includes CIK number, company name, form type, filing date, and the file name. The submission entries always start on line 12 of the file and fields are delimited with "|". Following is an excerpt from the the master.idx file for the first quarter of 2019.

```
Description:           Master Index of EDGAR Dissemination Feed
Last Data Received:    March 31, 2019
Comments:              webmaster@sec.gov
Anonymous FTP:         ftp://ftp.sec.gov/edgar/
Cloud HTTP:            https://www.sec.gov/Archives/


CIK|Company Name|Form Type|Date Filed|Filename
--------------------------------------------------------------------------------
1000045|NICHOLAS FINANCIAL INC|10-Q|2019-02-14|edgar/data/1000045/0001193125-19-039489.txt
1000045|NICHOLAS FINANCIAL INC|4|2019-01-15|edgar/data/1000045/0001357521-19-000001.txt
1000045|NICHOLAS FINANCIAL INC|4|2019-02-19|edgar/data/1000045/0001357521-19-000002.txt
1000045|NICHOLAS FINANCIAL INC|4|2019-03-15|edgar/data/1000045/0001357521-19-000003.txt
1000045|NICHOLAS FINANCIAL INC|8-K|2019-02-01|edgar/data/1000045/0001193125-19-024617.txt
1000045|NICHOLAS FINANCIAL INC|SC 13G/A|2019-02-04|edgar/data/1000045/0001104659-19-005360.txt
1000045|NICHOLAS FINANCIAL INC|SC 13G/A|2019-02-08|edgar/data/1000045/0001258897-19-001312.txt
1000045|NICHOLAS FINANCIAL INC|SC 13G/A|2019-02-11|edgar/data/1000045/0001019056-19-000082.txt
1000045|NICHOLAS FINANCIAL INC|SC 13G/A|2019-02-13|edgar/data/1000045/0000315066-19-001222.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|13F-HR|2019-02-14|edgar/data/1000097/0001000097-19-000001.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|4|2019-01-17|edgar/data/1000097/0000919574-19-000485.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|4|2019-01-24|edgar/data/1000097/0000919574-19-000578.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|SC 13G/A|2019-02-04|edgar/data/1000097/0000919574-19-000756.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|SC 13G/A|2019-02-08|edgar/data/1000097/0000919574-19-000865.txt
1000097|KINGDON CAPITAL MANAGEMENT, L.L.C.|SC 13G|2019-02-04|edgar/data/1000097/0000919574-19-000754.txt
1000148|LEGACY CAPITAL FUND, INC.|X-17A-5|2019-02-21|edgar/data/1000148/9999999997-19-000997.txt
1000177|NORDIC AMERICAN TANKERS Ltd|424B3|2019-03-29|edgar/data/1000177/0000919574-19-002650.txt
1000177|NORDIC AMERICAN TANKERS Ltd|6-K|2019-02-12|edgar/data/1000177/0000919574-19-000990.txt
1000177|NORDIC AMERICAN TANKERS Ltd|6-K|2019-02-13|edgar/data/1000177/0000919574-19-001212.txt
```

Figure 12.1: Excerpt From master.idx

The index files are archived in separate "qtr"-year folders structured as follows using the first calendar quarter of 2019 as an example:

https://www.sec.gov/Archives/edgar/full-index/2019/QTR1/master.idx

To download index files for a time period, we can create a loop that simply changes the year and the qtr number in the http address above and downloads the index file for the given qtr-year.[1] Note that the name

[1]The SEC encourages researchers to download files after business hours.

of the index file on EDGAR is always master.idx so we need to give it a new name when we download it to avoid overwriting existing files. For example, when we download the index file for the first quarter of 2019, we will assign it the name master20191.idx. Below is a function, `get_index` that downloads index files.

The function takes three parameters, `start_year`, `end_year`, and `down_direct`. `start_year` and `end_year` are the first and last year, respectively, of the index files you would like to download. `down_direct` is the name of the folder the index files should be downloaded to. After checking to make sure the folder exists, and creating it if it does not, the the program loops through each year and qtr and downloads the index file.

In:

```
import os
import urllib.request
from pathlib import Path

def get_index(start_year:int, end_year:int,
              down_direct:str):
    """Downloads SEC  EDGAR Index Files.
    start_year - > First Year to download
    end_year-> Last Year to download
    down_direct->Directory to download files to
    """
    print('Downloading Index Files')
    # Check if the download folder exists.
    if not os.path.exists(down_direct):
        # Create the directory if it does not exist.
        os.makedirs(down_direct)
    # Loop through each year and quarter
    for year in range(start_year, end_year+1):
        for qtr in range(1,5):
            # Specify the file you want to download.
            url = 'https://www.sec.gov/Archives/edgar/
   full-index/' + str(year) + '/' + 'QTR' + str(qtr) +
    '/master.idx'
            # Specify the file name and location
            # to download to.
            dl_file = down_direct + 'master' + str(year
   ) + str(qtr) + '.idx'
            # Download the file.
            urllib.request.urlretrieve(url, dl_file)
            # Print the name of the downloaded file
```

```
            print('Downloaded',dl_file,end='\n')
    print('Downloading of Index Files Complete')
    return

# Specify the location of the folder where index files
# will be downloaded to
down_direct = os.path.join(Path.home(),
                           'edgar',
                           'indexfiles')
# Execute the get_index function and download filings
# from 2018, 2019, to the folder /
get_index(2018, 2019, down_direct)
```

Following is the output generated from the above script. Each downloaded file is listed.

