

STUDENT MANUAL

Using Data Science Tools in Python®

Using Data Science Tools in Python®

Using Data Science Tools in Python®

Part Number: 094001

Course Edition: 1.0

Acknowledgements

PROJECT TEAM

<i>Author</i>	<i>Technical Consultant</i>	<i>Media Designer</i>	<i>Content Editor</i>
Jason Nufryk	Andrea Cogliati	Brian Sullivan	Pamela J. Taylor

Notices

DISCLAIMER

While Logical Operations, Inc. takes care to ensure the accuracy and quality of these materials, we cannot guarantee their accuracy, and all materials are provided without any warranty whatsoever, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. The name used in the data files for this course is that of a fictitious company. Any resemblance to current or future companies is purely coincidental. We do not believe we have used anyone's name in creating this course, but if we have, please notify us and we will change the name in the next revision of the course. Logical Operations is an independent provider of integrated training solutions for individuals, businesses, educational institutions, and government agencies. The use of screenshots, photographs of another entity's products, or another entity's product name or service in this book is for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the book by nor any affiliation of such entity with Logical Operations. This courseware may contain links to sites on the Internet that are owned and operated by third parties (the "External Sites"). Logical Operations is not responsible for the availability of, or the content located on or through, any External Site. Please contact Logical Operations if you have any concerns regarding such links or External Sites.

TRADEMARK NOTICES

Logical Operations and the Logical Operations logo are trademarks of Logical Operations, Inc. and its affiliates.

All other product and service names used may be common law or registered trademarks of their respective proprietors.

Copyright © 2020 Logical Operations, Inc. All rights reserved. Screenshots used for illustrative purposes are the property of the software proprietor. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without express written permission of Logical Operations, 3535 Winton Place, Rochester, NY 14623, 1-800-326-8724 in the United States and Canada, 1-585-350-7000 in all other countries. Logical Operations' World Wide Web site is located at www.logicaloperations.com.

This book conveys no rights in the software or other products about which it was written; all use or licensing of such software or other products is the responsibility of the user according to terms and conditions of the owner. Do not make illegal copies of books or software. If you believe that this book, related materials, or any other Logical Operations materials are being reproduced or transmitted without permission, please call 1-800-326-8724 in the United States and Canada, 1-585-350-7000 in all other countries.

Using Data Science Tools in Python®

Lesson 1: Setting Up a Python Data Science Environment.....	1
Topic A: Select Python Data Science Tools.....	2
Topic B: Install Python Using Anaconda.....	8
Topic C: Set Up an Environment Using Jupyter Notebook.....	14
Lesson 2: Managing and Analyzing Data with NumPy....	29
Topic A: Create NumPy Arrays.....	30
Topic B: Load and Save NumPy Data.....	42
Topic C: Analyze Data in NumPy Arrays.....	51
Lesson 3: Transforming Data with NumPy.....	71
Topic A: Manipulate Data in NumPy Arrays.....	72
Topic B: Modify Data in NumPy Arrays.....	90
Lesson 4: Managing and Analyzing Data with pandas.	111
Topic A: Create Series and DataFrames.....	112
Topic B: Load and Save pandas Data.....	124
Topic C: Analyze Data in DataFrames.....	134
Topic D: Slice and Filter Data in DataFrames.....	152

Lesson 5: Transforming and Visualizing Data with pandas..	165
Topic A: Manipulate Data in DataFrames.....	166
Topic B: Modify Data in DataFrames.....	185
Topic C: Plot DataFrame Data.....	200
Lesson 6: Visualizing Data with Matplotlib and Seaborn.....	213
Topic A: Create and Save Simple Line Plots.....	214
Topic B: Create Subplots.....	224
Topic C: Create Common Types of Plots.....	242
Topic D: Format Plots.....	265
Topic E: Streamline Plotting with Seaborn.....	292
Appendix A: Scraping Web Data Using Beautiful Soup.....	319
Topic A: Scrape Web Pages.....	320
Solutions.....	331
Glossary.....	337
Index.....	341

About This Course

More and more organizations are turning to data science to help guide business decisions. Regardless of industry, the ability to extract knowledge from data is crucial for a modern business to stay competitive. One of the tools at the forefront of data science is the Python® programming language. Python's robust libraries have given data scientists the ability to load, analyze, shape, clean, and visualize data in easy to use, yet powerful, ways. This course will teach you the skills you need to successfully use these key libraries to extract useful insights from data, and as a result, provide great value to the business.

Course Description

Target Student

This course is designed for students who wish to expand their ability to extract knowledge from business data. The target student for this course understands the principles and benefits of data science and has used basic data-driven tools like Microsoft® Excel® and Structured Query Language (SQL) queries, but wants to take the next steps into more advanced applications of data science.

So, the target student may be a programmer or data analyst looking to solve business problems using powerful programming libraries that go beyond the limitations of prepackaged GUI tools or database queries; libraries that give the data scientist more fine-tuned control over the analysis, manipulation, and presentation of data.

A typical student in this course should have several years of experience with computing technology, along with a proficiency in programming.

Course Prerequisites

To ensure your success in this course, you should have at least a high-level understanding of fundamental data science concepts, including but not limited to: data engineering, data analysis, data storage, data visualization, and statistics. You can obtain this level of knowledge by taking the CertNexus DSBIZ™ (*Exam DSZ-110: Data Science for Business Professionals*) course.

You should also be proficient in programming with Python. You can obtain this level of skills and knowledge by taking the following Logical Operations courses:

- *Python® Programming: Introduction*
- *Python® Programming: Advanced*

Course Objectives

In this course, you will use various Python tools to load, analyze, manipulate, and visualize business data.

You will:

- Set up a Python data science environment.
- Manage and analyze data with NumPy arrays.
- Manipulate and modify data with NumPy arrays.
- Manage and analyze data with pandas DataFrames.
- Manipulate, modify, and visualize data with pandas DataFrames.
- Visualize data with Matplotlib and Seaborn.

The CHOICE Home Screen

Logon and access information for your CHOICE environment will be provided with your class experience. The CHOICE platform is your entry point to the CHOICE learning experience, of which this course manual is only one part.

On the CHOICE Home screen, you can access the CHOICE Course screens for your specific courses. Visit the CHOICE Course screen both during and after class to make use of the world of support and instructional resources that make up the CHOICE experience.

Each CHOICE Course screen will give you access to the following resources:

- **Classroom:** A link to your training provider's classroom environment.
- **eBook:** An interactive electronic version of the printed book for your course.
- **Files:** Any course files available to download.
- **Checklists:** Step-by-step procedures and general guidelines you can use as a reference during and after class.
- **Spotlights:** Brief animated videos that enhance and extend the classroom learning experience.
- **Assessment:** A course assessment for your self-assessment of the course content.
- Social media resources that enable you to collaborate with others in the learning community using professional communications sites such as LinkedIn or microblogging tools such as Twitter.

Depending on the nature of your course and the components chosen by your learning provider, the CHOICE Course screen may also include access to elements such as:

- LogicalLABs, a virtual technical environment for your course.
- Various partner resources related to the courseware.
- Related certifications or credentials.
- A link to your training provider's website.
- Notices from the CHOICE administrator.
- Newsletters and other communications from your learning provider.
- Mentoring services.

Visit your CHOICE Home screen often to connect, communicate, and extend your learning experience!

How To Use This Book

As You Learn

This book is divided into lessons and topics, covering a subject or a set of related subjects. In most cases, lessons are arranged in order of increasing proficiency.

The results-oriented topics include relevant and supporting information you need to master the content. Each topic has various types of activities designed to enable you to solidify your understanding of the informational material presented in the course. Information is provided for reference and reflection to facilitate understanding and practice.

Data files for various activities as well as other supporting files for the course are available by download from the CHOICE Course screen. In addition to sample data for the course exercises, the

course files may contain media components to enhance your learning and additional reference materials for use both during and after the course.

Checklists of procedures and guidelines can be used during class and as after-class references when you're back on the job and need to refresh your understanding.

At the back of the book, you will find a glossary of the definitions of the terms and concepts used throughout the course. You will also find an index to assist in locating information within the instructional components of the book. In many electronic versions of the book, you can click links on key words in the content to move to the associated glossary definition, and on page references in the index to move to that term in the content. To return to the previous location in the document after clicking a link, use the appropriate functionality in your PDF viewing software.

As You Review

Any method of instruction is only as effective as the time and effort you, the student, are willing to invest in it. In addition, some of the information that you learn in class may not be important to you immediately, but it may become important later. For this reason, we encourage you to spend some time reviewing the content of the course after your time in the classroom.

As a Reference

The organization and layout of this book make it an easy-to-use resource for future reference. Taking advantage of the glossary, index, and table of contents, you can use this book as a first source of definitions, background information, and summaries.

Course Icons

Watch throughout the material for the following visual cues.

Icon	Description
	A Note provides additional information, guidance, or hints about a topic or task.
	A Caution note makes you aware of places where you need to be particularly careful with your actions, settings, or decisions so that you can be sure to get the desired results of an activity or task.
	Spotlight notes show you where an associated Spotlight is particularly relevant to the content. Access Spotlights from your CHOICE Course screen.
	Checklists provide job aids you can use after class as a reference to perform skills back on the job. Access checklists from your CHOICE Course screen.
	Social notes remind you to check your CHOICE Course screen for opportunities to interact with the CHOICE community using social media.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

1

Setting Up a Python Data Science Environment

Lesson Time: 1 hour, 30 minutes

Lesson Introduction

To get started with applying data science tasks to the business, you'll want to set up an environment that includes the necessary tools. Even though you can install each Python® library directly, you'll find it much easier to manage them in an overall data science platform.

Lesson Objectives

In this lesson, you will:

- Select prominent data science tools that support the Python programming language.
- Install Python and various data science tools using the Anaconda distribution.
- Set up a Python-based environment for performing data science by using Jupyter Notebook.

TOPIC A

Select Python Data Science Tools

Your first move, of course, will be to learn more about the tools you'll be using, as well as to select the specific tools that will best meet your business needs.

Python and Data Science

You're probably familiar with the general concept of data science and what it means for the organization. Still, to understand how it can be implemented using Python, a reminder of the definition is in order: **Data science** is the discipline that involves accumulating data, analyzing the data, extracting value from the data, and presenting the value of the data in a meaningful way.

When Guido van Rossum created Python in 1990, he did not intend for it to become one of the most popular platforms for data science; and yet, over the years, it has become just that. Python is used by data science professionals all over the world in both academic and business settings. In the last few years, it has overtaken other popular languages for data science, like R, in market share. There are several factors that contribute to this, including Python's simple syntax and relative ease of use for new programmers. Data science professionals often come from a background where they're accustomed to using graphical tools like Microsoft® Excel® and Tableau®, so they may not be seasoned programmers. Python makes this transition to coding less of a struggle.

However, the biggest reason for Python's success in data science is the availability of incredibly robust third-party libraries. These libraries provide all of the functionality of the aforementioned definition of data science in an accessible way, while also having a suitable amount of depth for industry veterans. The Python community also promotes an **open source** philosophy, so these third-party libraries are both free in terms of price and free in terms of the right to modify and distribute source code. As a result, many volunteers frequently collaborate to update and improve these libraries.

Machine Learning

Machine learning is a major discipline of artificial intelligence (AI) that is often included in definitions of data science. This discipline and its associated Python libraries are beyond the scope of this course, though you may be interested in exploring them after you've become more experienced with data science fundamentals.

Data Science Tools: An Overview

The following table presents an overview of several prominent data science tools that integrate with or otherwise support the Python programming language.

Tool	Description
NumPy	A Python library that enables you to create and perform mathematical calculations on large, multi-dimensional arrays (matrices). NumPy is a foundational library for data science using Python.
SciPy	Another library in the Python data science stack. It builds on NumPy by offering more powerful mathematical functions, particularly those used in the field of scientific computing. It also includes some of the other data science components discussed in this table, including pandas and Matplotlib.

Tool	Description
<i>statsmodels</i>	A statistical modeling library that also works alongside NumPy and other Python data science components.
<i>pandas</i>	Part of SciPy, a Python library that supports data structures and data analysis functions for Python programming. The primary data structure offered by pandas is the DataFrame.
<i>Matplotlib</i>	Also part of SciPy, a Python library that includes various methods for plotting data on graphs. Matplotlib supports many different kinds of visualization techniques.
<i>Seaborn</i>	A Python library that extends the functionality of Matplotlib by incorporating more types of plots, as well as more sophisticated versions of Matplotlib plots. Seaborn also makes data visualization more attractive.
<i>Jupyter Notebook</i>	A web application that enables users to create, view, and share interactive notebooks—files that include live, executable code, as well as explanatory markup text.
<i>Beautiful Soup</i>	A Python library for scraping and parsing HTML and XML documents.
<i>Anaconda</i>	A cross-platform, open source distribution that includes many Python data science libraries. It also features a package management environment.
 Note: You will use Anaconda in this course's activities.	

Machine Learning Libraries

One of the most common Python libraries for fundamental machine learning is scikit-learn. For deep learning, tools like TensorFlow™, PyTorch™, and Keras are also widely used.

Reference Documentation

You certainly shouldn't expect to memorize each and every module, function, and parameter that is provided by all of the various Python data science libraries. There will be times when you need to look up how to do something, including things that are beyond the scope of this course. When that time comes, you'll get a lot of use out of each library's reference documentation. Reference documentation typically includes a brief definition of a function, its required and optional arguments, its attributes, what it returns, and simple examples. Because many of these libraries are considered to be in the same family, the documentation structure will be fairly consistent. However, keep in mind that reference documentation is just that—a reference. It gives you the facts, not a robust instructional experience.

The following is a list of where to find each library's reference documentation:

- NumPy: <https://docs.scipy.org/doc/numpy/reference/>
- SciPy: <https://docs.scipy.org/doc/scipy/reference/>
- statsmodels: <http://www.statsmodels.org/stable/api.html>
- pandas: <https://pandas.pydata.org/pandas-docs/stable/reference/index.html>
- Matplotlib: <https://matplotlib.org/api/index.html>
- Seaborn: <https://seaborn.pydata.org/api.html>
- Jupyter® Notebook: <https://jupyter-notebook.readthedocs.io/en/stable/>
- Beautiful Soup: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- Anaconda®: <https://docs.anaconda.com/anaconda/reference/>

Hardware Considerations

Whether you use all, or only some, of these tools in testing and production environments, you should be mindful of the hardware they will run on. The hardware resources of your systems will directly affect the performance of your data science operations; if these resources are unsuitable, there may be delays, service outages, or other problems that make it difficult or impossible to achieve your business goals.

For data science projects, storage drive capacity and speed are of special importance. Traditional hard disk technology tends to be the most cost-effective option when large capacities are needed, while solid-state drives (SSDs) are more desirable for the speed of read and write operations. In addition to storage hardware, central processing unit (CPU) technology is also an important consideration. Modern multi-core CPUs, especially CPUs that target the server market, are crucial when it comes to running data science applications. If you plan on taking your data science projects further by incorporating machine learning, and especially deep learning, you'll want to consider the parallel computing power of GPUs.

Factors that should prompt you to carefully evaluate and possibly upgrade your hardware capabilities include:

- **Dataset size.** Large datasets, especially those that qualify as "big data," will require more computing resources to process, as well as more storage space.
- **Scalability.** Resources and requirements are constantly evolving, and your code will need to adapt to changes in scale.
- **Types of operations.** Certain operations, especially those that perform complex statistical analyses, may require a great deal of computing power.
- **Time constraints.** Lower-end hardware can complete most tasks just fine, but the deciding factor may be whether or not it can complete a task quickly enough to meet business requirements.
- **Virtualization.** Virtualization can help streamline resource allocation and management, but the hardware it runs on needs to have adequate support for virtualization techniques.

Rather than taking a reactive approach, you should proactively evaluate your hardware architecture needs before undertaking data science projects. That way, you'll be less likely to encounter disruptions during the project due to poor planning.

Cloud Considerations

Hardware challenges can be a burden, and, depending on your organization's needs, may be too costly or time consuming. An alternative approach is to offload these efforts to cloud services. Many organizations find it more cost-effective to move their data science projects to the cloud due to its elastic nature; the organization pays only for the resources it needs, and it can access virtually unlimited resources on demand. Organizations might move development, storage, processing, testing, or publishing tasks to the cloud. Or, they might offload some combination of these tasks to the cloud, or perhaps all of them. It depends on each organization's needs and its willingness to accept the effort and risks of keeping operations on-premises.

You may not necessarily have the final say in such matters, but if you have the opportunity to work with data science in the cloud, you should be aware that many of the Python tools and environments previously listed may still be available to you. For example, you can create and manage cloud-based virtual machines on a platform like Amazon™ Elastic Compute Cloud (EC2™), and then set up a data science environment like you would with any local computer. Or, you can leverage data-science-specific services like Google's CoLaboratory™, which supports integration with Jupyter Notebook. There are many such options available to you, provided by multiple vendors.

