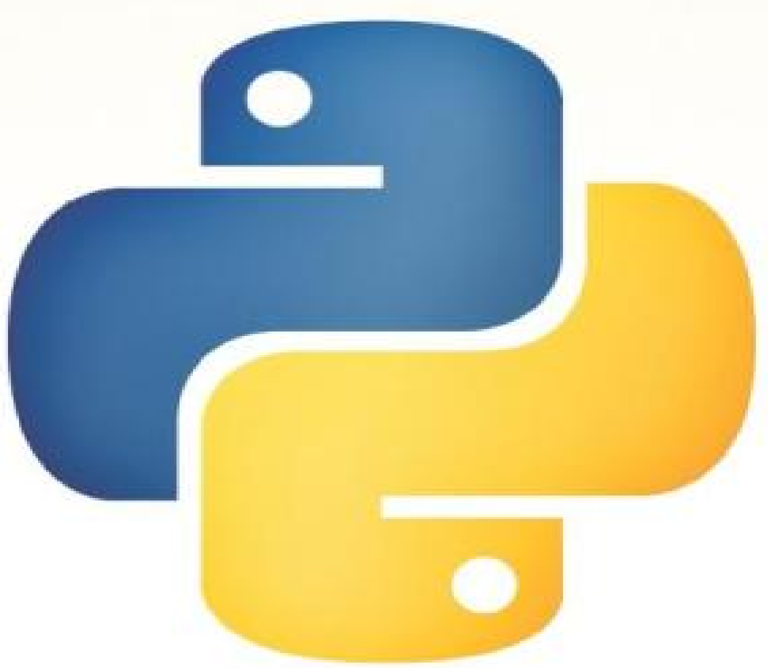
Copyrighted Material

PYTHON for Data Analysis

A Beginners Guide to Master the Fundamentals of Data Science and Data Analysis by Using Pandas, Numpy and Ipython



BRADY ELLISON

Copyrighted Material

Python for Data Analysis

A Beginners Guide to Master the Fundamentals of Data Science and Data Analysis by Using Pandas, Numpy and Ipython

---

Brady Ellison

© Copyright 2021 - All rights reserved.

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

Legal Notice:

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaged in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

Table of Contents

Introduction

What You Should Keep in Mind

All Tech Work Has A Creative Element

Some Things Will Be Harder at First

You Don't Know Everything

You Won't Work Alone

Some Rules

To Python Beginners

Chapter 1: What is Data Science/Analysis?

Data Science vs. Data Analysis

An Example

Data Life Cycle

Data Collection

Data Cleaning

Data Wrangling

Analysis

Application

Why Python?

Chapter 2: Setting Up Your Environment

Anaconda

Windows Anaconda Installation

macOS Anaconda Installation

Using the Installer

Using the Command-line

Linux Anaconda Installation

Chapter 3: iPython & Jupyter

iPython

iPython Installation & Getting Started

iPython Special Features

Getting Information About the Object

Magic Functions

List of Magic Functions

Running and Editing a Python Script

Running System Commands

Jupyter

What Does it Do?

A Quick Overview

Understanding Modality

Jupyter Cell Magic Functions

IPyWidgets

Interactives

Types of Widgets

Numeric Widgets

Boolean Widgets

Selection Widgets

Chapter 4: Pandas

Setting Up Your Environment

Pandas Data Structures

DataFrames & Series

Labelling Indexes In A Series

Converting Tuples & Dictionaries Into A Series

Accessing Data In A DataFrame

Deleting Columns

How to Read and Write Data in Pandas

Learning More About the Data

Writing A DataFrame to A File

Selecting Data

Creating Plots

Creating New Columns

Adding and Removing Columns

Doing Statistics

Combining Tables

Dealing With Textual Data

Find length

Resources

Table A : Reading and Writing data table

Table B:2019 Weekly Data

Table C: The second set of 2019 data for DataFrame combining exercises and others

Chapter 5: NumPy

Installation

The Importance of NumPy Arrays

What is a NumPy Array?

Creating Arrays

Learning About An Array

Basic Array Operations

Accessing Elements, Slicing and Iterating Arrays

Manipulating Shapes

Stacking Arrays

Splitting An Array

Final Words & FAQ

When Do I Know I Have Enough Projects in My Portfolio?

What Type of PC Do I Need for Data Science?

What Are Some of the Skills I Will Need?

Is There a Future in Data Science/Analytics?

What Will it Take for Me to Become a Data Analyst?

Other Books from the Author

References

Introduction

This book is designed to offer working knowledge of Python and data science and some of the tools required to apply that knowledge. It's possible that you have little experience with or knowledge of data analysis and are interested in it. You might have some experience in coding. You may have worked with data before and want to use Python. We have made this book in a way that will be helpful to all these groups and more besides in varying ways. This can serve as an introduction to the most current tools and functions of those tools used by data scientists.

We will cover the following topics:

- Data Science/Analysis and its applications

- IPython and Jupyter - an introduction to the basic tools and how to navigate and use them. You will also learn about its importance in a data scientist's ecosystem.

- Pandas - a powerful data management Python library that lets you do interesting things with data. You will learn all the basics you need to get started.

- NumPy - a powerful numerical library for Python. You will learn more about its advantages.

If you have no idea what any of these mean, don’t worry. This book will explain them in detail and get you started. Before we begin, there are a few things you should keep in mind.

What You Should Keep in Mind

It is important when learning something new to have a goal-oriented mindset as opposed to a limiting one. It makes things easier, giving you the grit you need to deal with difficult problems. Without a focused and goal-oriented mindset, you are prone to be demotivated and eventually giving up. Below are some of the principles that will enable you to thrive.

All Tech Work Has A Creative Element

Learning anything in tech or anything that involves tech has room for creativity or requires it. You will be learning the fundamentals in this book, but it is helpful not to think of these things as laws or rules. Rules and laws are like protocol. They tell us how things should be done, in what circumstances, and how. Tech is not like that. We are not teaching you rules and laws. We are giving you tools, techniques, and tricks to use how you see fit. Some ways will be a better fit for the individual than others, some ways will not be as productive for some tasks, some will be new, and some will be old, some will work instantly, and some won't work for everyone. I am not saying there is no etiquette in tech, there is and you will learn it, but the tech itself does not work that way. So, when you study this book, remember this.

Rote learning and similar methods might help, but they won't make you a better tech practitioner than your peers. It is helpful to know this because students worry when they don't remember precisely how to perform a specific task or fix a particular problem. You don't have to know the syntax off the top of your head (with practice, this will come). All you need to remember are the tools you have and how you can use them to accomplish a task. The concepts and logic are essential. If you need to remember the syntax, you can always look it up or the tools you use will help you with that.

Some Things Will Be Harder at First

As it is with learning anything, you will find some things about Python challenging. This is normal. It does not mean you are not equipped with the intelligence you need to succeed. Sometimes, tech makes sense the more you use it and the more you encounter it. Sometimes what you are learning is a smaller part of a bigger puzzle. Remember, when you encounter these feelings, they don't mean anything about your ability to understand the subject. These feelings are a sign that your brain is working on a problem, meaning it will connect things once they come into view.

You Don’t Know Everything

When you are done with this book, you will not know everything about the subject, and that is fine. You will not know everything because no one knows everything about all tech. You will find that you always have to learn some things. Most discursive fields [if not all] require us to adapt and expand our skills constantly. Sometimes, we may find ourselves in roles that don’t require us to this at all; in such fields, you might be sufficiently competent to perform your duties and advance your career. Your aim shouldn’t be to know all there is to know. It is to be capable enough to solve problems for those who will hire you or yourself. If you keep thinking there is a certain avalanche of information you need to master to start working, you will never begin. You need to be confident in your ability to learn new things when performing new tasks. As you gain more experience through this book and in life, you will do this more easily.

You Won’t Work Alone

In real life, you don’t do all the work alone. You will have other people to work with. You will balance each other's weaknesses and strengths. You don’t have to be rid of weaknesses. Prepare yourself to work with others. While you expand your skills and build on your work ethic, you should also work on your soft skills. Soft skills are about your ability to interact well with others. They are not necessarily social skills, although they can include social skills like good communication. Soft skills can be things like the ability to compromise, communicate ideas, listen, be reliable, sensitive, respect other people's roles, and others.

Some Rules

When reading this book, try to do the exercises yourself and follow along. Don't just try to do them in your head. When you write the code yourself rather than copying and pasting it, you will be better able to practice and apply what you have learned. You will find that you make certain assumptions when you code, things you take for granted. These will show themselves when you make mistakes while writing code. As the console spits errors, you will learn to shed those misguided assumptions and adopt new ones. This means as you go ahead, you will find it easy to do more complex things. Don't sit back even when the code in question looks like a very simple one-liner that you can recall later when dealing with real multi-liners.

To Python Beginners

If you don't know Python, you should read the first chapter to find out if data science/analysis is something that you want to get into. If you think it is, you should then turn your attention to learning Python. We will not go over the Python basics in this book. This shouldn't discourage you. Python is easy to understand and there are plenty of places offline that offer interactive classes on Python. Through them, you will learn the basics you need to start with this book. If you have some coding experience, especially with languages like C++ and Javascript, Python's syntax will look a lot more straightforward and easy to work with. You won't have much trouble learning as you follow along.

A useful resource you can use to learn the Python basics is SoloLearn. SoloLearn is a site and mobile app that hosts different interactive programming language courses for free. While learning, you will be tested with fun quizzes. We recommend it because it is something you can do anywhere at any time, it is very mobile friendly and it is a lot of fun to do. On the site, you find an active community, challenges, and other tasks to help you get comfortable with the language. It won't take you more than a week or two to learn. It will take you a lot less if you come from another programming language.

After you have learned more about Python, you can continue from chapter two.

Chapter 1: What is Data Science/Analysis?

This chapter will consist of a basic overview of data analysis. Having a broader understanding of what you are doing and the role it plays makes learning easier.

Data Science vs. Data Analysis

Data analysts are a subset of data scientists whose roles are focused on providing insights that help in the decision-making processes of institutions or business. Data scientists can perform data analyst duties, but they also build tools and products (Liberty, 2019).

Data scientists are behind powerful search engine algorithms that help find the best search results. They help with recommendation algorithms such as ones you might find on YouTube or Amazon (Data Science vs. Big Data vs. Data Analytics, 2016). They are also involved in things like machine learning, building statistical models, AI, and more. Their work is product-focused.

Data analysts use tools and techniques to arrive at conclusions and insights that can help in the decision-making of institutions or businesses. Data analysts are found in diverse fields like health care, business, gaming, travel, power management systems, and logistics (Data Science vs. Big Data vs. Data Analytics, 2016). They make reports, making the data tell a story and suggest actions based on findings.

Doing these things will require slightly different knowledge or specializations, but there is still a lot in common between them, both in the tools and the techniques used in handling data. There are technologies and subjects you don’t have to learn when you become a data analyst and the same applies to data scientists interested in a particular field. You will find there is a lot that data scientists and data analysts do that is similar in their day-to-day work.

An Example

Imagine Ray's Coffee Shop which has suddenly moved a part of its operation online, offering home delivery and so on. Like any company of its size, Ray's collects a lot of data, mostly for its operations. They keep track of what is selling, what is not, and so forth. For the sake of the example, a data analyst can work in two ways: to help with a specific goal or to mine data for insights that will help the business in its endeavors. In mining for insights, they might collect data from various sources: sales data, data from their web app, or even buy data they think might be helpful. Then, they clean the data and proceed to other stages. When done, they make certain observations about the data, using various tools to tease out helpful information. For instance, they might notice that certain drinks are popular among certain demographics or that certain drinks sell well during certain seasons.

They can then present these findings to the company and advice on what they should do. They can tell them to make more Ray's Hot Cocoa in the winter or advertise more to Generation Z. Ray's might choose to exploit this or they might decide otherwise. Ray's might also wonder if they should expand into a certain neighborhood. The data analysts would then use data from multiple sources to produce the best projections for the business so Ray's knows whether to expand.

In other roles, they can test marketing strategy assumptions, so Ray's can decide on the best strategy. They can help build new products, so Ray's captures a new market. They can forecast trends so that Ray's knows which products to discontinue and which ones to start making. They can engineer dashboards and live reporting tools so that other people in the business can get real-time information on how various aspects of the business are performing.

They can even build machine learning models so Ray's knows which customers are likely to be long-time customers instead of those who are just passing. They can help with logistics to make Ray's more efficient at delivering its product and improving customer satisfaction. They can help Ray's set the best prices for their products. They can help Ray's with their production schedule so that they can cut waste. In a nutshell, they help the business stay healthy and competent.

You will notice that people who do these things have always been around in businesses, even before data analysts became relevant. So, what has changed? In the past, the people who would make these decisions would often have to use premade tools to look at their data and manipulate it. These tools are still useful today, and they are still used by people who perform these tasks. However, these kinds of tools often involve looking at the data and inspecting it. Since these tools are pre-made, they have some limits. In the age we live in now, we can capture more data and usually have to work with a lot of more data because storing data has become very cheap. Inspecting data very closely in person is becoming impractical. This means we have to use new creative tools to look at our data and mine it. That is where Python, in conjunction with other tools, comes in.

Data Life Cycle

The data life cycle refers to the steps and procedures that data analysts are involved in when they work. These don’t have to happen step by step. They sometimes happen simultaneously or in various stages of a project, back and forth. The data life cycle is more of a presentational model that explains what happens or breaks down the job into distinguishable chunks. When you work, it is something you will experience, it is not really something that has to be taught. It makes sense when dealing with data to take these steps.

Data Collection

In the first stage, data analysts collect the data they need. Usually, this data comes from various sources and in multiple formats. They can get the data in file formats like CSV, JSON, and XML. Sometimes they scrape the internet, collect data from APIs, buy it from data brokers, or collect it from databases. If the data is useful and they need it, they will find ways to get it. It might also be important for them to determine what data they might need for a goal they are trying to achieve. So, ultimately, they are guided by their goals and interests in the data they collect.

Data Cleaning

In this stage, the data analysts prepare the data so that they can process it later. Data is cleaned by removing missing data, fixing incomplete data, removing irrelevant data, removing duplicates, and fixing poorly formatted data. Other things that are done are making sure that all the data is in the format that the analytical tools can work with. They also make sure there are no errors in it. The purpose of this is to make sure that the data does not provide false or misleading results. The cleaner the data, the more accurate and insightful it can be.

Data Wrangling

The purpose of this stage is to shape data and to put it together in a way that is more useful to us. The data will be given a structure, merged, and properly indexed. This will make more sense when we talk about data tables.

Analysis

In this stage, the analysis takes place. The data is explored. Statistical models are built. Correlations are investigated to find causations and meaningful correlations are added together. Various statistical analyses are run and hypotheses are tested. Then, visualizations are built, reports are written, and presentations are put together.

Application

In the last stage, all this knowledge is put together to make real life changes. This is where an intervention is suggested, models are built, and real-life tests are done. Think of this stage as the execution stage; it includes all the things we have discussed in our Ray's Coffee Shop example.

Why Python?

You can bully any coding language into doing whatever you want. You can use JavaScript - primarily a web programming language - to do data analysis. Furthermore, there are other languages that are used in data science: C, C++, Julia, and R. However, Python is the most widely used and the most used by data science courses to teach their students.

Unlike any other language I mentioned, Python is the easiest language to learn. You can almost read it in English and its math is intuitive. It is not that alien to the eye and it looks clean. This makes it the perfect language for beginners to learn. When you think of all the things a data analyst has to know and understand to do their job, a language like Python is appropriate because of its intuitive nature and easy to read syntax. Python is also a powerful language with many applications in many areas. Its ease does not make it any less powerful or useful. In any area in tech, you will find Python. It is a versatile tool and a good one to learn no matter what your ambitions are.

Another thing that Python has going for it is that it is free. It has many libraries and it is widely supported. This means you can always count on it to be popular and when you get stuck, you will find answers to your problems easily. Every tool we are going to use is Python-based and they all work with each 0ther. As we learn, you will see how they relate to each other.

Chapter 2: Setting Up Your Environment

In this chapter, we are going to take you through setting up your environment for data science. Since data analysis is a subset of data science, this shouldn't surprise you.

Anaconda

Anaconda is a distribution platform that will download all the packages and tools you need to do data science. Anaconda simplifies the installation of tools we will use in his book. It will also install Python on your system.

Before you attempt to install Anaconda, make sure that your machine meets the following system requirements:

Operating System

- Windows 8 or newer
- 64-bit macOS 10.13+
- Linux - Ubuntu, RedHat, CentOS 6+, and others

If you don't have the operating systems above, you can still install the older version of Anaconda, but you might run into some problems because, in this book, we will use the new version.

System Architecture:

- Windows- 64-bit x86, 32-bit x86
- macOS- 64-bit x86
- Linux- 64-bit x86, 64-bit Power8/Power9

Storage

- Minimum of 5 gigs of space

Windows Anaconda Installation

Before you install Anaconda, you need to verify that your system architecture is x86. Follow these instructions to do so:

Open Windows Explorer.

Double click on Drive C: It should take you to the root. In there, you should have a folder titled “Programs (x86).” If you find it, you can proceed.

Note: It doesn't matter if you have a 64-bit or 32-bit operating system.

Below are instructions on how to install Anaconda for Windows operating systems.

Go to this page: Anaconda | Individual Edition

Scroll down and click the “Download” button. The page will take you to installers at the bottom of the page.

Click on the appropriate installer to download. Make sure you download the file into the “Download” folder on your computer. There have been instances where downloading the installer in other file directories causes problems with the installation process.

If you don’t know if you should pick the 64-bit or 32-bit version:

● Right-click on the Start button.

● A menu will appear. Select “System.”

● A window will appear, and under “System type,” it will tell you if it is 64-bit or 32-bit.

Open the Anaconda installer as soon as it has loaded. If you run into problems, disable antivirus temporarily; you can start it again when the installation is complete.

A Licensing Agreement page will appear. Read it and click, “I Agree.”

By default, the next page has picked the “Just Me” radio button. Click “Next.”

If the directory path selected by default has spaces or Unicode characters, you should go to your root and make a new directory with no spaces or Unicode characters. For instance, a directory like this can cause errors because it has space: “C:\Users\Mike Wendling\.” Between the words “Mike” and “Wendling,” there is space. You can take the following steps to fix this.

● Click on “Browse.”

- Go to C: drive root.

- Click “Make Folder.”

- Give it a name without space or Unicode characters. Like this: “awesomestudent.” The directory will be: “C:\awesomestudent\.”

- Select the folder and click “Next.”

The next page will show you advanced options. Select the “Register Anaconda3 as my default for Python 3.8.” By default, this option will be selected for you already.

Click “Install” and wait. In some machines, it might appear stuck in some stages. Don’t start it over, just wait. To see all the progress that is made, click on the “Show Details” button.

After installation is complete, click “Next.”

Anaconda will ask you to install PyCharm. Click “Next.” This is optional.

You will see a “Thanks for installing Anaconda” dialog box. You can uncheck the checkboxes and click “Finish” or you could leave them as is and click “Finish.”

Anaconda is installed.

macOS Anaconda Installation

There are two ways to install Anaconda on macOS: using an installer or the command line. We recommend using the installer, but we will cover both methods here.

Follow these steps:

Using the Installer

Go to the following link: Anaconda | Individual Edition

Scroll down and click the “Download” button.

You will find two installers under the macOS installers section. Pick the “Graphical Installer” option and download.

Verify the installer file’s content by typing the following command in the terminal: shasum -a 256 filename

Note: You can skip this step if you prefer, but it is not recommended.

Open the file and click “Continue” to begin the installation.

Answer all the prompts in the Introduction and License pages.

You will be asked to choose a directory. Make sure that the installation directory is the “~/opt” directory. You can install it anywhere, but this is recommended.

On the next screen, select the “Install for me only” option.

Click continue.

A screen will appear asking you to click on a link if you want to install Pycharm. Click continue.

After installation is complete. You will see a screen that says “The installation was completed successfully.”

Congratulations, you have installed Anaconda.

Using the Command-line

To install Anaconda using the command-line, follow these steps.

Go to this page and click on the “Download” button: Anaconda | Individual Edition.

Download the Command Line Installer under the macOS section.

Verify the files by typing the following in the terminal:

```
shasum -a 256 /path/filename
```

Note: /path/filename/ should be replaced by the installation path and filename.

Install Anaconda for Python 3.7 by typing the following command in the terminal.

```
bash ~/Downloads/Anaconda3-2020.02-MacOSX-x86_64.sh
```

Note: You should use “bash” whether or not you are using the Bash shell.

You can replace the ~/Downloads with a directory you prefer.

You will get a prompt that says: “In order to continue the installation process, please review the license agreement.” Press Enter to see the license agreements.

Scroll the bottom and enter “yes” to agree.

You will be prompted to press Enter to confirm the chosen location. You can type CTRL-C to change to another location or cancel. It will display the following line when you accept the default setting: PREFIX=/home/<user>/anaconda<2 or 3>. Continue the installation.

Note: The installation might take a little while. That is normal.

You get a prompt that says: “Do you wish the installer to initialize Anaconda3 by running conda init?” Yes is the recommended response, so enter yes.

The installer will say: “Thank you for installing Anaconda!”

A link to install PyCharm will appear. Ignore it.

Close and open the terminal window for the Anaconda installation to take effect.

Congratulations, you have installed Anaconda.