Out:

```
Downloading Index Files
Downloaded /edgar/indexfiles/master20181.idx
Downloaded /edgar/indexfiles/master20182.idx
Downloaded /edgar/indexfiles/master20183.idx
Downloaded /edgar/indexfiles/master20184.idx
Downloaded /edgar/indexfiles/master20191.idx
Downloaded /edgar/indexfiles/master20192.idx
Downloaded /edgar/indexfiles/master20193.idx
Downloaded /edgar/indexfiles/master20194.idx
Downloading of Index Files Complete
```

12.2.2 Download SEC Filings

Now that the index files have been downloaded, we are now ready to download the filings. In the following example, we will download 10-Ks filed from 2018 through 2019. For illustration purposes, we will only download the first five 10-Ks for each quarter but this limit can easily be changed or removed by changing or removing the "if statement" that checks the count.

The `get_files` function takes five parameters, `start_year`, `end_year`, `reform`, `inddirect`, and `odirect`. As with the `get_index` function, `start_year` and `end_year` specify the year range to be downloaded. `reform`, will contain the regular expression specifying the filings to be downloaded (e.g., 10-K). `inddirect` is the folder containing the index files, and `odirect` is the directory the filings will be downloaded to.

To limit the total number of files in a given folder, the filings will be downloaded into separate year folders.

Most of the logic below is straightforward but there are two regular expressions that need further explanation. The first regular expression is used to identify the forms to download, which are 10-Ks in this example. Following is a line containing a 10-K from the index file for the first quarter of 2018 (master20181.idx).

```
1000228|HENRY SCHEIN INC|10-K|2018-02-21|edgar/data/1000228/0001000228-18-000012.txt
```

We need a regular expression that finds "10-K", but we also want to download 10-KSB, 10-KSB40, and 10K-405, which are all 10-K type filings. Therefore, we are going to create a regular expression that accommodates those form types as well. Following is the expression used in this example.

```
reform='(\|10-?k(sb|sb40|405)?\s*\|)'
re_formtype = re.compile(reform, re.IGNORECASE)
```

In the example above, we are looking for a `'10'` that follows the delimiter, `'|'`. It is a good idea to start with the delimiter because there are some filings that we want to ignore, such as `'NT 10-K'`, which a company files if does not expect to file its 10-K on a timely basis. By requiring the `'10'` to come right after the delimiter we are able to skip over "'NT" filings. In most cases, the `'10'` is followed by a `'-'` but it is possible that the `'-'` is omitted, so we follow the `'-'` with a `'?'`, which makes the `'-'` optional. Next we require `'K'`, which we follow with `'(sb|sb40|405)?'`. This part of the expression means that the `'K'` is optionally followed by `'sb'`, `'sb40'`, or `'405'`. Finally,`'\s*\|'` means we need zero or more spaces followed by `'|'`. Note that this expression intentionally excludes amended 10-Ks, which has a form type of '10-K/A'. If you want to download amended 10-Ks as well, simply remove `'\s*\|'` from the end of the regular expression.

The second regular expression we will use is:

```
re_fullfilename=re.compile(r"\|(edgar/data.*\/([\d-]+\.txt)
    )",re.IGNORECASE)
```

This expression gets the location and name of the filing to download. Using the HENRY SCHEIN example, above, the expression will obtain

"edgar/data/1000228/0001000228-18-000012.txt" as the value for the first regex group, and "0001000228-18-000012.txt" as the second.