However, there are downsides to relying on the cloud. The organization doesn't have clear ownership of data when it's hosted outside of its domain, which can be an issue when it comes to protecting the security and privacy of that data. Cloud resources are also not impervious to downtime, and the organization may suffer from adverse events that are out of its control.

Guidelines for Selecting Python Data Science Tools



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are selecting Python data science tools.

Select Python Data Science Tools

When selecting Python data science tools:

- Identify your business needs first.
- Identify the goals of your data science tasks.
- Identify which specific Python libraries might help you achieve your goals.
- Identify which libraries may be worth further investigation or experimentation.
- Consider using NumPy to perform calculations on datasets of numeric data.
- Consider using pandas to clean and analyze datasets of varying data types.
- Consider using Matplotlib and Seaborn to help visualize your data.
- Consider installing an overall Python environment like Anaconda to streamline the development process.
- Consult reference documentation for each applicable library whenever you need help with a particular function or module.
- Consider how the hardware you're running these tools on may affect your data science operations.
- Consider how offloading some data science processing to the cloud may be beneficial.
- Consider the risks of moving your operations to the cloud.

ACTIVITY 1–1

Selecting Python Data Science Tools

Scenario

You work for Greene City Emporium (GCE), a small chain of retail stores that operates in the fictitious state of Richland. GCE sells a variety of products from many different manufacturers—everything from food, to housewares, to sporting goods, and more. The business is currently expanding into new territories within Richland, and it's also expanding its product catalog to meet the needs of its customers.

You were hired by GCE a few years ago as an application developer, but recently you've transitioned into a data analyst role. The company is launching a new internal initiative to improve its business processes, so they need someone like you to extract insights from the data they're generating. Before you can start, however, you need to select the right tools for the job.

1. You and your team members are considering using a Python data science platform for most of your analysis and visualization.

Why is Python such a popular programming language for data science? What are its advantages?

2. Which Python library is the foundation on which the other libraries are built?

- pandas
- NumPy
- Matplotlib
- SciPy

3. How does Seaborn differ from Matplotlib?

4. The data analysis team is looking for a single distribution from which they can easily acquire and update their Python data science libraries.

Which tool suits this need?

- Anaconda
- Jupyter Notebook
- SciPy
- BeautifulSoup

5. Because the data analysis team will eventually work with large volumes of data that GCE collects, it will need to decide how best to run its data projects. The projects can run on local hardware, in the cloud, or some mix of the two. The choice comes down to what the company is willing to handle itself vs. what it needs to outsource due to monetary constraints, lack of available resources, or lack of expertise.

If the company were to run these data science projects on local hardware, what are some of the things they need to consider?

6. If the company were to run these data science projects on cloud services, what are some of the things they need to consider?
-

TOPIC B

Install Python Using Anaconda

Anaconda includes many fundamental Python data science libraries, as well as other features that make it easy to manage a development environment. In this topic, you'll establish the environment you'll use for the rest of the course.

Anaconda

Anaconda is a cross-platform, open source distribution that supports two of the most common programming languages used in data science: Python and R. While you can certainly install Python on its own and manually install the libraries you need, Anaconda is actually the preferred way to acquire a data science development environment. This is because Anaconda not only comes pre-packaged with many of these data science tools already installed, but it also enables you to easily acquire more tools if desired.

Some of the major data science libraries that come installed with Anaconda include:

- NumPy
- SciPy
- statsmodels
- pandas
- Matplotlib
- Seaborn
- BeautifulSoup

In addition, there are thousands of other tools available through Anaconda that you might be interested in, such as integrated development environments (IDEs) like Visual Studio Code, and data analysis frameworks like Orange.



Note: The Anaconda distribution installs Python by default, but you can install R later if you wish.

Anaconda Navigator

Anaconda Navigator is a graphical user interface (GUI) for managing software that is available through the Anaconda distribution. It also serves as a portal for software documentation and community interaction.

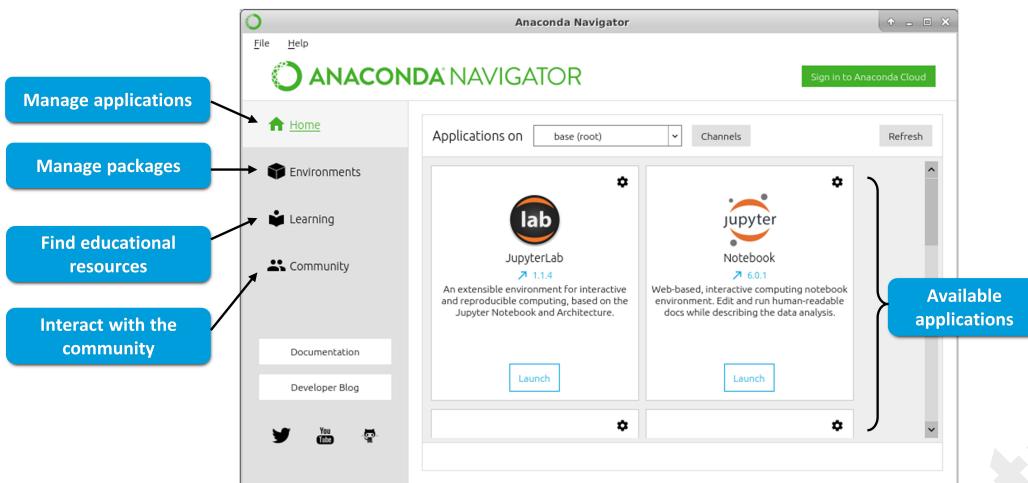


Figure 1–1: The home page of the Anaconda Navigator app.

Anaconda Navigator has a tabular interface, where each tab leads to a page with specific functionality.

Tab	Description
Home	This page presents a dashboard for all of the full-fledged applications that are either installed or available to be installed within Anaconda. Depending on its current state, you can install, update, uninstall, or launch an application from this page.
Environments	This page shows a list of all available packages, environments, and channels (repository locations). From here, you can manage each component by installing, updating, or uninstalling it. There are thousands of available packages, so consider using the sort and search features to find the ones you're looking for more easily.
Learning	This page provides links to documentation for the applications, libraries, and environments you may be working with. In addition, the page features other educational resources like videos and presentations.
Community	This page provides links to community-driven events like developer conferences, as well as to discussion forums like Stack Overflow.

Conda

If you've used Python or even an operating system like Linux® for any significant amount of time, you'll probably be familiar with **package managers**—programs that streamline the process of querying, installing, updating, and uninstalling packaged software. In particular, the most common package manager for installing Python libraries is **pip**. Anaconda, however, provides an alternative package manager called **Conda**.

Conda® is used on the back end by Anaconda Navigator to download and install packages, but like pip, it is also available as a command-line tool. One major difference between pip and Conda is that pip works only with Python packages, while Conda is language agnostic, i.e., it can manage both Python packages and non-Python packages. This distinction is most useful in environments that have dependencies on non-Python packages, like data science environments and full stack development environments. However, Conda can install packages only within a Conda

environment, and not in a pre-existing Python environment (e.g., one installed manually, rather than through Anaconda).

Another major difference is that Conda is much smarter about how it handles package dependencies. When you download and install a package, pip also installs any dependencies that are associated with that package. These dependencies may end up conflicting with other packages installed on the system. Conda, on the other hand, minimizes dependency issues by determining what packages are required for your particular environment. It helps to avoid or otherwise resolve conflicts between packages, guaranteeing a level of compatibility.

The last major difference is that Conda provides isolated virtual environments that contain sets of installed packages. You can use the packages in one environment without affecting the packages in another. This is similar to the isolated nature of a virtual machine or container. This type of setup is common for developers who work on multiple projects that each have different package requirements. It's also helpful for experimenting with certain libraries without causing issues in other, more stable environments.

Deciding Between Conda and pip

When you are making the decision to use Conda or pip, consider the following scenarios:

- If your project has dependencies on non-Python packages and you're fine with working in an isolated Conda environment, then Conda is your best choice.
- If you have a pre-existing Python environment outside of Conda, then you'll need to use pip.
- If you're fine with working in an isolated Conda environment and you're only working in Python, then you can choose either Conda or pip.

The conda Command

You can use the `conda` command to query, install, update, and uninstall Conda packages. You're free to use Anaconda Navigator for these purposes, but you may prefer the flexibility and expedience of a command-line interface. The `conda` command includes several subcommands for working with packages, several of which are described in the following table.

Subcommand	Used To
<code>clean</code>	Remove unused packages.
<code>help</code>	Show documentation for the <code>conda</code> command.
<code>install</code>	Download and install the specified package(s).
<code>list</code>	List all linked packages in the Conda environment.
<code>remove</code>	Uninstall the specified package(s).
<code>search</code>	Search for available packages by name and other details.
<code>update</code>	Update the specified package(s) to the latest version.

For example, to see detailed information about the `numpy` package: `conda search numpy --info`.

Syntax

The syntax of the `conda` command is `conda [subcommand] [package name] [options]`

Anaconda Cloud

Anaconda Cloud is an online repository that enables any user to upload their own software package or programming project free of charge. All files uploaded by free accounts are made public and can

be accessed by any other user. A paid version of Anaconda Cloud provides private repository functionality to organizations.



Access the Checklist tile on your CHOICE Course screen for reference information and job aids on How to Install Anaconda.

Do Not Duplicate or Distribute

ACTIVITY 1–2

Installing Anaconda

Data File

/home/student/DSTIP/Setup/Anaconda3-2020.02-Linux-x86_64.sh

Scenario

The data analysis team at GCE has decided to begin exploring Python's data science capabilities. The Anaconda distribution is a convenient way to acquire what the team needs as far as Python libraries go. So, you've agreed to install Anaconda on your local machine to get started.

	Note: The system on the VM is configured to log the user in automatically. If you are prompted at any time to log in, the account is named student and the password is Pa22w0rd .
	Note: Activities may vary slightly if the software vendor has issued digital updates. Your instructor will notify you of any changes.

1. Start the VM.

- Start the **Oracle VM VirtualBox** application.
- In the **Oracle VM VirtualBox Manager**, from the list on the left, select **DSTIP** and select **Machine→Start→Normal Start**.
- Wait as the virtual machine finishes loading and the operating system starts.



2. Install Anaconda.

- From the top panel on the desktop, select **Applications→Terminal Emulator**.

	Note: You can also open the Terminal from the hidden panel at the bottom of the desktop.
---	---

- At the prompt, type `bash ~/DSTIP/Setup/Anaconda` but don't press **Enter**.
- Press **Tab** to automatically fill in the script name so that the command is now `bash ~/DSTIP/Setup/Anaconda3-2020.02-Linux-x86_64.sh`
- Enter the command.
- Press **Enter** to begin reviewing the license terms.
- Press **Spacebar** to scroll through the license terms, then enter `yes` to accept them.

- g) Press **Enter** to confirm the default installation location.
- h) Wait for the installation to progress. Then, when prompted to initialize Anaconda, enter **yes**
- i) After the installation completes, enter `source ~/.bashrc` to reset the shell session.

3. Start Anaconda Navigator.

- a) At the prompt, enter `anaconda-navigator`
- b) In the **Choose password for new keyring** dialog box, in the **Password** and **Confirm** text boxes, type **Pa22w0rd**
- c) Select **Continue**.
- d) In the **ANACONDA NAVIGATOR** dialog box, select **Ok, and don't show again**.
- e) Verify that you are on the Anaconda Navigator Home page, which displays the dashboard.



4. Get acquainted with the Anaconda Navigator interface and its available features.

- a) Examine the available applications.
Anaconda installs a few different applications out of the box, but the one you'll be working with is Jupyter Notebook.
- b) From the navigation pane, select **Environments**.
By default, you are shown a list of all installed packages for the base environment. Many of these packages were installed automatically.
- c) Scroll down and look for Python data science libraries like **matplotlib**, **numpy**, and **pandas** to verify they've been installed.
You'll be using these libraries and more throughout the course.
- d) From the navigation pane, select **Learning**.
- e) Examine the documentation available to you.
There's everything from general Python programming to library-specific documentation.
- f) Select the **Community** tab.
- g) Examine the Python-related events, forums, and other community resources available to you.
- h) From the navigation pane, select **Home** to return to the list of applications.
- i) Stay on this page.

TOPIC C

Set Up an Environment Using Jupyter Notebook

Now that various Python libraries are installed through Anaconda, you'll want to begin programming. You could start writing scripts or entering statements into a shell, but there is an alternative approach that is commonly used in the Python data science community: Jupyter Notebook.

IPython

IPython is an interactive command shell for the Python programming language. In other words, IPython provides an interactive command-line interface through which users can enter and run Python code. You may already be familiar with the standard Python shell, typically invoked by running the `python` or `python3` command. IPython offers the same basic environment, but it extends the environment with some features that have made it a popular alternative, especially in the realm of data science. Some of those extended features include:

- The IPython kernel, which is what the shell and other programs built on top of the shell access in order to run code interactively.
- A more visually interesting interface.
- Enhanced support for data visualization libraries and GUI frameworks.
- Support for parallel computing, improving the performance of code in execution.
- The ability to run operating system commands within the IPython shell.
- Support for *magic commands*, which are commands unique to IPython that provide additional functionality. For example, the `%timeit` command times the execution of the expression that follows the command.
- Support for ease-of-use features like interactive code completion.

The screenshot shows a terminal window titled "Terminal - IPython: home/student". The window contains the following IPython session:

```
(base) student@debian:~$ ipython
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: var_1 = 5
In [2]: var_2 = 85
In [3]: %timeit var_1 * var_2
254 ns ± 3.24 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [4]: print("The answer is: " + str(var_1 * var_2))
The answer is: 4550
```

Annotations highlight specific features:

- An arrow points from the text "Magic command" to the line `%timeit var_1 * var_2`.
- An arrow points from the text "Code completion" to the word `var_2` in the line `print("The answer is: " + str(var_1 * var_2))`.

Figure 1–2: IPython's interactive shell, showing use of a magic command and code completion.

As development on IPython progressed, the developers added more and more features that extended beyond the shell, and eventually started supporting languages other than Python. Starting with IPython version 4.0 in 2015, the developers decided to split the language-agnostic features into a separate project. Today, IPython is once again focused on providing an interactive environment for Python.

Project Jupyter

The project that inherited IPython's language-agnostic functionality is called **Project Jupyter**. The name *Jupyter* is in part a reference to the three core languages that are commonly used within the project: Julia, Python, and R. However, Project Jupyter supports over 100 programming languages, as one of its goals is to create a flexible, interactive computing environment without placing restraints on the user's preferred language.

Project Jupyter provides several features and applications, the most prominent of which is Jupyter Notebook. The following are additional features and applications:

- Various execution environments, called kernels, for each supported language (including IPython).
- A client for managing kernels.
- Qt console, an enhanced command shell that integrates with kernels.
- Voilà, a program that converts a Jupyter Notebook into a form that is more suitable for sharing and publishing.
- JupyterHub, server software that provides multi-user functionality for Jupyter Notebook.
- JupyterLab, a web-based interface that consolidates some of the above functionality into a unified and highly integrated environment.

Jupyter Notebook

Jupyter Notebook is the core application of Project Jupyter. It is an execution environment derived from the original IPython Notebook, and it provides the user with a web-based interface for entering, running, outputting, and analyzing code in a configurable document format. The Notebook interface goes beyond a typical interactive shell by incorporating rich media capabilities, explanatory markup, modular execution, and other features that make it suitable as a single platform from which a developer can work with every step of the execution process. This makes Jupyter Notebook a popular tool for teaching concepts like data science, but it is also used in both testing and production capacities.

As Jupyter Notebook is a web application, it runs as a web server on the local machine by default and is accessible on port 8888. The user merely connects to the web server by entering **`http://127.0.0.1:8888`** in a browser. The Notebook server can also run as a publicly accessible web server, although for security purposes, you should configure it with an SSL/TLS certificate so that users can connect over HTTPS. After users connect to the server, they are presented with a dashboard from which they can begin working with Jupyter Notebook documents.



Note: Jupyter Notebook will eventually be replaced by JupyterLab. JupyterLab uses the same document format as Jupyter Notebook, however.

Notebook Documents

The Jupyter Notebook document is both the file and the interface through which you work with that file. There are two main elements to the interface:

- The **menu**. This is not part of the document itself, but it enables you to apply various commands to the document.
- The **cells**. These make up the document itself, and they contain the code, markup, and any output that results from execution.