Linux Anaconda Installation

In order to be able to use GUI features, which is what is required for Anaconda and many of it's other offerings, you will need the following packages installed. This table is taken from Anaconda documentation (Installing on Linux — Anaconda documentation, n.d.):

Debian	apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2 libxrandr2 libxss1 libxcursor1 libxc
RedHat	yum install libXcomposite libXcursor libXi libXtst libXrandr alsa-lib mesa-libEGL libXdam
ArchLinux	pacman -Sy libxau libxi libxss libxtst libxcursor libxcomposite libxdamage libxfixes libxra
OpenSuse/SLES	zypper install libXcomposite1 libXi6 libXext6 libXau6 libX11-6 libXrandr2 libXrender1 lib
Gentoo	emerge x11-libs/libXau x11-libs/libxcb x11-libs/libX11 x11-libs/libXext x11-libs/libXfixes x

Then, all you need to do is follow the instructions below. Do this if you have an x86 system.

Go to this link: Anaconda | Individual Edition

Scroll down and click the Download button. It will take you down the page.

Under the Linux section, you will see two versions of Anaconda presented. Select the x86 one.

Note: For the sake of these instructions, we will assume that is your system architecture.

Once the installer has downloaded, open your terminal and run the command below:

sha256sum /path/filename

Note: This is to check the data integrity of the installation package. It is important to do this before proceeding. Where there is “path,” enter the file’s path. Where there is “filename,” enter the package’s file name.

To install Anaconda for Python 3.7, enter the following command in the terminal.

bash ~/Downloads/Anaconda3-2020.02-Linux-x86_64.sh

You should use the bash command even when you are not using the Bash shell. You have to also replace the /Downloads/ directory with the directory you downloaded the installation file.

You will get a prompt that says: “In order to continue the installation process, please review the license agreement.” Click Enter to see the license agreements.

Scroll the bottom and enter yes to agree.

You will be prompted to press Enter to conform to the chosen location. You can type CTRL-C to change to another location or cancel. When you accept the default setting, it will display the following line: PREFIX=/home/<user>/anaconda<2 or 3>. Continue the installation.

It is recommended you install in the default location that the installer suggests.

You will get a prompt that says: “Do you wish the installer to initialize Anaconda3 by running conda init?” Yes is the recommended response, so enter yes.

The installer will say: “Thank you for installing Anaconda!”

A link to install PyCharm will appear. Ignore it.

Close and open the terminal window for the Anaconda installation to take effect. Alternatively, you can enter the following command:

```
source ~/.bashrc
```

Congratulations, Anaconda is installed.

Chapter 3: iPython & Jupyter

In this chapter, we are going to look at iPython and then jump to Jupyter. The two are related because iPython works as the kernel for Jupyter. A lot of what we are going to do in this book will happen in Jupyter. This explains why the two share this chapter.

iPython

It is helpful to think of iPython as a more advanced Python REPL, if you happen to be familiar. Python is interactive and has many exciting features you would expect to get in an IDE. For instance, when an expression is not complete, iPython senses this and doesn't run your code right then, just like hitting Enter in an IDE would work.

Here are some features you can expect from iPython (iPython Documentation — iPython 7.19.0 documentation, n.d.):

- Penetrative object introspection

- History across different sessions

- Caching outputs with references

- Comprehensive tab completion that supports variables, keywords, filenames, and functions

- Magic commands for controlling the iPython environment and the operating system

- Easy watching between different setups.

- Logging and reloading of sessions

- Comprehsenves syntax processing for special purposes

- Access system shell with an alias system

- Easy integration into other tools and programs

- Include access to PDB debugger and Python profile.

If these don’t make sense, don't be intimidated. We will get to them soon.

iPython Installation & Getting Started

When we installed Anaconda in the last chapter, iPython was automatically installed. To make sure that this happened, you can follow these simple steps:

Open Anaconda Powershell Prompt. You can find this by searching for it.

Once it is open, type the following command and hit enter:

```
ipython
```

If you get a message like below, it means you have iPython:

iPython 7.19.0—An enhanced Interactive Python. Type '?' for help.

Look carefully because the message should be three lines down or so.

If you don't get this, you will have to type the following command to install iPython on your system:

```
conda install -c anaconda ipython
```

If you typed in the command and it appears iPython is already installed, you are ready to use it. To start using iPython, you don't need to know a whole lot. As we said, it functions just like a Python REPL with extra features.

Let's do a few exercises to illustrate this.

If you haven't yet, open your Anaconda Prompt. Type: ipython.

In iPython In[1] appears before an expression and this number increments automatically as you type more lines. In regular Python REPL, all we have is the >>> before an expression. Entering the code below should be able to illustrate this point:

```
x = 20

y = 26

Print ( x + y)
```

When you press enter, the result will be 46.

```
(base) PS C:\Users                > ipython
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: x = 20

In [2]: y = 26

In [3]: print(x + y)
46

In [4]:
```

As you can see, the In[1] are incremented with every line of code. The Out[1], which functions similarly, gives us the In[x] line output. To see this, run the following code:

66 * 100

34 / 2

You will see that Out[x] lines have a number corresponding to the line of the executed code. This is shown below:

```
(base) PS C:\Users\          > ipython
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: x = 20

In [2]: y = 26

In [3]: print(x + y)
46

In [4]: 66 * 100
Out[4]: 6600

In [5]: 34 / 2
Out[5]: 17.0

In [6]:
```

Another difference between iPython and regular REPL is the syntax highlighting. Syntax highlighting, as you will know, is valuable for a variety of reasons. It improves readability and disambiguates code, especially when dealing with more lines of code.

iPython Special Features

We are going to go over iPython's special features briefly.

Tat completion

Like in most IDEs, iPython allows code completion to access the methods we want quickly. A feature likes this stops us from having to look at syntax all the time.

As we know, every data type has its methods in Python. We are going to take a look at an example of Tab completion. To do this, type the following code in your iPython REPL:

```
Str = "love is everything"
```

On the next line, type:

```
Str.
```

Press the Tab button. Methods will appear.

```
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: Str = 'love is everything'

In [2]: Str.
        capitalize() expandtabs   isalpha      isnumeric    ljust        rfind        split        translate
        casefold     find         isascii      isprintable  lower        rindex       splitlines   upper
        center       format       isdecimal    isspace      lstrip       rjust        startswith   zfill
        count        format_map   isdigit      istitle      maketrans    rpartition   strip
        encode       index        isidentifier isupper      partition    rsplit       swapcase
        endswith     isalnum      islower      join         replace      rstrip       title
```

You can navigate these with your arrow keys. Navigate to the upper() method using your arrow keys and select it by pressing enter. You will have this:

Str.upper

Finish the code with parentheses, like this:

Str.upper()

Press Enter. You will see in the output that the sentence is all uppercase now.

```
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: Str = "love is everything"

In [2]: Str.upper()
Out[2]: 'LOVE IS EVERYTHING'

In [3]:
```

Getting Information About the Object

In iPython, you can get information about an object (any element) by using this syntax:

```
Object?
```

The ? is the operator that tells iPython to give you information about the element.

Test this with the following code:

```
People = [“Amy”, “James”, “Kamil”]

People?
```

After the last line, press Enter, and you should get this as your output:

```
Type:  list

String form: ['Amy', 'James', 'Kamil']

Length:  3
```

Docstring:

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.

The argument must be an iterable if specified.

For extra details about the object, you can use double question marks, like this:

Str = “life is good”

Str??

Output:

Type: str

String form: life is good

Length: 12

Docstring:

str(object='') -> str

str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or

errors is specified, then the object must expose a data buffer

that will be decoded using the given encoding and error handler.

Otherwise, returns the result of object.__str__() (if defined)

or repr(object).

encoding defaults to sys.getdefaultencoding().

errors defaults to 'strict'.

Magic Functions

iPython has a range of built-in “magic” functions. There are two kinds of them: line-orientated functions and cell-orientated functions. Magic functions solve the problem of awkward Python syntax on the console. Line magics are always preceded by % and they work like command line calls. Line magics get, as an argument, the entire line. Arguments are passed without any delimiters. They can return results that can be used on the right-hand side of an assignment. Cell-magics are always preceded by “%%” and they get the line as the argument and the line below as another argument (iPython - Magic Commands - Tutorialspoint, n.d.).

Let’s look at the %timeit magic functions in both modes. This function s tells us how long it takes for a piece of code to do what it needs to do.

Type in the following code in the iPython prompt to see it for yourself:

```
%timeit range(1000000)
```

You should get an output like this:

```
555 ns ± 3.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Notice that results may vary according to your system.

Now let us see how the same functions work as cell magic. Type in this code:

```
%%timeit x= range(200000)

max(x)
```

Press Enter a couple of times, and you will get a result like this one:

```
13.6 ms ± 72.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

There are many other magic functions that will come in handy as you grow.

List of Magic Functions

%autocall [mode]

Autocall allows you to call functions, without using the parentheses.
It comes in three modes:

- 0 - which means off
- 1 - which is the default
- 2 - which is means it is always on

To illustrate, a simple multiplication function like the one below could be called in different ways under the three settings:

```
def  multi(x , y) :

z = x * y

print (z)
```

Under OFF or setting 1, you would need to call it this way:

```
multi( 3, 9)
```

The output will be: 27

Without the parentheses, you would get the following error:

File "<ipython-input-24-3ed107be4e98>", line 1

multi 3,9

^

SyntaxError: invalid syntax

When autocall is on default, you would be able to call the function without parentheses. In the always-on scenarios, you can call any function without parentheses all the time.

Return autocall to default by typing: %autocall 1. The output will be: Automatic calling is: Smart.

To turn on autocall all the time, type: %autocall 2. The output will be: Automatic calling is: Full

%automagic

Automagic allows us to use magic functions without the % that comes before them. By default, this feature is turned on. For instance, if you type in pwd in your iPython prompt, you will get the present working directory as your output. The function is helpful because if we know where we are, we can navigate to where we want to be.

If you type in %automagic 0 you will turn the automagic function off. This means when you type pwd, the system will throw an error like

the one below.

```
NameError  Traceback (most recent call last)

<ipython-input-30-86938b1e80ee> in <module>

——> 1 pwd


NameError: name 'pwd' is not defined
```

To get the current directory, you will have to enter pwd with the % prefix, like this:

%pwd

It’s a bit cumbersome to do, so turn automagic on by typing: %automagic 1. You will get this message: Automagic is ON, % prefix IS NOT needed for line magics.

%cd

This one works the same way as your machine works. It allows you to do change directories. We are going to be using it in the next section. Officially, the syntax is like this: %cd <directory>. Because automagic is on, you can drop the % sign.

%dhist

This magic command will print all the directories you have visited. It is useful when you are in a flurry of work and you want to get back to a specific directory and can’t remember where it is. It is a quick way of looking it up and going there. Every time you use cd command,

the list is updated in the _dh variable. Below is an example of dhist command output:

Directory history (kept in _dh)

0: C:\Users\awesmestudent

1: C:\Users\awesmestudent\python script

2: C:\Users\awesmestudent\run

3: C:\Users\awesomestudent\python script\assets

%edit

This prompts the text editor and allows you to edit a Python script. We are going to see how it works in the next section.

%matplotlib

This will activate matplotlib support. It won't import a library. TkAgg is the default GUI toolkit for matplotlib. It is possible to request a different GUI backend. To see the list of those available, enter :

%matplotlib—list

Just a reminder: If you don't understand any of this or you find it confusing, it will become clear as we use these tools.

Running and Editing a Python Script

We are going to use iPython to run a script and edit that script. Open your text editor and write the following code, and save it under with the py. Extension:

```
journey = ["Don't", "Stop", "Beleiving"]

for x in journey:

print(x + "\n")
```

You can copy and paste it into the file, and don’t fix the spelling error. We will do that later.

Open iPython and cd to the directory where the file is stored.

The syntax is like this:

```
cd <directory>
```

Here’s an example:

```
cd C:\Users\acer\python script
```

Once you have done that, type “run” followed by the filename. Like this:

```
Run believe.py
```

Our output from the code should be:

```
Don’t

Stop

Beleiving
```

Immediately we see the lyric is not right. The last word is misspelled, and maybe it could also use an exclamation point. To edit the file, we don’t need to navigate back to the text editor. We can open it through the iPython Prompt.

On the next like type: edit filename, like this:

```
edit believe.py
```

iPython will open the file and you can then edit it. Remember to fix the spelling error and add an exclamation point:

```
journey = ["Don't", "Stop", "Believing!"]

for x in journey:

print(x + "\n")
```

When you are done editing, save the file and close the editor. iPython will execute the following result:

```
Don't

Stop

Believing!
```

Here’s how it all looks:

```
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: cd C:\Users\          \python script
C:\Users\          \python script

In [2]: run believe.py
Don't

Stop

Beleiving


In [3]: edit believe.py
Editing... done. Executing edited code...
Don't

Stop

Believing!


In [4]:
```

If you want to check the command you have run, you can type, "history" in the next line and press Enter. The console would display the commands you have run in this session in their order.

In this example it would be:

cd C:\Users\awesomestudent\python script

run believe.py

edit believe.py

history

If you look at the code, these are all the commands you have run unless you have been doing your exercises in one window and building on all the others in this chapter. In that case, you would have something a little bit more verbose.

Running System Commands

You will be able to run system commands from the iPython prompt. Any statement that starts with an exclamation point will be treated as a system command for your operating system. Here are some examples of system commands you can run to test this.

!dir = This will give you information on the directory you are in.

!date – This will give you the date according to your system.

What is also interesting about system commands is that you can assign some of them to Python variables. For instance, you can have code like this:

var = !date

print(var)

The output will be:

[‘The current date is: Thu 12/10/2020 ‘, ‘Enter the new date: (mm-dd-yy) ‘]

As you can see, the output is split where a new line forms, making a list. If you type: var?, this is what it would show you:

Type: Slist

String form: [‘The current date is: Thu 12/10/2020 ‘, ‘Enter the new date: (mm-dd-yy) ‘]

Length: 2

File: c:\awesomestudent\lib\site-packages\ipython\utils\text.py

Docstring:

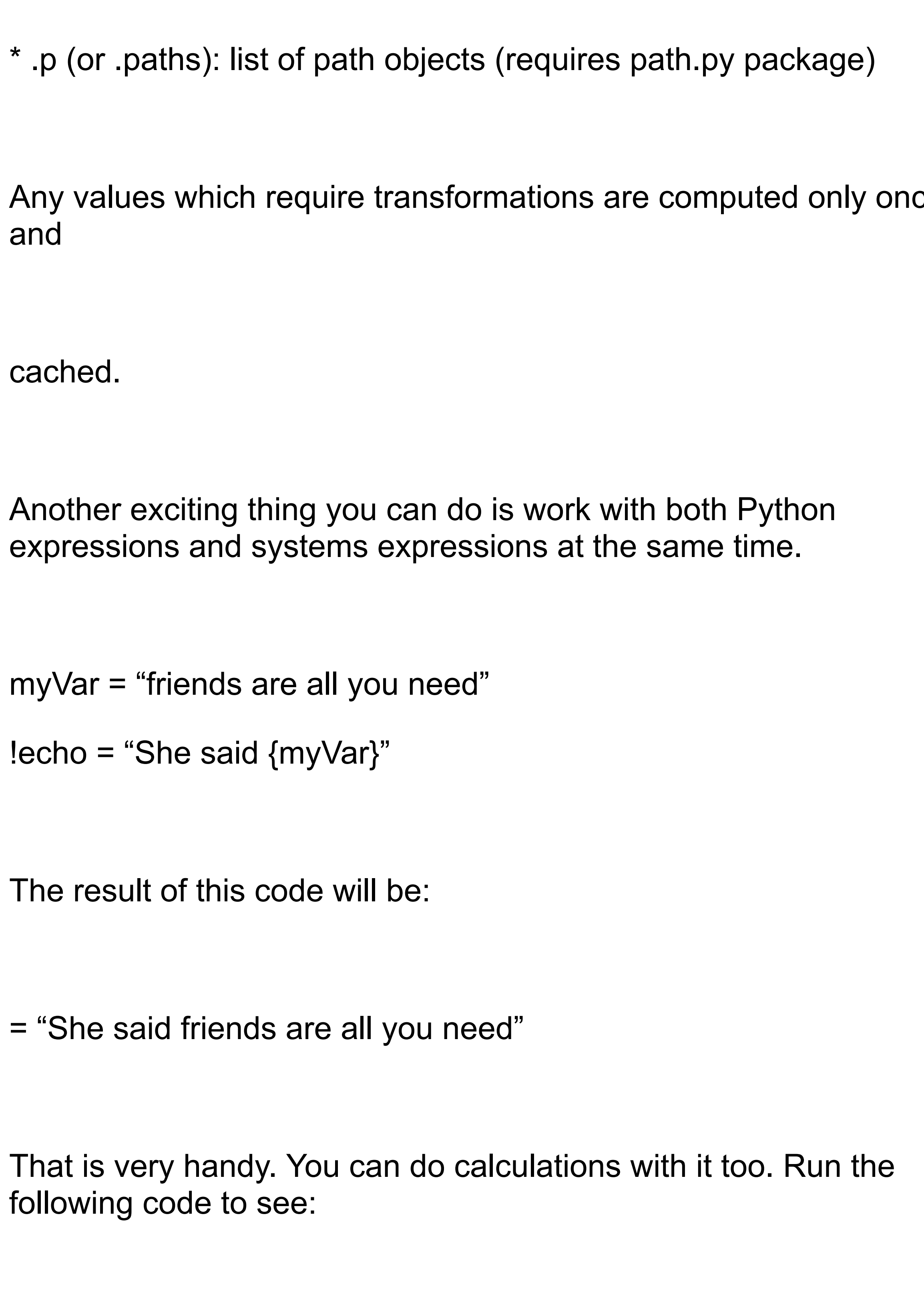
List derivative with a special access attributes.

These are normal lists, but with the special attributes:

* .l (or .list) : value as list (the list itself).

* .n (or .nlstr): value as a string, joined on newlines.

* .s (or .spstr): value as a string, joined on spaces.



* .p (or .paths): list of path objects (requires path.py package)

Any values which require transformations are computed only once and

cached.

Another exciting thing you can do is work with both Python expressions and systems expressions at the same time.

myVar = “friends are all you need”

!echo = “She said {myVar}”

The result of this code will be:

= “She said friends are all you need”

That is very handy. You can do calculations with it too. Run the following code to see:

x = 2

: y = 20

z = y / x

!echo “{y} divided by {x} equals to {z}”

The output will be:

“20 divided by 2 equals to 10.0”

When you lose the quotation marks at !echo “{y} divided by {x} equals to {z}”, the output won’t have any quotations, but it will work

Jupyter

Now, we will start working with Jupyter.

What Does it Do?

Jupyter is an important data science tool that has the unique ability to accommodate code, human language, and a variety of other technologies. A data scientist can make calculations in several languages and display graphs and construct reports in the same space. This data can also be exported in popular formats like HTML, pdf, and Excel files. Jupyter's integrative nature, ease of use, and intuitive design make it a popular choice. Jupyter is also a server side program that opens up in the browser. There is also a cloud version that can be run online. We are going to use the local one.

A Quick Overview

The purpose of this section is to get you familiar with working with Jupyter. One of the best ways to teach people is to show them how something works. So this section, we will be doing this most of the time. Please follow along.

The Jupyter interface is divided into two parts: the header and the body. It is web-based and runs in a browser, so these terms carry the same meaning as those associated with websites.

The body is composed of cells. Cells are one of the most critical concepts in Jupyter. Each cell can house different content, one can house code and run it and another can host markdown (a language similar to HTML) that can serve as an explainer or other purposes. You can add as many cells as you want and a cell can have as much code as is necessary. The markdown allows you to add text, images and make documents appealing to the eye. You can see how it can become a tool for crunching numbers and putting together beautiful reports. It shrinks down the number of tools one has to use by integrating them in one place.

When you installed Anaconda, Jupyter Notebooks should be installed too. You can access it the long way or the quick way.

The long way is going to your OS search and finding the Anaconda Navigator. The window will open a suite of apps that came with Anaconda, some already installed and some to be installed. Launch Jupyter Notebooks by clicking on the launch button. Do not open JupyterLabs.

The short way is to open the Anaconda Prompt and type: jupyter notebook or to click on the program icon from the Start menu.

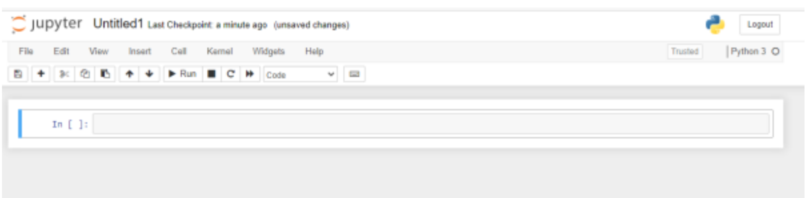
When you have done this, your default browser will open to a page like this.

You will notice that some of your folders and files are listed. You can navigate your computer from the browser to open the files you want in Jupyter. On the top left corner, there are three tabs: files, running, and clusters. The file tab is the one we are currently under. It is rooted in the directory the server is started in.

The running tab will display all the notebooks that are running at any given moment. As you open more notebooks and work, you can come here to keep track of everything or shut down some of the notebooks you no longer need. This is because closing the page/tabs of a notebook is not the same as closing its kernel. To shut it down completely, you have to shut down the kernel. Under this tab, you will also find running terminals if the notebook server is running locally.

The “clusters” tab will give you a summary of iPython Parallel clusters, but you will need to install the iPython Parallel extension for this feature to work.