Using these two regular expressions with the following logic, we will obtain 10-Ks filed in 2018 and 2019.

```
import urllib.request
import shutil
import os
import re
from pathlib import Path

def get_files(start_year:int, end_year:int,
              reform:str,
              inddirect:str, odirect:str):
    """
    Downloads SEC filings for specific companies
    start_year -> First Year to download
    end_year -> Last Year to download
    reform -> Regex to specify forms to be downloaded
    inddirect -> Directory containing index files
    odirect -> Directory the filings will be downloaded to
    """

    print('Downloading Filings')

    # Regex to identify the form to download.
    re_formtype = re.compile(reform, re.IGNORECASE)
    # Regex to extract file name information
    # from a line
    re_fullfilename = re.compile(r"\|(edgar/data.*\/([\d
    -]+\.txt))", re.IGNORECASE)

    #loop through the index files based on year
    for year in range(start_year, end_year+1):
        #check whether the directory exists and create one
        # if it does not.
        download_path = os.path.join(odirect, str(year))
        if not os.path.exists(download_path):
            os.makedirs(download_path)

        for qtr in range(1,5):
            #name of index file to be read.
            dl_file = os.path.join(inddirect, 'master' +
    str(year) + str(qtr) + '.idx')

            # check to see if the index file exists.
```

```
            if not os.access(dl_file, os.R_OK):
                # Download the index file if it does not
                # already exist
                url='https://www.sec.gov/Archives/edgar/
    full-index/' + str(year) + '/' + 'QTR' + str(qtr) + '/
    master.idx'
                # download the file defined as url and
                # download to the file defined a dl_fle.
                urllib.request.urlretrieve(url, dl_file)
            # open the index file
            with open(dl_file, 'r') as f:
                # set a counter called count to 1. Note
                # that the counter will only be incremented
                # after it downloads a file.
                count=1

                # loop through each line in the index file,
                # assigning to a variable called line
                for line in f:
                    # Only download a file if the counter
                    # is less than 5.
                    # Remove this if statement if you want
                    # to download all the files for the
                    # time period
                    if count<5:
                        # Check to see if the the line
                        # matches the form type
                        rematch=re.search(re_formtype,line)
                        #If there is a match then download
                        # the filing
                        if rematch:
                            # The following line searches
                            # for filename information.
                            # The first grouping will
                            # contain the location and
                            # filename of the file to be
                            # downloaded. The second
                            # grouping will contain just
                            # the filename o
                            matches = re.search(
    re_fullfilename, line)
                            if matches:
                                # Construct the url to for
                                # retrieving the filing
                                url = str('https://www.sec.
    gov/Archives/') + str(matches.group(1))
```

```
                                # Create the filename to
                                # download the file to.
                                outfile = os.path.join(
    download_path,
                                                       str(
    matches.group(2)))
                                # Check to make sure the
                                # file hasn't already
                                # been downloaded

                                if not (os.path.isfile(
    outfile) and os.access(outfile, os.R_OK)):
                                    # Print the name of the
                                    # file to be downloaded
    .
                                    print("Downloading:"+
    str(outfile),end='\n')
                                    #downlaod the file
                                    urllib.request.
    urlretrieve(url, outfile)
                                    count += 1
     print('Downloading of Filings Complete',end='\n')
     return

# Specify, in regular expression format, the filing
# you are looking for.  Following is the for 10-k.
reform='(\|10-?k(sb|sb40|405)?\s*\|)'

# Specify location of the index files.
inddirect = os.path.join(Path.home(), 'edgar', 'indexfiles'
    )

# Specify where to download filings to
odirect = os.path.join(Path.home(), 'edgar', '10K')

# Execute the get filings function
get_files(2018, 2019, reform, inddirect, odirect)
```

12.2.3 Read Filings

Now that the filings have been downloaded, we can draw from some of the previous chapters to extract data we are interested in. Let's first take a look at what is contained in the files we downloaded, using the 2018, 10-K submission (0001000228-18-000012.txt) by Henry Schein,

Inc. If you view this submission on the SEC website from your browser, it looks as in Figure 12.2.

Figure 12.2: 10-K Submission

As shown in Figure 12.2, the submission contains some basic information about the filing (e.g., filing date, report date, and form type) and about the company (e.g., company name, CIK number, and business address). A typical submission involves multiple files (e.g., the10k_2017.htm), of various types (e.g., HTML, text, jpeg, XBRL, etc.) and all of these files are "embedded" into one large text file. In this example, the "Complete submission text file" is 0001000228-18-

000012.txt. The Format of the file is Standard Generalized Markup Language (SGML). The best way to think about SGML, is to contrast it with HTML. HTML tells the browser how text should be rendered (displayed), and SGML describes what the text is. For example, the first section of this file begins with the tag "<SEC-HEADER>" and ends with the tag "</SEC-HEADER>". All header information is contained within these two tags. Following the header, each file is embedded sequentially and sandwiched between these tags, <DOCUMENT> and </DOCUMENT>. There is a pair of document tags for every embedded file. The SGML tags are very helpful in finding what you are looking for. In this particular script, for example, we are only in interested in gathering header information, so we limit our text processing to data between <SEC-HEADER> and </SEC-HEADER>.

In the script, below, we extract basic header information (e.g., CIK number) from the files we downloaded from EDGAR. The header part of the file is relatively easy to work with because, unlike most of a filing, the formatting is standard. The code within the `process_header` function, below, is documented but there are two topics that warrant more discussion.

The edgar_vars Dictionary: The variable, `edgar_vars`, is a dictionary, like those covered in Section 4.7. Each entry in this dictionary contains a regular expression used to extract certain information. In this script we extract the file name, CIK number, report date, filing date, company name, SIC code, and a hyperlink to the submission. There is a dictionary entry containing a regular expression to extract each of these data items. For example, the key-value pair for the CIK number, looks like this:

```
"cik":re.compile('^\s*CENTRAL\s*INDEX\s*KEY:\s*(\d{10})',
    re.IGNORECASE),
```

Using a dictionary like this allows us to process all the regular expressions in a loop, which reduces the amount of code we have to write.

Read File Names into a List: To read in and process all the files in a folder, we first need to create a list containing all the file names. There

is a very useful package, called `glob`, that makes this step easy. The following statement reads all .txt files contained in the folder located at '/foldername' path into a list named `files`:

```
files = glob.glob('/foldername/*.txt)
```

Once the list is created, it is easy to loop through each file using a for loop. The following script reads 10-K filings, creates a pandas dataframe containing the header information and writes the dataframe out to a csv file.