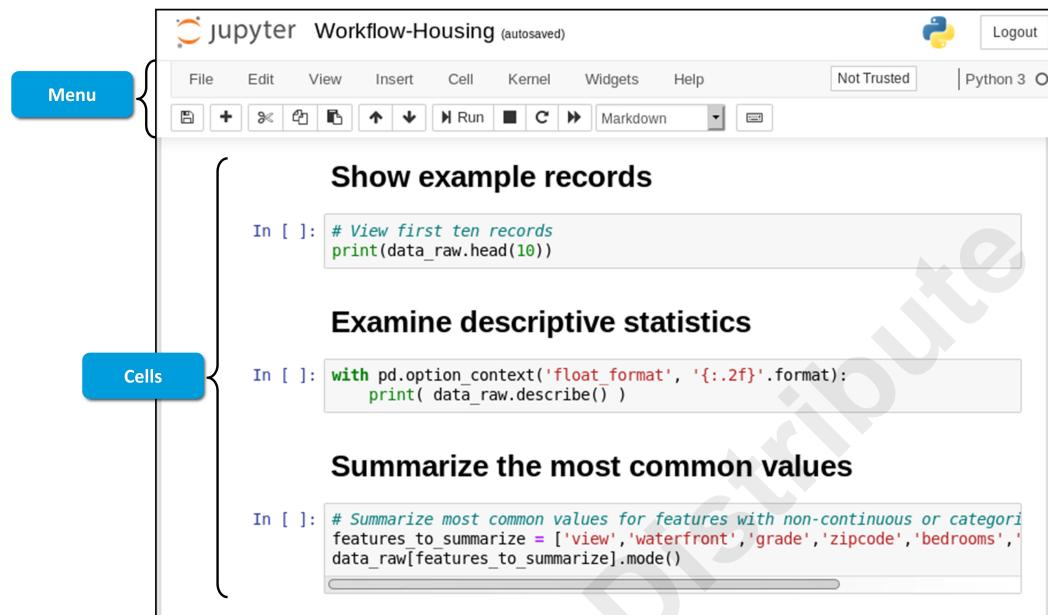


Figure 1–3: The Jupyter Notebook document interface.

The typical workflow of a Notebook document involves entering a block of code into a cell, optionally adding markup to explain the code, then executing the cell that contains the code. Then, you enter more code into the next block, execute that code, and so on. Executing code in this sequential fashion is the most common way to use Jupyter Notebook. The beauty of this approach is that the execution environment maintains its state; in other words, code executed in Cell 1 can affect the code in Cell 2. Depending on the code itself, you may receive textual or graphical output after a cell is executed.

Notebook documents are really nothing more than text files. You can open them in any text editor and see that everything from the main language code, to the markup, to the structure and type of the cells, is defined in the JavaScript Object Notation (JSON) format. The JSON is simple enough to understand, and you would likely have no trouble editing the document in a text editor, if you so desired. When these document files are saved, they are given the .ipynb extension so that they are easily identified by the Jupyter Notebook web app. Documents are easy to share, as another user needs only Jupyter Notebook and the relevant kernel to load and execute any .ipynb file.

Kernels

As mentioned, a *kernel* is a code execution environment that is usually specific to one programming language implementation. Jupyter Notebook leverages kernels for executing code within Notebook documents. The document, which holds the user's code, opens an interactive session with a kernel. Code is passed to the kernel, which returns any output back to the session, enabling the document to return any output to the user. In other words, the kernel is the back end to the document GUI's front end. Among the more than 100 kernels supported by Jupyter Notebook, IPython is the original and is installed by default.

Kernels exist independently of the document(s) they support. Multiple sessions can be opened with a single kernel, meaning that multiple users can work with the same kernel. This can help centralize the allocation of resources in a multi-user environment, especially if each user works from a thin client that has no real processing power of its own. However, this also means that simply closing a document will not stop the running process associated with the kernel; the process will continue to run in the background. A user must deliberately shut down the associated kernel to free up those resources.

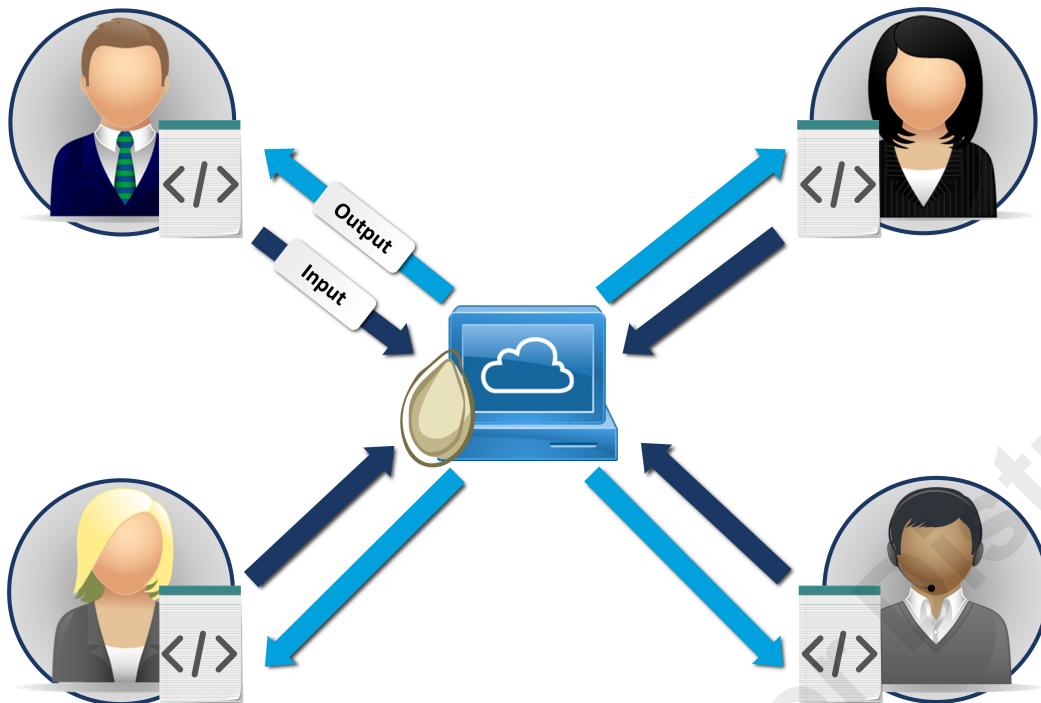


Figure 1–4: Multiple users accessing the same kernel.

Jupyter Notebook Dashboard

When you first connect to Jupyter Notebook through a browser, you are presented with a dashboard. This dashboard shows the **Files** tab by default—a directory structure of the currently logged-in user's home directory. You can select folders to drill down into them, or you can check the check box next to a folder to edit it, similar to most operating system file browsers. Every non-hidden file in a folder, regardless of whether or not it's an .ipynb file, is shown. You simply select the file to open it, or check its check box to perform various actions on it. You can also use the menu to "upload" a file (which essentially copies a file from one location to the current directory) or to create a new file. There are also sorting and filtering options available. The **Files** tab is primarily how you will find and open Notebook documents, as well as other resources like data files and images, when you are working with Jupyter Notebook.

In addition to the **Files** tab, the **Running** tab shows all of the Notebook documents that are currently attached to running kernels. You can shut down multiple kernel processes from this page. The **Clusters** tab shows any running clusters if you are leveraging IPython's parallel computing capabilities.

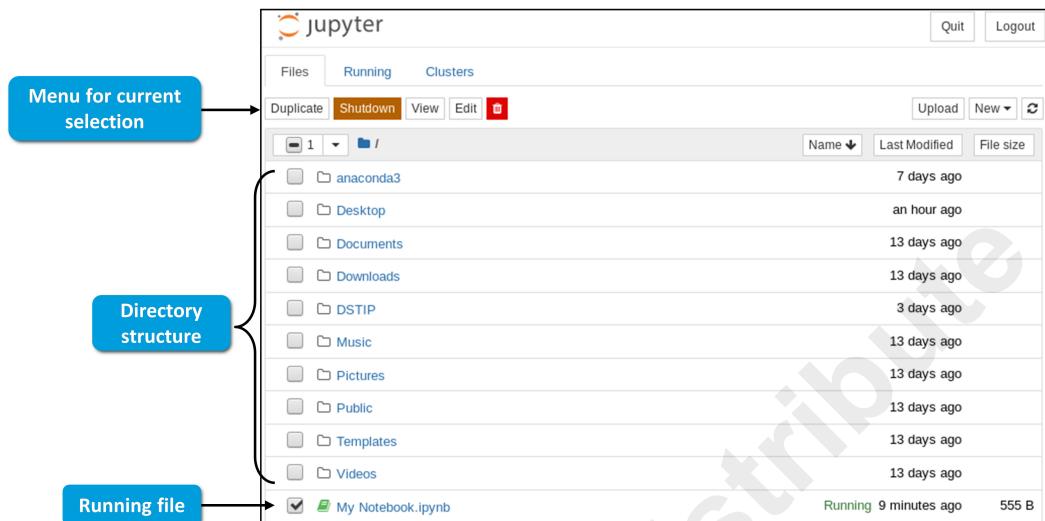


Figure 1–5: The Jupyter Notebook dashboard.

Notebook Document Cells

The cells in a Notebook document typically come in two flavors: code cells and markup cells.

- **Code cells** contain the programming language code that you intend for the kernel to execute. For example, if you're using the default IPython kernel, you'll be writing your Python code here. Any code you can write in a terminal can go here; Jupyter Notebook will send the code to the kernel for execution, which will return an output (if there is any). The output for a code cell is displayed after the code itself, although it is still considered part of the same code cell.
- **Markup cells** contain text that is not executed by the kernel, but will still display within the document. The markup language that Jupyter Notebook uses is called Markdown, which is relatively simple to learn and easy to use. The advantage of Markdown is that it enables you to format text in different ways, including formatting text as a heading, selectively adding or removing formatting to specific characters, configuring mathematical script by using LaTeX notation, and more. Markup cells are often used to explain, describe, or otherwise provide context for one or more code cells that are either before or after the markup cell.

Each code cell has one of three possible states: not executed, currently executing, and finished executing. This is indicated by the `In` text to the side of each code cell:

- `In []` —The code has not yet been executed.
- `In [*]` —The kernel is currently executing the code.
- `In [n]` —The code has finished executing, where n is a number indicating the order in which the cell was executed. The first cell typically shows `In [1]`, the second `In [2]`, and so on. However, you can re-execute a code cell at any point, so that cell will adopt the next highest number.

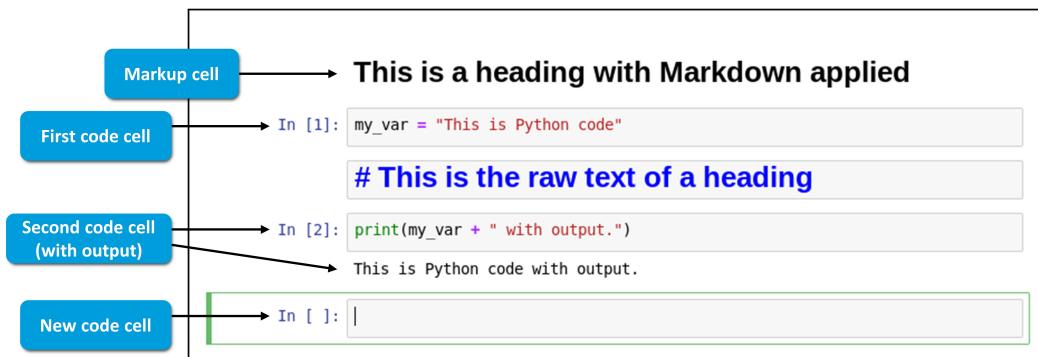


Figure 1–6: A Notebook document with various cells. Notice how the variable in the first code cell is usable by the second code cell.

Common Markdown Formatting

A full discussion of Markdown is beyond the scope of this course, but here are a few common formatting options:

- # —Formats the following text as a heading.
- *text* —Formats the enclosed text with italics.
- **text** —Formats the enclosed text as bold.
- `text` —Formats the enclosed text as in-line code.

Notebook Document Menu

The menu portion of the Notebook document interface enables you to interact with a document in many ways. There are two components of the menu: the menu bar and the toolbar. The menu bar includes all available commands in a typical drop-down menu format; the toolbar includes buttons for some of the most common commands, as well as a drop-down list for selecting the cell type.

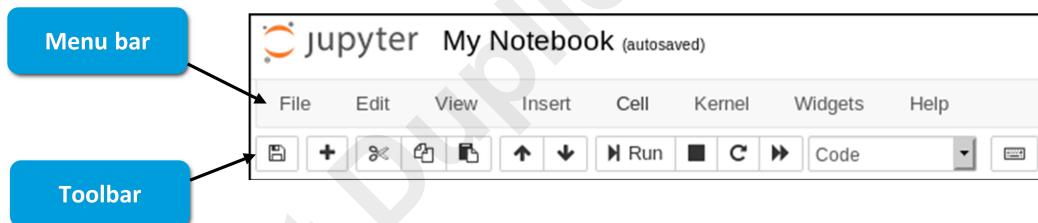


Figure 1–7: The document menu.

The following table describes each menu bar category and the commands it provides.

Category	Includes Commands For
File	Creating new documents, opening existing documents, saving documents, renaming documents, converting documents to other formats, and closing documents and halting the associated kernel.
Edit	Cutting, copying, pasting, deleting, splitting, merging, and moving cells. Also includes a find and replace command.
View	Hiding/unhiding screen elements and toggling line numbers.
Insert	Inserting cells.

Category	Includes Commands For
Cell	Running cells, changing cell type, and managing cell outputs. This gives you the flexibility to run only the selected cell, run all cells at once, or run only cells below or above the current selection.
Kernel	Interrupting, restarting, reconnecting to, and shutting down the attached kernel. You can also change kernels here. Restarting a kernel is useful when you want to clear the document, or you've made some change in code and want to reset the execution environment to a clean state.
Widgets	Saving and clearing the state of Notebook widgets. Widgets extend the GUI functionality of a document, such as adding sliders, check boxes, radio buttons, drop-down menus, etc.
Help	Accessing various tutorials and reference documentation for Jupyter Notebook and installed programming libraries.

	Access the Checklist tile on your CHOICE Course screen for reference information and job aids on How to Create, Modify, and Delete Jupyter Notebook Documents.
---	--

	Access the Checklist tile on your CHOICE Course screen for reference information and job aids on How to Run Jupyter Notebook Documents.
---	---

ACTIVITY 1–3

Setting Up a Jupyter Notebook Environment

Before You Begin

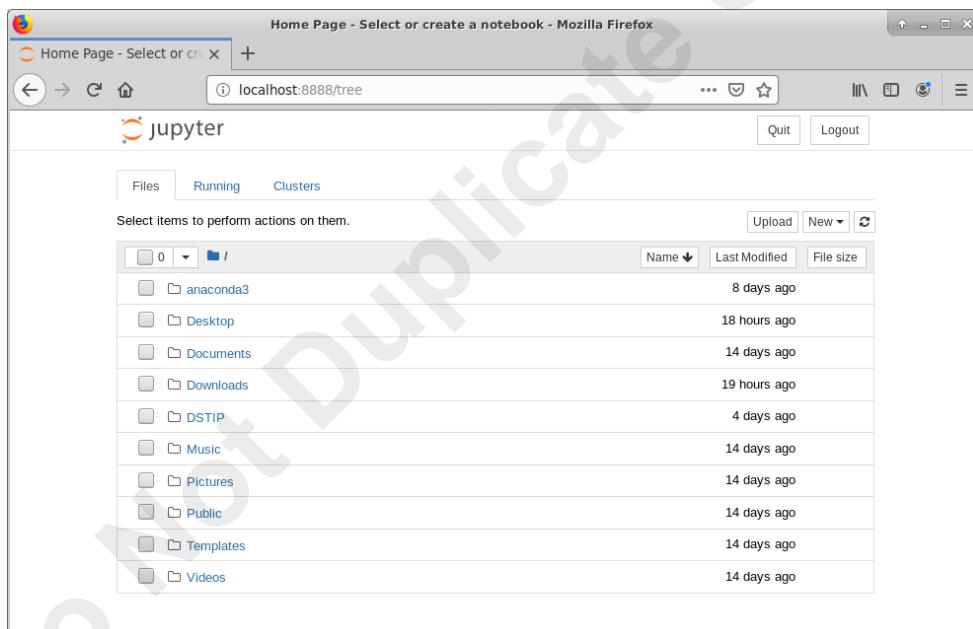
Anaconda Navigator is open to the **Home** tab.

Scenario

While the data analysis team may eventually use a large-scale development and deployment platform, for now it needs the convenience of running interactive Python code without much heavy integration. Several team members have used Python shells in the past, but they're a little too limited. Thankfully, Anaconda automatically installed Jupyter Notebook, a graphical scripting and execution environment. Jupyter Notebook can help the team process data in many ways and see the results, all within an interface that flows well and is easy to edit. You've volunteered to test this environment to see how well it can accommodate the team's workflow.

1. Open Jupyter Notebook.

- In the list of available applications, under **Jupyter Notebook**, select **Launch**.
- Verify that Firefox opens to the Jupyter Notebook dashboard.