To create and open a new notebook, you will need to navigate to the top right corner. Click on “new” and select “Python 3.” You will be taken to another page with an interface like this:



The horizontal block with In[ ] next to it is an example of a cell. Cells change their behavior according to their settings. Click on the cell and write the following code in it:

```
8 + 9
```

When you press Enter, the code will not execute. You can execute it by pressing CTRL + Enter or clicking on the play/run button in the header’s menu section.

The output of the code will be: 17

You will also notice that both the input cell and the output now share the same number. We need to run more code to show the significance of that.

To add another cell, press the + button on the menu in the header.

Now, add another cell below the new one by pressing B on your keyboard (for this to work, click elsewhere on the page or press ESC). Click on the cell above and write this code:

```
print(“I am first”)
```

Run the code and press A on your keyboard. Run the following code in the new cell you just created:

```
print(“I am first”)
```

As you will see, the code puts out the same message. If it were in any other situation, we would not know which of these two code blocks executed first. We would assume that the one that comes before the other fired first, but that would be false. This is where the numbers in the In and Out bracket help. They help tell which of the cells’ code was executed before which: 2 came before 3 but the way cells are arranged would not tell us this.

If we have multiple blocks running different kinds of codes, it might be tedious to click on one cell,execute, and then click another, and so on. The notebook solves this by allowing us to press Shift + Enter so we can execute the code easily by pressing Enter again when it focuses on the next one.

Write the following code in the empty cell:

```
if 10 > 2:

print(“Hello Jupyter” * 3)
```

Unfocus (don’t run the code), press B and write this code in the new cell:

```
20 * 20
```

Unfocus and add another cell. In it, write the following code:

```
print(“I am getting the hang of this”)
```

Warning: Do not copy and paste any of the code above. This may lead to errors.

Go back to the first cell. Hold Shift and press Enter. You will see the code execute, then the next cell will be highlighted. When you press Enter again, that code will run. You can imagine doing this multiple times so you can quickly run code instead of selecting a cell and pressing Enter or navigating to the run button multiple times.

Now, select or make a new cell. Go to the drop-down menu and select Markdown. You will see that the iPython like In[] expression disappears. This is because we have told Jupyter that we will not be running Python in that cell. Type the following in the cell:

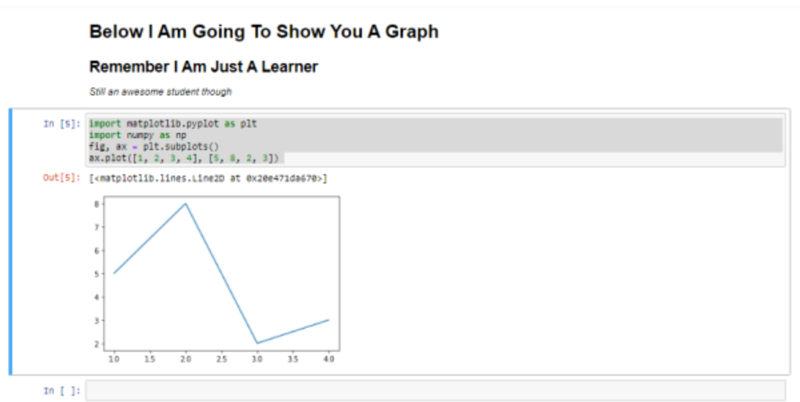
```
# Below I Am Going To Show You A Graph

## Remember I Am Just A Learner

*Still an awesome student though*
```

Now, run the code and see that these three lines are the equivalent of the following HTML tags: <H1>, <H2>, and <em>, respectively.

Add a cell below and type this code in it:

```
import matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

ax.plot([1, 2, 3, 4], [5, 8, 2, 3])
```

Now, the markdown above makes sense. We have displayed a graph in the notebook, and coupled with that is the data we ran. You can see how one can build a compelling document with all these tools. You can do a lot with markdown to customize your document and add multimedia to it. Combining that with all the other Python tools takes it further.



If you want to download the document, you can go to File and select Download. You can also explore some of the options there like “Save as” options.

Markdown is a fairly easy language to learn. Here is a quick cheat sheet below to get you started (Lee, 2018):

H1 to H6 Headings	# Heading Text ## Heading Text ### Heading Text #### Heading Text ##### Heading
Italics	*This text is italicized*
Bold	**This text is bold**
Blockquote	> Blockquote paragraphs must have > a right-arrow bracket at the start > of every sing
Unordered List	- Bullet item - Bullet item - Bullet item - Use a two-spaces to indent the next line
Ordered List	1. Numbered list item 2. Numbered list item 3. Numbered list item 1. Ordered lists can
Mixed List	1. Look as I mix list types! - Awesome right
Horizontal Line	———— Note: You choose between three hyphens, asterisks and underscores. The
Hyperlink	This is an [example link](https://www.example.com)
Image	`![Alt Text](http://example.com/image/path.jpg)`

<table>
<tr><td>Ignore markdown</td><td>When you prefix markdown with *backslashes* formatting will be ignored</td></tr>
<tr><td>Code Inline</td><td>`Some code inline`</td></tr>
<tr><td>Code Block</td><td>``` This will support a block of code Several lines of code Good for illustration purpose</td></tr>
<tr><td>Strikethrough</td><td>~~The text in between will be crossed out~~</td></tr>
<tr><td>Hard Line Break</td><td>Some text\ A new line but not a new paragraph</td></tr>
<tr><td>Table</td><td>| First Header | Second Header | |——————|——————-—| | cell 1 content | cell 2</td></tr>
<tr><td>Task Lists</td><td>- [x] Completed task - [ ] Something I am yet to do - [ ] \(Optional) Parenthese to be igr</td></tr>
<tr><td>Emoji</td><td>:emojicode:</td></tr>
</table>

Understanding Modality

The Jupyter notebook has modality, which means the behavior of the keyboard changes according to working mode. In the last section, although it might have appeared so, we were always careful about when you should use keyboard shortcuts and when you shouldn't. This was because of modality..

Two modes determine how the notebook will act. There is editing mode and command mode. You know you are in command mode when the margins on the left side of a cell are blue. You are in edit mode when a cell is highlighted green instead.

Let's illustrate modality. Click on a cell. Type d and then press Tab. This functions as code completion such as we learned in the iPython section. You will get suggestions that you can navigate. Because the cell has a green border, you are in editing mode. Now press Esc. Pressing Esc activates the command mode. You will notice when you press Tab in this mode, the notebook highlights the next cell in green instead of offering code completion.

Below is a table of commands according to what mode is activated. It is important to learn commands because commands give us a quick, simple way of working through notebooks. It might not be a big issue in small projects to navigate back and forth using your cursor, but this can slow you down if you are working on a large document.

The following tables are taken from Tutorialspoint (Jupyter Notebook - Editing - Tutorialspoint, n.d.).

Command Mode (press Esc to enable)

F	find and replace
Ctrl-Shift-F	open the command palette
Ctrl-Shift-P	open the command palette
Enter	enter edit mode
P	open the command palette
Shift-Enter	run cell, select below
Ctrl-Enter	run selected cells
Alt-Enter	run the cell and insert below
Y	change cell to code
M	change cell to markdown
R	change cell to raw
K	select cell above
Up	select cell above
Down	select cell below
J	select cell below
Shift-K	extend selected cells above
Shift-Up	extend selected cells above
Shift-Down	extend selected cells below
Shift-J	extend selected cells below
Ctrl-S	Save and Checkpoint
S	Save and Checkpoint
Shift-L	toggles line numbers in all cells, and persist the setting
Shift-Space	scroll notebook up

1	change cell to heading 1
2	change cell to heading 2
3	change cell to heading 3
4	change cell to heading 4
5	change cell to heading 5
6	change cell to heading 6
A	insert cell above
B	insert cell below
X	cut selected cells
C	copy selected cells
V	paste cells below
Z	undo cell deletion
D,D	delete selected cells
Shift-M	merge selected cells, or curren
Shift-V	paste cells above
L	toggle line numbers
O	toggle output of selected cells
Shift-O	toggle output scrolling of selec
I,I	interrupt the kernel
0,0	restart the kernel (with dialog)
Esc	close the page
Q	close the page
Space	scroll notebook down

Edit Mode (press Enter to enable)

Tab	code completion or indent	Ctrl-Home	go to cell start
Shift-Tab	tooltip	Ctrl-Up	go to cell start
Ctrl-]	indent	Ctrl-End	go to cell end
Ctrl-[	dedent	Ctrl-Down	go to cell end
Ctrl-A	select all	Ctrl-Left	go one word left
Ctrl-Z	undo	Ctrl-Right	go one word right
Ctrl-/	comment	Ctrl-M	enter command mode
Ctrl-D	delete the whole line	Ctrl-Shift-F	open the command palette
Ctrl-U	undo selection	Ctrl-Shift-P	open the command palette
Insert	toggle overwrite flag	Esc	enter command mode
Ctrl-Backspace	delete word before	Ctrl-Y	redo
Ctrl-Delete	delete word after	Alt-U	redo selection
Shift-Enter	run cell, select below	Ctrl-Shift-Minus	split cell at cursor
Ctrl-Enter	run selected cells	Down	move cursor down
Alt-Enter	run cell and insert below	Up	move cursor up
Ctrl-S	Save and Checkpoint		

Jupyter Cell Magic Functions

We will briefly look at some of the most helpful magic functions you can use in Jupyter.

%%js/javascript

This magic function allows you to work with Javascript code in a cell. This can be handy when you want to use other tools available to JS to improve your work. Some prefer using JavaScript for certain tasks. Let's illustrate how this works.

In a cell, type the following code:

```
%%js

let x = [2, 22, 9, 10, 50, 100]

function containsOdd(x){

console.log(x)

x.forEach(item => {

let be = item % 2

console.log(item)

if(be != 0){

alert("We found an odd number")

console.log("inner")

}

} )

}

containsOdd(x);
```

You should get an alert that says: We found an odd number. We have intentionally illustrated this with multiple code lines involving loops, arrays, and functions to show that functionality is not limited. You can make a mini-app or add js visualizations in the cell and run it.

%%html

If markdown is not your strong suit and you are more comfortable with HTML, you can use the magic function above to use HTML. Just make sure that your cell is set to Code not markdown. This is because Python needs to run and render your HTML.

Write the following in a code cell:

%%html

<h1> This is H1 Tag </h1>

<p> This is a paragraph about how interesting it is that I have managed to do this in a notebook that essentially runs on Python. </br> Look, I have added a new line</p>

Press run and you will see HTML is rendered for the page.

%who_ls

This magic command will list all the variables in the cell. You can also specify the type of information you want to see like functions, lists, or strings.

The following tables list some of the most important magic functions you can use (Mueller, n.d.):

Magic Function	Type Alone Provides Status?	Description
%alias	Yes	Assigns or displays an alias for a system command.
%autosave	Yes	Displays or modifies the intervals between automatic Notebo
%cls	No	Clears the screen.
%colors	No	Specifies the colors used to display text associated with pror
%config	Yes	Enables you to configure IPython.
%file	No	Outputs the name of the file that contains the source code fo
%hist	Yes	Displays a list of magic function commands issued during th
%install_ext	No	Installs the specified extension.
%load	No	Loads application code from another source, such as an onli
%load_ext	No	Loads a Python extension using its module name.
%lsmagic	Yes	Displays a list of the currently available magic functions.
%magic	Yes	Displays a help screen showing information about the magic
%matplotlib	Yes	Sets the backend processor used for plots. Using the inline v
%paste	No	Pastes the content of the Clipboard into the IPython environ
%pdef	No	Shows how to call the object (assuming that the object is cal
%pdoc	No	Displays the docstring for an object.
%pinfo	No	Displays detailed information about the object (often more th
%pinfo2	No	Displays extra detailed information about the object (when a
%reload_ext	No	Reloads a previously installed extension.
%source	No	Displays the source code for the object (assuming that the s
%unalias	No	Removes a previously created alias from the list.
%unload_ext	No	Unloads the specified extension.
%%writefile	No	Writes the contents of a cell to the specified file.

IPyWidgets

IPyWidgets are a Python library that allows you to create interactive HTML GUIs, which are called widgets. There is a wide variety of them. You can make sliders, textboxes, buttons and more. You can use them to explore data in creative ways and synchronize information across two different languages like Python and Javascript. The widgets will be rendered through Python, but the result on the user's end (the front end) will be HTML and JavaScript. This means you can use Javascript to communicate data.

Widget Basics

To use iPyWidgets, you need to import them first. You will need to use the following code to do so:

```
import ipywidgets as widgets
```

Widgets have a display mechanism which allows them to be displayed. Widgets are always displayed in the output area below the cell. You can also use the display() function to display widgets.

This means you can display a widget in one of two ways:

```
widgets.IntSlider()
```

Or

a = widgets.IntSlider()

display(a)

Try it. The output will be like this:

```
In [1]: import ipywidgets as widgets

In [3]: widgets.IntSlider()
        35

In [6]:  a = widgets.IntSlider()
        display(a)
        0
```

Depending on your situation, one will be more appropriate than the other. You can close a widget by using the close() method. Usually, you would call this method in the next cell below the widget.

You can learn more about a widget's properties by using the value property. It works because widgets are objects. Here is an example of it below:

widgets .value

You can also use the dot notation to change the value of the widget. Like this:

widgets.value = 50

Widgets all also have keys, descriptions, and disabled properties. You can view the entire lists of a widgets stateful properties by using:

widgets.keys

You set a widget's properties like this:

widgets.Text(value="I believe in life", disabled=False)

```
In [11]: widgets.IntSlider()

         0

In [10]:  a = widgets.IntSlider()
         a.value = 57
         display(a)

         57

In [7]: widgets.Text(value='I believe in life', disabled=False)

         I believe in life
```

We can give the widget properties as we are declaring it in the cell. We can also use link or jslink method to link values of two widgets so when the value of one gets updated, the other gets updated too.

Like this:

s = widgets.IntText()

b = widgets.IntSlider()

display(s, b)

link = widgets.jslink((s, 'value'),(b, 'value'))

Warning: do not copy and paste. This might give you errors. Type manually.

```
In [13]: s = widgets.IntText()
         b = widgets.IntSlider()
         display(s, b)

         link = widgets.jslink((s, 'value'),(b, 'value'))
```

49

49

When you replace the ‘Int’ with ‘Float,’ the slider will display and work with float numbers using widgets keyword to instantiate widgets. In reality, we have to do so because we imported them under the name(widgets) here:

```
import ipywidgets as widgets
```

We could have done this, for instance:

```
import ipywidgets as wids
```

And we would be using the keyword, “wids,” for our work. like this:

```
wids.Text(value=”life is peachy”, disabled=True)
```

Once you have linked widgets, you can unlink them with the following code:

```
#link.unlink()
```

Interact

This widget lets you explore data and code interactively. This is one of the most exciting features that widgets have to offer, so let’s go through some of its features and usefulness below.

Interact generates UI controls for arguments of functions. When you manipulate the controls, the function is called. Before you can explore a function, you will need to define that function first. As you work with

the UI and change the argument's value, the function executes and gives out answers, which are then displayed in real-time. However, it will act differently based on the arguments you pass through it. Take the following function for an example:

```
def h(x):

return x
```

If you pass x=10, a slider will appear which will keep passing whatever figure is indicated by the slider, and the answer will be returned on the right-hand side. You would pass the function like this:

```
interact(h, x=10);
```

Depending on other arguments types you pass, interact will display something different. If you pass a string, interact will give you a text box instead of a slider. If you pass a boolean, it will give you a checkbox. Here is an example of both:

```
interact(h, x="Chocolate rocks!")

interact(h, x="True")
```

Interact can work with functions that have many arguments. You just have to make sure that the "x" in this case matches the argument you want to bind UI to.

So if you have a function like this:

```
def f(o, l):
return (o, l)
```

You can create a different interact UI for each argument like this:

```
interact(o=”Life is summer”, l = 10)
```

O will produce a textbox with the words “Life is summer” and l will give us a slider at ten.

In some cases, you might want to create a UI with a fixed value. This will prevent that value from being changed as you explore the function of the data. When you do this, interact will not produce UI for the arguments with the fixed value. Here is some example code:

```
def wix(a, b):
return (a, b)
interact(wix, a=fixed(50), b= 15)
```

As we saw, when you pass an integer, you will get an integer slider. This also happens when you pass a float. In some cases, you might want to define some of the features of the slider. You might want the slider to have limits or to increment by ten or three instead of one.

"Interact" gives you the ability to do so with these properties: min, max, step, and value.

"Min" allows you to set the minimum number for the slider. This is the bottom limit of your slider which sits on the left side. You can set this number to zero, a negative number, or a positive number. "Max" sets the highest number that the slider can reach. "Step" sets the value by which the slider will increment or decrement as you move it. "Value" is the number that the slider gets instantiated at.

Here is an example of one below:

```
interact(f, x=widgets.FloatSlider(min=-50.0, max=75.0, step=2, value=5)
```

This can be simplified by using the following syntax:

```
def f(a):

return (a)

interact(f, a=(10.0, 90.0, 1.0)
```

The code above sets a float slider with a minimum of 10.0, a maximum of 90.0 that increments of decrements by 1.0. If you provide only the first two arguments, you will only set the min and the max value. You might wonder from the code above how you will set a default value. To do so, you have to pass the initial value when you define the function, like this:

```
@interact(a=(-10,50, 0.5))

def f(a=10):



return a
```

The code above will give you a slider with a default value of ten.

Another thing we haven’t discussed is how you can use interact to build dropdown menus. To do this, you have to pass a list instead of strings. Like this:

```
interact(f, a=[‘Pop’,’Rock’, ‘Classic’, ‘R&B’, ‘Hip Hop’])
```

The code above will produce a dropdown menu of music genres contained in the list. In some cases, you want a drop-down menu that can pass non-string values. To this, you must pass label/value pairs. When a person pics a label, the value that is paired with that label will be passed. Here’s an example below:

```
interact(f, a=[('option one', 30), ('option two', 28),(‘option three’, 75),
(‘option four’, 99)]);
```

In this case, when you choose “option tw0,” you will get twenty-eight and so on.

Interactives

Interactive is another function that works much like interact except it allows you to reuse widgets and access data that is bound to the UI. Interactives do not display data automatically like interact does. So, you have to define a display method in the function. Here is how the syntax of the function has to look for interactives to display the data:

```
from IPython.display import display

def f(x, y):

display(x + y)

return x+y
```

Interactives return widget instances when you run:

```
widget = interactive(f, x =20, y= 70)
```

Types of Widgets

Below, we are going to look at types of widgets and their properties. You will be able to use the syntax contained to make your own widgets. This section will show you how to control the different characteristics of widgets. We are going to start with widgets we are already familiar with and build to new ones.

Numeric Widgets

The first types of widgets are numerics. They are made to display numbers. Integers and floats, bounded and unbounded, share the same kind of structure whether they are dealing with floats or integers.

Below is an example of a float slider with the properties you can manipulate:

```
widgets.FloatSlider(

value=90.0,

min=0.0,

max=100.0,

step=5.0,

description='This is a Float slider:',

disabled=False,

continuous_update=False,

orientation='horizontal',

readout=True,

readout_format='.1f',

)
```

The first four values in the list will be familiar. We have already talked about those. The description variable allows you to set the label that will be displayed next to the slider. In this case, it will say, “This is a Float slider.” Orientation tells us how the slider should be displayed. The structure will be similar with an Int slider too.

```
widgets.IntSlider(

value=7,

min=0,

max=100,

step=1,

description='This is an Int slider',

disabled=False,

continuous_update=False,

orientation='horizontal',

readout=True,

readout_format='d'

)
```

If you want either of the sliders to be displayed vertically, you will need to change orientation to vertical. This way, it looks more like a fader. It might be nice if you labeled it “volume” by changing the

description. Perhaps give the volume the value of thirty because it is not too quiet or too loud. Like this:

```
widgets.IntSlider(

value=30,

min=0,

max=100,

step=1,

description='Volume',

disabled=False,

continuous_update=False,

orientation='vertical',

readout=True,

readout_format='d'

)
```

Range sliders

Range sliders produce a UI which allows you to define a range by controlling two points on one the slider. There are Int range sliders and Float range sliders. Here is the syntax for both:

```
widgets.IntRangeSlider(
value=[10, 50],
min=0,
max=100,
step=1,
description='Range Test:',
disabled=False,
continuous_update=False,
orientation='horizontal',
readout=True,
readout_format='d',
)
widgets.FloatRangeSlider(
value=[2.3, 8.0],
min=0,
max=10.0,
step=0.1,
description='Float Range Test:',
disabled=False,
continuous_update=False,
```

```
orientation='horizontal',

readout=True,

readout_format='.1f',

)
```

You will notice that the value argument now passes a list with two values. The values inside define the default points where the control button will be on the slider. By default we will have a range of 2.3 to 8.0 for the Float range slider(2.3 - 8.0) and a range of ten to fifty for the Int range slider (10-50). By moving the button in either direction, we can widen and shrink the range or increase or decrease it in volume.

Progress

IntProgress allows the user to visualize progress of different functions. You can give these bars Bootstrap-like themes - success, info, warning, or nothing under the bar_style attribute. You can use FloatProgress or IntProgress to do this. Here is examples of both:

```
widgets.IntProgress(

value=50,

min=0,

max=100,

step=1,

description='Loading:',
```

```
bar_style="info”,
orientation='horizontal'
)
```

This will display a bar that is at the halfway mark.

```
widgets.FloatProgress(
value=9.0,
min=0,
max=10.0,
step=0.1,
description='Loading:',
bar_style='warning',
orientation='horizontal'
)
```

This will display a bar that is 90% filled with red color.