In:

```
import os
import re
import pandas as pd
import glob

def process_header(start_year:int, end_year:int,
                   filings:str, outfile:str):
    """ Extracts header information from 10-K filings.
    Parameters:
    start_year -> First Year to process
    end_year -> Last Year to process
    filings -> Directory containing files to process
    outfile -> CSV file output
    """

    # Create a "dictionary" of regular expressions for
    # each of the variables we want to get. A
    # dictionary contains "keys" and "values."  In
    # this case, for example, the key "cik", refers
    # to the regular expression for the cik number.
    edgar_vars={
        "file" : re.compile('<SEC-DOCUMENT>(.*\.txt)',
    re.IGNORECASE),
        "cik" : re.compile('^\s*CENTRAL\s*INDEX\s*KEY:\
    s*(\d{10})', re.IGNORECASE),
        "report_date" : re.compile('^\s*CONFORMED\s*
    PERIOD\s*OF\s*REPORT:\s*(\d{8})', re.IGNORECASE),
        "file_date" : re.compile('^\s*FILED\s*AS\s*OF\s
    *DATE:\s*(\d{8})', re.IGNORECASE),
        "name" : re.compile('^\s*COMPANY\s*CONFORMED\s*
    NAME:\s*(.+)', re.IGNORECASE),
        "sic" : re.compile('^\s*STANDARD\s*INDUSTRIAL\s
    *CLASSIFICATION:.*?(\d{4})', re.IGNORECASE),
        "hlink" : re.compile(r'(.*?([0]*(\d+))\-(\d
    {2})\-(\d{6})))', re.IGNORECASE)
```

```
}
# create a regular expression representing the
# last row of the file you want to read. The tag
# '</SEC-HEADER>' represents the end of the
# Header information in the .txt file.  All the
# header information should be found before this
# line
regex_endheader = re.compile(r'</SEC-HEADER>', re.
IGNORECASE)

# Create a dataframe that has column names
# identical to those we defined in our
# dictionary. The "keys()" method creates
# a list of just the keys in the dictionary
# This means, edgar_vars.keys(), is a list
# that looks like this:
# ["file", "cik", "report_date", "file_date", "name
", "sic", "hlink"]
eframe = pd.DataFrame(columns = edgar_vars.keys())

# loop through each of the year folders in
# filings
for year in range(start_year, end_year+1):
    # specify the files to process
    path = os.path.join(filings,str(year),'*.txt')
    # read in the names of each of the files
    # contained in the folder
    files=glob.glob(path)
    # process one file at a time.
    for file in files:
        # Create a dictionary to hold the
        # information we are obtaining
        # (e.g., cik number)
        header_vars={}
        # For each of the keys contained
        # in the dictionary, set the initial
        # value to -99. This way we are sure
        # that each item is defined in the
        # dictionary even if we cannot
        # find the value.
        for x in edgar_vars.keys():
            header_vars[x]=-99
        # Open the file we are processing
        # and read it in one line at a time.
        f=open(file, 'r')
        for line in f:
```

```
                # The "items" method converts
                # the dictionary into a list
                # that is easy to operate on
                tems=edgar_vars.items()
                # Loop through the dictionary
                # and assign the key to "k"
                # and the value to "v"
                # For example, the first time
                # through, k="file", and
                # v="re.compile('<SEC-DOCUMENT>(.*\.txt
)', re.IGNORECASE)"
                for k, v in tems:
                    match = v.search(line)
                    # if a match is found for the first
                    # time, add it to the dictionary
                    # containing the header values.
                    # The purpose of the expression,
                    # re_key!="hlink" is to not try and
                    # match the hlink expression.
                    # The hlink expression is used at
                    # the end to create a hyperlink to
                    # the file on edgar.
                    if match and header_vars[k]==-99
and k!="hlink":
                        header_vars[k]=match.group(1)
                # Check to see if we are at the end
                # of the header part of the filing.
                # Exit if we are there
                match = regex_endheader.search(line)
                if match:
                    break
            f.close()
            # Create a link to the file on edgar
            if header_vars['file'] != -99:
                #Construct a link to the actual filing
                match = edgar_vars['hlink'].search(
header_vars['file'])
                if match:
                    header_vars['hlink'] = str('http://
www.sec.gov/Archives/edgar/data/')+str(header_vars[
'cik'].lstrip('0'))+str("/")+str(match.group(3))+
str(match.group(5))+str(match.group(6))+str("/")+
str(match.group(2))+str("-index.htm")
                #add the row to our dataframe
                eframe.loc[len(eframe)] = header_vars
```

```
        # Write to csv file
        eframe.to_csv(outfile, sep=",", encoding='utf-8')
        print(f'Header File: {outfile} created')
        return eframe

# You can change the name of the output file below.
outfile = os.path.join(Path.home(),
                       'edgar',
                       'filingsoutput.csv')
 #Location of filings to be processed
filings = os.path.join(Path.home(), 'edgar', '10K')

edgar_dat = process_header(2018,2019,filings,outfile)
edgar_dat.head()
```

After the script executes, it should produce output like that shown below.

Out[6]:

Header File: /Users/andrewleone/edgar/filingsoutput.csv created

	file	cik	report_date	file_date	name	sic	hlink
0	0001558370-18-002201.txt	0001000232	20171231	20180316	KENTUCKY BANCSHARES INC /KY/	6022	http://www.sec.gov/Archives/edgar/data/1000232...
1	0001213900-18-003785.txt	0001000683	20171231	20180402	BLONDER TONGUE LABORATORIES INC	3663	http://www.sec.gov/Archives/edgar/data/1000683...
2	0001000229-18-000025.txt	0001000229	20171231	20180212	CORE LABORATORIES N V	1389	http://www.sec.gov/Archives/edgar/data/1000229...
3	0001193125-18-082402.txt	0001000209	20171231	20180314	MEDALLION FINANCIAL CORP	6199	http://www.sec.gov/Archives/edgar/data/1000209...
4	0001437749-18-022311.txt	0001000230	20181031	20181219	OPTICAL CABLE CORP	3357	http://www.sec.gov/Archives/edgar/data/1000230...