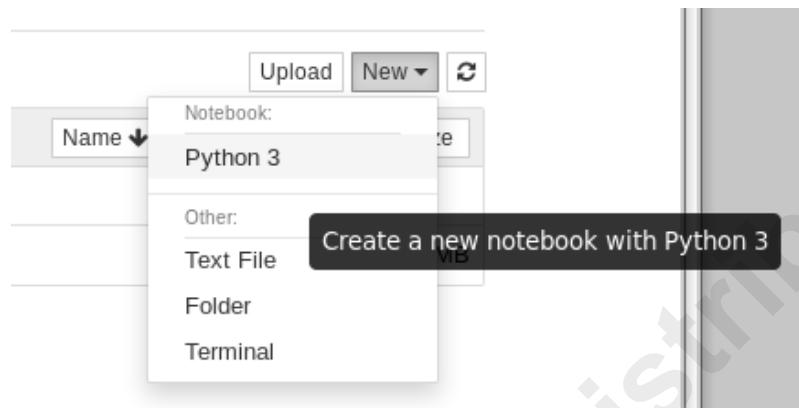
Note: If any additional tabs or panes appear, close them.

2. Create a new Jupyter Notebook document.

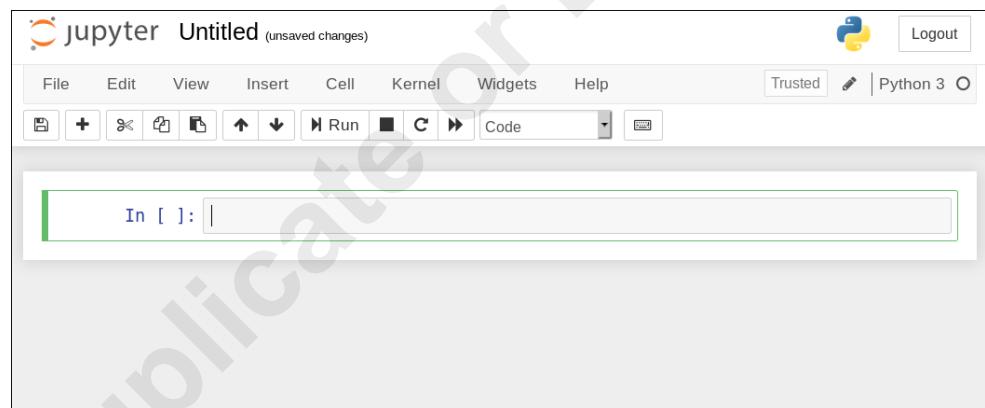
- Examine the directory tree to verify that it is displaying the contents of your home directory. This includes folders like **Documents**, **Downloads**, **Music**, **Pictures**, and more.
- Select the **DSTIP** folder to open it.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

- c) Select **Setup**.
- d) In the **Setup** folder, verify that Jupyter Notebook is displaying the Anaconda installation file that is in this directory.
- e) Select **New→Python 3**.



- f) Verify that a new tab opens in Firefox and that a blank Notebook document is displayed.



3. Rename the Notebook document.

- a) On the menu bar, select **File→Rename**.
- b) In the dialog box, type **Hello Jupyter** and then select **Rename**.
- c) Verify that the name of the document has been changed.

jupyter Hello Jupyter (autosaved)

4. Write and execute a simple Python statement.

- a) In the first blank code cell, type the following code:

```
In [ ]: print('Hello, Jupyter!')
```

- b) On the toolbar, select the **Run** button.



Note: You can also press **Shift+Enter** to run a cell.

- c) Observe that the code cell produced an output, and that a new blank cell was automatically added.

```
In [1]: print('Hello, Jupyter!')
Hello, Jupyter!
```

In []: |

The `In [1]` next to the first code cell indicates that it was the first cell to run in this kernel session.

5. Add explanatory markup to the document.

- Ensure that the blank cell is selected.
- On the toolbar, select the **Code** drop-down list, then select **Markdown**.
- Observe that the `In []` next to the cell was removed, as markup cells are not executed by the kernel.
- Select inside the markup cell and type **# Test Jupyter Notebook functionality**
- Verify that you can see the raw markup.

```
In [1]: print('Hello, Jupyter!')
Hello, Jupyter!
```

Test Jupyter Notebook functionality

Although the Markdown text isn't executed by the kernel, it still needs to "run" in order to be converted to rich text.

- With the markup cell selected, select **Run**.
- Verify that the text now appears as a heading.

```
In [1]: print('Hello, Jupyter!')
Hello, Jupyter!
```

Test Jupyter Notebook functionality

In []: |

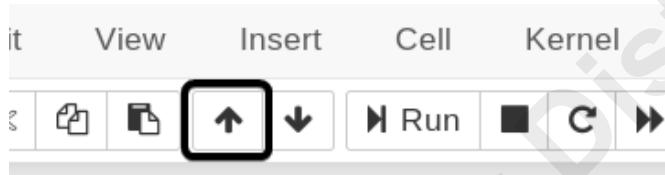
- h) Select the markup cell once to make it the current selection.

In [1]: 1 print('Hello, Jupyter!')
Hello, Jupyter!

Test Jupyter Notebook functionality

In []: 1

- i) On the toolbar, select the **move selected cells up** button.



- j) Verify that the heading now comes before the code cell it describes.

Test Jupyter Notebook functionality

In [1]: print('Hello, Jupyter!')
Hello, Jupyter!

In []:

6. Test the execution state from one code block to the next.

- a) In the blank code cell, type the following code:

In []: my_var = [1, 2, 3]

- b) Select **Run** to run the cell.
c) Observe that, since your code is just defining a variable, no output appears.
d) In the next blank cell, type the following code:

In []: print(my_var)

- e) Run the cell.
f) Verify that the list is printed with the expected values.
g) Select inside the second code cell (the one that defines `my_var`).

- h) In the list definition, change the integer 1 to 0.
- i) Select the next code cell—the one that prints `my_var`—and run it.
- j) Observe that, despite changing the value of the variable in the previous cell, this cell's output does not change when re-run.

7. Why do you think this is?

8. Correct the issue.

- a) Select the cell that defines `my_var` and run it.
- b) Observe that it now has the highest number in the order of execution (i.e., In [5]).
- c) Run the next cell, which has automatically been selected.
- d) Observe that the output has been updated to include the new value for `my_var`.

9. Restart the kernel.

- a) On the menu bar, select **Kernel→Restart & Run All**.
- b) In the dialog box, select **Restart and Run All Cells**.
- c) Verify that every cell was run sequentially.

Test Jupyter Notebook functionality

```
In [1]: print('Hello, Jupyter!')
Hello, Jupyter!

In [2]: my_var = [0, 2, 3]

In [3]: print(my_var)
[0, 2, 3]
```

- d) Select **Kernel→Restart & Clear Output**.
- e) In the dialog box, select **Restart and Clear All Outputs**.
- f) Verify that the Notebook document is now in a clean, unexecuted state.

Test Jupyter Notebook functionality

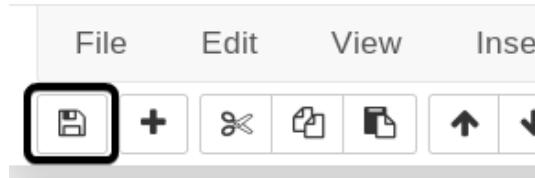
```
In [ ]: print('Hello, Jupyter!')
Hello, Jupyter!

In [ ]: my_var = [0, 2, 3]

In [ ]: print(my_var)
```

10. Save the document, and then shut down the kernel.

- a) On the toolbar, select the **Save and Checkpoint** button.



- b) Select **File→Close and Halt**.

This shuts down the associated kernel and closes the Notebook document.

11. Shut down Jupyter Notebook.

- a) Close Firefox.
 - b) Close Anaconda Navigator.
 - c) In the **Close running applications** dialog box, check **notebook**, then select **Quit**.
 - d) Select **Yes** to quit.
 - e) Close the terminal.
-

Summary

In this lesson, you set up a Python data science environment using Anaconda and Jupyter Notebook. You'll be able to leverage these powerful tools to perform critical data science tasks in a helpful and feature-rich environment. While this is not the only way to process and analyze data in Python, these tools provide a workflow that is helpful to data science beginners and veterans alike.

Would you consider using Jupyter Notebook in your own job? Why or why not?

What other non-Python data analysis tools have you used in the past, or still currently use?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

2

Managing and Analyzing Data with NumPy

Lesson Time: 1 hour, 45 minutes

Lesson Introduction

The foundation of data science in Python® is NumPy. Most of your work will involve NumPy, whether directly or indirectly. So, you'll leverage the power of this library to manage your data and extract useful insights from that data.

Lesson Objectives

In this lesson, you will:

- Create basic NumPy arrays.
- Load existing datasets into NumPy arrays, and save NumPy arrays as files.
- Analyze data in NumPy arrays.

TOPIC A

Create NumPy Arrays

To begin, you'll write code to create the fundamental data structure in Python data science—the NumPy array.

Traditional Python Lists

In Python, a list is an ordered, mutable data type that holds a sequence of values. Recall that there are many ways to create a list object, but the most basic way is to enclose the values in brackets `[]` and separate each value by a comma:

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

Lists are great for processing collections of data, especially data of varying types. But, lists have a couple of notable flaws. The first has to do with how Python handles any of its standard objects, not just lists. Unlike languages like C, Python is dynamically typed; you don't need to explicitly declare a variable's data type when you define that variable. For example, `my_list` is not declared with any data type—Python automatically interprets this as a list because of the syntax used to define it.

Dynamic typing increases programming flexibility and tends to simplify code, but it comes at a cost. A Python variable does not just point to a location in memory that contains the proper data type encoding, like a variable in a statically typed language would. A Python variable is an object with multiple attributes, including the variable's type, its actual value, its size, how many times it's been referenced, and potentially more. These additional components add overhead to a Python object's performance. Objects like lists are even more susceptible to performance issues because of their flexibility. A list is not only its own Python object, but it also points to a block of pointers in memory, and those memory pointers point to the individual Python objects that comprise the list. This adds quite a bit more overhead. If all of a list's values are of the same data type—integers, for example—then pointing to each individual integer becomes redundant.

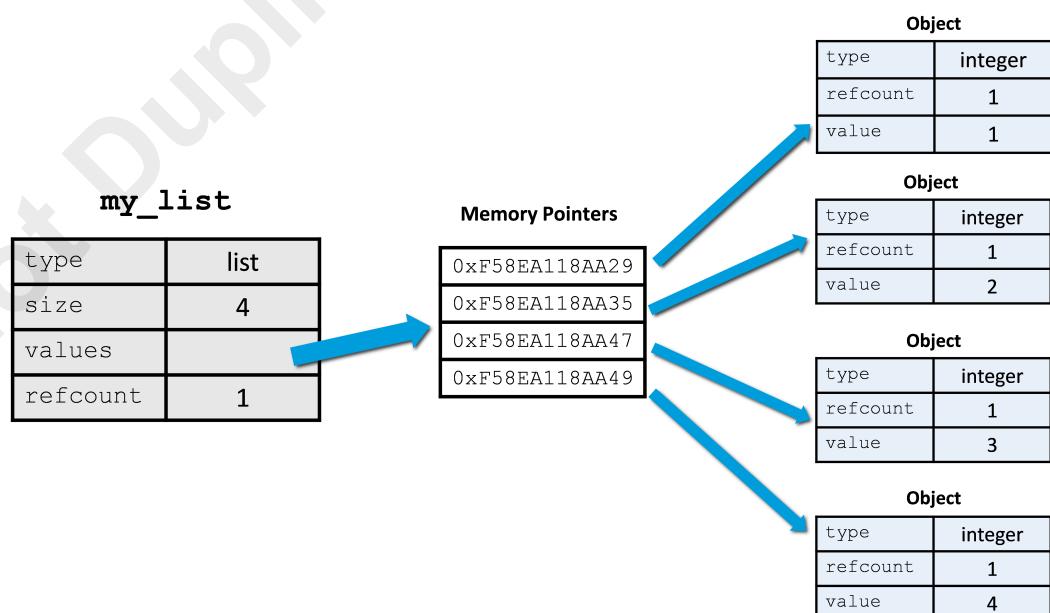


Figure 2–1: The basic structure of a Python list.

The ndarray Object

In the field of data science, large datasets are common. As more data goes into a list, the performance issue worsens considerably. To address this problem, a different approach to storing data is needed. Python actually includes a statically typed `array` object to alleviate some of the performance issues with lists, but this still falls short when it comes to manipulating the data in arrays. Instead, NumPy introduces the highly efficient `ndarray` object.

The `ndarray` object is the fundamental component of NumPy, and it is usually just referred to as a [NumPy array](#). It is a multi-dimensional, single-type structure that is widely used in data science for the efficient storage and processing of data. Unlike Python lists, NumPy arrays don't have the overhead of pointing to a group of pointers, which in turn point to individual objects. Instead, a NumPy array object points directly to a contiguous block of data in memory, also called a data buffer.

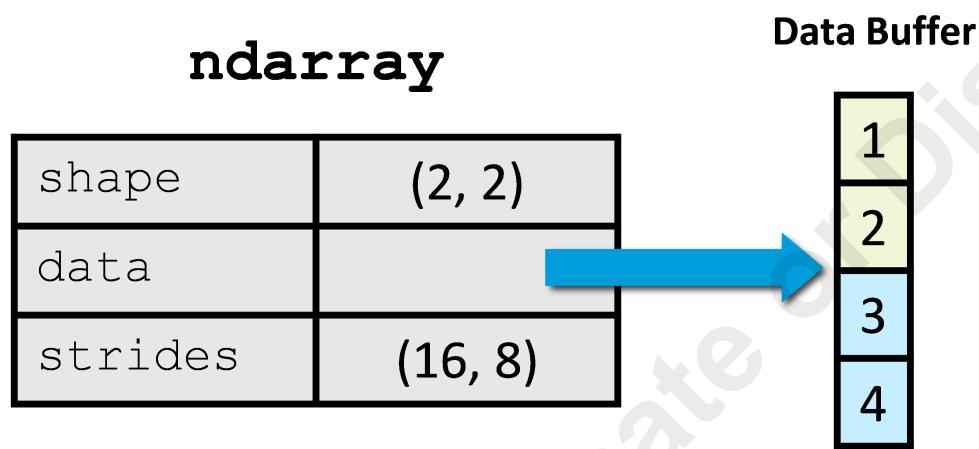


Figure 2-2: The basic structure of a NumPy array.

NumPy arrays, therefore, increase performance, not just in storing data in memory, but for applying mathematical and other transformative operations to an array. If you were to perform an operation on every member of a Python list (multiplication, for example), the interpreter would need to verify each member's data type in order to determine the function to use for the operation. In a NumPy array, no such data type lookup is necessary, and the operation can be carried out quickly.

Homogeneous Data

The main drawback to NumPy arrays is that all members of the array must be the same data type. You cannot have an array with a mix of integers, floats, and strings. It must be all integers, all floats, all strings, etc. Therefore, NumPy is most useful in situations where data is homogeneous or can be homogenized. Even if your raw data is of a mixed type, you may still be able to load it into a NumPy array without an unacceptable loss in data quality. NumPy will automatically perform type conversion in certain situations.

For example, in the following array, all of the items are integers except one (a float):

```
[1, 2.2, 3, 4]
```

When you create a NumPy array object with these values, NumPy will automatically convert all of the integers to floats:

```
[1., 2.2, 3., 4.]
```

This is done to preserve data accuracy. If the opposite conversion were performed—turning the float into an integer—important data would be lost. When string types are thrown into the mix, NumPy will convert all values to strings. This array:

```
[1.3, '2', 3, 4]
```

Is turned into this array:

```
['1.3', '2', '3', '4']
```



Note: NumPy arrays are most commonly used with numerical data, though you can certainly store and process string data with them.

The `numpy.array()` Function

The `numpy.array()` function is used to construct an `ndarray` object. In other words, you can use this function to create a NumPy array with given data. For example:

```
numpy.array([1, 2, 3, 4, 5, 6])
```

This creates a NumPy array of integers. The syntax should look familiar—you're essentially just passing a list into the function, which will use that list to construct the array. If you supply mixed data types, the array will do type conversion here.

As mentioned previously, NumPy arrays are multi-dimensional. This is one of the other major advantages that they have over traditional Python lists. The NumPy array in the prior example has one dimension. However, the following array has two dimensions:

```
numpy.array([[1, 2, 3], [4, 5, 6]])
```

Notice how multiple lists are now nested within another list. This creates a second dimension. It's usually easier to grasp this shift to two dimensions when you depict the array as a table:

```
numpy.array([[1, 2, 3],
            [4, 5, 6]])
```

So, each inner list represents a row, and each column is filled out according to the order of each list item. Being able to represent data in two dimensions is extremely useful when working with tabular datasets or performing matrix calculations. In addition, you can continue to nest lists to increase the array's dimensionality past two dimensions.