Bounded Text Widgets

These allow you to display text box-looking widgets that accept numbers within a specified range. When they are focused on, users may choose to use the arrows on their right or arrow keys to go

increments or decrements. Alternatively, they can type the number they want in the field, but if the number they type is out of bounds, the text box will be outlined red and will reject the value entered.

Below is the syntax for the BoundedInText widget. This means it will only accept integers between the specified range.

```
widgets.BoundedIntText(

value=5,

min=0,

max= 18,

step=1,

description='Bounded Int Text:',

disabled=False

)
```

Below is an example of BoundedFloatText configured to accept values ranging from 1.0 to 20.0:

```
widgets.BoundedFloatText(

value=10.0,

min=1.0,

max=20.0,
```

```
step=0.5,

description='Bounded Float Text:',

disabled=False

)
```

An alternative to BoundedText widgets are those that are not bounded. These widgets can take any value as long as that value fits the formats specified. For instance, an IntText will only take integers and a FloatText will only take float numbers. When the number does not fit the format, the text box will be lined red.

Below are are examples of both:

```
widgets.IntText(

value=200,

description='Put in any integer:',

disabled=False

)
```

The default value in the IntText will be 200.

```
widgets.FloatText(

value=11.5,
```

```
description='Put in any float number:',
disabled=False
)
```

The default in the text box will be 11.5

Boolean Widgets

As the name suggests, Boolean widgets are on or off widgets that can be either a or b, True or False. In this category, we have only three widgets.

The first widget is the toggle button. Depending on the use case, you will want to set its value to either True or False. They have a tooltip feature that displays an explanation when a user hovers over the button with the cursor. The tooltip is defined under the tooltip attribute. They also are given Bootstrap-like styles or no styles at all.

Here is an example of one:

```
widgets.ToggleButton(

value=False,

description='Click me',

disabled=False,

button_style='',

tooltip='You will see magic',

icon='check'

)
```

The next Boolean widget is the checkbox. You can label it under the “description” argument. Below is an example of one:

```
widgets.Checkbox(

value=False,

description='Check to show',

disabled=False
```

The last one is a read-only widget which indicates if something is valid or not. The widgets display two values, Valid and Invalid. The description attribute specifies which one is the case. Here is how it looks:

```
widgets.Valid(

value=False,

description='Valid',

)
```

Selection Widgets

Selection widgets are widgets that allow one to choose from a variety of options. These include dropdown lists, radio buttons, and so on. To pass options, use lists or value pairs. We looked at dropdown widgets in our discussion of Numerical widgets, so we will not go over them here. However, I will show another syntax you can use for dropdown widgets:

```
widgets.Dropdown(

options=['1', '2', '3'],

value='2',

description='Number:',

disabled=False,

)
```

Next on the list are radio buttons which are optimal when you want only one option to be selected at a time. Here is an example of one:

```
widgets.RadioButtons(

options=['The Matrix', 'Matrix: Reloaded', 'Matrix: Revolutions'],

value='The Matrix’,
```

description='Which is the best Matrix movie?:',

disabled=False

)

The widgets will present the Matrix movies and a user will have to choose which one they think is better. They cannot pick one or two because that would defeat the purpose of the question.

Another type similar to radio buttons is selection widgets because they allow users only to pick one option. They are able to use a variety of UIs to do this. We look at one example of differing UIs.

The following selection widget will display what looks like a text box with rows. When a user selects a value, that value gets highlighted. It also has the rows attribute which specifies the number of rows in the box.

widgets.Select(

options=['The Matrix', 'Matrix: Reloaded', 'Matrix: Revolutions', Matrix 4’],

value='Matrix: Revolutions',

description=’Choose a Matrix movie:',

```
disabled=False

)
```

You can also present the same options in a slider. As the user moves the slider, the values will change to the movie they want. Here is the example of that:

```
widgets.SelectionSlider(

options=['The Matrix', 'Matrix: Reloaded', 'Matrix: Revolutions', 'Matrix 4'],

value='sunny side up',

description='Slide to pick a Matrix movie',

disabled=False,

continuous_update=False,
```

orientation='horizontal',

readout=True

)

SelectMultiple lets you select more than one option when you hold Shift or Ctrl and click on an option using arrow keys.

widgets.SelectMultiple(

options=['The Matrix', 'Matrix: Reloaded', 'Matrix: Revolutions',’Matrix 4’],

value=['Matrix 4'],

description=’Add the Matrix movies you want',

disabled=False

)

The last select widget we will discuss is the ToggleButtons select widget. This one works by presenting a set of buttons. Only one can be selected at a time. In our example, let's imagine that when you press a button with a movie name, that movie plays. The button will be highlighted to show the film that is now playing. You will also be able to add tooltips in Bootstrap-like styles.

```
widgets.ToggleButtons(

options=['The Matrix', 'Matrix: Reloaded', 'Matrix: Revolutions',’Matrix 4’],

description='Play a movie by clicking:',

disabled=False,

button_style='success',

tooltips=['Neo has to choose', 'Neo Meets the Architect', 'Machines rain fire', ‘No one knows’],

)
```

There are many more widgets out there that can add to your document to bring interactivity to them.

Chapter 4: Pandas

Pandas is an open-source Python library that provides easy to use data structures and other data analysis tools. It has a wide range of applications in a wide range of fields. In this chapter, we will introduce you to pandas. As we go on, you will see how these various tools we are talking about create the data analytics toolkit.

Here the benefits of pandas as listed in the documentation (Package overview — pandas 1.1.5 documentation, n.d.):

- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data

- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects

- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations

- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and

transforming data

- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects

- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets

- Intuitive merging and joining data sets

- Flexible reshaping and pivoting of data sets

- Hierarchical labeling of axes (possible to have multiple labels per tick)

- Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving/loading data from the ultrafast HDF5 format

- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging

Setting Up Your Environment

When you installed Anaconda, pandas should have also been installed. To make sure that was the case, open your Anaconda Prompt and run the following command:

```
conda install pandas
```

If pandas is installed, it will tell you. If it isn’t, pandas will install.

Pandas Data Structures

pandas has two data structures: DataFrame and Series. These are best construed as containers for lower dimensional data types. This means that more complex data structures hold those with few dimensions inside. In our case, DataFrames contain Series and Series contain scalar. Scalars are data structures that only have a single value. They can be a character, number, or Boolean.

The data inside DataFrames or Series can be changed. Series length cannot be changed, but columns can be added to a DataFrame. This shouldn't worry you because most of the methods used in pandas make new objects and do not alter the original data. pandas offers immutability where it is appropriate. DataFrames are two-dimensional, size-mutable, and often contain differently labeled columns. Series are one-dimensional, their size cannot be changed, and they have homogeneously labeled columns (Package overview — pandas 1.1.5 documentation, n.d.) .

To start working with panda, import it using the following code:

```
import pandas as pd
```

pd is the agreed-upon alias for pandas.

We will work with pandas in Jupyter. Write that code and any that follow after in Jupyter notebook. Make a new notebook and start working.

DataFrames & Series

We will make a Dataframe structure to illustrate how data is stored and displayed by pandas DataFrame. If you know JSON or are familiar with JavaScript objects, the syntax will be easy to understand. For instance, columns and their labels are properties and rows are indexes.

The following DataFrame Structure captures South Park's main characters, their names, ages, and sexess. Go to Jupyter Notebooks and type in this code after you import pandas:

```
df = pd.DataFrame({

"Name": ["Eric Cartman",

"Stan Marsh",

"Kyle Broflovlski",

“Kenny McCormick”],

"Age": [10, 10, 10,10],

"Sex": ["male", "male", "male",”male”]}

)
```

When you type df in the next line and run, it should give you a table like this one:

Name   Age Sex

0 Eric Cartman 10 male

1 Stan Marsh 10 male

2 Kyle Broflovlski 10 `male

3 Kenny McCormick 10 `male

You will notice that this looks more like a table you would use in a program like Excel to store that. That is the concept behind DataFrames, but this data can be presented in Series data types. We have chosen this example to show that pandas is capable of storing different kinds of data. For instance, in the example above, there is textual and numerical data stored.

If you wanted to access names on the table alone, you would type:

df[“name”]

When you do this, pandas will retrieve a Series which is a column. The numbers on the left are row numbers or indexes. It will output something like this:

0 Eric Cartman

1 Stan Marsh

2 Kyle Broflovlski

3 Kenny McCormick

In some circumstances, it might be best to work with a Series. The syntax for creating one is like this:

shoppingList = pd.Series([‘eggs’,’bread’,’milk’,’butter’])

If you call shoppingList, you will get an output like this:

0 eggs

1 bread

2 milk

3 butter

This is why we say that columns in DataFrames are Series because they have the same structure. Series are one-dimensional structures and you can access their data by their index. Series indexes can be customized so that you don’t have to use numbers to access them.

Labelling Indexes In A Series

So if we had to log how many push-ups you did in a day, we could present that in one of two ways.

We could write the following code:

```
pushups = p.Series([200, 100, 20, 50, 12, 20, 34])
```

This will give us a series like this:

```
0 200
1 100
2 20
3 50
4 12
5 20
6 34
```

So, if you wanted to know how many pushups you did on Wednesday you would type: pushups[2]. Wouldn't it be better if you

could just access the data by the name of the day instead of a number that might as well be arbitrary? It turns out you can do this in pandas. Here is how:

pushups = [200, 100, 20, 50, 12, 20, 34]

#first, build an array/list.

pushups = pd.Series(pushups, index=[‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’, ‘Sunday’])

#then, you pass the array index argument with the labels for your indexes

The output of the Series will look like this

Monday 200

Tuesday 100

Wednesday 20

Thursday 50

Friday 12

Saturday 20

Sunday 34

To find out how many pushups you did on Wednesday, you will type: pushups['Wednesday'].

You can still access data with index numbers. If you want to access Wednesday, you can still use pushups[2].

Converting Tuples & Dictionaries Into A Series

In pandas, you can convert tuples into Series too. So, if you have a tuple like this one:

tups = (‘Mind Freeze’, 2020, ‘May, ‘15’)

To convert it into a Series, you would use the following syntax:

s = pd.Series(tups)

The print(s) function will give you this output:

0 Mind Freeze

1 2020

2 May

3 15

You can also convert dictionaries into Series. So let’s assume you have a dictionary like this one:

dict = {‘Album’ : ‘Fires Within Fires’, ‘Artist’ : ‘Neurosis’, ‘Tracks’ : 5, ‘Year’ : 2020, ‘Rating’: 85 }

#conversion below

Ds = pd.Series(dict)

You can retrieve specific elements this way:

Ds[[‘Album’,’Artist’]

Your output will be:

Album Fires Within Fires

Artist Neurosis

Accessing Data In A DataFrame

DataFrames are two-dimensional unlike Series. This means there are two ways that one can access their data. They can access it by using the row index or the column index. Earlier, we illustrated how data in a DataFrame can be accessed using a column index in our South Park characters example. You can access a data row in a DataFrame with the following syntax.

```
df.ix[0]
```

This will give us the following output, working with our South Park example:

```
Name  Eric Cartman
Age 10
Sex male
```

This way, we have accessed Cartman's profile.

As you can see, all of our rows are still numbers. If we wanted to change them, we would use the following syntax:

df.index[‘first one’, ‘second one’, ‘third one’, ‘forth one’]

Our output of the South Park DataFrame would look this:

	Name	Age	Sex
first one	Eric Cartman	10	male
second one	Stan Marsh	10	male
third one	Kyle Broflovlski	10	male
fourth one	Kenny McCormick	10	male

As you can see, our table’s row column does not have a name and is left empty. We can fix that with the following syntax.

```
df = df.set_index([‘Order’])
```

The output will then be:

Order	Name	Age	Sex
first one	Eric Cartman	10	male
second one	Stan Marsh	10	male
third one	Kyle Broflovlski	10	male
fourth one	Kenny McCormick	10	male

Now, the index row has a column attribute so we can access it’s content. Like this:

```
df.ix[:, ‘Order’]
```

If you wanted to access a particular value in a table, you would have to use the following syntax:

```
df.ix[row , column]
```

Here is an example:

```
df.ix[‘third one’, ‘Name’]
```

Output: Kyle Broflovlski

Deleting Columns

Let's say you have data in DataFrame and you find some of the data in it not relevant to your interest. You can delete that column in the following ways:

```
del df['sex']
```

It will return a table without the sex column.

Order	Name	Age
first one	Eric Cartman	10
second one	Stan Marsh	10
third one	Kyle Broflovlski	10
forth one	Kenny McCormick	10

df.drop(‘age’, axis=1), will return:

Order	Name
first one	Eric Cartman
second one	Stan Marsh
third one	Kyle Broflovlski
forth one	Kenny McCormick

How to Read and Write Data in Pandas

You can use the pandas read function to read data from various formats — Excel, SQL, JSON, HTML, gbq, CSV, and more. Prefix every file format with the read keyword like this:

```
computerUsage= pd.read_csv('data/computerUsage.csv')
```

An underscore follows the read method, and then the file format that is to be read is entered. This time, that file format is CSV. In parenthesis is the directory on the computer where the file is saved so pandas can access it.

When you call the DataFrame, entering the name of it and pressing Enter, you will be shown the first five and the last five rows of the data.

Note: A version of this table is available at the end of the chapter you can use it to follow along. That version is used extensively later, so it is optional if you participate now.

	From	To	Code	Hearthstone	Passive	Reading
1	1/1/2019	1/6/2019	0.09 h	20.60 h	2.13 h	9.88 h
2	1/7/2019	1/13/2019	0.00 h	42.71 h	1.60 h	11.05 h
3	1/14/2019	1/20/2019	0.09 h	14.95 h	0.00 h	21.78 h
4	1/21/2019	1/27/2019	0.00 h	12.61 h	0.09 h	22.91 h
5	1/28/2019	2/3/2019	0.00 h	26.97 h	5.83 h	22.91 h
26	7/1/2019	7/7/2019	11.44 h	0.70 h	9.64 h	1.51 h
27	7/8/2019	7/14/2019	11.80 h	4.87 h	3.74 h	7.32 h
28	7/15/2019	7/21/2019	0.74 h	4.09 h	4.15 h	17.29 h

29	7/22/2019	7/28/2019	4.37 h	2.52 h	8.77 h	24.42 h
30	7/29/2019	7/31/2019	5.01 h	1.07 h	3.89 h	12.18 h

[32 rows x 6 columns]

Above is an example that shows a CSV of computer usage from a single user spanning weeks. We have the first five entries and the last five. On the bottom, pandas gives us the dimension of the DataFrame.

If you want to access a specified number of the first rows of the DataFrame, you will do this:

```
computerUsage.head(10)
```

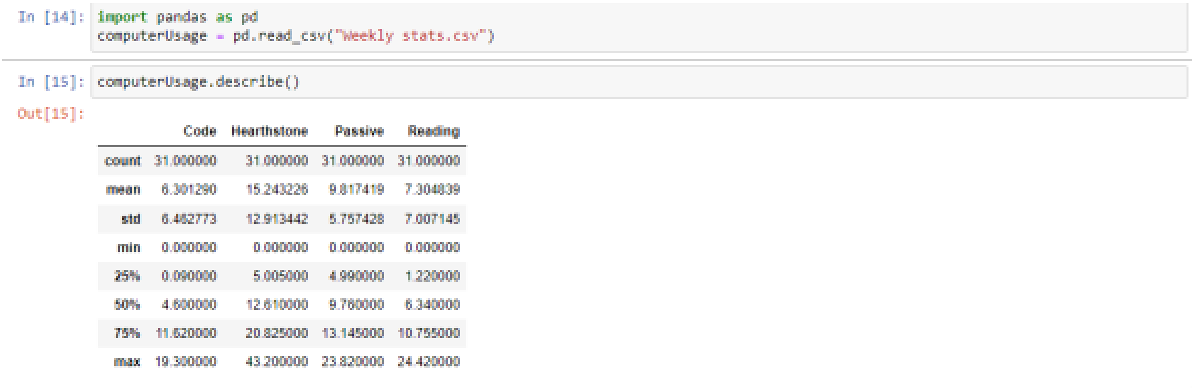
Note: Head is the method used to extract rows from the front, tails is the method used to get the last. So computerUsage.tail(10) would return the last ten in the DataFrame.

Head extracts the first ten rows, so your output will be:

	From	To	Code	Hearthstone	Passive	Reading
0	1/1/2019	1/6/2019	0.09 h	20.60 h	2.13 h	9.88 h
1	1/7/2019	1/13/2019	0.00 h	42.71 h	1.60 h	11.05 h
2	1/14/2019	1/20/2019	0.09 h	14.95 h	0.00 h	21.78 h
3	1/21/2019	1/27/2019	0.00 h	12.61 h	0.09 h	22.91 h
4	1/28/2019	2/3/2019	0.00 h	26.97 h	5.83 h	22.91 h
6	2/4/2019	2/10/2019	0.22 h	33.45 h	3.49 h	10.72 h
7	2/11/2019	2/17/2019	0.01 h	39.99 h	9.52 h	7.54 h
8	2/18/2019	2/24/2019	0.00 h	43.20 h	10.99 h	5.54 h
9	2/25/2019	3/3/2019	0.00 h	10.63 h	19.78 h	6.71 h

[10 rows x 6 columns]

When you get data, it is always good to have a basic overview of the data. The described method helps with that. Here is an illustration of it below:



```
In [14]: import pandas as pd
         computerUsage = pd.read_csv("Weekly stats.csv")

In [15]: computerUsage.describe()

Out[15]:
```

	Code	Hearthstone	Passive	Reading
count	31.000000	31.000000	31.000000	31.000000
mean	6.301290	15.243226	9.817419	7.304839
std	6.462773	12.913442	5.757428	7.007145
min	0.000000	0.000000	0.000000	0.000000
25%	0.090000	5.005000	4.990000	1.220000
50%	4.600000	12.610000	9.760000	6.340000
75%	11.620000	20.825000	13.145000	10.755000
max	19.300000	43.200000	23.820000	24.420000

The data had to be cleaned so that the ‘h’ values that were in the table did not change numerical values into strings. The CSV file was copied into the directory of Jupyter that was being run, that is why the directory(this part “(“Weekly stats.csv”)”) only includes the file name.

Learning More About the Data

We can learn more about the data types that we have by using:

computerUsage.dtypes

The output will be:

From object

To   object

Code   float64

Hearthstone float64

Passive   float64

Reading   float64

dtype: object

Notice: We didn't need to use parentheses to call this function because dtypes is an attribute of the DataFrame.

The results tell us there are no integers, only floats and strings. Strings are referred to as objects in the output. If the DataFrame had integers, they would be indicated like this: int64.

To get a technical summary of the data, use the following method:

computerUsage.info()

The output will be:

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 31 entries, 0 to 30

Data columns (total 6 columns):

Column Non-Null Count Dtype

—- ——— ————— ——-

```
0  From    31 non-null   object
1  To    31 non-null   object
2  Code    31 non-null   float64
3  Hearthstone   31 non-null   float64
4  Passive   31 non-null   float64
5  Reading   31 non-null   float64
dtypes: float64(4), object(2)
memory usage: 1.6+ KB
```

The method above starts by telling us that this is a DataFrame. It also, tells us how many entries are in the DataFrame and gives us the names of the columns. These are now indexed at the far left side. Then, it tells us what sorts of data types are held in each row in the far right column. It then tells us how many data instances of those data types appear by row. At the bottom, we have the amount of RAM occupied by the data.

Writing A DataFrame to A File

When working on a project, your colleagues might want you to send a file in a particular format. To read any files containing data you use the read method. To convert that data into a required format, we use the t0_ method.

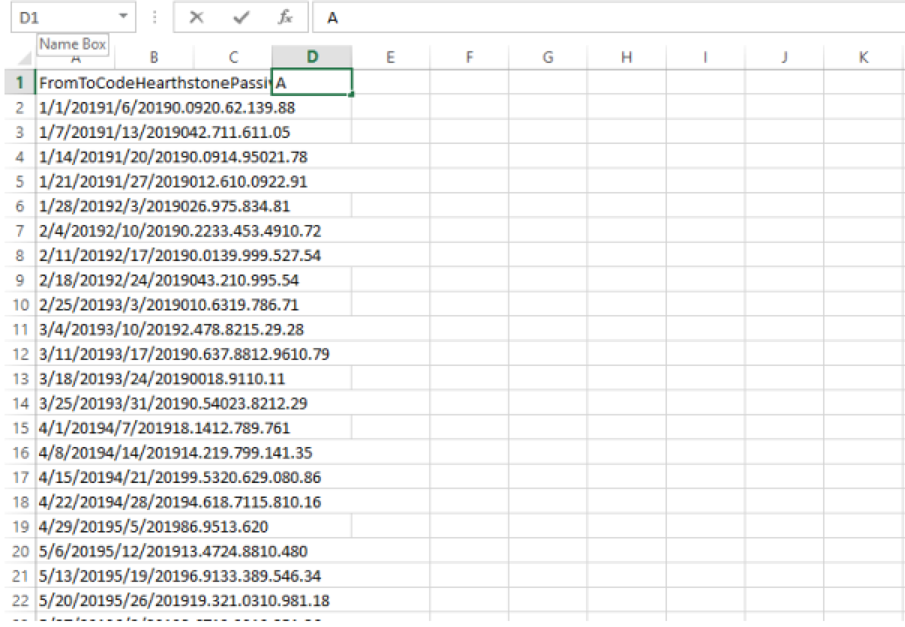
Here is an example of it:

computerUsage.to_excel('computerUsage.xlsx', sheet_name='computer usage', index=False)

The data on your DataFrame will be stored as an excel file if we follow the syntax above. Because we don't need to log indexes, we have said "False." If we do that, the index values will be saved as part of the data in Excel. Let's have a look:

The file is stored in the directory with the "xlsx" extension.