This program should provide a reader with a useful framework for analyzing SEC filings. For example, the reader can incorporate functions illustrated in Chapter 12 to extract the MD&A section of the 10-K, and Chapter 7 to measure the tone of the MD&A.

12.3 Web Scraping

There are times when we want to obtain data from a website, and the process of doing so is often referred to as web scraping. The process requires some familiarity with HTML, which was briefly described in the previous chapter, and a lot of patience! The ease with which we can extract data depends on the consistency with which data is reported on a website. The challenge is that we need to extract the data from HTML files, and some websites change the HTML fairly often. That means, we could write code to scrape data from a website that works one day but not the next. In this section, we will introduce the basics of web scraping with the task of obtaining all Accounting and Auditing

Enforcement Releases (AAERS) available on the SEC's website as an example.[2]

The objective of this exercise is to create a csv file that contains the AAER Number, AAER Date, and name of the Defendant, for each AAER on the SEC website; and download each enforcement release, which can later be further analyzed. If we go to https://www.sec.gov/divisions/enforce/friactions.shtml, we will see that there is a list of AAERs for every year from 1999 to 2020. If we click on 1999, for example, we will see a list of all AAERS for 1999, and the URL address is: https://www.sec.gov/divisions/enforce/friactions/friactions1999.shtml. We will obtain all of the table entries listed in the table that looks like this:

Release No.	***Date***	***Action***
AAER-1213	Dec. 13, 1999	Solucorp Industries Ltd., Joseph S. Kemprowski, Peter R. Mantia, James G. Spartz, Robert Kuhn, Victor Herman, Arle Pierro and W. Bryan Fair ***Release No.:*** LR-16388
AAER-1212	Dec. 14, 1999	Jose E. Rivera, CPA ***Release No.:*** 34-42229

To extract this information from the table, we need to examine the HTML to see how the data is organized. We can view the HTML in Google Chrome browser by selecting View->Developer->View Source from the menu. If we scroll down, we will eventually come to a table and see that the first three rows of the table shown above, are rendered with the following HTML:

```
<table border="0" cellpadding="0" cellspacing="7">
<tr valign="top">
<td nowrap><b><i>Release No.</i></b></td>
<td><b><i>Date</i></b></td>
<td><b><i>Action</i></b>
</td></tr>

<tr vAlign=top>
<td noWrap><a href="/litigation/litreleases/lr16388.htm">
    AAER-1213</a></td>
```

[2]It is important to respect website policies on scraping. Many sites prohibit certain types of scraping, especially when the process consumes significant resources and slows down the site. There is a file at the root of most websites, "robots.txt ", that specifies what is and is not allowed.

```
<td nowrap>Dec. 13, 1999 </td>
<td>Solucorp Industries Ltd., Joseph S. Kemprowski, Peter R
    . Mantia, James G. Spartz, Robert Kuhn, Victor Herman,
    Arle Pierro and W. Bryan Fair
<br><b><i>Release No.:</i></b> LR-16388</td></tr>
<tr vAlign=top>
<td noWrap><a href="/litigation/admin/34-42229.htm">AAER
    -1212</a></td>
<td>Dec. 14, 1999 </td>
<td>Jose E. Rivera, CPA
<br><b><i>Release No.:</i></b> 34-42229</td></tr>
```