The `numpy.ndarray` Class

The `numpy.array()` function actually calls the `numpy.ndarray` class to create the object. You could call `numpy.ndarray` directly, but it is better practice to just use the `numpy.array()` function.

NumPy Array Attributes

Now that you know how to construct a simple NumPy array, you can begin to learn more about that array. Each array object has multiple attributes, and you can retrieve these attributes by making the appropriate call on the array object. Some of these attributes are described in the following table, along with examples. Each example output assumes that `x` is defined as follows:

```
x = numpy.array([[1, 2], [3, 4], [5, 6]])
```

Attribute	Description and Example
<code>x.ndim</code>	Returns the number of the array's dimensions. Example output: 2
<code>x.shape</code>	Returns a tuple of the array's shape, i.e., number of rows by number of columns. Example output: (3, 2)

Attribute	Description and Example
<code>x.size</code>	Returns the number of items in the array. Example output: 6
<code>x.itemsize</code>	Returns the number of bytes long each element in the array is. Example output: 8
<code>x.nbytes</code>	Returns the number of bytes long the entire array is. Example output: 48
<code>x.dtype</code>	Returns the type of data in the array. Example output: int64

NumPy Data Types

As mentioned earlier, one NumPy array can include only a single type of data. However, there are many data types to choose from. You can rely on NumPy to infer this type based on the Python syntax you use, and you can also rely on NumPy to perform type conversion. However, there are times when you may want to explicitly define the type a NumPy array should use. You can do this while creating an array by passing a data type value into the `dtype` parameter for the `numpy.array()` function.

For example, say you have a dataset that is mostly integers, but also has a few integers that are mistakenly defined as string literals:

```
numpy.array([[1, '2', 3], ['4', 5, 6]])
```

NumPy will automatically convert every item in this array into a string. If you need them all to be integers, and you know that all of the mistaken strings literals are otherwise in an integer-ready format, then you can create the array as follows:

```
numpy.array([[1, '2', 3], ['4', 5, 6]], dtype = 'int64')
```

Note that, if any of the string literals aren't in an integer-ready format (e.g., one of the items is 'A'), then NumPy will throw an error. So, you need to be sure all of your data will fit into the type you're trying to convert to.

Data Types Supported by NumPy

The following table lists some of the most common data types supported by NumPy.

Data Type	Description
<code>int8</code>	8-bit signed integer (-128 to 127)
<code>int16</code>	16-bit signed integer (-32,768 to 32,767)
<code>int32</code>	32-bit signed integer (-2,147,483,648 to 2,147,483,647)
<code>int64</code>	64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
<code>uint8</code>	8-bit unsigned integer (0 to 255)
<code>uint16</code>	16-bit signed integer (0 to 65,535)
<code>uint32</code>	32-bit signed integer (0 to 4,294,967,295)
<code>uint64</code>	64-bit signed integer (0 to 18,446,744,073,709,551,615)
<code>float16</code>	16-bit (half-precision) floating-point value

Data Type	Description
float32	32-bit (single-precision) floating-point value
float64	64-bit (double-precision) floating-point value
complex64	64-bit complex number represented by two 32-bit floats (one for real numbers, one for imaginary numbers)
complex128	128-bit complex number represented by two 64-bit floats (one for real numbers, one for imaginary numbers)
bool_	Boolean (true or false) value stored as a byte

Additional Array Creation Functions

You can use the `numpy.array()` function to manually populate an array or to programmatically populate an array (e.g., by looping over ranges). However, NumPy offers different functions for automatically creating an array with certain kinds of data. Some of these functions might be useful for creating "dummy" data, whereas others are helpful as operands in mathematical calculations. These functions also serve a purpose when you are using techniques like regression.

These functions have arguments that you can use to further specify the desired output; otherwise, the default arguments are used. Like `numpy.array()`, all of these functions enable you to specify the data type to use through the `dtype` argument. The other available arguments differ by function. The following table describes several alternative array creation functions.

	Note: Argument names are delineated by <i>italic</i> font. In the examples, the >>> symbols indicate the Python command being provided as input and the ... symbols indicate a continuation of the previous line's input. The output is indicated by a lack of preceding symbols.
Function	Description and Example
<code>numpy.zeros(shape)</code>	Returns an array filled with all zeros in the given shape. <pre>>>> numpy.zeros((2, 3)) array([[0., 0., 0.], [0., 0., 0.]])</pre>
<code>numpy.ones(shape)</code>	Returns an array filled with all ones in the given shape. <pre>>>> numpy.ones((2, 3)) array([[1., 1., 1.], [1., 1., 1.]])</pre>
<code>numpy.full(shape, fill_value)</code>	Returns an array filled with the provided fill values in the given shape. <pre>>>> numpy.full((2, 3), 5) array([[5, 5, 5], [5, 5, 5]])</pre>
<code>numpy.eye(N)</code>	Returns an identity matrix (a 2-D array in which all values are 0 except for values on the diagonal, which are 1). The N argument is the number of rows. <pre>>>> numpy.eye(3) array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])</pre>

Function	Description and Example
<code>numpy.linspace(start, stop, num)</code>	Returns an array of evenly spaced numbers that are between the specified start and stop points. The num argument is the number of values to place in the array. <pre>>>> numpy.linspace(0, 10, 3) array([0., 5., 10.])</pre>
<code>numpy.arange(start, stop, step)</code>	Returns an array similar to <code>numpy.linspace()</code> , but instead of specifying the number of values in the array, you specify the space between numbers (the step). <pre>>>> numpy.arange(0, 10, 3) array([0, 3, 6, 9])</pre>

The `numpy.random` Module

The `numpy.random` module provides numerous functions for creating random data. Random generation is often used for testing, but it can also be useful for simulating natural processes, making random routines more deterministic, and more. The `numpy.random.seed()` function, for example, seeds the random number generator. NumPy, like most software, doesn't use a true random number generator, but a pseudorandom number generator (PRNG). PRNGs are initialized from a seed value, and if two instances of a PRNG use the same seed value, they will return the same output. So, a seed is used to make output more deterministic. You can provide any integer as the seed value argument.

Other than seeding, some of the most common randomization functions are described in the following table. The example outputs are determined by a random seed value of 20.

	Note: In the table, the <code>numpy</code> namespace is omitted for brevity.
Function	Description and Example
<code>random.randint(low, high, size)</code>	Returns an array of the given shape (<code>size</code>) filled with random integers. The <code>low</code> and <code>high</code> parameters are the lower and upper bounds of the random numbers, respectively. <pre>>>> random.randint(0, 5, (2, 3)) array([[3, 2, 4], [2, 1, 4]])</pre>
<code>random.random(size)</code>	Returns an array of the given shape filled with random floats between 0.0 and 1.0. <pre>>>> random.random((2, 3)) array([[0.03588, 0.69175, 0.37868], [0.51851, 0.65795, 0.19385]])</pre>
<code>random.shuffle(x)</code>	Returns an array whose sequence has been randomly shuffled. This requires an existing array to be passed in as <code>x</code> . In multi-dimensional arrays, only the first axis (the axis of rows) is shuffled. <pre>>>> x = arange(0, 10, 3) >>> random.shuffle(x) >>> x array([6, 9, 3, 0])</pre>

Function	Description and Example
<code>random.normal(loc, scale, size)</code>	Returns an array of the given shape whose values follow a normal (Gaussian) distribution. The <code>loc</code> argument specifies the mean of the distribution, and <code>scale</code> specifies the standard deviation. <pre>>>> random.normal(5, 2, (2, 3)) array([[4.66873, 8.44657, 4.19854], [6.59097, 3.26662, 10.13637]])</pre>

Guidelines for Creating NumPy Arrays



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are creating NumPy arrays.

Create NumPy Arrays

When you are creating NumPy arrays:

- Consider using the NumPy `ndarray` object over traditional Python lists for performance benefits.
- Use NumPy when you are working with homogeneous data.
- Use array attributes to retrieve metadata about an array.
- Familiarize yourself with the types of data supported by NumPy.
- Consider how mixed data is converted to a single data type when imported into a NumPy array.
- Use array creation functions like `linspace()` and `randint()` to generate artificial data.

ACTIVITY 2–1

Creating NumPy Arrays

Data File

/home/student/DSTIP/NumPy/Creating Arrays.ipynb

Scenario

To get started with NumPy, you'll explore the basics of creating NumPy arrays. While most of Greene City Emporium's data will come from existing sources, you should be comfortable with creating arrays on the fly, as this can serve several purposes.

1. Open Jupyter Notebook.

- On the desktop, double-click the **Jupyter** shortcut.
- Verify that Firefox opens to the Jupyter Notebook dashboard.
- Navigate to **DSTIP/NumPy/**.
- Select **Creating Arrays.ipynb** to open it.
- Select **View→Toggle Line Numbers**.

2. Import the relevant software libraries.

- View the cell titled **Import software libraries**, and examine the code listing below it.

```

1 import sys
2 import numpy as np
3
4 # Summarize software libraries used.
5 print('Libraries used in this project:')
6 print('- Python {}'.format(sys.version))
7 print('- NumPy {}'.format(np.__version__))

```

- Select the cell that contains the code listing, then select **Run**.
- Verify that the version of Python is displayed, as is the version of NumPy that was imported.

```

Libraries used in this project:
- Python 3.7.6 (default, Jan  8 2020, 19:59:22)
[GCC 7.3.0]
- NumPy 1.18.1

```



Note: Importing NumPy as `np` is a convention, though not required.

3. Create a simple NumPy array.

- Scroll down and view the cell titled **Create a simple NumPy array**, then select the code cell below it.

- b) In the code cell, type the following:

```

1 my_arr = np.array([[1.5, 2, 3],
2                   [4, 5, 6],
3                   [7, 8, 9],
4                   [10, 11, 12]])
5 my_arr

```

- c) Run the code cell.
d) Examine the output.

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ],
       [ 7. ,  8. ,  9. ],
       [10. , 11. , 12. ]])
```

- You create a two-dimensional array using double brackets.
 - The array is arranged similar to a table with rows and columns, and each number is in its own cell.
 - By referencing the array variable as the last line in a cell, the kernel outputs the object `array()` and its data. If you use a `print()` statement, the output shows just the data.
 - All of the values in the array include a decimal point, implying that the array is of a float type. However, all but one of your input values was an integer. NumPy automatically converted the entire array into a float type to maintain precision, as an array can only hold one type of data.
- e) Select the next code cell, and then type the following:

```

1 print('There are {} dimensions in the array.'.format(my_arr.ndim))
2 print('There are {} rows and {} columns in the array.' \
      .format(my_arr.shape[0], my_arr.shape[1]))
3
4 print('There are {} items in the array.'.format(my_arr.size))
5 print('The array holds data of type {}.'.format(my_arr.dtype))

```



Note: The backslash character (\) indicates that the code statement continues on the next line.

- f) Run the code cell.
g) Examine the output.

```
There are 2 dimensions in the array.
There are 4 rows and 3 columns in the array.
There are 12 items in the array.
The array holds data of type float64.
```

The shape, size, and type of the array are as expected.

4. Attempt to create a NumPy array with rows of unequal length.

- a) Scroll down and view the cell titled **Attempt to create a NumPy array with rows of unequal length**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 my_arr = np.array([[0, 1.5, 2, 3],
2                   [4, 5, 6],
3                   [7, 8, 9],
4                   [10, 11, 12]])
5 my_arr
```

- c) Run the code cell.
d) Examine the output.

```
array([list([0, 1.5, 2, 3]), list([4, 5, 6]), list([7, 8, 9]),
       list([10, 11, 12])], dtype=object)
```

Instead of a normally formatted NumPy array of numbers, Python returned an array of lists with an object-based data type. There might be some scenarios where this is desired, but for the most part, it'll be difficult to work with something like this. It's important to make sure your arrays have rows of equal length.

5. Generate NumPy arrays using ranges of data.

- a) Scroll down and view the cell titled **Generate NumPy arrays using ranges of data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 lin_arr = np.linspace(0, 100, 20, dtype = 'int64')
2 print(lin_arr)
3 print('\nThis array has {} dimension(s.)'.format(lin_arr.ndim))
```

- c) Run the code cell.
d) Examine the output.

```
[ 0  5 10 15 21 26 31 36 42 47 52 57 63 68 73 78 84 89
94 100]
```

```
This array has 1 dimension(s).
```

- The `linspace()` function created an array where the values range from 0 to 100, given 20 total values.
 - The values are evenly spaced as much as possible while maintaining an integer data type.
- e) Select the next code cell, and then type the following:

```
1 arange_arr = np.arange(0, 100, 20, dtype = 'int64')
2 print(arange_arr)
3 print('\nThis array has {} dimension(s.)'.format(arange_arr.ndim))
```

Notice how the arguments are the exact same as when you called `linspace()`, but this time you're calling `arange()`.

- f) Run the code cell.

- g) Examine the output.

```
[ 0 20 40 60 80]
This array has 1 dimension(s).
```

- The `arange()` function also created an array with values between 0 and 100.
- Instead of 20 being the total number of values, it is now the step between each value. This results in fewer values to put in the array.
- The `stop` value in this function is exclusive, so 100 does not appear.

6. Generate a NumPy array with random data.

- Scroll down and view the cell titled **Generate a NumPy array with random data**, then select the code cell below it.
- In the code cell, type the following:

```
1 np.random.seed(1975)
2 print('Seed initialized.')
```

- Run the code cell.
- Examine the output.

```
Seed initialized.
```

This makes the rest of the execution environment deterministic; in other words, given the same set of parameters, every call to the same pseudorandom function will produce the same output.

- Select the next code cell, and then type the following:

```
1 rand_arr = np.random.randint(0, 100, (4, 3))
2 print(rand_arr)
3 print('\nThis array has {} dimension(s.)'.format(rand_arr.ndim))
```

This creates an array of random values in the following manner:

- The values will be between 0 and 100 (exclusive).
 - The array will be 4×3 in shape.
- Run the code cell.
 - Examine the output.

```
[[81 20  0]
 [ 1 35 69]
 [23 27 75]
 [62 72 79]]
This array has 2 dimension(s).
```

- Select the next code cell, and then type the following:

```
1 np.random.shuffle(rand_arr)
2 print(rand_arr)
```

- Run the code cell.

-
- j) Examine the output.

```
[[ 1 35 69]
 [62 72 79]
 [81 20  0]
 [23 27 75]]
```

The values in the first axis (the rows) were randomly reordered. For example, the first value in the first row used to be 81, but it is now the first value in the third row.

7. When might you use functions like `linspace()`, `arange()`, and `random.randint()`?

8. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel->Shutdown**.
 - b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
 - c) Close the **Creating Arrays** tab in Firefox, but keep a tab open to **DSTIP/NumPy/** in the file hierarchy.
-

TOPIC B

Load and Save NumPy Data

While you may create NumPy arrays as part of your data science project, you'll likely have some other source of data that you'll want to easily load into NumPy. Likewise, you may want to save any changes to that data while you are working in NumPy. In this topic, you'll write code to both load data and save NumPy arrays as files.

Loading Data into a NumPy Array

It's likely that, at least at the beginning of a data science project, you'll have an existing dataset to work with. Getting this data into a Python environment and constructing an `ndarray` object from that data is an important step in the process. You could load data manually—for example, copying the raw text and formatting it to fit within a `numpy.array()` call—but this will obviously be very tedious and time consuming, even for relatively small datasets. You could try a more programmatic approach by using Python's stock `read()` and `readline()` functions, and then parsing out the input so that it can fit in a NumPy array, but this is not optimal either. Instead, NumPy provides several functions that not only load external data sources, but automatically format those sources as `ndarray` objects.

There are two main types of files in modern computing technology: text files and binary files. A text file, as you would imagine, contains raw text. A binary is anything that's not a text file—executables, images, videos, archives, etc. NumPy can load both types as arrays, though it is limited when it comes to binary files. There are several functions that you can use to load content into NumPy, but the two most common are `numpy.loadtxt()` and `numpy.load()`.

The `numpy.loadtxt()` Function

The `numpy.loadtxt()` function enables you to load data stored in a text file into a NumPy array object. The format of the text file can vary since you do have some control over how NumPy interprets the file, but by default, NumPy expects the data to be in tabular format, with each cell value delimited by a space. So, for example, say you have a text file `my_data.txt` with the following data:

```
6 10 9 3 6 1
10 4 3 5 1 2
```

When you load the file with `loadtxt()`, the resulting `ndarray` will be:

```
array([[6., 10., 9., 3., 6., 1.],
       [10., 4., 3., 5., 1., 2.]])
```

When you call `loadtxt()`, you need to specify the file with the `fname` argument. This is a string that corresponds to either the relative or absolute path where the file is stored. If `my_data.txt` is in the same directory as the Python script or Notebook document that calls the function, then you can simply write the following:

```
numpy.loadtxt('my_data.txt')
```

However, you may need to provide an absolute path to where the file is stored on your local device (or network share), such as:

```
numpy.loadtxt('/home/student/my_data.txt')
```

In either case, once the file is loaded, you can work with the data as you would any NumPy array.



Note: This function will not work if your file has missing values. Each row must have the same number of values.

Additional Parameters

Aside from `fname`, some additional parameters of note include:

- `dtype` —Specify the data type to use, rather than the default (float).
- `comments` —Specify what character(s) are considered the start of comments; the information on these comment lines will not be loaded.
- `delimiter` —Specify what character(s) are used to separate values, rather than the default (space).
- `skiprows` —Specify the first n rows to skip when loading the file.
- `usecols` —Specify which column(s) to read from the file (by number).

The `numpy.genfromtxt()` Function

If you know or suspect your text file has missing values, you can use `numpy.genfromtxt()` to specify how those missing values will be handled.

The `numpy.load()` Function

While you can use `loadtxt()` to load text files, the `numpy.load()` function is used to load binary files. Specifically, the function supports binary files that are in the **pickle** format. The concept of *pickling* is Python's version of serializing data—transforming a data object into a series of bits that can be stored as a file and later reconstructed in the programming environment as an exact replica. The resulting serialized object is called a pickle, and typically has the extension .npy or .npz. Pickles are great for storing and sharing complex data objects (like NumPy arrays) that are created during the execution of code. You can easily reuse these pickles with other data science projects.

To load the pickle file `my_data.npy` using a relative path:

```
numpy.load('my_data.npy')
```

The result will be an `ndarray` that matches the `ndarray` saved in the pickle.



Caution: Although useful, pickling can present a security risk, as an untrusted pickle file may contain malicious code that executes in the environment once it is loaded.

Saving NumPy Array Data

Just as you can load pre-existing data into a NumPy array, you can also save data from a NumPy array. For example, part of the data science lifecycle is to manipulate and modify data in ways that suit your particular needs. You start with a dataset, load it into a NumPy array, and then make some changes to the array. If you ever need to reuse this new array in a new Python project, or if you need the array to serve as input to an external tool, you'll need some way to save that data.

As with loaded data, saved data typically takes one of two forms: text or binary. Text files are easier for people to read, and many tools have text parsing capabilities. However, pickle files are essentially clones of the array object itself, so Python won't need to parse the text, and you won't need to worry about how the text is formatted. Ultimately, the choice comes down to what is most applicable to your own situation.

The two most common functions for saving data are `numpy.savetxt()` and `numpy.save()`.

The `numpy.savetxt()` Function

The `numpy.savetxt()` function enables you to save a NumPy array as a simple text file. The two required arguments are `fname`, which specifies where to save the file (and what name to give it); and `x`, which is the array object to save. For example:

```
my_array = numpy.array([[1, 3, 5],
                      [5, 3, 8]])
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

```
numpy.savetxt('my_data.txt', my_array)
```

As with loading, you'll need to either use a relative file path or an absolute one, depending on what you're trying to do. By default, the file is saved in a tabular format with a space as the delimiter. Note that, by default, the values will be saved as floats. This is what the text file will look like using the previous example:

```
1.000000000000000e+00 3.000000000000000e+00 5.000000000000000e+00
5.000000000000000e+00 3.000000000000000e+00 8.000000000000000e+00
```

To make the output a little cleaner, you can specify an output format using the `fmt` parameter. For example:

```
numpy.savetxt('my_data.txt', my_array, '%.4f')
```

Will result in:

```
1.0000 3.0000 5.0000
5.0000 3.0000 8.0000
```

Additional Parameters

Some additional parameters of note include:

- `delimiter` —Specify what character(s) are used to separate values, rather than the default (space).
- `newline` —Specify what character(s) are used to separate rows, rather than the default (newline).
- `header` —Add a header to the file.
- `footer` —Add a footer to the file.
- `comments` —Specify the character(s) that will signify the start of the header and footer.

The `numpy.save()` Function

Just as you can load a pickle, you can save one, too. The `numpy.save()` function saves a NumPy array in the binary pickle format. There are two required arguments: `file`, to specify the file path and name you want to save the array to; and `arr`, which is the name of the array object you want to save. For example:

```
my_array = numpy.array([[1, 3, 5],
                      [5, 3, 8]])
```

```
numpy.save('my_data.npy', my_array)
```

The `my_data.npy` file can now be loaded at any time by using the `numpy.load()` function.



Note: You can set `allow_pickle` to `False` to save the file in a binary format other than a pickle.

Guidelines for Loading and Saving NumPy Data

Follow these guidelines when you are loading and saving NumPy data.

Load and Save NumPy Data

When loading and saving NumPy data:

- Weigh the advantages and disadvantages of working with text vs. binary files, if you're given the choice.
 - Text files are easy for a human to read and can be used as input to other applications.
 - Binary files represent the array exactly as an object in memory.
- Use `loadtxt()` when loading from a text file.

- Use `load()` when loading from a binary pickle file.
- Use `savetxt()` to save NumPy array data in a text file.
- Use `save()` to save NumPy array data in a binary format.

Do Not Duplicate or Distribute

ACTIVITY 2–2

Loading and Saving NumPy Data

Data Files

/home/student/DSTIP/NumPy/Loading and Saving Arrays.ipynb
 /home/student/DSTIP/NumPy/data/rate_temp.txt
 /home/student/DSTIP/NumPy/data/price_temp.txt

Before You Begin

Jupyter Notebook is open.

Scenario

You've been supplied with some of GCE's sales data. The data is in a loose form, as the company is still working on improving its data collection and storage capabilities. The text files you're given contain information about each individual sale invoice, as well as the satisfaction rating that the customer gave for each transaction. Before you do anything else, you need to get this data into a NumPy array format to make it easier to work with. You anticipate needing to share the data with other members of your team, so you'll also save each array as a file.

1. Examine the data files.

- From the top panel on the desktop, select **Applications→File Manager**.
- Navigate to **/home/student/DSTIP/NumPy/data**.
- Double-click **rate_temp.txt** to open it in a text editor.
- Observe how the data is structured in this file.
 - The customer satisfaction ratings are on a scale from 1 to 10, with the possibility of a decimal point for more precise scores.
 - Each data point is on its own line.
 - There is a header at the top of the file. You'll need to handle this when loading the file into NumPy.
- Close the file when you're done.
- In the same directory, open **price_temp.txt**.
- Observe how the data is structured in this file.

The data doesn't appear to be labeled, but your colleague informed you that each row represents one part of the transaction, and each column is a single transaction:

- Row 1 contains the unit price for each transaction.
- Row 2 contains the tax price for each transaction (based on a 5% rate).
- Row 3 contains the total price for each transaction. This is the unit price multiplied by the item count, then added to the tax price. The item count isn't yet available.

Note that each column is separated by a space.

- Close the file when you're done.

- Switch back to Jupyter Notebook, and then open the **DSTIP/NumPy/Loading and Saving Arrays.ipynb** file.
- Import the relevant software libraries.
 - View the cell titled **Import software libraries**, and examine the code listing below it.
 - Run the code cell.

- c) Verify that the versions of Python and NumPy are returned.

4. Print information about a NumPy array.

- a) Scroll down and view the cell titled **Print information about a NumPy array**, and examine the code listing below it.

```

1 def array_info(array):
2     print('The array has {} dimension(s)'.format(array.ndim))
3     print('The shape of the array is: {}'.format(array.shape))
4     print('There are {} items in the array.'.format(array.size))
5     print('The array holds data of type {}'.format(array.dtype))
6
7 print('The function to print array info has been defined.')

```

This function, when called, will output information about the array passed in as an argument.

- b) Run the code cell.

5. Load a text file as a NumPy array.

- a) Scroll down and view the cell titled **Load a text file as a NumPy array**, then select the code cell below it.
- b) In the code cell, type the following:

```

1 ratings = np.loadtxt('data/rate_temp.txt', skiprows = 1)
2 ratings

```

- Line 1 loads the first text file you looked at into a NumPy array. The first argument uses a relative path to point to the data file.
 - To avoid problems with the text header, you're telling NumPy to skip the first row when it loads the data into the array.
 - Line 2 returns the array.
- c) Run the code cell.
- d) Examine the output.

```

array([ 9.1,  9.6,  7.4,  8.4,  5.3,  4.1,  5.8,  8. ,  7.2,  5.9,  4.5,
       6.8,  7.1,  8.2,  5.7,  4.5,  4.6,  6.9,  8.6,  4.4,  4.8,  5.1,
       4.4,  9.9,  6. ,  8.5,  6.7,  7.7,  9.6,  7.4,  4.8,  4.5,  5.1,
       5.1,  7.5,  6.8,  7. ,  4.7,  7.6,  7.7,  7.9,  6.3,  5.6,  7.6,
       7.2,  9.5,  8.4,  4.1,  8.1,  7.9,  9.5,  8.5,  6.5,  6.1,  6.5,
       8.2,  5.8,  6.6,  5.4,  9.3,  10. ,  7. ,  10. ,  8.6,  7.6,  5.8,
       6.7,  9.9,  6.4,  4.3,  9.6,  5.9,  4. ,  8.7,  9.4,  5.4,  8.6,
       5.7,  6.6,  6. ,  5.5,  6.4,  6.6,  8.3,  6.6,  4. ,  9.9,  7.3,
       5.7,  6.1,  7.1,  8.2,  5.1,  8.6,  6.6,  7.2,  5.1,  4.1,  9.3,
       7.4])

```

All of the data has been loaded into an `ndarray` object, as intended.

- e) Select the next code cell, and then type the following:

```
1 array_info(ratings)
```

- f) Run the code cell.

- g) Examine the output.

```
The array has 1 dimension(s).
The shape of the array is: (100,)
There are 100 items in the array.
The array holds data of type float64.
```

- The `loadtxt()` function created a one-dimensional array out of the data file. This is because there was only one "column" of data.
- There are 100 total data points in the array.
- The array contains floats.

6. Load another text file.

- Scroll down and view the cell titled **Load another text file**, then select the code cell below it.
- In the code cell, type the following:

```
1 prices = np.loadtxt('data/price_temp.txt')
2 prices
```

This loads the prices data (unit, tax, and total). The loading operation is similar to before, but this time there are no text headers, so you don't need to skip any rows.

- Run the code cell.
- Examine the output.

```
array([[7.469000e+01, 1.528000e+01, 4.633000e+01, 5.822000e+01,
       8.631000e+01, 8.539000e+01, 6.884000e+01, 7.356000e+01,
       3.626000e+01, 5.484000e+01, 1.448000e+01, 2.551000e+01,
       4.695000e+01, 4.319000e+01, 7.138000e+01, 9.372000e+01,
       6.893000e+01, 7.261000e+01, 5.467000e+01, 4.030000e+01,
       8.604000e+01, 8.798000e+01, 3.320000e+01, 3.456000e+01,
       8.863000e+01, 5.259000e+01, 3.352000e+01, 8.767000e+01,
       8.836000e+01, 2.489000e+01, 9.413000e+01, 7.807000e+01,
       8.378000e+01, 9.658000e+01, 9.942000e+01, 6.812000e+01,
       6.262000e+01, 6.088000e+01, 5.492000e+01, 3.012000e+01,
       8.672000e+01, 5.611000e+01, 6.912000e+01, 9.870000e+01,
       1.537000e+01, 0.206000e+01, 5.660000e+01, 2.001000e+01]]
```

- There's much more information here than with the ratings data.
- The prices are cast as floats with many digits, which may be more than is necessary.



Note: Unlike with standard Python, NumPy does not have a `Decimal` data type, which is more commonly used than floats to store pricing data, due to its higher level of precision. For demonstration purposes, the GCE prices will be left as floats, but in a real-world situation, you should consider a workaround. One potential workaround is multiplying all prices by 100, then casting them as integers. In output, you can simply divide by 100 to show the true price out to two decimal places.

- Select the next code cell, and then type the following:

```
1 array_info(prices)
```

- Run the code cell.

- g) Examine the output.

```
The array has 2 dimension(s).
The shape of the array is: (3, 100)
There are 300 items in the array.
The array holds data of type float64.
```

- The created array has two dimensions (i.e., rows and columns).
- There are 3 rows and 100 columns, for a total of 300 data points.

7. Save the arrays as files.

- a) Scroll down and view the cell titled **Save the arrays as files**, then select the code cell below it.
 b) In the code cell, type the following:

```
1 np.save('customer_ratings.npy', ratings)
2 print('Ratings array has been saved.')
```

This saves the `ratings` array as a Python pickle file in the current directory.

- c) Run the code cell.
 d) Examine the output.

```
Ratings array has been saved.
```

- e) Select the next code cell, then type the following:

```
1 np.savetxt('unit_tax_total.csv', prices, '%.2f', delimiter = ',')
2 print('Prices array has been saved.')
```

- This saves the `prices` array as a comma-separated value (CSV) file in the current directory.
 - The format string `'%.2f'` ensures that only two decimal points will be recorded in the file, rather than the full precision that the array currently has.
 - The `delimiter` argument ensures that each value is separated by a comma.
- f) Run the code cell.
 g) Examine the output.

```
Prices array has been saved.
```

8. Examine the files you saved.

- From the top panel on the desktop, select the **data - File Manager** tab to open it.
- Navigate back one directory so you're in `/home/student/DSTIP/NumPy/`.
- Right-click `unit_tax_total.csv`, then select **Open with "Mousepad"**.
- Examine the file, and verify that the data points are separated by commas.
- Close the file when you're done.
- Double-click `customer_ratings.npy`.
- Verify that the file did not open, and then select **Cancel**.

9. Why were you unable to open this file in a text editor?

10.What are some advantages and disadvantages of saving this type of file as compared to a CSV file?

11.Shut down this Jupyter Notebook kernel.

- a) Return to Jupyter Notebook.
 - b) From the menu, select **Kernel→Shutdown**.
 - c) In the **Shutdown kernel?** dialog box, select **Shutdown**.
 - d) Close the **Loading and Saving Arrays** tab in Firefox, but keep a tab open to **DSTIP/NumPy/** in the file hierarchy.
-

TOPIC C

Analyze Data in NumPy Arrays

Now that you've constructed array objects in various ways, it's time to start analyzing the data in those arrays. NumPy offers many ways for you to extract useful insights from your data.

NumPy Array Indexing

You can index a NumPy array similar to how you would index a Python list. For one-dimensional arrays, you simply provide the index number for the item you want to access within brackets. For example:

```
x1d = numpy.array([8, 3, 9, 16])
x1d[0]
```

The result is 8, because that is the value in the first index. Remember that indices start with 0. The index `x1d[1]` is 3, `x1d[2]` is 9, and so on. You can also use a negative index, like `x1d[-1]` (16).

Indexing with multi-dimensional arrays is a little different, but it's not that complicated. You provide a tuple within the brackets, where each dimension is separated by a comma. The following example indexes a two-dimensional array:

```
x2d = numpy.array([[8, 3, 9, 16],
                  [20, 4, 1, 6]])
x2d[1, 2]
```

The 1 index refers to the second row, and the 2 index refers to the third column. Therefore, this returns 1.