When opened, you can see it contains all the data:



It might look strange, but the data is still accessible and usable. To read, you will only need to type:

```
comp = pd.read_excel('computerUsage.xlsx', sheet_name='computer usage')

comp.head()
```

The data we are using here is available at the end of this chapter as Table A. You can follow the instructions there and begin following along. It is important you do so.

As you can see, the output is similar to the first five rows we showed at the beginning with "h" (hours) missing:

	From	To	Code	Hearthstone	Passive	Reading
0	1/1/2019	1/6/2019	0.09	20.60	2.13	9.88
1	1/7/2019	1/13/2019	0.00	42.71	1.60	11.05
2	1/14/2019	1/20/2019	0.09	14.95	0.00	21.78
3	1/21/2019	1/27/2019	0.00	12.61	0.09	22.91
4	1/28/2019	2/3/2019	0.00	26.97	5.83	22.91

We have seen earlier how we can access specific data in a DataFrame, but the data we used was converted from another format which made it display differently. For instance, columns became row indexes. Let's see what happens when we try the same thing with this data.

We will use the following code to access the first five Heartstone and Code entries, to see if the student was coding more or playing a video games during the first five weeks of classes:

```
firstweek = comp[['Code','Hearthstone']]

firstweek.head()
```

The output is:

	Code	Hearthstone
0	0.09	20.60
1	0.00	42.71
2	0.09	14.95
3	0.00	12.61
4	0.00	26.97

If you wanted to see the last five entries of the data you would type:

```
firstweek.tail()
```

As we can see, the student was doing a whole lot of gaming instead of coding in the first five weeks.

Look at how both columns still occupy the same positions they did at the top as well.

To get the dimensions of the DataFrame, you would type the following:

```
comp[['Hearthstone','Code']].shape
```

Your output will be:

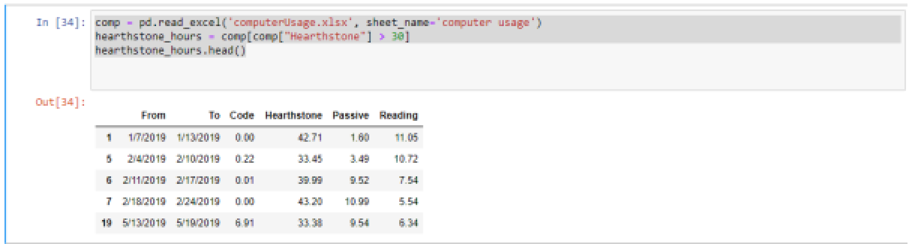
```
(31, 2)
```

This tells you how many columns and rows are in the extracted data. It says thirty-one rows and two columns.

According to the demands of the project you're working on, you might want to filter data that meets a certain criteria. You might want to count the weeks the student spent thirty hours playing Hearthstone and comparing that with how much time they spent on other activities. The example below does that:

```
comp = pd.read_excel('computerUsage.xlsx', sheet_name='computer usage')

hearthstone_hours = comp[comp["Hearthstone"] > 30]

hearthstone_hours.head()
```

In the square brackets, we have pulled the Hearthstone column which gives us a set of numbers. Then, we use a conditional operator (>) to test if each number is greater than thirty. If it is, we will pull rows where that is true. This gives us the table below.


In [34]:
```
comp = pd.read_excel('computerUsage.xlsx', sheet_name='computer usage')
hearthstone_hours = comp[comp["Hearthstone"] > 30]
hearthstone_hours.head()
```
Out[34]:

	From	To	Code	Hearthstone	Passive	Reading
1	1/7/2019	1/13/2019	0.00	42.71	1.60	11.05
5	2/4/2019	2/10/2019	0.22	33.45	3.49	10.72
6	2/11/2019	2/17/2019	0.01	39.99	9.52	7.54
7	2/18/2019	2/24/2019	0.00	43.20	10.99	5.54
19	5/13/2019	5/19/2019	6.91	33.38	9.54	6.34

As we can see, whenever the student games a lot, they struggle to keep up with other activities except maybe reading.

Let's say you want to find out if, in all this time, the student has exceeded a certain number of hours in coding. You can use the following code that checks every row to see if that condition is met.

```
comp = pd.read_excel('computerUsage.xlsx', sheet_name='computer usage')

comp["Code"] > 3.00
```

The code will check if there was a week when the student coded for more than three hours. It returns a Series as opposed to a DataFrame, shown below:

```
0  False

1  False

2  False

3  False

4  False

5  False

6  False

7  False

8  False

9  False

10  False

11  False
```

```
12  False
13  True
14  True
15  True
16  True
17  True
18  True
19  True
20  True
21  True
22  True
23  True
24  True
25  True
26  True
27  True
28  False
29  True
30  True
Name: Code, dtype: bool
```

In real-world scenarios, the Series returned will be thousands of lines long or hundreds. It would be impractical to count each one by hand. If we wanted to show how many of these met our criteria, we can do so with the following code:

```
import pandas as pd

computerUsage = pd.read_excel('op.xlsx')

code_hours = computerUsage[computerUsage['Code'] > 3.00]
```

```
code_hours.shape
```

The output will be: (17, 6)

The output tells us that there are seventeen weeks that the student spent more than three hours coding. We can try to run the same operation with all Python conditional operators like the ones below. They allow us to write code that can comb through the data for more complex characteristics than we can ever do ourselves. Even if the data files were thousands of lines long, it would still have taken our code time to retrieve the information we want.

Operator	Function
==	Equal to
!=	Not Equal
<	Lesser than
>	Greater than
>=	Equal to or greater than
<=	Less than or equal to

To illustrate this, we took data on computer usage spanning more than three years tracking different kinds of activities. Here are the first five and the last ten rows of the data.

Note: You do not have access to this data, but you can try to use what is available at the end of the chapter to follow along in your own capacity.

From	To	Active Energy	Code	Hearthstone	Noise/Admin/Random	Passive	Production	Reading	Work
7/24/2017	7/24/2017	1.92	0.00	0.00	0.00	0.00	0.97	0.00	0.96
7/25/2017	7/25/2017	6.84	1.83	0.25	0.71	0.00	4.02	0.02	0.92
7/26/2017	7/26/2017	7.68	0.94	0.00	0.07	0.00	2.83	0.58	0.04
7/27/2017	7/27/2017	6.55	0.00	0.00	0.00	0.00	1.25	0.00	0.60
12/11/2020	12/11/2020	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12/12/2020	12/12/2020	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12/13/2020	12/13/2020	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12/14/2020	12/14/2020	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12/15/2020	12/15/2020	0.10	0.00	0.00	0.00	0.00	0.02	0.00	0.00

In the last five days, we see that there was no activity at all. Let us count the days that the user read more than one hour a day using the code from earlier and let us see how long it takes the code to complete the task.

```
In [32]: Usage = pd.read_excel('UsageOverYrs.xlsx')
         read_hours = Usage[Usage['Reading'] > 1.00]
         %timeit read_hours.shape
         read_hours.shape

         1.97 µs ± 75.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

Out[32]: (225, 10)
```

The code tells us that in all that time, the user read for more than an hour in 225 days. Look at how long it took for the code to complete the task. You can not imagine gleaning that information as quickly as we did with Python. Interestingly, Python can do this with even larger sets of data with even more complex logic. You can also factor in that this code was run on a much slower machine.

Let's say you want to find out how much energy a student has on days they read for three hours or more. You will use the following code to find out. This code matches the column data with the row that matches our criteria. For instance, if this was a data set of students at a university and we wanted the names of the students who are twenty, we would use code like this.

```
students_20 = students.loc[students["Age"] == 20, "Name"]
```

So, in our reading example, the code will be like this:

```
read_hours = Usage.loc[Usage['Reading'] > 3.00 ,'Active Energy']
```

The output from the data was fifty-three, showing us that there are fifty-three rows in the DataFRame.

Here were the first five results of the data:

427 6.268611

428 4.928056

429 5.991111

433 5.494167

```
436  4.469167
Name: Active Energy, dtype: float64
```

The data so far suggests that the student needs at least 4.5 hours of active energy to read for more than three hours.

Selecting Data

In some cases, you want to select data from point A to B, whether it be columns or rows of data. Our years old data has more than 1200 days in it. We may be interested in the days between 500 to 600 for columns two to four.

Here's the code for that:

```
five_six = Usage.iloc[500:601, 2:5]

five_six
```

Below are the first five and the last five in the section, showing only the three columns we wanted.

Active Energy Code Hearthstone

500 2.167500 0.0 0.0

501 0.731944 0.0 0.0

502 4.881667 0.0 0.0

503 0.155556 0.0 0.0

504 0.000000 0.0 0.0

Active Energy Code Hearthstone

596 10.151389 1.642222 1.481944

597 6.120278 0.640556 0.412222

598 8.585556 1.075278 1.415833

599 6.008333 1.128333 0.000000

600 5.645556 0.218611 0.000000

When you look at the syntax of both these selections, you will notice the loc/iloc operators before the square brackets. These are required. Square brackets alone will not select a subset of data using our expressions. In both examples, before the comma, we selected a row we wanted, and in the part after the comma, we selected a column we wanted. The loc operator is best used in instances where you will use a condition expression like in our reading-hours example. The iloc operator, which we used to select the days between 500 and 600, has taken no conditional operators. Instead, we have just used numbers

and colon to select the parts of the data we are interested in. Both always go before the square brackets.

They also have another function. They can assign values to the selected data if we desire. Take the hundred days we selected above. If we wanted to assign the number ten to the first ten elements of the code column, we would write the following code.

```
Usage.iloc[0:10, 3] = 10

Usage.head()
```

Note: Assigning the code to a variable will not work.

The output for the first five rows of this code is:

	From	To	Active Energy	Code	Hearthstone	Noise/Admin/Random	Passive	Production	Reading	Work
0	7/24/2017	7/24/2017	1.924444	10	0	0	0	0.965556	0	0.960556
1	7/25/2017	7/25/2017	6.836111	10	0.248889	0.707222	0	4.024722	0.016389	0.921111
2	7/26/2017	7/26/2017	7.682222	10	0	0.068333	0	2.834167	0.578056	0.040556
3	7/27/2017	7/27/2017	6.554444	10	0	0.003333	0	1.247778	0	0.603889
4	7/28/2017	7/28/2017	3.916111	10	0	0	0	0.053889	0	0.041111

Suddenly, the student has spent the first ten days coding for ten hours.

Earlier, we looked at how the square bracket in conjunction with conditional operators can filter out certain kinds of data from the table. The isin() method works the same way, but it is more sophisticated in what it allows you to do. For instance, you may be interested in rows when the student either didn't read or spend an hour reading. To do that, you would need to use the following code.

```
reading_01 = Usage[Usage["Reading"].isin([0, 1.0])]
```

The output of the first five rows would be as follows:

From	To	Active Energy	Code	Hearthstone	Noise/Admin/Random	Passive	Production	Reading	Work
7/24/2017	7/24/2017	1.924444444	10	0	0	0	0.9655555556	0	0.9605555556
7/27/2017	7/27/2017	6.554444444	10	0	0.003333333333	0	1.247777778	0	0.6038888889
7/28/2017	7/28/2017	3.916111111	10	0	0	0	0.05388888889	0	0.04111111111
7/30/2017	7/30/2017	6.076666667	10	0	0	0	2.275833333	0	1.373055556
8/2/2017	8/2/2017	0	10	0	0	0	0	0	0.001666666667

You can tell from the date columns on the left that the dates do not follow chronologically. The function has only retrieved dates the student didn't read or read for an hour. In our first five dates, it only shows dates he did not read.

Isin() is the equivalent of the following code:

```
reading_01 = Usage[(Usage["Reading"] == 0) | (Usage["Reading"] == 1.0)]
```

When you chain multiple conditional statements like this, each condition must be put inside parentheses. Also, pandas won't accept the or/and operators, to use them, you will need to use the | for the or and the & for the and.

Creating Plots

It would be nice to display all this data from computer usage as a graph or plot so that we can glean more insights. pandas can work with Matplotlib to do this. We are going to discuss ways you can do this in this section. We will need to import pandas and Matplotlib together before we can start using it.

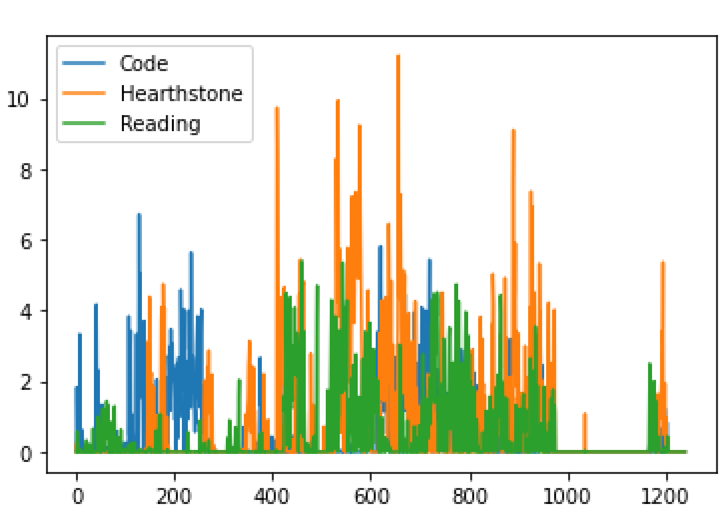
```
import pandas as pd

import matplotlib.pyplot as plt
```

If we were investigating if gaming has any impact on the student's productivity, it might do us well to focus on three items in our data: Hearthstone, Reading, and Coding. Now, the data we will be working on within this section has been edited to look like this:

Tip: To make yours look like this edit it in a spreadsheet program like Google sheets or Excel, you don't have access to this data, but you can practice with the one provided. Table A at the end of the chapter would be a good choice.

Days	Code	Hearthstone	Reading
7/24/2017	0	0	0
7/25/2017	1.83	0.25	0.02
7/26/2017	0.94	0	0.58
7/27/2017	0	0	0
7/28/2017	0.01	0	0

We need to use set_index to convert the first column into an index or when we plot the data, we will not have a presentation of time on the x-axis as shown below.



All we get are numbers, not dates.

We convert the days column like this:

```
Usage = pd.read_excel('UsageOverYrs.xlsx')

dataUsage = Usage.set_index('Days')

dataUsage.head()
```

We pull the data first, then convert it and show the result to see if we were successful. Now the data has dates as the index.

To make a quick plot of the data, all we have to type:

dataUsage.plot()

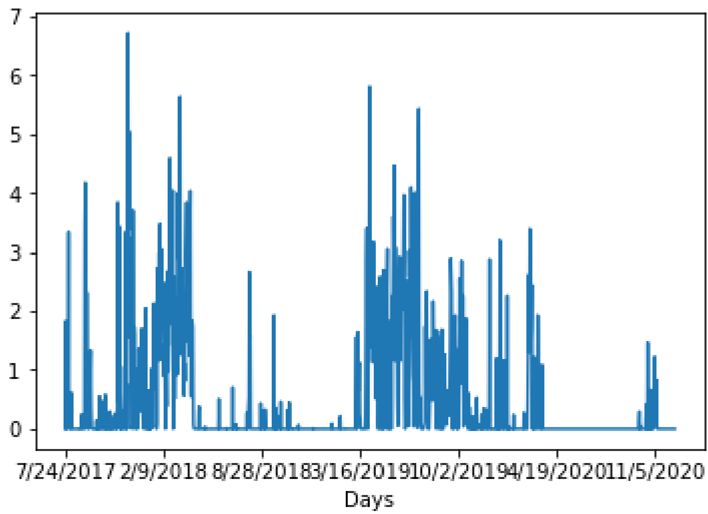
We get the following graph:

```
In [13]: Usage = pd.read_excel('UsageOverYrs.xlsx')
         dataUsage = Usage.set_index('Days')
         dataUsage.plot()

Out[13]: <AxesSubplot:xlabel='Days'>
```

If you wanted to plot one column on the table, you would use the following syntax.

dataUsage[‘Code’].plot()

That would give you the following result:



This is a combination of the selection methods we learned earlier and the plot() method.

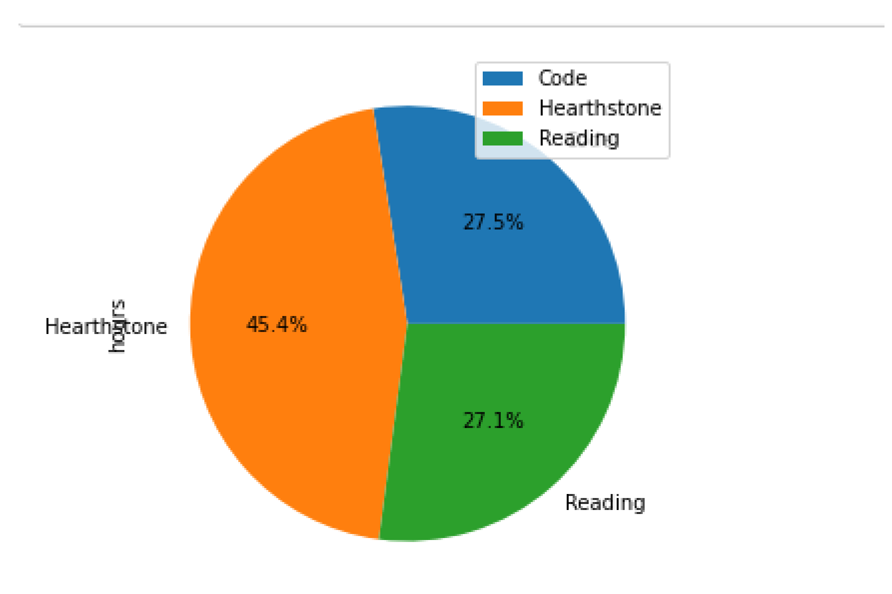
You want to use a scatterplot to compare two columns in the table. Take Reading and Hearthstone for an example. You would use the following code:

```
dataUsage.plot.scatter(x="Reading",

y= "Hearthstone",

alpha=0.5)
```

We can choose from many plot types that are available to us. In the example above, we have used scatter. There are area, bar, barh, box, density, hist, kde, and pie. In some circumstances, you might be required to make some adjustments to present some data. For instance, if I wanted to display the usage data in a pie chart, I would need the total hours of each activity in the given time and plot it like this:

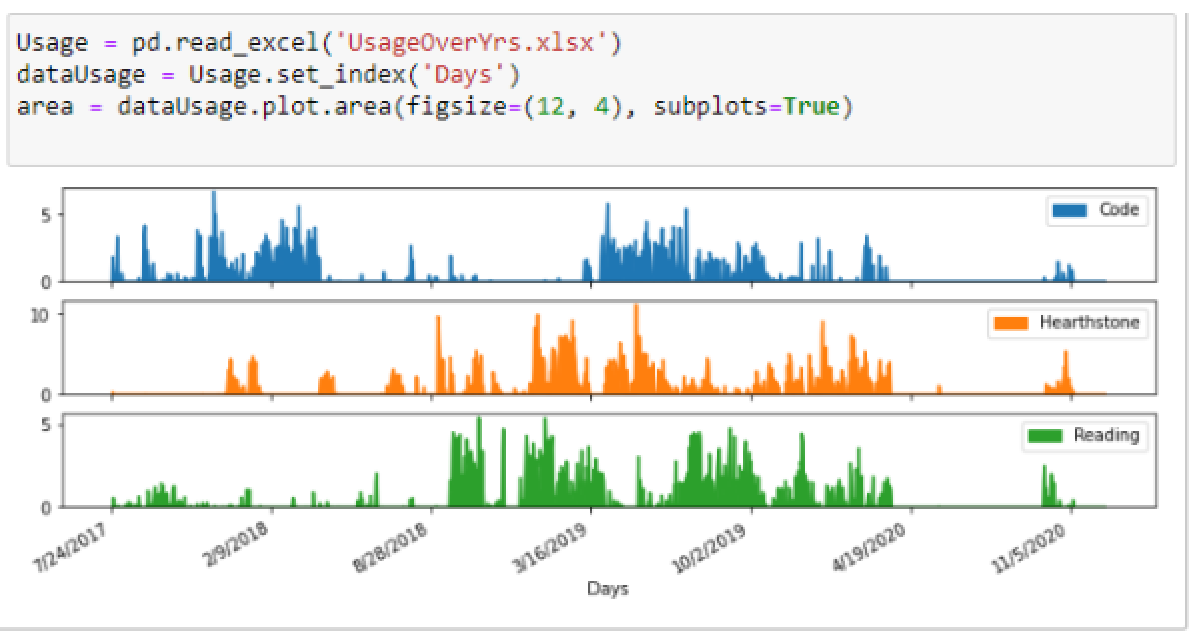
```
df = pd.DataFrame({'hours': [603.41, 995.68 , 594.07]},

index=['Code', 'Hearthstone', 'Reading'])

plot = df.plot.pie(y='hours', figsize=(5, 5), autopct='%1.1f%%')
```

These are the results:



As shown in the line graph, we can see that gaming had taken the most time while reading and coding shared close to equal time.

We might want to look at the data on separate plots to gain more insights from the timeline. We do that using subplots, like this:



All of these tasks are enabled by Matplotlib which is a Python library that allows us to visualize and customize data. This means anything you would do in Matplotlib, you would be able to do together with pandas.