Everything looks a bit messy but we can observe a pattern. Each row in the table starts with the <tr> tag and each column starts with the <td> tag. The first column contains the AAER number and a link to the AAER. The second column contains the date and the third column names the defendants. The third column also includes a link to the litigation release, if applicable, but we will ignore the litigation releases in this exercise. To get the data we need, for each year, we need to: 1) find the table containing the AAERs within the HTML file, 2) loop through each row, 3) extract the AAER Number, link to the AAER document, AAER date, and defendant, and 3) download the AAERs. Along the way, we will also create a pandas dataframe with the basic AAER information and then output the dataframe to a csv file. The script below will handle each of these tasks. In addition to comments in the source code of the script, we explain how it works in the text that follows the source code.

```
1  import pandas as pd
2  from bs4 import BeautifulSoup
3  import requests
4  from dateutil import parser as dateparse
5  import re
6  import os
7  import urllib.request
8
9  def get_aaers(start_year:int, end_year:int,
10                down_folder:str, csv_file:str):
11     # create a dataframe to store the list of AAERS
12     aaer_table = pd.DataFrame(columns = ['aaer_number',
13                                          'date',
14                                          'defendant',
15                                          'link'])
```

```
16      # loop through each year of AAERS
17      for year in range(start_year, end_year+1):
18          print(year)
19          # Define the URL to get
20          url = 'https://www.sec.gov/divisions/enforce/
    friactions/friactions' + str(year) + '.shtml'
21          # download the file
22          response = requests.get(url)
23
24          print('here')
25
26          # extract html from response object
27          data = response.text
28          # parse the HTML.
29          soup = BeautifulSoup(data, 'lxml')
30          # The AAER table is the 5th table in the
31          # document prior to 2016. It is the first
32          # table after that. So the table number
33          # is specified as follows:
34          if year > 2015:
35              idx = 0
36          else:
37              idx = 4
38          # Grab the table
39          table = soup.find_all('table')[idx]
40          # loop through each row - each AAER
41          for row in table.find_all('tr'):
42              # Get the columns contained in the row
43              columns = row.find_all('td')
44              # make sure there is at least one column
45              if columns:
46                  # make sure that the first column contains
47                  # the tag 'a'
48                  if columns[0].find('a'):
49                      # make sure that the first column
50                      # contains a hyperlink
51                      if columns[0].find('a').get('href'):
52                          # create a variable containing a
53                          # link to the AAER
54                          link = 'https://www.sec.gov' + str(
    columns[0].find('a').get('href'))
55                          # get the AAER number
56                          # (e.g., AAER-1209)
57                          aaer = columns[0].find('a').
    contents[0]
58                          # remove the "AAER-" from AAER
```

```
                        # so there is just number
                        aaer_num = re.findall(r"\d+",aaer)
    [0]
                        # Use date parsing package to
                        # parse the date.
                        dt = dateparse.parse(columns[1].
    contents[0])
                        aaerdate = str(dt.year) + str(dt.
    month).zfill(2) + str(dt.day).zfill(2)
                        defendant = columns[2].contents[0]
                        # add the row to our dataframe
                        aaer_table.loc[len(aaer_table)] = [
    aaer_num,aaerdate,defendant,link]
                        # check to see if the AAER is
                        # a pdf or htm file.
                        if link.find('pdf')!= -1:
                            ftype='.pdf'
                        else:
                            ftype='.htm'
                        # name of file to download to
                        dl_file = os.path.join(down_folder,
     str(aaer_num) + ftype)

                        print(dl_file)

                        # download the file if not already
                        # been downloaded.
                        if not (os.path.isfile(dl_file) and
     os.access(dl_file, os.R_OK)):
                            urllib.request.urlretrieve(link
    , dl_file)
    # output the pandas table to a csv file
    aaer_table.to_csv(csv_file,index=False)
    return aaer_table

# Name of folder to download aaers to
down_folder = os.path.join(Path.home(), 'aaers')
#Name of file that will include list of aaers
csv_file = os.path.join(down_folder, 'aaers.csv')
#Download and create a pandas dataframe of AAERs
aaers = get_aaers(1999, 2019, down_folder, csv_file)
#display pandas table
aaers
```

We will use a package called `BeautifulSoup` to parse the HTML files and extract the information we need. The first function, `get_aaers`,

does most of the work. After a pandas dataframe, `aaer_table`, is created, a `for` loop is defined that takes `start_year`, `end_year`, `down_folder`, and `csv_file` as inputs. Like loops used to process index files in the previous section, `start_year` and `end_year` are inputs that specify the range of years to be processed. The variable `down_folder` is the folder the AAER's should be downloaded to, and `csv_file` is the name of the csv file to be created. In line 14, the url for the web page containing the list of AAER's for the year being processed. Line 16 makes a url request with the `requests` package and line 18, puts the HTML into the string named data. Line 20 parses the HTML with the `BeatifulSoup` package, which can now be referenced with `soup`.

Now that the file has been parsed, we can extract the data we are looking for. The structure of the files changed somewhat in 2016, which is the sort of thing that is typically found through trial and error. Prior to 2016, the table containing the AAER listings, is found in the fifth table in the document. This can be seen by searching for "<table" in the source. If we search the source in the url, https://www.sec.gov/divisions/enforce/friactions/friactions1999.shtml , we will find the text containing the HTML shown above in the fifth table. Starting in 2016, this is the first table. As a result, we want to obtain the fifth table before 2016 and the first table beginning in 2016. To obtain the proper table, lines 24-27 set the index to 0 (first table) for years after 2015 and 4 (fifth table) for years prior to 2016. Line 29 obtains the table and then line 31 begins the process of looping through rows which are indentifed by the <tr> tag. For each row, the columns (<td> tag) are obtained in line 33.

Before extracting information from the columns, we need to check to make sure we are dealing with a row that contains an AAER. Referring again to the HTML snip-it provided above, note that the first row consists of column headings and not data to be extracted. The first column of every row with columns of data contains a link to an AAER. Therefore, we will only extract data from rows if the first column contains a hyperlink ("href"). We have three "if" statements beginning in line 35, to make sure a column exists, the column contains an "a" tag, and the column contains an "href." If those "if" statements are satisfied, we can extract the data. Lines 41-49 extract the data from each of the

three columns. Note that column 45, removes the "AAER-" from the AAER number so that we are left with just the integer. For example, we remove "AAER-" from "AAER-1213" and are left with "1213". In rows 47 and 48, we convert the Date, into YYYYMMDD format and so that it can be saved as an integer. This is done using the `dateutil` package.

After the column information is obtained, it is added to the pandas dataframe (line 51). The last step in the routine is to download the specific AAER. The AAERs will be saved by AAER number followed by an extension. The extension will be saved as a PDF or HTML file depending on the file type. Lines 53-56 get the appropriate file extension. Line 60 checks to see if the file has already been downloaded and, if it has not, line 61 downloads it. Finally, line 63 writes the pandas dataframe to a csv file.