As with any data structure, indexing a NumPy array is helpful when you need to access a single item. For example, in your analysis of the data, you might discover an anomalous value that you want to change. You can use indexing to change that value, much like a Python list:

```
x2d[1, 2] = 5
```

Now the array is:

```
array([[8, 3, 9, 16],
       [20, 4, 5, 6]])
```

Fancy Indexing

Normal indexing involves providing a *scalar* for the index—in other words, a single value. **Fancy indexing** involves providing a *list* or *array* of indices. If you need to access multiple items in an array, and those items are staggered throughout the array (i.e., not side by side), then fancy indexing is a good solution. For example:

```
x1d = numpy.array([10, 5, 32, 16, 1, 2, 9, 4])
```

```
x1d[[0, 4, 7]]
```

This selects index 0, 4, and 7 and returns each item in those indices as an array:

```
array([10, 1, 4])
```

When you use fancy indexing for multi-dimensional arrays, you can provide NumPy arrays as the actual indices. For example, you have the following array:

```
x2d = numpy.array([[8, 3, 9, 16],
                  [20, 4, 1, 6]])
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

Let's say you want to access value 16 (index [0, 3]) and value 1 (index [1, 2]). You can provide each row reference as an array, as well as each column reference as a separate array:

```
row = numpy.array([0, 1])
col = numpy.array([3, 2])

x2d[row, col]
```

This will produce `array([16, 1])`.

When you use fancy indexing, it's important to note that the shape of the resulting array will be the shape of the index arrays, not the shape of the array you're indexing. In the previous example, the array after indexing is in one dimension, but the array initially had two dimensions.

NumPy Array Slicing

Fancy indexing is useful for selecting non-contiguous cells, but it can be somewhat tedious if you're trying to return an entire row, or one section of a column, or some other range of values. This is where slicing comes into play. Slicing retrieves a contiguous part of an array, also called a *subarray*. Slicing a NumPy array, like indexing, is performed very similar to how it's done in a Python list. You separate each element of the slice with a colon (:). The basic syntax is as follows:

```
my_arr[start:stop:step]
```

Here's a simple slicing example on a one-dimensional array:

```
x = numpy.array([9, 10, 3, 78, 5, 4, 18, 100, 34, 51])

x[0:3:1]
```

The output of this operation is the following array:

```
array([9, 10, 3])
```

So, this operation returned the values in indices 0 through 2, without skipping any. Note that the stop criterion is exclusive, so it stops before index 3.

You don't need to provide each slice component if you want to use the defaults. The default values are:

- A start of 0 (i.e., the first cell of that dimension)
- A stop equal to that dimension's total length (i.e., the last cell of that dimension).
- A step of 1.

So, the prior slicing example can be simplified to:

```
x[:3]
```

As another example, let's say you wanted to retrieve every other cell in the array starting with index 2:

```
x[2::2]
```

This results in:

```
array([3, 5, 18, 34])
```

You didn't specify a value for the stop criterion, so it uses the default (the last cell).

Slicing Multi-Dimensional Arrays

As you might expect, slicing multi-dimensional arrays follows the same basic pattern as indexing: you separate each dimension by a comma. For example, let's say you want to slice out the middle part of the following array:

```
x = np.array([[16, 8, 5, 24, 57],
              [61, 1, 19, 11, 84],
              [2, 7, 14, 15, 23]])
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

```
x[:, 1:4]
```

The first half of this expression slices by rows. Since no values are specified, it will slice by all rows. The second half of this expression slices by columns. Columns 1 through 3 will be sliced. So, this results in:

```
array([[8, 5, 24],
       [1, 19, 11],
       [7, 14, 15]])
```

You can even combine slicing with indexing to access entire rows or columns:

```
>>> x[1, :]
array([61, 1, 19, 11, 84])
>>> x[:, 3]
array([24, 11, 15])
```

Or, combine slicing with fancy indexing:

```
>>> x[1:, [1, 4]]
array([[1, 84],
       [7, 23]])
```

NumPy Array Iteration

Now that you can better "see" your data through slicing and indexing, you'll likely want to be able to summarize that data in useful ways. As with any other Python object, you can do this by iterating over an array using a `for` loop. For example, let's say you want to print the sum of each item in an array:

```
x = numpy.arange(1, 1e7)

arr_sum = 0
for i in x:
    arr_sum += i
print(arr_sum)
```

This is rather intuitive to someone who's used to object-oriented programming. However, iterating over NumPy arrays, especially when using a typical Python `for` loop, is very inefficient. The loop operates on the array cell by cell; on every cell, the Python interpreter must bind `arr_sum` with the next object that is iterated over, then it must look up `my_arr` in memory again. This repeated process consumes a lot of CPU time. Therefore, iteration is very much discouraged when using NumPy arrays. Thankfully, there is an alternative approach to operating on an array that is much faster.

The `numpy.nditer` Object

The `numpy.nditer` object was introduced to provide a little more flexibility in iterating over a NumPy array. The equivalent syntax to the previous example is as follows:

```
for i in numpy.nditer(x):
    arr_sum += 1
print(arr_sum)
```

However, this is not necessarily any faster than a normal Python `for` loop, so you should still try to avoid iteration whenever possible.

Vectorization

The alternative to NumPy array iteration is vectorization. **Vectorization** is the process of leveraging pre-existing NumPy functions to operate over an entire array, rather than operate one-by-one on each item as they would in a `for` loop. While iteration is susceptible to overhead due to Python's

dynamic nature, vectorization functions call highly optimized C code to loop over the array data where it appears in memory. This makes vectorized operations much faster than Python loops. Not only that, but vectorization often simplifies the code's syntax. The following example uses a vectorization function to sum up the items in an array:

```
x = np.arange(1, 1e7)
numpy.sum(x)
```

This function operates on the entire array to calculate the sum of its items. That's all it takes. Compared to the `for` loop example mentioned earlier, the vectorization function returns the same result several *orders of magnitude* faster. This is the power of the NumPy array, and it's why you should always prefer to vectorize, rather than iterate, whenever possible.



Note: Iteration might be necessary in cases where NumPy does not provide a vectorization function for a particular operation. However, most fundamental types of operations are supported.

NumPy Statistical Summary Functions

NumPy provides many statistical summary functions, and each one can be used to vectorize your array operations.

In the following table, some of the most prominent statistical summary functions are described. Each function has, at minimum, the first two arguments: an `a` argument that takes an array as input, and the `axis` argument that enables you to specify one or more dimensions (axes) to apply the operation to. If no `axis` value(s) are specified, NumPy reduces the array to a single dimension—a process called *flattening*—and performs the operation on the entire array.

The example outputs work with the following array:

```
x = numpy.array([[39, 67, 43],
                 [45, 46, 51]])
```

Function	Description and Example
<code>numpy.amin()</code>	Returns the lowest value in the array. <pre>>>> numpy.amin(x) 39</pre>
<code>numpy.amax()</code>	Returns the highest value in the array. <pre>>>> numpy.amax(x) 67</pre>
<code>numpy.mean()</code>	Returns the arithmetic mean of values in the array. <pre>>>> numpy.mean(x) 48.5</pre>
<code>numpy.average(weights)</code>	Returns the weighted average of values in the array. The <code>weights</code> argument takes an array of weights that correspond to each value in the array <code>a</code> . <pre>>>> w = numpy.array([[1.6, 1, 1], ... [1, 1.3, 1.2]]) >>> numpy.average(x, weights = w) 47.66197</pre>
<code>numpy.median()</code>	Returns the median of values in the array. <pre>>>> numpy.median(x) 45.5</pre>

Function	Description and Example
<code>numpy.std()</code>	Returns the standard deviation of values in the array. <pre>>>> numpy.std(x) 9.01387</pre>
<code>numpy.var()</code>	Returns the variance of values in the array. <pre>>>> numpy.var(x) 81.25</pre>
<code>numpy.sum()</code>	Returns the sum of all values in the array. <pre>>>> numpy.sum(x) 291</pre>



Note: Summary statistics are also referred to as descriptive statistics.

Class Object Syntax

Some of these statistical functions can also be used as methods of a NumPy array object. For example, this:

```
numpy.mean(x)
```

Is equivalent to this:

```
x.mean()
```

You can use either type of syntax according to your preference.

The `numpy.unique()` Function

The `numpy.unique()` function enables you to find all of the unique values in a NumPy array. This can be useful for several reasons. For example, you might be interested in seeing only unique part numbers in a purchase history, to distinguish individual products. The function also removes duplicate values from the array, which might be part of your data cleaning process.

The required parameter is `arr`, which is the array it will perform the operation on. A basic example is as follows:

```
x = numpy.array([[1, 1, 3, 3],
                 [1, 1, 3, 3],
                 [3, 3, 1, 0]])
```

```
numpy.unique(x)
```

This returns:

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

As you can see, the two-dimensional array is flattened to a one-dimensional array on output. Like with statistical summary functions, you can supply an `axis` argument to specify that the operation should only be performed on the desired dimensions. For example:

```
>>> numpy.unique(x, axis = 0)
array([[1, 1, 3, 3],
       [3, 3, 1, 0]])
```

In this case, the function returned the unique *rows* (axis 0), not unique values. Also, the resulting array maintains its two dimensions, rather than being flattened to one.

Additional Parameters

Another argument you might find useful is `return_index`, which accepts a Boolean value. This returns the index in which each unique value is found:

```
>>> u, i = numpy.unique(x, return_index = True)
>>> i
array([11,  0,  2])
```

The 0 value was first found at index 11; 1 was first found at index 0; and 3 was first found at index 2. Recall that the default behavior is to flatten the array, which explains the indices going up to 11.

You can also activate `return_counts` to retrieve the number of time each value appears in the array:

```
>>> u, c = numpy.unique(x, return_counts = True)
>>> c
array([1, 5, 6])
```

The 0 value occurs once; the 1 value occurs five times; and the 3 value occurs six times.

SciPy Integration

SciPy is sometimes used to refer to the Python data science stack itself, but SciPy is also an API that provides specialized functionality that isn't limited to just data science. The API actually builds on top of NumPy, and its functions are therefore designed to work directly with NumPy arrays. Primarily, SciPy provides scientific and advanced mathematical algorithms that scientists can use to analyze, manipulate, and visualize data—algorithms that go beyond what is included with standard NumPy. SciPy is, therefore, most commonly used in scientific research, but you may still find some useful functionality for your business needs.

SciPy functions can easily interface with NumPy arrays. In the following example, a SciPy function called `linalg.inv()` is able to take the inverse of a square matrix (i.e., it performs a type of linear algebra):

```
from scipy import linalg

x = numpy.array([[0, 5],
                [2, 1]])
```

`linalg.inv(x)`

The result is a NumPy array:

```
array([[-0.1,  0.5],
       [ 0.2,  0.]])
```



Note: NumPy also has an equivalent module, `numpy.linalg`, but SciPy's version has additional functionality.

SciPy Subpackages

The SciPy API is organized by scientific computing domains, where each domain has an associated subpackage. Each subpackage includes various functions. For example, `scipy.linalg` is a subpackage, and `scipy.linalg.inv()` is one of several functions in that subpackage.

A deeper dive into the subpackages and their functions is beyond the scope of this course. However, the following table briefly summarizes each subpackage.

Subpackage	Description and Examples
<code>scipy.cluster</code>	<p>Includes functions related to clustering algorithms.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>cluster.vq.kmeans()</code> • <code>cluster.hierarchy.ward()</code>
<code>scipy.constants</code>	<p>Includes functions related to physical and mathematical constants.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>constants.value()</code> • <code>constants.unit()</code>
<code>scipy.fft</code>	<p>Includes functions related to Fourier transformations.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>fft.fft()</code> • <code>fft.dct()</code>
<code>scipy.integrate</code>	<p>Includes functions related to integrals and ordinary differential equations.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>integrate.quad()</code> • <code>integrate.solve_ivp()</code>
<code>scipy.interpolate</code>	<p>Includes functions related to interpolation and splines.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>interpolate.interp1d()</code> • <code>interpolate.BSpline()</code>
<code>scipy.io</code>	<p>Includes functions related to input and output.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>io.loadmat()</code> • <code>io.savemat()</code>
<code>scipy.linalg</code>	<p>Includes functions related to linear algebra.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>linalg.solve()</code> • <code>linalg.lu()</code>
<code>scipy.misc</code>	<p>Includes functions related to miscellaneous algorithms.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>misc.derivative()</code> • <code>misc.face()</code>
<code>scipy.ndimage</code>	<p>Includes functions related to multi-dimensional image processing.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>ndimage.convolve()</code> • <code>ndimage.affine_transform()</code>

Subpackage	Description and Examples
<code>scipy.odr</code>	Includes functions related to orthogonal distance regression. Example functions: <ul style="list-style-type: none">• <code>odr.Data()</code>• <code>odr.ODR()</code>
<code>scipy.optimize</code>	Includes functions related to optimizing object functions. Example functions: <ul style="list-style-type: none">• <code>optimize.minimize()</code>• <code>optimize.root()</code>
<code>scipy.signal</code>	Includes functions related to signal processing. Example functions: <ul style="list-style-type: none">• <code>signal.order_filter()</code>• <code>signal.bilinear()</code>
<code>scipy.sparse</code>	Includes functions related to sparse matrices. Example functions: <ul style="list-style-type: none">• <code>sparse.bsr_matrix()</code>• <code>sparse.identity()</code>
<code>scipy.spatial</code>	Includes functions related to spatial algorithms and data structures. Example functions: <ul style="list-style-type: none">• <code>spatial.KDTree()</code>• <code>spatial.Rectangle()</code>
<code>scipy.special</code>	Includes functions related to specialized algorithms. Example functions: <ul style="list-style-type: none">• <code>special.ellipj()</code>• <code>special.gamma()</code>
<code>scipy.stats</code>	Includes functions related to statistics. Example functions: <ul style="list-style-type: none">• <code>stats.cosine()</code>• <code>stats.bernoulli()</code>

statsmodels Integration

Like SciPy, statsmodels builds on NumPy to provide additional functionality that the base NumPy package does not. In this case, statsmodels focuses on creating statistical models from data and exploring more advanced statistical attributes of that data. Applying statsmodels functionality to NumPy arrays is not much different than when using SciPy. The following example calculates measures of skewness in a random sample of data:

```
import statsmodels.stats as sm

numpy.random.seed(1975)
x = numpy.random.randint(0, 1e4, (3, 3))
```

```
sm.statsmodels.robust_skewness(x) [0]
```

The output (which, in this case, just grabs the first measure of skewness) is as follows:

```
array([0.18301123, -0.20729364, 0.65438286])
```

Although much of statsmodels' functionality doesn't always have a business application, you may still find that certain statistical routines enhance your understanding of the data. So, consider investigating statsmodels if NumPy's built-in statistics functions don't fulfill your needs.



Note: In addition to working with NumPy arrays, statsmodels also works with pandas data structures.

statsmodels Statistics Submodules

In addition to its statical modeling functions, statsmodels also provides the `statsmodels.stats` module and various submodules for testing and exploration. The following table lists the submodules in this module, along with some example functions for each submodule.

Submodule	Description and Examples
<code>stats.statsmodels</code>	<p>Includes functions related to residual diagnostics and specification tests.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>statsmodels.durbin_watson()</code> • <code>statsmodels.robust_kurtosis()</code>
<code>stats.sandwich_covariance</code>	<p>Includes functions related to sandwich (robust covariance matrix) calculations.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>sandwich_covariance.cov_hac()</code> • <code>sandwich_covariance.cov_cluster()</code>
<code>stats.gof</code>	<p>Includes functions related to goodness of fit tests and measures.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>gof.powerdiscrepancy()</code> • <code>gof.chisquare_effectsize()</code>
<code>stats.runs</code>	<p>Includes functions related to non-parametric tests.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>runs.mcnemar()</code> • <code>runs.cochrans_q()</code>
<code>stats.inter_rater</code>	<p>Includes functions related to inter-rater reliability and agreement.</p> <p>Example functions:</p> <ul style="list-style-type: none"> • <code>inter_rater.cohens_kappa()</code> • <code>inter_rater.to_table()</code>

Submodule	Description and Examples
<code>stats.multitest</code>	<p>Includes functions related to multiple tests and multiple comparison procedures.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>multitest.multipletests()</code> <code>multitest.NullDistribution()</code>
<code>stats.weightstats</code>	<p>Includes functions related to basic statistics and t-tests with frequency weights.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>weightstats.ttest_ind()</code> <code>weightstats.ztest()</code>
<code>stats.power</code>	<p>Includes functions related to power and sample size calculations.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>power.TTestIndPower()</code> <code>power.GofSquaredPower()</code>
<code>stats.proportion</code>	<p>Includes functions related to proportion calculations.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>proportion.proportion_confint()</code> <code>proportion.binom_test()</code>
<code>stats.moment_helpers</code>	<p>Includes functions related to moment helpers.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>moment_helpers.cov2corr()</code> <code>moment_helpers.se_cov()</code>
<code>stats.mediation</code>	<p>Includes functions related to mediation analysis.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>mediation.Mediation()</code> <code>mediation.MediationResults()</code>
<code>stats.oaxaca</code>	<p>Includes functions related to Blinder–Oaxaca decomposition.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>oaxaca.OaxacaBlinder()</code> <code>oaxaca.OaxacaResults()</code>
<code>stats.dist_dependence_measures</code>	<p>Includes functions related to distance dependence measures.</p> <p>Example functions:</p> <ul style="list-style-type: none"> <code>dist_dependence_measures.distance_statistics()</code> <code>dist_dependence_measures.distance_covariance()</code>

Guidelines for Analyzing Data in NumPy Arrays

Follow these guidelines when you are analyzing data in NumPy arrays.

Analyze Data in NumPy Arrays

When analyzing data in NumPy arrays:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

- Use vectorization instead of traditional Python looping when you perform operations on arrays.
- Index an array to retrieve individual items and/or change items.
- Use fancy indexing to retrieve multiple staggered items in an array.
 - Keep in mind that the shape of the result will be the shape of the index.
- Slice an array to retrieve contiguous items from the array.
- Apply summary functions to retrieve statistical information about your arrays.
- Consider how libraries like SciPy and statsmodels can provide additional mathematical operations to use on your NumPy arrays.

ACTIVITY 2–3

Analyzing Data in NumPy Arrays

Data Files

/home/student/DSTIP/NumPy/Analyzing Arrays.ipynb
 /home/student/DSTIP/NumPy/data/customer_ratings.npy
 /home/student/DSTIP/NumPy/data/unit_tax_total.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've loaded some preliminary data into NumPy arrays, so now you can freely analyze that data. While you could potentially view the data all at once, the larger your arrays are, the more difficult this becomes. So, you'll want to select only the subset of data that you're interested in, be it a single customer rating, a range of total prices, or some other combination. Once you have the subset you want, you can use various statistical functions to describe the data in that subset. You'll get a better feel for the nature of that data, including its measures of central tendency as well as how disperse the values are. Analysis of this kind may reveal interesting patterns in the data, such as sales trends or customer sentiment. It can also help you identify errors in the dataset itself.