Creating New Columns

We can make new columns from existing columns. For instance, we might decide to have a “Productive” column that combines values of Coding and Reading columns since the two can be thought of as productive activities. We would use the following code.

```
Usage = pd.read_excel('UsageOverYrs.xlsx')

dataUsage = Usage.set_index('Days')

dataUsage["Productive"] = dataUsage["Reading"] +
dataUsage["Code"]

dataUsage.head(10)
```

The third line is where the magic happens. We start by giving our new column a name in the square bracket on the left, and then we assign to it the values of the two columns on the right after the assignment operator. The code will go between Code and Reading columns, adding each row together. If we had used a multiplication operator, it would have multiplied the values together. The sum of each added to the new “Productive” column.

Note: User Table A data for this exercise.

Here is the output of this code:

Code	Hearthstone	Reading	Productive
0	0	0	0

1.831389	0.248889	0.016389	1.847778
0.942778	0	0.578056	1.520833
0	0	0	0
0.012778	0	0	0.012778
0.091389	0	0.034167	0.125556
0.904722	0	0	0.904722
3.3375	0	0.0125	3.35
0.140556	0	0.080278	0.220833
0	0	0	0

We can also create a new column that gives us the multiplied values of another column. Making a column where the values or the reading column are double would be as easy as this:

```
dataUsage["Doubled Reading"] = dataUsage["Reading"] * 2
```

The output is:

Code	Hearthstone	Reading	Doubled Reading
0	0	0	0
1.831389	0.248889	0.016389	0.032778
0.942778	0	0.578056	1.156111
0	0	0	0
0.012778	0	0	0

You can use other numerical and logical operators to work with the data and save the results of those calculations into another column.

We can also rename the column names. Here is an example of that:

```
computerUsageNew = dataUsage.rename(

columns={"Code": "Programming",

"Hearthstone": "Gaming",

"Reading": "Books/Research"})

computerUsageNew.head()
```

Programming	Gaming	Books/Research
0	0	0
1.831389	0.248889	0.016389
0.942778	0	0.578056
0	0	0
0.012778	0	0

As you can see, the columns in this DataFrame make more sense. If someone doesn't know what Hearthstone is, they will not understand what the column is measuring, and Code is also a vague label.

You can also manipulate the column names in other ways, like changing them to all caps or lowercase. Like this:

```
computerUsageNew = dataUsage.rename(

columns={"Code": "Programming",

"Hearthstone": "Gaming",

"Reading": "Books/Research"})

columnsUp = computerUsageNew.rename(columns=str.upper)
```

This will give us the new column names in uppercase. Of course, you can decide to enter upper case labels manually. In a situation where you are dealing with many columns, it might take more time. So functions like these help make things easier.

PROGRAMMING	GAMING	BOOKS/RESEARCH
0	0	0
1.831389	0.248889	0.016389
0.942778	0	0.578056
0	0	0
0.012778	0	0

Adding and Removing Columns

If you think of DataFrames as a dict, adding and removing columns will come naturally to you because the syntax is the same.

Take our computer usage DataFrame (the slimmed-down version). We may want to add an “Away” column for tracking when the student is sleeping or busy with other activities. We may get our Away data from subtracting the values of all the other columns from hours in a day. It is not a perfect measure, but it will give us a healthy estimate. Our code for that will look like this:

```
dataUsage['Away'] = 24 - dataUsage['Code'] + dataUsage['Reading']
+ dataUsage['Hearthstone']
```

Here are the first ten rows of that code:

Code	Hearthstone	Reading	Away
0	0	0	24
1.831389	0.248889	0.016389	22.43389
0.942778	0	0.578056	23.63528
0	0	0	24
0.012778	0	0	23.98722
0.091389	0	0.034167	23.94278
0.904722	0	0	23.09528
3.3375	0	0.0125	20.675
0.140556	0	0.080278	23.93972
0	0	0	24

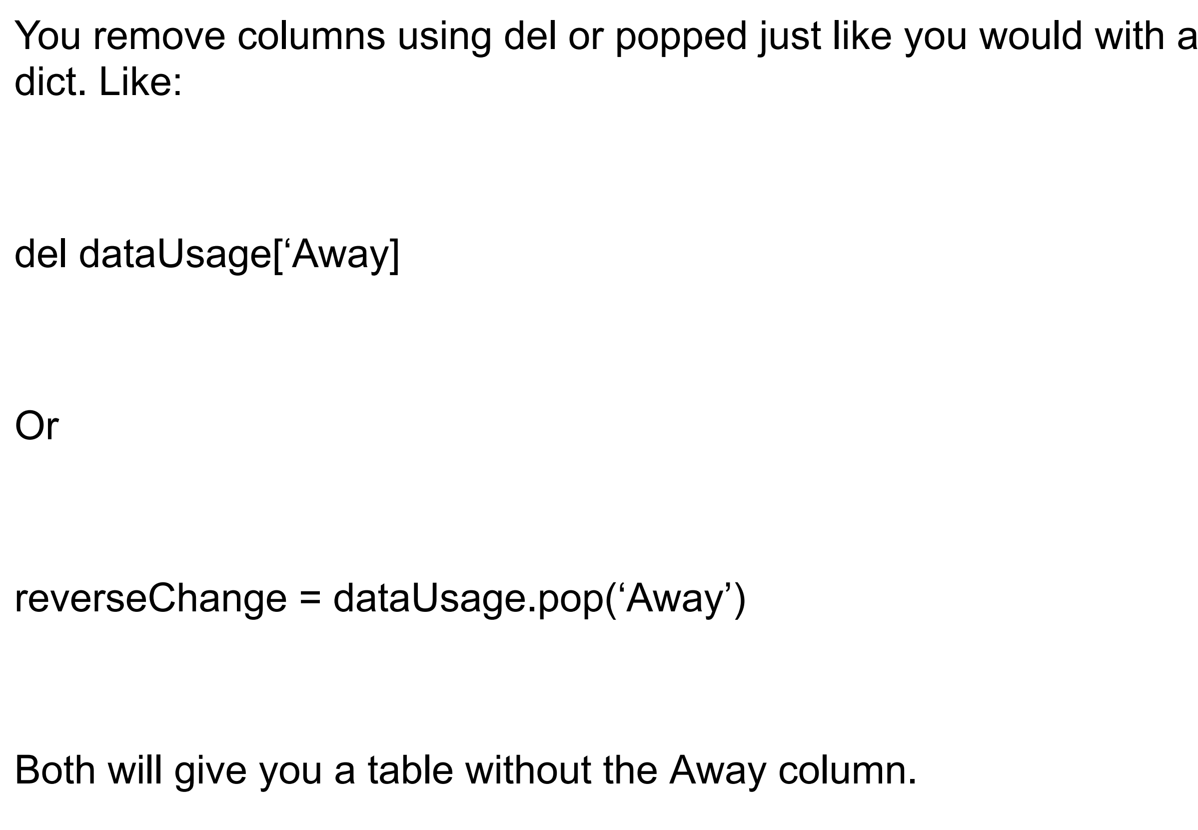
When we add a new column, we have to fill it. And this shows us how we can do that.

So when I give a column a scalar value like below, that value will fill the entire column:

usage[‘Away’] = ‘Unreliable’

The output will

Code	Hearthstone	Reading	Away
0	0	0	Unreliable
1.831389	0.248889	0.016389	Unreliable
0.942778	0	0.578056	Unreliable
0	0	0	Unreliable
0.012778	0	0	Unreliable



You remove columns using del or popped just like you would with a dict. Like:

del dataUsage[‘Away]

Or

reverseChange = dataUsage.pop(‘Away’)

Both will give you a table without the Away column.

Doing Statistics

We will discuss how to do some simple statistics in this section using pandas. Earlier, I showed how you can get a general summary of the DataFrame statistics using the describe method. We will show you how you would make other calculations of the data.

If you want to find the mean of a particular column, you will do this:

```
dataUsage[‘Reading’].mean()
```

The output is: 0.4786995254722894

You might want to find the median for Code and Reading columns. This is how you would go about it:

```
dataUsage[['Code', 'Reading']].median()
```

The output from our data sample is:

```
Code   0.0

Reading   0.0

dtype: float64
```

When you look closely, you can see that what is returned is a DataFrame.

You can use describe on columns of your own interest, like this:

```
dataUsage[['Code', 'Reading', 'Hearthstone']].describe()
```

The output on our data is:

	Code	Reading	Hearthstone
count	1241	1241	1241
mean	0.486232	0.4787	0.80232
std	0.962752	0.960654	1.597422
min	0	0	0
25%	0	0	0
50%	0	0	0
75%	0.475	0.462222	1.015278
max	6.713611	5.389722	11.208333

In some cases, you may not want to use the describe method. You might want to define the information you want from each column. You can use the agg method for that. Here is the agg method in action:

```
dataUsage.agg({'Hearthstone': ['min', 'max', 'skew'],

'Reading': ['min', 'max', 'mean']})
```

This is the output of the calculations:

	Hearthstone	Reading
max	11.208333	5.3897223
mean	NaN	0.478700
min	0.0000002	0.000000
skew	2.637625	NaN

For our next examples, we are going to look at computer usage data from the year 2019. To participate, use Table B in the resources section at the end of this chapter. The data is on a weekly basis instead of days.

It looks like this:

Note: Table Bs Month's column is in numbers instead of strings. It is intentional, we are trying to illustrate something.

Code	Hearthstone	Reading	Month
0	20.6	9.88	Jan
0	42.71	11.05	Jan
0.09	14.95	21.78	Jan
0	12.61	22.91	Jan
0	26.97	4.81	Jan
0.22	33.45	10.72	Feb
0.01	39.99	7.54	Feb
0	43.2	5.54	Feb
0	10.63	6.71	Feb

Our rows cover 53 weeks instead of 52 to fully cover the dates between 1 Jan 2019 and 31 Dec. So the last week in the table starts the year 2020.

Faced with a DataFrame like this, we might want to find the mean of all these activities by month. That is where the groupby method comes in.

Here is an example:

```
weeklyData = pd.read_excel('weekly.xlsx')

weeklyData.groupby('Month').mean()
```

Our output will be:

Month	Code	Hearthstone	Reading
Apr	11.24939	15.76844	0.672833
Aug	6.303056	5.132153	10.03028
Dec	1.502056	8.988222	5.132333
Feb	0.055833	31.81813	7.626875
Jan	0.017833	23.56694	14.08711
Jul	6.37	3.027556	13.16456
Jun	13.77431	6.486389	2.040347
Mar	4.258958	4.176181	10.61986
May	12.10278	23.16799	2.194167
Nov	0.526111	7.864167	3.972083
Oct	4.618889	7.696389	1.974028
Sep	5.839222	0.871	10.44817

The indexes are not presented chronologically, but we have the information we want. They are sorted alphabetically. If we must have the DataFrame sorted chronologically, we can do so by using numbers to present months like this:

Code	Hearthstone	Reading	Month
0	20.6	9.88	1
0	42.71	11.05	1
0.09	14.95	21.78	1
0	12.61	22.91	1
0	26.97	4.81	1
0.22	33.45	10.72	2
0.01	39.99	7.54	2
0	43.2	5.54	2
0	10.63	6.71	2

Our results will be ordered chronologically like this:

Month	Code	Hearthstone	Reading
1	0.017833	23.56694	14.08711
2	0.055833	31.81813	7.626875
3	4.258958	4.176181	10.61986
4	11.24939	15.76844	0.672833
5	12.10278	23.16799	2.194167
6	13.77431	6.486389	2.040347
7	6.37	3.027556	13.16456
8	6.303056	5.132153	10.03028
9	5.839222	0.871	10.44817
10	4.618889	7.696389	1.974028
11	0.526111	7.864167	3.972083
12	1.502056	8.988222	5.132333

In some cases, you might want to pull only one activity instead of the entire DataFrame. Here is how you would do that:

```
weeklyData.groupby('Month')['Reading'].mean()
```

In the code above, I retrieved the mean of reading data alone, grouping it by months. I have selected it using the square bracket while I grouped the elements using parenthesis.

The output is:

Month	Reading
1	14.08711
2	7.626875
3	10.61986
4	0.672833
5	2.194167
6	2.040347
7	13.16456
8	10.03028
9	10.44817
10	1.974028
11	3.972083
12	5.132333

We can group elements by more than one trait. So far, we have used months. We can also try seasons to see if there are seasonal differences we should be aware of. Let us label the seasons A for Winter up to D for Autumn. The reason for this is that we saw our data gets arranged alphabetically when strings are involved. Our Dataframe would look like this:

Note: Table B looks like this already.

Code	Hearthstone	Reading	Month	Seasons
0	20.6	9.88	1	A-WINTER
0	42.71	11.05	1	A-WINTER
0.09	14.95	21.78	1	A-WINTER
0	12.61	22.91	1	A-WINTER
0	26.97	4.81	1	A-WINTER
0.22	33.45	10.72	2	A-WINTER
0.01	39.99	7.54	2	A-WINTER
0	43.2	5.54	2	A-WINTER
0	10.63	6.71	2	A-WINTER

Now, let us group Reading by Seasons and Months respectively like this:

```
weeklyData.groupby(['Seasons','Month'])['Reading'].mean()
```

Our results should look like this:

Seasons	Month	Reading
A-WINTER	1	14.08711
2	7.626875	
12	5.132333	
B-SPRING	3	10.61986
4	0.672833	
5	2.194167	
C-SUMMER	6	2.040347
7	13.16456	
8	10.03028	
D-AUTUMN	9	10.44817
10	1.974028	
11	3.972083	

If you get something different, check the order you have entered Seasons and Month in the square brackets.

To get all of the DataFrame values, you need to omit the selecting square bracket. Like this:

```
weeklyData.groupby(['Seasons','Month']).mean()
```

The output will be:

Seasons	Month	Code	Hearthstone	Reading
A-WINTER	1	0.017833	23.56694	14.08711
2	0.055833	31.81813	7.626875	
12	1.502056	8.988222	5.132333	
B-SPRING	3	4.258958	4.176181	10.61986
4	11.24939	15.76844	0.672833	
5	12.10278	23.16799	2.194167	
C-SUMMER	6	13.77431	6.486389	2.040347
7	6.37	3.027556	13.16456	
8	6.303056	5.132153	10.03028	
D-AUTUMN	9	5.839222	0.871	10.44817
10	4.618889	7.696389	1.974028	
11	0.526111	7.864167	3.972083	

The count-value method can be used to find the number of entries by category. You may want to find it if all the seasons have the same number of entries or if some have more weeks than others, and that could explain some of the differences in the data. Before we get to that, let’s get a sense of what this function does.

We will use it in the Reading category and see what happens. Here is the code we ran below:

#To get a clear idea, we had to convert values in the reading column into integers.

weeklyData['Reading'] = weeklyData['Reading'].astype(int)

#The we ran the value_counts method on the column

weeklyData['Reading'].value_counts()

This was the output we got:

0 10

1 7

7 5

9 5

10 4

11 3

6 3

3 3

15 2

5 2

2 2

14 1

12 1

22 1

17 1

4 1

21 1

24 1

Name: Reading, dtype: int64

The data tells you how many times values in the left appear in the data. So there are ten weeks that the student did not read or read less than an hour. For seven weeks, he read for an hour or more until we got to the end where it tells us the student has one week where he spent twenty four hours or more reading.

Now, let's group the reading data by seasons and see what comes up:

```
#we convert from floats to integers again

weeklyData['Reading'] = weeklyData['Reading'].astype(int)

#we apply the count here

results = weeklyData.groupby('Seasons')['Reading'].count()
```

Remember, we are trying to find out how many weeks or entries belonged to what seasons. We should get seasons in the index, and the number of entries in the right like our output shows:

Seasons	Reading
A-WINTER	14
B-SPRING	13
C-SUMMER	13
D-AUTUMN	13

You can imagine doing this to find out all sorts of traits like how many people bought different types of tickets to a particular concert. You can replace the season column with ticket tiers like regular, gold, platinum, diamond, and VIP. You just have to be creative.

Combining Tables

In the real world, you might find yourself working with multiple data sources that you would need to combine. We will show how we can do that. We will introduce another data table that can help us get a better understanding of the student's computer usage in the year 2019. The table will represent weekly data but focusing on another set of activities that will give us a wider appreciation of the student's activities. This is Table C in the Resources section.

It looks like this:

Away	Random/Admin	Work
24.8	11.59	0
24.09	22.25	0.05
18.65	31.92	1.61
20.9	18.03	15.23
23	20.31	7.01
27.75	25.13	11.71
18.35	17.14	8.72
20.85	19.02	7.7
34.13	22.91	8.79

The hours spent working on assignments and other school-related things and the hours spent doing random things like watching YouTube videos or answering emails are logged into this table.

To remind you, our original data looks like this:

Code	Hearthstone	Reading	Month	Seasons
0	20.6	9.88	1	A-WINTER
0	42.71	11.05	1	A-WINTER
0.09	14.95	21.78	1	A-WINTER
0	12.61	22.91	1	A-WINTER
0	26.97	4.81	1	A-WINTER
0.22	33.45	10.72	2	A-WINTER
0.01	39.99	7.54	2	A-WINTER
0	43.2	5.54	2	A-WINTER
0	10.63	6.71	2	A-WINTER

Now let’s explore how we can combine the data to create a complete picture.

The concatenate method is the one way we can combine these tables. By default, it will add the second table at the last row first, but you can choose if you want the table to be added row-wise or column-wise.

Since we will not show the whole DataFrame here, we will use the shape method to find out if the new DataFrame is larger.

Here is the code for declaring tables and checking their lengths.

```
table1 = pd.read_excel('weeklyR.xlsx')

table2 = pd.read_excel('weeklyT2.xlsx')

print('First DataFrame is' ,table1.shape)

print('Second DataFrame is', table2.shape)
```

The output was:

```
First DataFrame is (53, 5)

Second DataFrame is (53, 3)
```

The code correctly tells us they have fifty-three rows each, but the first table has five columns and the second has three.

Now, to concatenate:

```
table1 = pd.read_excel('weeklyR.xlsx')

table2 = pd.read_excel('weeklyT2.xlsx')

newTable = pd.concat([table1, table2], axis=0)

newTable.head()
```

We will get the following output:

	Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
0	0	20.598333	9.8775	1	A-WINTER	NaN	NaN	NaN
1	0	42.7075	11.048889	1	A-WINTER	NaN	NaN	NaN
2	0.089167	14.953333	21.784167	1	A-WINTER	NaN	NaN	NaN
3	0	12.609167	22.914167	1	A-WINTER	NaN	NaN	NaN
4	0	26.966389	4.810833	1	A-WINTER	NaN	NaN	NaN

We can see the new columns are added, but they have no values in them. This is because their values are added after the last row of the last table, as we discussed.

Now let's check the size of the new DataFrame:

```
newTable.shape()
```

The output gives us double the rows we had:

```
(106, 8)
```

Rows forty-nine to fifty-seven look like this:

49	0.009722	7.311667	7.174444	12	A-WINTER	NaN	NaN	NaN
50	1.243056	9.056667	2.830556	12	A-WINTER	NaN	NaN	NaN
51	3.204444	5.182778	0.401944	12	A-WINTER	NaN	NaN	NaN
52	0	16.67278	0	12	A-WINTER	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	NaN	24.79667	11.58806	0
1	NaN	NaN	NaN	NaN	NaN	24.08917	22.24861	0.049722
2	NaN	NaN	NaN	NaN	NaN	18.64694	31.91833	1.611944
3	NaN	NaN	NaN	NaN	NaN	20.90222	18.03278	15.22861

When concatenating the tables, we used the default axis=0 setting. If we want to add the new table column-wise so that the rows remain the same size, we can change it to 1. The axis describes two directions that are inherent in the DataFrames. The 0 axis runs vertically, downward, and the 1 axis runs horizontally from left to right. Let's have a look at it:

```
newTable = pd.concat([table1, table2], axis=1)

newTable.head()
```

The output is:

	Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
0	0	20.59833	9.8775	1	A-WINTER	24.79667	11.58806	0
1	0	42.7075	11.04889	1	A-WINTER	24.08917	22.24861	0.049722
2	0.089167	14.95333	21.78417	1	A-WINTER	18.64694	31.91833	1.611944
3	0	12.60917	22.91417	1	A-WINTER	20.90222	18.03278	15.22861
4	0	26.96639	4.810833	1	A-WINTER	22.99833	20.30917	7.011944

As you can see, the new table is added to the right, and we can confirm that the new rows were added with the following code:

```
newTable.shape
```

Output is:

```
(53, 8)
```

In some instances, it is helpful to know where particular data in the new DataFrame comes from. To have this information, we will need to use the keys argument in the concat method. The keys arguments allow us to mark the index where values in a table come from.

Let’s combine and our tables again, this time we will use the argument:

```
table1 = pd.read_excel('weeklyR.xlsx')

table2 = pd.read_excel('weeklyT2.xlsx')

newTable = pd.concat([table1, table2], axis=1

keys=['tab1', 'tab2'])

newTable.head()
```

Our output is this:

	tab1	tab2						
	Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
0	0	20.598333	9.8775	1	A-WINTER	24.796667	11.588056	0
1	0	42.7075	11.048889	1	A-WINTER	24.089167	22.248611	0.049722
2	0.089167	14.953333	21.784167	1	A-WINTER	18.646944	31.918333	1.611944
3	0	12.609167	22.914167	1	A-WINTER	20.902222	18.032778	15.228611
4	0	26.966389	4.810833	1	A-WINTER	22.998333	20.309167	7.011944

As you can see, data from the first table is labeled tab1, and data from the second is labeled tab2 adding more clarity to the DataFrame.