This is a relatively simple example of web scraping that demonstrates the potential for collecting data from web pages on the internet. For more advanced web scraping, have a look at Scrapy, a comprehensive Python framework for web scraping.

12.4 A Note on API's

The most convenient and reliable way to obtain data on the internet is through an Application Programming Interface (API). Many popular websites (e.g., Twitter, Facebook, Lexis/Nexis, etc.) offer API's to developers and researchers, which provide more direct access to data on the website's server. Essentially, we can pass the server a query and, rather than rendering the results in HTML to our browser, the results are returned directly to our (Python) program. One will typically be required to request an API access for a specific website/service. For example, we need to apply for developer access to use the Twitter API, which usually takes a few days for approval. We will also likely need to pay an access fee if we want to obtain a large amount of data.

Most APIs have a Python "wrapper", or package, that simplifies programming. Because API implementations vary greatly from website to website, we will not cover a specific example. However, if you need to collect a significant amount of data, we strongly suggest that you

attempt use an API whenever possible. Following are some websites that provide APIs along with links to apply for access.

Web Site	Data	Python Wrapper Available
Twitter	Social Media	Yes
LinkedIn	Social Media	Yes
Lexis/Nexis	News	No
RavenPack	News	Yes
CalcBench	SEC Data	Yes
WRDS	Various	Yes

Acknowledgements

We would like to thank Stefan Reichelstein, the editor of Foundations and Trends in Accounting, for giving us the opportunity to write this monograph. We also thank our colleagues at the University of Illinois at Urbana-Champaign, University of Miami, and Northwestern University for their continuous support and frequent discussions on the importance of examining texts in corporate disclosures. Finally, we would like to thank organizers, Mary Ellen Carter and Marlene Plumlee, and participants of the technical workshop, "Igniting Your Coding Engine" at the 2019 FARS Mid-Year meeting for inspiring us to develop this methodological guide for textual analysis.

References

Bentley, J. W., T. E. Christensen, K. H. Gee, and B. C. Whipple. 2018. "Disentangling managers' and analysts' non-GAAP reporting". *Journal of Accounting Research*. 56(4): 1039–1081.

Bentley, J. W., K. Stubbs, Y. Tian, and R. L. Whited. 2019. "Manipulating the Narrative: Managerial Discretion in the Emphasis of GAAP Metrics in Earnings Announcement Press Releases". Available at SSRN: https://ssrn.com/abstract=349773.

Blankespoor, E. 2019. "The impact of information processing costs on firm disclosure choice: Evidence from the XBRL mandate". *Journal of Accounting Research*. 57(4): 919–967.

Bochkay, K., R. Chychyla, and D. Nanda. 2019. "Dynamics of CEO disclosure style". *The Accounting Review*. 94(4): 103–140.

Bochkay, K., J. Hales, and S. Chava. 2020. "Hyperbole or reality? Investor response to extreme language in earnings conference calls". *The Accounting Review*. 95(2): 31–60.

Bochkay, K. and C. B. Levine. 2019. "Using MD&A to improve earnings forecasts". *Journal of Accounting, Auditing & Finance*. 34(3): 458–482.

Bonsall, S. B., A. J. Leone, B. P. Miller, and K. Rennekamp. 2017. "A plain English measure of financial reporting readability". *Journal of Accounting and Economics*. 63(2): 329–357.

Bozanic, Z., D. T. Roulstone, and A. Van Buskirk. 2018. "Management earnings forecasts and other forward-looking statements". *Journal of Accounting and Economics*. 65(1): 1–20.

Brochet, F., K. Kolev, and A. Lerman. 2018. "Information transfer and conference calls". *Review of Accounting Studies*. 23(3): 907–957.

Brown, S. V. and J. W. Tucker. 2011. "Large-sample evidence on firms' year-over-year MD&A modifications". *Journal of Accounting Research*. 49(2): 309–346.

Butler, M., A. J. Leone, and M. Willenborg. 2004. "An empirical analysis of auditor reporting and its association with abnormal accruals". *Journal of Accounting and Economics*. 37(2): 139–165.

Campbell, J. L., H. Chen, D. S. Dhaliwal, H.-m. Lu, and L. B. Steele. 2014. "The information content of mandatory risk factor disclosures in corporate filings". *Review of Accounting Studies*. 19(1): 396–455.

Cecchini, M., H. Aytug, G. J. Koehler, and P. Pathak. 2010. "Making words work: Using financial text as a predictor of financial events". *Decision Support Systems*. 50(1): 164–175.

Chychyla, R., A. J. Leone, and M. Minutti-Meza. 2019. "Complexity of financial reporting standards and accounting expertise". *Journal of Accounting and Economics*. 67(1): 226–253.

Dyer, T., M. Lang, and L. Stice-Lawrence. 2017. "The evolution of 10-K textual disclosure: Evidence from Latent Dirichlet Allocation". *Journal of Accounting and Economics*. 64(2): 221–245.

Filzen, J. J. and K. Peterson. 2015. "Financial statement complexity and meeting analysts' expectations". *Contemporary Accounting Research*. 32(4): 1560–1594.

Gow, I. D., D. F. Larcker, and A. A. Zakolyukina. 2019. "Non-answers during conference calls". *Chicago Booth Research Paper*. (19-01).

Guay, W., D. Samuels, and D. Taylor. 2016. "Guiding through the Fog: Financial statement complexity and voluntary disclosure". *Journal of Accounting and Economics*. 62(2): 234–269.

Gunning, R. *et al.* 1952. "Technique of clear writing".

Hanley, K. W. and G. Hoberg. 2010. "The information content of IPO prospectuses". *The Review of Financial Studies*. 23(7): 2821–2864.

Heitmann, M., C. Siebert, J. Hartmann, and C. Schamp. 2020. "More Than a Feeling: Benchmarks for Sentiment Analysis Accuracy". *Working Paper, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3489963*.

Henry, E. and A. J. Leone. 2016. "Measuring qualitative information in capital markets research: Comparison of alternative methodologies to measure disclosure tone". *The Accounting Review*. 91(1): 153–178.

Hoberg, G. and G. Phillips. 2016. "Text-based network industries and endogenous product differentiation". *Journal of Political Economy*. 124(5): 1423–1465.

Hoitash, R. and U. Hoitash. 2018. "Measuring accounting reporting complexity with XBRL". *The Accounting Review*. 93(1): 259–287.

Hope, O.-K., D. Hu, and H. Lu. 2016. "The benefits of specific risk-factor disclosures". *Review of Accounting Studies*. 21(4): 1005–1045.

Huang, X., S. H. Teoh, and Y. Zhang. 2014. "Tone management". *The Accounting Review*. 89(3): 1083–1113.

Jegadeesh, N. and D. Wu. 2013. "Word power: A new approach for content analysis". *Journal of Financial Economics*. 110(3): 712–729.

Kincaid, J. P., R. P. Fishburne Jr, R. L. Rogers, and B. S. Chissom. 1975. "Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel".

Kravet, T. and V. Muslu. 2013. "Textual risk disclosures and investors' risk perceptions". *Review of Accounting Studies*. 18(4): 1088–1122.

Lang, M. and L. Stice-Lawrence. 2015. "Textual analysis and international financial reporting: Large sample evidence". *Journal of Accounting and Economics*. 60(2-3): 110–135.

Larcker, D. F. and A. A. Zakolyukina. 2012. "Detecting deceptive discussions in conference calls". *Journal of Accounting Research*. 50(2): 495–540.

Lehavy, R., F. Li, and K. Merkley. 2011. "The effect of annual report readability on analyst following and the properties of their earnings forecasts". *The Accounting Review*. 86(3): 1087–1115. ISSN: 00014826.

Li, F. 2008. "Annual report readability, current earnings, and earnings persistence". *Journal of Accounting and Economics*. 45(2-3): 221–247.

Li, F. 2010a. "Survey of the Literature". *Journal of Accounting Literature*. 29: 143–165.

Li, F. 2010b. "The information content of forward-looking statements in corporate filings—A naïve Bayesian machine learning approach". *Journal of Accounting Research*. 48(5): 1049–1102.

Li, F., M. Minnis, V. Nagar, and M. Rajan. 2014. "Knowledge, compensation, and firm value: An empirical analysis of firm communication". *Journal of Accounting and Economics*. 58(1): 96–116.

Lo, K., F. Ramos, and R. Rogo. 2017. "Earnings management and annual report readability". *Journal of Accounting and Economics*. 63(1): 1–25.

Loughran, T. and B. McDonald. 2011. "When is a liability not a liability? Textual analysis, dictionaries, and 10-Ks". *The Journal of Finance*. 66(1): 35–65.

Loughran, T. and B. McDonald. 2013. "IPO first-day returns, offer price revisions, volatility, and form S-1 language". *Journal of Financial Economics*. 109(2): 307–326.

Loughran, T. and B. McDonald. 2016. "Textual analysis in accounting and finance: A survey". *Journal of Accounting Research*. 54(4): 1187–1230.

Mikolov, T., K. Chen, G. Corrado, and J. Dean. 2013a. "Efficient estimation of word representations in vector space". *ICLR Workshop*.

Mikolov, T., I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013b. "Distributed representations of words and phrases and their compositionality". In: *Advances in Neural Information Processing Systems*. 3111–3119.

Muslu, V., S. Radhakrishnan, K. Subramanyam, and D. Lim. 2015. "Forward-looking MD&A disclosures and the information environment". *Management Science*. 61(5): 931–948.

Price, S. M., J. S. Doran, D. R. Peterson, and B. A. Bliss. 2012. "Earnings conference calls and stock returns: The incremental informativeness of textual tone". *Journal of Banking & Finance*. 36(4): 992–1011.

Project Jupyter. 2018. "JupyterLab Documentation". https://jupyterlab.readthedocs.io/en/stable/. (accessed: 22 Mar 2020).

Project Jupyter. 2020. "About Us". https://jupyter.org/about. (accessed: 22 Mar 2020).

Securities and E. Commission. 1998. "A Plain English Handbook: How to Create Clear SECDisclosure".

Tetlock, P. C. 2007. "Giving content to investor sentiment: The role of media in the stock market". *The Journal of Finance*. 62(3): 1139–1168.

Tetlock, P. C., M. Saar-Tsechansky, and S. Macskassy. 2008. "More than words: Quantifying language to measure firms' fundamentals". *The Journal of Finance*. 63(3): 1437–1467.

You, H. and X.-J. J. Zhang. 2009. "Financial reporting complexity and investor underreaction to 10-K information". *Review of Accounting Studies*. 14(4): 559–586.