Earlier, when we combined the new table, we saw some missing values that gave us NaN. If this were a larger data set, it would have been very difficult to know if this were the case. It is important to be able to know if there are any missing data in our DataFrame, and if there is, how many elements are missing:

When we combined the tables using the default (axis=0), we got this:

49	0.009722	7.311667	7.174444	12	A-WINTER	NaN	NaN	NaN
50	1.243056	9.056667	2.830556	12	A-WINTER	NaN	NaN	NaN
51	3.204444	5.182778	0.401944	12	A-WINTER	NaN	NaN	NaN
52	0	16.67278	0	12	A-WINTER	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	NaN	24.79667	11.58806	0
1	NaN	NaN	NaN	NaN	NaN	24.08917	22.24861	0.049722
2	NaN	NaN	NaN	NaN	NaN	18.64694	31.91833	1.611944
3	NaN	NaN	NaN	NaN	NaN	20.90222	18.03278	15.22861

To find out how many values are missing in the table would take quite a long time. Luckily, the following function does that perfectly:

```
newTable = pd.concat([table1, table2], axis=0)

newTable.isna().sum()
```

The output is detailed:

```
Code    53

Hearthstone   53

Reading    53

Month    53

Seasons    53

Away    53

Random/Admin   53

Work    53

dtype: int64
```

NaN values may get in the way of doing some calculations. In that case, we may want to fill them with predetermined values. In this case, we may want the value to be 0. Fillna is a pandas method that allows us to fill NA values with data. Here is how we would do that:

```
newTable.fillna(0)
```

The output from this would be:

	Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
0	0	20.598333	9.8775	1	A-WINTER	0	0	0
1	0	42.7075	11.048889	1	A-WINTER	0	0	0

2	0.089167	14.953333	21.784167	1	A-WINTER	0	0	0
3	0	12.609167	22.914167	1	A-WINTER	0	0	0
4	0	26.966389	4.810833	1	A-WINTER	0	0	0
...	...	...	...	...	...	...	...	...
48	0	0	0	0	0	8.023889	18.531667	35.808333
49	0	0	0	0	0	12.909167	32.3575	17.559722
50	0	0	0	0	0	11.698611	26.2175	29.279444
51	0	0	0	0	0	15.596389	23.173056	20.033333
52	0	0	0	0	0	1.918889	3.990556	0.289722

As you can see, all of our NaN values have been replaced with the value specified in the parentheses of our fillna method.

We may not want to fill the DataFrame with a predetermined value. We want the box below or above to be filled with the last recorded value. The fill method allows you to do this.

Here’s an example:

```
newTable.fillna(method='ffill')
```

Our output is:

		Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
tab1	0	0	20.598333	9.8775	1	A-WINTER	NaN	NaN	NaN
1	0	42.7075	11.048889	1	A-WINTER	NaN	NaN	NaN	
2	0.089167	14.953333	21.784167	1	A-WINTER	NaN	NaN	NaN	
3	0	12.609167	22.914167	1	A-WINTER	NaN	NaN	NaN	
4	0	26.966389	4.810833	1	A-WINTER	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	...
tab2	48	0	16.672778	0	12	A-WINTER	8.023889	18.531667	35.808333
49	0	16.672778	0	12	A-WINTER	12.909167	32.3575	17.559722	
50	0	16.672778	0	12	A-WINTER	11.698611	26.2175	29.279444	
51	0	16.672778	0	12	A-WINTER	15.596389	23.173056	20.033333	
52	0	16.672778	0	12	A-WINTER	1.918889	3.990556	0.289722	

The area with red numbers should be NaN, like the area that is in purple. The area in red, down to the last row, is filled with the last value of the first table at row fifty three. Because the columns on the top right have no value above them, they cannot be assigned a value although they are empty. The direction of the method is forward. That is what the ‘ffill’ arguments passed mean: forward fill. If we wanted areas in the purples to be filled with the values of the first row of the tab2, we would do this:

```
newTable.fillna(method='bfill')
```

Our first five rows will look like this:

		Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
tab1	0	0	20.598333	9.8775	1	A-WINTER	24.796667	11.588056	0
1	0	42.7075	11.048889	1	A-WINTER	24.796667	11.588056	0	
2	0.089167	14.953333	21.784167	1	A-WINTER	24.796667	11.588056	0	
3	0	12.609167	22.914167	1	A-WINTER	24.796667	11.588056	0	
4	0	26.966389	4.810833	1	A-WINTER	24.796667	11.588056	0	

See how the area in purple has been filled, but the area that was in red will not be filled because we used bfill.

You can limit the number of rows you want to be filled at a time by adding a limit argument as illustrated with the code:

```
newTable.fillna(method='bfill', limit=3)
```

Looking at row forty-nine to fifty-five created by this function, we see this:

		Code	Hearthstone	Reading	Month	Seasons	Away	Random/Admin	Work
tab1	49	0.009722	7.311667	7.174444	12	A-WINTER	NaN	NaN	NaN
50	1.243056	9.056667	2.830556	12	A-WINTER	24.796667	11.588056	0	
51	3.204444	5.182778	0.401944	12	A-WINTER	24.796667	11.588056	0	
52	0	16.672778	0	12	A-WINTER	24.796667	11.588056	0	
tab2	0	NaN	NaN	NaN	NaN	NaN	24.796667	11.588056	0
1	NaN	NaN	NaN	NaN	NaN	24.089167	22.248611	0.049722	

This is not a complete list of the methods that can be used to add or remove data, but these are some of the most common.

Dealing With Textual Data

This section will illustrate how you can use pandas methods to manipulate textual data. The table we will be using looks like this:

Names	Genre	Age	Followers
Hughes, Doretha	Pop	34	50000000
Crissman, Cornell	Alternative	23	1000000
Cerny, Matthew	Dance	19	20000000
Chase, Gladys	Metal	45	30000000
Piasecki, Danika	Jazz	50	400000
Sosa, Eustolia	Indie	26	7000000
Critchlow, Effie	Progressive	36	800000
Urich, Arthur	Classical	50	3000000
Mcgeorge, Babara	Rap	17	1000000
Steve, Abraham	Instrumental	30	10000000

This is a list of fictional artists, their genres, age, and how many followers they have on Twitter.

The first thing we will learn is how to change the case. So, let us write code to change the genre column value into uppercase.

We used code to export this table from Excel so we can work with it. You can copy this table into a spreadsheet program and import it with pandas.

Here’s the code we used to change the case of Genre column:

```
artists['Genre'].str.upper()
```

We used the square bracket to select the column. Then, we used the str accessor which helped us gain access to string methods. Then, we applied our uppercase method.

Here was the output:

```
0   POP
1   ALTERNATIVE
2   DANCE
3   METAL
4    JAZZ
5   INDIE
6   PROGRESSIVE
```

7   CLASSICAL

8   RAP

9 INSTRUMENTAL

Name: Genre, dtype: object

The code went through each element and changed it to uppercase. It is important to remember this.

We can create a surname column from the names column.

When we use the following code, we split the string at the comma turning it into a list:

```
artists['Names'].str.split(",")
```

The code will return:

0   [Hughes, Doretha]

1   [Crissman, Cornell]

2   [Cerny, Matthew]

3   [Chase, Gladys]

4   [Piasecki, Danika]

5   [Sosa, Eustolia]

6 [Critchlow, Effie]

7 [Urich, Arthur]

8 [Mcgeorge, Babara]

9 [Steve, Abraham]

Name: Names, dtype: object

To turn these two item lists into another column, we would need to do this:

```
artists['Surname'] = artists['Names'].str.split(",").str.get(0)

artists['Surname']
```

The outcome of this code is a column of surnames:

0 Hughes

1 Crissman

2 Cerny

3 Chase

4 Piasecki

5 Sosa

6 Critchlow

7 Urich

8 Mcgeorge

9 Steve

Name: Surname, dtype: object

Our DataFrame looks like this:

	Names	Genre	Age	Followers	Surname
0	Hughes, Doretha	Pop	34	50000000	Hughes
1	Crissman, Cornell	Alternative	23	1000000	Crissman
2	Cerny, Matthew	Dance	19	20000000	Cerny
3	Chase, Gladys	Metal	45	30000000	Chase
4	Piasecki, Danika	Jazz	50	400000	Piasecki

You will have to find a way of removing the surnames from the “Names” column and maybe the “Surnames” column next to the “Names” column. You can figure it out.

You can use the contain method to find out if a string is matched. The contains method will return boolean values.

Here’s an example:

```
artists['Names'].str.contains('Crissman')
```

The output for the code is:

```
0  False
1  True
2  False
3  False
4  False
5  False
6  False
7  False
8  False
9  False
```

Name: Names, dtype: bool

The code checks the conditional on every element. It does not stop once it has found the string that matches it. In a very long list, you want to know which row your name appears and have it pulled so you can access all the other information about them.

The following code helps us achieve that:

```
artists[artists['Names'].str.contains('Crissman')]
```

It returns us the profile of the artists instead of booleans.

	Names	Genre	Age	Followers
1	Crissman, Cornell	Alternative	23	1000000

If the list contained many similar names, this code would return a DataFrame that contains the information we request. In our example, there is only one person whose surname is Crissman.

Other ways of extracting names DataFrames or Series are supported, but these are common examples.

Find length

Then the len method allows us to extract the length of strings. You can use it in a DataFrame. In our example, we would use the code below to access a column and return a string length for each element:

```
artists['Names'].str.len()
```

The output will be:

```
0  15
1  18
2  15
3  14
4  16
5  15
6  16
7  13
8  16
```

```
9  14
Name: Names, dtype: int64
```

If you are only interested in learning where the longest name in the DataFrame is, you must first find out all the lengths of all names in the table and get the index of the largest number.

```
artists['Names'].str.len().idxmax()
```

The output is:

```
1
```

Index 1.

To discover what the name at that index is, we must use the code below:

```
artists.loc[artists['Names'].str.len().idxmax(), 'Names']
```

The output will be:

'Crissman, Cornell '

To reiterate, these functions might seem not that useful, but as soon as you start dealing with large data sets, their importance becomes apparent. If this were a list with a hundred thousand names, we would be better off using them than counting ourselves. The idxmax method does not work on strings. It works on the numbers returned by the len method, picking the biggest one. Idxmin does the opposite.

We hope our hands-on approach has helped you get started with pandas.

Resources

This is a list of tables you can use to practice or follow along with the exercises as promised.

Copy and paste them into Excel or Google Sheets and download them to your computer in a format you are comfortable with (we have used the Excel format here: xlsx). Add them to the same directory as your notebook instance and then import them into Jupyter with the read function to follow along.

Table A : Reading and Writing data table

From	To	Code	Hearthstone	Passive	Reading
1/1/2019	1/6/2019	0.09	20.6	2.13	9.88
1/7/2019	1/13/2019	0	42.71	1.6	11.05
1/14/2019	1/20/2019	0.09	14.95	0	21.78
1/21/2019	1/27/2019	0	12.61	0.09	22.91
1/28/2019	2/3/2019	0	26.97	5.83	4.81
2/4/2019	2/10/2019	0.22	33.45	3.49	10.72
2/11/2019	2/17/2019	0.01	39.99	9.52	7.54
2/18/2019	2/24/2019	0	43.2	10.99	5.54
2/25/2019	3/3/2019	0	10.63	19.78	6.71
3/4/2019	3/10/2019	2.47	8.82	15.2	9.28
3/11/2019	3/17/2019	0.63	7.88	12.96	10.79
3/18/2019	3/24/2019	0	0	18.91	10.11
3/25/2019	3/31/2019	0.54	0	23.82	12.29
4/1/2019	4/7/2019	18.14	12.78	9.76	1
4/8/2019	4/14/2019	14.2	19.79	9.14	1.35
4/15/2019	4/21/2019	9.53	20.62	9.08	0.86
4/22/2019	4/28/2019	4.6	18.71	15.81	0.16
4/29/2019	5/5/2019	8	6.95	13.62	0
5/6/2019	5/12/2019	13.47	24.88	10.48	0
5/13/2019	5/19/2019	6.91	33.38	9.54	6.34
5/20/2019	5/26/2019	19.3	21.03	10.98	1.18
5/27/2019	6/2/2019	8.67	13.39	10.25	1.26
6/3/2019	6/9/2019	15.79	5.14	12.48	0.26
6/10/2019	6/16/2019	13.17	12.53	10.68	0.72
6/17/2019	6/23/2019	10.43	6.45	14.68	1.46
6/24/2019	6/30/2019	15.72	1.83	13.33	5.73
7/1/2019	7/7/2019	11.44	0.7	9.64	1.51
7/8/2019	7/14/2019	11.8	4.87	3.74	7.32
7/15/2019	7/21/2019	0.74	4.09	4.15	17.29
7/22/2019	7/28/2019	4.37	2.52	8.77	24.42
7/29/2019	7/31/2019	5.01	1.07	3.89	12.18

Table B:2019 Weekly Data

Note: To do exercises where you only need Months without the seasons, remove the Seasons column before importing.

Code	Hearthstone	Reading	Month	Seasons
0	20.6	9.88	1	A-WINTER
0	42.71	11.05	1	A-WINTER
0.09	14.95	21.78	1	A-WINTER
0	12.61	22.91	1	A-WINTER
0	26.97	4.81	1	A-WINTER
0.22	33.45	10.72	2	A-WINTER
0.01	39.99	7.54	2	A-WINTER
0	43.2	5.54	2	A-WINTER
0	10.63	6.71	2	A-WINTER
2.47	8.82	9.28	3	B-SPRING
5.2	7.88	10.79	3	B-SPRING
0	0	10.11	3	B-SPRING
9.36	0	12.29	3	B-SPRING
18.28	12.78	1	4	B-SPRING
14.2	19.79	1.35	4	B-SPRING
10.77	20.62	0.86	4	B-SPRING
5	18.71	0.16	4	B-SPRING
8	6.95	0	4	B-SPRING
13.47	24.88	0	5	B-SPRING
6.98	33.38	6.34	5	B-SPRING
19.3	21.03	1.18	5	B-SPRING
8.67	13.39	1.26	5	B-SPRING
15.79	5.14	0.26	6	C-SUMMER
13.17	12.53	0.72	6	C-SUMMER
10.43	6.45	1.46	6	C-SUMMER
15.71	1.83	5.73	6	C-SUMMER
11.44	0.7	1.51	7	C-SUMMER
11.13	4.87	7.32	7	C-SUMMER
0.72	4.09	17.29	7	C-SUMMER
3.21	2.52	24.42	7	C-SUMMER

5.34	2.96	15.27	7	C-SUMMER
9.14	15.44	11.5	8	C-SUMMER
6.47	2.71	6.77	8	C-SUMMER
6.89	1.34	11.97	8	C-SUMMER
2.72	1.04	9.88	8	C-SUMMER
2.52	0	14.41	9	D-AUTUMN
4.2	0.75	7.78	9	D-AUTUMN
9.11	0.62	9.3	9	D-AUTUMN
5.49	1.83	10.96	9	D-AUTUMN
7.87	1.15	9.79	9	D-AUTUMN
10.91	7.8	3.76	10	D-AUTUMN
6.26	3.35	2.81	10	D-AUTUMN
1	9.62	0.21	10	D-AUTUMN
0.32	10.02	1.12	10	D-AUTUMN
0.59	1.07	3.02	11	D-AUTUMN
0.12	5.13	1.98	11	D-AUTUMN
0.63	16.38	3.06	11	D-AUTUMN
0.77	8.88	7.84	11	D-AUTUMN
3.05	6.72	15.25	12	A-WINTER
0.01	7.31	7.17	12	A-WINTER
1.24	9.06	2.83	12	A-WINTER
3.2	5.18	0.4	12	A-WINTER
0	16.67	0	12	A-WINTER

Table C: The second set of 2019 data for DataFrame combining exercises and others

Away	Random/Admin	Work
24.8	11.59	0
24.09	22.25	0.05
18.65	31.92	1.61
20.9	18.03	15.23
23	20.31	7.01
27.75	25.13	11.71
18.35	17.14	8.72
20.85	19.02	7.7
34.13	22.91	8.79
29.05	30.62	9.79
18.55	25.06	4.85
23.86	6.57	3.46
32.68	3.3	8.08
22.61	24.05	0.38
17.56	16.61	0.43
29.92	16.05	0.9
20.29	5.64	0.17
26.12	18.62	0.02
23.07	12.91	0.02
17.8	18.31	0.54
21.18	14.22	0.1
38.21	13.5	0.01
28.59	26.15	0.68
23.43	15.65	0.01
25.25	17.2	0.01
30.54	18.8	0.07
24.93	18.08	0.13
46.76	18.44	0.02
39.41	28.07	0.14
36.44	5.45	3.81
28.65	20.47	5.46

20.41	18.85	1.94
17.49	34.24	0.55
15.01	25.49	1.93
28.39	19.22	5.53
22.24	27.33	4.64
23.02	29.68	6.01
14.74	21.93	4.42
20.07	28.38	3.77
24.1	31.73	0.61
20.06	24.44	4.87
11.87	25.59	9.64
24.94	21.58	17.92
21.42	18.77	19.34
22.14	31.97	14.09
15.99	41.38	6.96
16.48	16.37	22.41
16.29	13.6	27.28
8.02	18.53	35.81
12.91	32.36	17.56
11.7	26.22	29.28
15.6	23.17	20.03
1.92	3.99	0.29

Chapter 5: NumPy

NumPy stands for Numerical Python. It is an open-source library used in a wide variety of fields, and it is the universal standard for working with numerical data using Python. Anyone can use NumPy, whether working on a small personal project or doing in-depth research. It integrates well with other data science tools.

NumPy can work with multidimensional arrays, ndarrays, and matrix data structures. It also provides a variety of high-level mathematical functions you can use to operate on the data. It gives Python the ability to work with powerful data structures that are naturally unavailable to it.

Installation

When you installed Anaconda, NumPy should have also been installed, but just to check, open your Anaconda Prompt and enter the following command:

```
conda install numpy
```

If it is installed, it will say: “All requested packages already installed.”

The Importance of NumPy Arrays

We already suggested that NumPy arrays are an important feature of NumPy. That might have led to you wondering why we can't just use Python to do our work. The simple answer is that it is inefficient.

Python lists contain different kinds of data in one list. NumPy arrays require the data entered within them to be homogenous. This means mathematical operations can be carried a lot quicker than they would be in a list where data types differ. This is because the language interpreter doesn't have to go back and forth checking if the data type fits an operation. It doesn't happen in NumPy because NumPy arrays data types are specified. Another advantage of using NumPy is that it requires less memory.

What is a NumPy Array?

A NumPy array, simply referred to as an array, is a grid of values that contains data that is indexed and has information about how it should be interpreted. All the elements in the array are of the same data type.

Arrays can be indexed by a tuple of integers, booleans, or another array. The array's rank is its dimensions and the shape is the tuple of integers that give it size along its dimensions. NumPy array dimensions are referred to as axes. Arrays can be accessed using square brackets just like you would with Python lists (NumPy: the absolute basics for beginners — NumPy v1.19.dev0 Manual, n.d.).

Creating Arrays

To work with NumPy arrays, you have to first import NumPy like this:

```
import numpy as np
```

You create an array by passing a Python list in it. Like this:

```
arr = np.array([1, 2, 3, 4, 5])
```

NumPy gives you the ability to create an array filled with zeros. You pass how long you want the array to be in the parenthesis, like this:

```
np.zeros(8)
```

The output will be:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

With NumPy, you can also create an empty array that is filled with random numbers.

Here is how:

```
np.empty(10)
```

Just like above, the parentheses tells NumPy how many values should be in the array. The output on our system was:

```
array([2.78151378e-307, 1.24610994e-306, 3.44900369e-307,
4.22792214e-307,

3.11526468e-307, 3.11522054e-307, 3.56011818e-307,
2.33644129e-307,

4.00540117e-307, 4.89539676e-307])
```

It will differ for you. Each time you run it, it will. People use np.empty because it is faster. The numbers are entered based on the state of your memory. It's all good to create an array this way, but you need to remember to fill it with the value you want later.

You can also quickly create a range with a range of elements, like this:

```
np.arange(5)
```

It will give you:

```
array([0, 1, 2, 3, 4])
```

You can make an array with evenly spaced values. This type of NumPy takes three arguments: the starting point, the finishing point, and the step or interval. Below, we make an array that starts from nine and ends at twenty-one and increases in increments of three.

```
np.arange(9, 21, 3)
```

The output should be:

```
array([ 9, 12, 15, 18])
```

We can also make arrays that are spaced in a linear fashion in a specified interval (NumPy: the absolute basics for beginners — NumPy v1.19.dev0 Manual, n.d.). The following array will have five values from 0 to 30. In other words, we split the range according to the value given in the num argument.

```
np.linspace(0, 30, num=5)
```

The output will be:

```
array([ 0. , 7.5, 15. , 22.5, 30. ])
```

The default datatype of arrays is np.float64. We may not want to work with that data type. So, we use the following method to specify a data type in the array:

```
np.ones(5 , dtype=np.int64)
```

Our output will be:

```
array([1, 1, 1, 1, 1], dtype=int64)
```

If we hadn't had specified, the data type output would be:

```
array([1., 1., 1., 1., 1.])
```

To make two-dimensional arrays, you have to pass multiple sequences like this:

```
arr2D = np.array([(1, 2, 4 , 6), (34, 23, 45, 23)])

arr2D
```

Our output will be like this:

```
array([[ 1, 2, 4, 6],

[34, 23, 45, 23]])
```

Arrays have formats or shapes just like pandas' DataFrames. This array has two rows and four columns. We can use this knowledge to make empty, zeros, and ones arrays that are two dimensional a lot quickly. We must first give the value for the rows and the value for the column.

Below we make an array that is filled with zeros and has five rows and four columns.

```
ze = np.zeros((5, 4))

ze
```

Our output is:

```
array([[0., 0., 0., 0.],

[0., 0., 0., 0.],

[0., 0., 0., 0.],
```

```
[0., 0., 0., 0.],

[0., 0., 0., 0.]])
```

We can specify what data type we want the array to be filled with like this:

```
np.zeros((5, 4), dtype=np.int16)
```

Our output will be:

```
array([[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]], dtype=int16)
```

The reshape method allows you to shape an array.

Take the following array:

```
arr = np.arange(10)

arr
```

This is how this array looks now:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If you want the array to be two-dimensional and have rows and columns, you would use reshape like this:

```
arr.reshape(2, 5)
```

The reshape method above turns the array into two rows of five numbers. So, our output is this:

```
array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9]])
```

If you wanted to make the array into a three-dimensional array, you would have to include another argument in the reshape method that specifies how many two-dimensional arrays you want to be held in the three-dimensional array.

Take the following two-dimensional array.

```
arr = np.arange(30).reshape(6, 5)

print(arr)

[[ 0  1 2 3 4]

[ 5  6  7  8  9]

[10 11 12 13 14]

[15 16 17 18 19]

[20 21 22 23 24]

[25 26 27 28 29]]
```

It has six rows and five columns. We may want to have two two-dimensional arrays out of this in a three-dimensional array. This is how we would do it:

```
arr = np.arange(30)

arr.reshape(2, 3, 5)
```

The first argument tells NumPy how I would like to divide the created array. The second gives how many rows each array should have and the third specifies columns.

Our array looks like this now:

```
array([[[ 0,  1,  2,  3,  4],
```

```
[ 5,  6,  7,  8,  9],
[10, 11, 12, 13, 14]],

[[15, 16, 17, 18, 19],
[20, 21, 22, 23, 24],
[25, 26, 27, 28, 29]]])
```

Printed, it looks like this:

```
[[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]]

[[15 16 17 18 19]
[20 21 22 23 24]
[25 26 27 28 29]]]
```

To print an array, you have to assign it to a variable and pass the variable in Python's print method or declare the array in the print method.

To change an array into a two-dimensional or three-dimensional array, you have to get your dimensions right. If your array cannot be shaped into three rows and four columns, NumPy will throw an error. You need to be sure that your array can give you the dimensions you want it to have.

NumPy's print method is clever. If the array you want to print is too large, it will not print the whole array. It will only print the corner of that array.

Let's take a look:

```
print(np.arange(2000))
```

The output of the code is:

```
[  0  1  2 ... 1997 1998 1999]
```

If the array was a two-dimensional array:

```
print(np.arange(2000).reshape(100,20))
```

The output is:

```
[[  0  1  2 ...  17  18  19]
```

```
[  20  21  22 ...  37  38  39]
[  40  41  42 ...  57  58  59]
...
[1940 1941 1942 ... 1957 1958 1959]
[1960 1961 1962 ... 1977 1978 1979]
[1980 1981 1982 ... 1997 1998 1999]]
```

Learning About An Array

To find out information about an array, you have the following methods to choose from:

ndim: gives information about the dimensions of the array.

size: size will give you the total number of elements in an array.

shape: will tell you how many rows and rolums your array has.

Let's use them on the previously printed array.

```
arr.ndim
```

Output is: 2

It gives us two because it is a two-dimensional array.

```
arr.size
```

Output is: 2000

We made an array with two thousand elements.

```
arr.shape
```

Output is: (100,20)

We made a two-dimensional array with a hundred rows and twenty columns.

Basic Array Operations

Arithmetic operations in NumPy are executed element wise. Let's illustrate this.

```
arr1 = np.array([2, 4, 6, 8, 10])

arr2 = np.array([1, 2, 3, 4, 5])

ans = arr1 + arr2

ans
```

The outcome is:

```
array([ 3,  6,  9, 12, 15])
```

As you can see, every value was added to a value that sits in the same position as itself in the other array. Index 0 added to index 0, 1 to 1 and so on.

In other languages, you can multiply a matrix with another by using the * operator, but in NumPy, that does not happen. Let's is illustrate:

```
matrix1 = np.array([[0, 1, 2],
```

```
[3, 4, 6],

[7, 8, 9]])

matrix2 = np.array([[2, 4, 6],

[8, 10, 12],

[14, 16, 18]])

matrix1 * matrix2
```

Our output will be:

```
array([[  0,  4,  12],

[ 24,  40,  72],

[ 98, 128, 162]])
```

As you can see, each number got multiplied by another just like we saw in the last example. If this were another language, it would have given us a matrix product. To do that in NumPy we need to use @ or dot.

Here’s how:

```
matrix1 @ matrix2
```

The output is:

```
array([[ 36,  42,  48],
[122, 148, 174],
[204, 252, 300]])
```

The alternative is:

```
matrix1.dot(matrix2)
```

The output is the same:

```
array([[ 36,  42,  48],
[122, 148, 174],
[204, 252, 300]])
```

The methods we have used so far output another array as the answer instead of just changing the elements in the original array. If you are interesting in changing the elements in an array, you have to use *= or += operators.

Let's illustrate this:

```
arr = np.zeros((2, 5), dtype=int)

arr += 4

arr
```

Output is:

```
array([[4, 4, 4, 4, 4],

[4, 4, 4, 4, 4]])
```

We have added four to all the zeros that occupied our array. If the array had been filled with twos we would have had an array of sixes. If we had used *=, we would have multiplied each number by four, so our array would be filled with eights.

```
arr = np.array([[2, 2, 2, 2, 2],

[2, 2, 2, 2, 2]])

arr *= 4

arr
```

Output:

```
array([[8, 8, 8, 8, 8],
```

[8, 8, 8, 8, 8]]

Many of the unary operations we perform on an array are available as methods in NumPy. If we need to add the sum of all elements in an array we use the sum method. If we want to find the biggest number, we use the max method, and if we want to find the smallest, we use the min method.

Below is an example of those operations:

arr22 = np.array([[0, 1, 2, 3, 4],

[5, 6, 7, 8, 9]])

arr22.min()

Output: 0

arr22.max()

Output: 10

arr22.sum()

Output: 45

You can run methods like these and others on axis 0 or 1. For instance, we may want to find the sum of each row.

```
arr = np.array([[ 36,  42,  48],
[122, 148, 174],
[204, 252, 300]])
arr.sum(axis=0)
```

It will give us:

```
array([362, 442, 522])
```

To find the sum of each row, we use:

```
arr.sum(axis=1)
```

Which gives us:

```
array([126, 444, 756])
```

We can use other operators in this manner. In some cases, you want the cumulative sum along every row.

```
arr.cumsum(axis=1)
```

Our output is:

```
array([[ 36,  78, 126],
[122, 270, 444],
[204, 456, 756]], dtype=int32)
```

Accessing Elements, Slicing and Iterating Arrays

One-dimensional arrays in NumPy function similarly to Python lists. They are accessed, integrated, and sliced the same way. What should interest you is how you perform similar functions on a multidimensional array. The best way to think of a multidimensional array is to think of it as a chessboard where each square or index can be located by row number and column number. Each axis can only have one value, but that is not limiting in any way.

Let's say we have a multidimensional array like this one:

```
#this array represents squares on a chessboard

arr =  np.arange(64).reshape(8, 8)

arr

array([[ 0,  1,  2,  3,  4,  5,  6,  7],

[ 8,  9, 10, 11, 12, 13, 14, 15],

[16, 17, 18, 19, 20, 21, 22, 23],

[24, 25, 26, 27, 28, 29, 30, 31],

[32, 33, 34, 35, 36, 37, 38, 39],

[40, 41, 42, 43, 44, 45, 46, 47],

[48, 49, 50, 51, 52, 53, 54, 55],
```

```
[56, 57, 58, 59, 60, 61, 62, 63]])
```

If we wanted to print fifty from the board, we would need to use the following syntax that takes two arguments. The first is the row and the second indicates the column.

Fifty is on the seventh row and the third column. To access it, we need to type:

```
arr[6, 2]
```

Remember that array indexes start at zero. That is why we have six and two instead of seven and three in the square brackets.

We can slice entire rows or columns if we like. We just have to pass the slice as one of the arguments and pick a row or column.

Below, we are slicing at all rows in the fourth column:

```
arr[0:7, 3]
```

Our output is the entire fourth column:

```
array([ 3, 11, 19, 27, 35, 43, 51])
```

This can also be achieved with his code:

```
arr[:, 3]
```

Iteration of multidimensional arrays goes along the first axis. Let's try an example so you can see:

```
for row in arr:

print(row)
```

Our output is:

```
[0 1 2 3 4 5 6 7]

[ 8  9 10 11 12 13 14 15]

[16 17 18 19 20 21 22 23]

[24 25 26 27 28 29 30 31]

[32 33 34 35 36 37 38 39]

[40 41 42 43 44 45 46 47]

[48 49 50 51 52 53 54 55]

[56 57 58 59 60 61 62 63]
```

The output is identical to our array. The code does not iterate over each element, it iterates rows. If you need it to iterate over each element, you will use the flat method. Let's use the flat method:

```
for row in arr.flat:
print(row)
```

This will be the output:

```
0
1
2
3
4
5
6
7
8
9
10
11
```

12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51

52

53

54

55

56

57

58

59

60

61

62

63

As you can see, the code printed each element in the array, not every row.

Manipulating Shapes

We discussed reshape, but there are other methods that let you manipulate the shape of an array. We will continue working with our chessboard array from the last section.

The shape of our array is (8, 8).

We can use the ravel method to flatten the array, like this:

```
arr.ravel()
```

The output will be:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,

17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,

34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,

51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63])
```

If you check the shape with arr.shape you will get: (64,)

To check the shape, you will have to assign the ravel array to a variable first.

The next method is one you can use to transpose the array:

```
arr.T
```

The output is:

```
array([[ 0,  8, 16, 24, 32, 40, 48, 56],
[ 1,  9, 17, 25, 33, 41, 49, 57],
[ 2, 10, 18, 26, 34, 42, 50, 58],
[ 3, 11, 19, 27, 35, 43, 51, 59],
[ 4, 12, 20, 28, 36, 44, 52, 60],
[ 5, 13, 21, 29, 37, 45, 53, 61],
[ 6, 14, 22, 30, 38, 46, 54, 62],
[ 7, 15, 23, 31, 39, 47, 55, 63]])
```

The array still has the same shape but the elements have changed places. If the array's shape was (5, 6), the transpose method would change the shape to (6, 5). The method changes rows into columns and columns into rows.

The methods we have looked at to manipulate the shape of the array don't change the original array but produce a new array from the original. If you want to change the original array, you have to use resize. Here's an example below

```
arr.resize(4, 16)

arr
```

Our output is:

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],

[16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],

[32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],

[48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63]])
```

If we had used reshape, we would not get the same result as when we called the variable holding the array. It would have given us the original array.

Stacking Arrays

You can stack arrays horizontally or vertically. Stacking is exactly what it sounds like.

Say you have two arrays, like these:

```
arrA = ([[9, 7, 5, 8],

[5, 2, 19, 12]])

arrB = ([[20, 23, 55, 18],

[50, 42, 55, 89]])
```

You can stack them vertically, like this:

```
np.vstack((arrA, arrB))
```

Our output will be:

```
array([[ 9,  7,  5,  8],

[ 5,  2, 19, 12],
```

[20, 23, 55, 18],

[50, 42, 55, 89]])

You can stack them horizontally like this:

np.hstack((arrA, arrB))

Our output will be:

array([[9, 7, 5, 8, 20, 23, 55, 18],

[5, 2, 19, 12, 50, 42, 55, 89]])

column_stack is another method that returns a two-dimensional array. Here is how it works.

arrA = ([9, 7, 5, 8, 20, 23, 55, 18])

arrB = ([5, 2, 19, 12, 50, 42, 55, 89])

np.column_stack((arrA, arrB))

Our output will be:

```
array([[ 9,  5],
[ 7,  2],
[ 5, 19],
[ 8, 12],
[20, 50],
[23, 42],
[55, 55],
[18, 89]])
```

Look closely, this isn't vstack. The column_stack method has paired the same indexed elements together in each row. That does not happen with vstack.

Splitting An Array

You can split an array into other, smaller arrays. One possible reason why you might want to do this is because it's possible that numbers in a certain row are relevant to calculations being made. In this section, we will work with the chessboard array. Here is how you would go about it:

```
np.hsplit(arr, 4)
```

The code tells NumPy to split the array into four separate arrays. Here's our output:

```
[array([[ 0,  1],

[ 8,  9],

[16, 17],

[24, 25],

[32, 33],

[40, 41],

[48, 49],

[56, 57]]),

array([[ 2,  3],

[10, 11],
```

```
[18, 19],
[26, 27],
[34, 35],
[42, 43],
[50, 51],
[58, 59]]),
array([[ 4,  5],
[12, 13],
[20, 21],
[28, 29],
[36, 37],
[44, 45],
[52, 53],
[60, 61]]),
array([[ 6,  7],
[14, 15],
[22, 23],
[30, 31],
[38, 39],
[46, 47],
```

```
[54, 55],

[62, 63]])]
```

We can tell NumPy to begin cutting in a certain place. Below, we tell it to split the array at the second and fifth column.

```
np.hsplit(arr, (2, 5))
```

This is the output we get:

```
[array([[ 0,  1],

[ 8,  9],

[16, 17],

[24, 25],

[32, 33],

[40, 41],

[48, 49],

[56, 57]]),

array([[ 2,  3,  4],

[10, 11, 12],

[18, 19, 20],
```

```
[26, 27, 28],
[34, 35, 36],
[42, 43, 44],
[50, 51, 52],
[58, 59, 60]]),
array([[ 5,  6,  7],
[13, 14, 15],
[21, 22, 23],
[29, 30, 31],
[37, 38, 39],
[45, 46, 47],
[53, 54, 55],
[61, 62, 63]])]
```

Final Words & FAQ

We have introduced you to three data science topics that are relevant to the field of data analysis. You have always worked with us through some examples that will help you familiarize yourself with the functions of these tools. Your next step should be to test your skills and sharpen your tools by using them in other contexts to fix problems that you encounter. It is the only way you will grow.

If you are someone with some tech experience, you already know how to research things when you get stuck and find solutions. If you are entirely new, you might have started doing this as you were learning with this book. We expected you to do this because it will be important in your role as a data analysis.The easiest way to determine what you don’t know is by thinking about what you want to do and asking yourself if you can use your skills to achieve that. Another way is to find freely available data and ask yourself how you can get an insight you are curious about in the data.

For instance, we haven't discussed in detail about data visualizations using matplotlib. Maybe you could focus on learning how to use it more proficiently to meet your needs. You can also find free public datasets to work with and practice. When practicing, you learn all sorts of skills, like how to clean the data for the country that is relevant to you. Some of this data is available to download, and some are available via an API. Do not be intimidated to go and try. My message is this: you have basics that will enable you to do interesting things with data even when it might require some research. Data analysts do this all the time, and it is an effective way to build your portfolio.

Here are some places you can find data to practice with:

IMF Data

The IMF publishes data on a wide range of economic sectors. Its data is available from various world regions, on gender, trade, and many other subjects. The data is free to download. You can use this for data cleaning exercises, data visualizations, and investigating interesting statistical phenomena. Visit the site here.

Enron Email

To practice your textual data analysis skills, you can use the Enron emails. When Enron collapsed, these emails became public. They are available to download here.

British Government Data

The UK has data that is open to the public on a wide range of topics like government spending, health, education, defense, crime, business, the economy, and more—this is the best place to learn how to clean data and do data visualizations or reports. Visit the site here.

UNICEF

UNICEF can be a wonderful place to find new projects to do. They provide all kinds of data about the lives of children internationally. They have data sets on education, nutrition, child mortality, HIV/AIDS, education, poverty, and more. You can use this data to practice visualizations, data cleaning, and more. The data UNICEF provides is free and comes from all over the globe. Visit the site here.

Walmart's Sales data

For something more commercial, you can try Walmart's sales data of over forty-five stores in the US. If you are a curious person, you have a lot of questions to ask the data. This data allows freedom. It is best for data processing projects. Visit the site here.

Many of the data cycle concepts we have discussed in chapter one will make more sense and present themselves to you as you work. They will train your research skills, the ability to determine what data is relevant and which it isn't, and the ability to mine the data for the information that is of value to you. You might find yourself in a situation where more data sources are needed, getting it and converting it into the same format so your results may be more accurate. Data analysts have to do this all the time.

You can also try a site like Kaggle which has many exercises you can do to expand your skills and keep them sharp. It also offers thousands of data sets and code samples that you can use in almost any domain. It is free and easy to use.

We have given you enough skills to take the next steps. Data science is all about using the fundamentals to crack novel problems and learn new things when required, adaptability is a must.

To get the job you want, you need to choose a field you are interested in and do exercises and projects based on those fields. Find a data set for that field and do the sorts of things that would be required of you in that field. When you do this, you are building the skills your employer will want and showing that you have the ability to perform as required. Your portfolio is likely to stand out because it would be tailored for the industry. For instance, if you are looking to get into finance, focus on financial data. Find out what sorts of things are important for financial institutions to know and to do and what data scientists do for them and practice those things. When done, put it in your portfolio.

When Do I Know I Have Enough Projects in My Portfolio?

When thinking of a portfolio, you should think in terms of numbers. One great project can be enough to make you hireable. You need to focus on what a given project will help you develop as a data analyst. Find out what companies are looking for in a data scientist and focus on expanding to those skills in projects you pick. Companies post for the perfect candidate, but perfect candidates either don’t exist, are working somewhere else, or are in the making..

What Type of PC Do I Need for Data Science?

At the bare minimum, you want a computer with an Intel core i3 processor and eight gigabytes of RAM. You will also need space on your computer. For learning alone, you can do with any computer, and many datasets you can use for practice won't require these specs. Many people prefer a computer with an Intel core i5 or better, with sixteen gigabytes of RAM and 1TB HDD. An SSD will also help, but they can be expensive. In reality, you can buy a computer with a weaker processor and four gigabytes of RAM to do the work you need, but it won’t be a good experience. There are cloud-based platforms that allow you to run more powerful collections and other tasks on the cloud instead on your own system. This means your computer doesn’t have to do the hard work all the time. You can use it for tests and run your actual work online.

What Are Some of the Skills I Will Need?

You need to have a basic analytics mindset. You also need to understand statistics. So, it will not hurt to learn basic statistics if you don’t know any. You will practice working with APIs and pulling data from databases. SQL is worth learning.

Is There a Future in Data Science/Analytics?

There is increasing automation in many fields, and data science is one of them. Certain tasks are being automated. However, data science will be around for a very long time because no one can predict the sorts of problems businesses might face. A lot of critical work goes into data science, and much of it is hard to automate. Automation tools only serve to make the jobs of data scientists easier, not to take them away.

What Will it Take for Me to Become a Data Analyst?

This will largely depend on you. There isn't a strict limit or timeline. If you are committed, a year is reasonable, assuming you have other commitments. Time should not be what you are focusing on. You should focus on getting the skills you need and honing the ones you have. It will be your work that speaks for you not how long you have been doing data science.

Other Books from the Author

Python For Beginners: A crash course to learn Python Programming in 1 Week

Python Machine Learning: A Step by Step Beginner's Guide to Learn Machine Learning Using Python

Python Data Analysis: A Beginners Guide to Master the Fundamentals of Data Science and Data Analysis by Using Pandas, Numpy and Ipython

References

Data Science vs. Big Data vs. Data Analytics. (2016, April 5). Simplilearn.com. https://www.simplilearn.com/data-science-vs-big-data-vs-data-analytics-article#:~:text=Data%20Analytics%20the%20science%20of%20examining%20raw%20data

Installing on Linux — Anaconda documentation. (n.d.). Docs.Anaconda.com. Retrieved December 22, 2020, from https://docs.anaconda.com/anaconda/install/linux/

IPython - Magic Commands - Tutorialspoint. (n.d.). Www.Tutorialspoint.com. Retrieved December 22, 2020, from https://www.tutorialspoint.com/jupyter/ipython_magic_commands.htm

IPython Documentation — IPython 7.19.0 documentation. (n.d.). Ipython.Readthedocs.Io. Retrieved December 22, 2020, from https://ipython.readthedocs.io/en/stable/

Jupyter Notebook - Editing - Tutorialspoint. (n.d.). Www.Tutorialspoint.com. Retrieved December 22, 2020, from https://www.tutorialspoint.com/jupyter/jupyter_notebook_editing.htm

Lee, J. (2018, July 18). The Printable Markdown Cheat Sheet for Beginners and Experts. MakeUseOf. https://www.makeuseof.com/tag/printable-markdown-cheat-sheet/

Liberty, D. (2019, January 2). Data Science vs. Data Analytics - What's the Difference? I Sisense. Sisense; Sisense. https://www.sisense.com/blog/data-science-vs-data-analytics/

Mueller, J. P. (n.d.). Common Jupyter Notebook Magic Functions. Dummies. Retrieved December 22, 2020, from https://www.dummies.com/programming/python/common-jupyter-notebook-magic-functions/

NumPy: the absolute basics for beginners — NumPy v1.19.dev0 Manual. (n.d.). Numpy.org. https://numpy.org/devdocs/user/absolute_beginners.html

Package overview — pandas 1.1.5 documentation. (n.d.). Pandas.Pydata.org. Retrieved December 22, 2020, from https://pandas.pydata.org/docs/getting_started/overview.html