1. In Jupyter Notebook, open the **DSTIP/NumPy/Analyzing Arrays.ipynb** file.
2. Import the relevant software libraries, and load the datasets.
 - a) View the cell titled **Import software libraries and load the datasets**, and examine the code listing below it.
 In addition to loading the libraries, this code also loads the `ratings` and `prices` datasets.
 - b) Run the code cell.
 - c) Verify that the versions of Python and NumPy are returned.
3. Confirm the speed advantage of vectorization.
 - a) Scroll down and view the cell titled **Confirm the speed advantage of vectorization**, then select the code cell below it.
 - b) In the code cell, type the following:

```

1 %%time
2 arr_sum = 0
3 for i in np.arange(1, 1e7):
4     arr_sum += i
5 arr_sum

```

- The `%%time` magic command returns the time it took to execute this entire cell.
 - A standard `for` loop iterates through each element of the array and returns the sum.
- c) Run the code cell.

- d) Examine the output.

```
CPU times: user 13.5 s, sys: 5.17 ms, total: 13.5 s
Wall time: 13.8 s

49999995000000.0
```



Note: The time you see in your environment may differ from what's shown in this screenshot.

- e) Select the next code cell, and then type the following:

```
1 %%time
2 arr_sum = np.sum(np.arange(1, 1e7))
3 arr_sum
```

This approach uses a NumPy vectorization function to do the same basic thing as the `for` loop.

- f) Run the code cell.
g) Examine the output.

```
CPU times: user 71.7 ms, sys: 7.85 ms, total: 79.5 ms
Wall time: 217 ms

49999995000000.0
```

The vectorization method finished much faster than the `for` loop, despite arriving at the same answer. This difference is more noticeable the larger the array is.

4. Use indexing to retrieve individual customer ratings.

- a) Scroll down and view the cell titled **Use indexing to retrieve individual customer ratings**, then select the code cell below it.
b) In the code cell, type the following:

```
1 ratings
```

- c) Run the code cell.
d) Examine the output.

```
array([ 9.1,  9.6,  7.4,  8.4,  5.3,  4.1,  5.8,  8. ,  7.2,  5.9,  4.5,
       6.8,  7.1,  8.2,  5.7,  4.5,  4.6,  6.9,  8.6,  4.4,  4.8,  5.1,
       4.4,  9.9,  6. ,  8.5,  6.7,  7.7,  9.6,  7.4,  4.8,  4.5,  5.1,
       5.1,  7.5,  6.8,  7. ,  4.7,  7.6,  7.7,  7.9,  6.3,  5.6,  7.6,
       7.2,  9.5,  8.4,  4.1,  8.1,  7.9,  9.5,  8.5,  6.5,  6.1,  6.5,
       8.2,  5.8,  6.6,  5.4,  9.3,  10. ,  7. ,  10. ,  8.6,  7.6,  5.8,
       6.7,  9.9,  6.4,  4.3,  9.6,  5.9,  4. ,  8.7,  9.4,  5.4,  8.6,
       5.7,  6.6,  6. ,  5.5,  6.4,  6.6,  8.3,  6.6,  4. ,  9.9,  7.3,
       5.7,  6.1,  7.1,  8.2,  5.1,  8.6,  6.6,  7.2,  5.1,  4.1,  9.3,
      7.4])
```

This is to remind you how the `ratings` dataset is structured.

- e) Select the next code cell, and then type the following:

```
1 print('First customer rating: {}'.format(ratings[0]))
2 print('Ninth customer rating: {}'.format(ratings[8]))
```

- f) Run the code cell.
g) Examine the output.

```
First customer rating: 9.1.
Ninth customer rating: 7.2.
```

Since indices start at 0 in a NumPy array—just as they do in a Python list—the first rating is at index 0, while the ninth rating is at index 8. You can look back at the `ratings` array to verify that the correct values were retrieved.

- h) Select the next code cell, and then type the following:

```
1 print('First and ninth customer ratings: {}'.format(ratings[[0, 8]]))
```

Providing multiple indices wrapped in double brackets is an example of fancy indexing.

- i) Run the code cell.
j) Examine the output.

```
First and ninth customer ratings: [9.1 7.2]
```

Fancy indexing has retrieved two values from two indices, placing them both into an array.

5. Use slicing to retrieve multiple customer ratings.

- a) Scroll down and view the cell titled **Use slicing to retrieve multiple customer ratings**, then select the code cell below it.
b) In the code cell, type the following:

```
1 print('The first 10 ratings are: {}'.format(ratings[0:10]))
2
3 # Streamlined notation.
4 print('\nThe first 10 ratings are: {}'.format(ratings[:10]))
5
6 print('\nRatings 20 through 30 are: {}'.format(ratings[19:30]))
7 print('\nObject type: {}'.format(type(ratings[19:30])))
```

- Line 1 uses the full notation to slice: `[start:stop:step]`
 - Line 4 does the same thing as line 1, but uses streamlined notation.
 - Line 6 slices from different starting and stopping points.
 - Line 7 prints the type of object that a slice returns.
- c) Run the code cell.
d) Examine the output.

```
The first 10 ratings are: [9.1 9.6 7.4 8.4 5.3 4.1 5.8 8.  7.2 5.9]
The first 10 ratings are: [9.1 9.6 7.4 8.4 5.3 4.1 5.8 8.  7.2 5.9]
Ratings 20 through 30 are: [4.4 4.8 5.1 4.4 9.9 6.  8.5 6.7 7.7 9.6 7.4]
Object type: <class 'numpy.ndarray'>
```

- The outputs show ranges of numbers extracted from the `ratings` array, according to how it was sliced.
- The last output line confirms that slicing a NumPy array returns a NumPy array.

- e) Select the next code cell, and then type the following:

```
1 print('Every other rating:\n {}'.format(ratings[::2]))
2 print('\nEvery third rating between 10 and 50:\n {}' \
3       .format(ratings[9:50:3]))
```

This time, you're providing different step values than the default (1).

- f) Run the code cell.
g) Examine the output.

```
Every other rating:
[ 9.1  7.4  5.3  5.8  7.2  4.5  7.1  5.7  4.6  8.6  4.8  4.4  6.   6.7
  9.6  4.8  5.1  7.5  7.   7.6  7.9  5.6  7.2  8.4  8.1  9.5  6.5  6.5
  5.8  5.4  10.  10.   7.6  6.7  6.4  9.6  4.   9.4  8.6  6.6  5.5  6.6
  6.6  9.9  5.7  7.1  5.1  6.6  5.1  9.3]

Every third rating between 10 and 50:
[5.9 7.1 4.5 8.6 5.1 6.  7.7 4.8 5.1 7.  7.7 5.6 9.5 8.1]
```

Providing a custom step value is helpful for selecting a portion of the data, where that portion still follows a consistent order.

6. Index a multi-dimensional array.

- a) Scroll down and view the cell titled **Index a multi-dimensional array**, then select the code cell below it.
b) In the code cell, type the following:

```
1 | prices
```

- c) Run the code cell.
d) Examine the output.

```
array([[ 74.69,  15.28,  46.33,  58.22,  86.31,  85.39,  68.84,  73.56,
        36.26,  54.84,  14.48,  25.51,  46.95,  43.19,  71.38,  93.72,
       68.93,  72.61,  54.67,  40.3 ,  86.04,  87.98,  33.2 ,  34.56,
      88.63,  52.59,  33.52,  87.67,  88.36,  24.89,  94.13,  78.07,
      83.78,  96.58,  99.42,  68.12,  62.62,  60.88,  54.92,  30.12,
     86.72,  56.11,  69.12,  98.7 ,  15.37,  93.96,  56.69,  20.01,
     18.93,  82.63,  91.4 ,  44.59,  17.87,  15.43,  16.16,  85.98,
     44.34,  89.6 ,  72.35,  30.61,  24.74,  55.73,  55.07,  15.81,
     75.74,  15.87,  33.47,  97.61,  78.77,  18.33,  89.48,  62.12,
     48.52,  75.91,  74.67,  41.65,  49.04,  20.01,  78.31,  20.38,
     0.  10.  06.  68.  10.  25.  88.  26.  48.  01.  88.  06.  76.  52.  40.  28]]))
```

Recall that this dataset has three rows, where each row is a component of the overall purchase, and each column is an individual transaction. The rows are:

- 0: Unit price.
- 1: Tax price.
- 2: Total price.

- e) Select the next code cell, and then type the following:

```
1 print('First unit price: ${}'.format(prices[0, 0]))
2 print('Third tax price: ${}'.format(prices[1, 2]))
3 print('Ninth total price: ${}'.format(prices[2, 8]))
```

- For a multi-dimensional dataset, you can provide indices for both dimensions (e.g., rows and columns).
 - For example, on line 3, you are selecting the value at row index 2, column index 8. The row at index 2 contains total prices, and the column at index 8 is the ninth overall transaction.
- f) Run the code cell.
g) Examine the output.

```
First unit price: $74.69.
Third tax price: $16.22.
Ninth total price: $76.15.
```

- h) Select the next code cell, and then type the following:

```
1 row = np.array([0, 1, 2])
2 col = np.array([0, 2, 8])
3 print('First unit price, third tax price, and ninth total price:\n {}' \
        .format(prices[row, col]))
```

- This does some fancy indexing on the `prices` array, where there are multiple row–column pairs that you're trying to extract.
 - On lines 1 and 2, each row reference is part of an array, as is each column reference.
- i) Run the code cell.
j) Examine the output.

```
First unit price, third tax price, and ninth total price:
[74.69 16.22 76.15]
```

The fancy indexing retrieved the values at (in order of the returned array):

- Row 0, column 0.
- Row 1, column 2.
- Row 2, column 8.

7. Slice a multi-dimensional array.

- a) Scroll down and view the cell titled **Slice a multi-dimensional array**, then select the code cell below it.
b) In the code cell, type the following:

```
1 print('First 10 sale prices:\n {}'.format(prices[0, :9]))
2 print('\nTax prices from 10 to 20:\n {}'.format(prices[1, 9:19]))
```

- Both lines 1 and 2 combine indexing and slicing on a multi-dimensional array. They both use an index for the row reference, with a slice for the column.
 - The syntax is essentially the same as indexing and slicing a one-dimensional array, except you need to provide the index/slice for both dimensions.
- c) Run the code cell.

- d) Examine the output.

```
First 10 sale prices:  
[74.69 15.28 46.33 58.22 86.31 85.39 68.84 73.56 36.26]  
  
Tax prices from 10 to 20:  
[ 8.23  2.9   5.1   11.74 21.59 35.69 28.12 24.13 21.78  8.2 ]
```

In both cases, a range of prices was retrieved from a single row.

- e) Select the next code cell, and then type the following:

```
1 print('First 10 tax prices and total prices:\n {}' \  
2     .format(prices[1:3, :10]))
```

In this case, both the rows and columns are being sliced.

- f) Run the code cell.
g) Examine the output.

```
First 10 tax prices and total prices:  
[[ 26.14   3.82   16.22  23.29  30.21  29.89  20.65  36.78   3.63   8.23]  
[548.97  80.22 340.53 489.05 634.38 627.62 433.69 772.38  76.15 172.75]]
```

- A range of prices was extracted from two rows, and the result is a two-dimensional array.
- In this sliced array, row 0 holds tax prices, and row 1 holds total prices.

8. Summarize statistics about the pricing data.

- a) Scroll down and view the cell titled **Summarize statistics about the pricing data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 low = np.amin(prices[0])  
2 high = np.amax(prices[0])  
3 print('The lowest unit price is: ${:.2f}'.format(low))  
4 print('The highest unit price is: ${:.2f}'.format(high))
```

- Lines 1 and 2 identify the lowest and highest unit prices, respectively.
- You might want to identify the lowest and highest prices from the entire array, but in this case, it's probably more useful to do this per variable. That way, you can find the minimum and maximum for each price type. So, the code is indexing the entire first row, which holds the unit prices.

- c) Run the code cell.
d) Examine the output.

```
The lowest unit price is: $12.45.  
The highest unit price is: $99.42.
```

- e) Select the next code cell, and then type the following:

```

1 mean = prices[1].mean()
2 median = np.median(prices[1])
3 print('The mean tax price is: ${:.2f}'.format(mean))
4 print('The median tax price is: ${:.2f}'.format(median))

```

- Lines 1 and 2 identify the mean and median of the tax prices, respectively.
- Line 1 calls a method on the array class object, while line 2 calls the function directly and passes in the array as an argument. Some functions, like `numpy.median()`, don't support class method syntax.
- As before, you're calling these summary functions on a single row in the array.

- f) Run the code cell.
g) Examine the output.

```

The mean tax price is: $17.44.
The median tax price is: $16.14.

```

- h) Select the next code cell, and then type the following:

```

1 std = prices[2].std()
2 var = prices[2].var()
3 print('The standard deviation of total price is: {:.4f}'.format(std))
4 print('The variance of total price is: {:.4f}'.format(var))

```

- Lines 1 and 2 identify the standard deviation and variance of the total prices, respectively.
 - This time, the functions summarize the third row of the array.
- i) Run the code cell.
j) Examine the output.

```

The standard deviation of total price is: 250.6676.
The variance of total price is: 62834.2251.

```

- k) Select the next code cell, and then type the following:

```

1 total = prices[2].sum()
2 print('Total sales: ${:.2f}'.format(total))

```

- Line 1 identifies the sum total prices.
 - Because unit price and tax price both contribute to total price, it makes the most sense to just get the sum of total price to see how much money the store is generating through its customer transactions.
- l) Run the code cell.
m) Examine the output.

```
Total sales: $36627.35.
```

9. What other libraries can you use to perform more specialized or advanced statistical analyses?

10. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel**–**Shutdown**.
 - b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
 - c) Close the **Analyzing Arrays** tab in Firefox, but keep a tab open to **DSTIP/NumPy/** in the file hierarchy.
-

Do Not Duplicate or Distribute

Summary

In this lesson, you created and analyzed data loaded into NumPy arrays, and you saved NumPy data as shareable files. NumPy arrays are the fundamental data structure for data science operations in Python, so knowing how to work with them is crucial.

Do you plan on saving the NumPy arrays you work with? Why or why not? If so, what format(s) would you prefer to save the data in?

Do you plan on using any other statistical or scientific libraries to supplement your NumPy array analyses? If so, which libraries might you use?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

3

Transforming Data with NumPy

Lesson Time: 2 hours

Lesson Introduction

While analyzing data is an important part of the data science process, so is changing that data to meet your needs. Whether it's to prepare and clean the data, or to modify it for easier analysis and presentation, being able to transform your NumPy arrays is crucial.

Lesson Objectives

In this lesson, you will:

- Manipulate data in NumPy arrays.
- Modify data in NumPy arrays.

TOPIC A

Manipulate Data in NumPy Arrays

Before changing data outright, you'll likely want to reshape and reorder the data. After all, the data is not necessarily going to be in the perfect form for every task you're trying to accomplish.

Views vs. Copies

When you slice a normal Python® list, it returns a copy of the original list. For example:

```
>>> py_list = [0, 5, 10, 100, 1000, 10000]
>>> slice = py_list[1:3]
>>> slice
[5, 10]
```

Now, let's say you update `slice`:

```
>>> slice[0] = 1
>>> slice
[1, 10]
>>> py_list
[0, 5, 10, 100, 1000, 10000]
```

As you can see, `py_list` remains unchanged.

However, this is not the case with NumPy arrays:

```
>>> num_arr = numpy.array([0, 5, 10, 100, 1000, 10000])
>>> slice = num_arr[1:3]
>>> slice
array([5, 10])
>>> slice[0] = 1
>>> slice
array([1, 10])
>>> num_arr
array([0, 1, 10, 100, 1000, 10000])
```

As you can see, the original `num_arr` had its second value change, despite the fact that only `slice` was explicitly modified. This is because NumPy slicing returns a *view* of the array, not an independent copy. In this case, `slice` is merely a view into `num_arr`. The array data between each object is shared. If you were to create another slice, it would also be a view and not a separate copy. Therefore, it's important to be aware of how NumPy array slicing can impact the flow of your code. If you want to retrieve data and put it in a separate array, while keeping the original data intact, the default slicing method will not be adequate. Still, creating views is useful when you are processing large datasets, as it avoids duplicating the entire data buffer in memory.

The `numpy.copy()` Function

If you ever need to explicitly copy the data in an array when slicing, you can do so by using the `numpy.copy()` function. For example:

```
>>> num_arr = numpy.array([0, 5, 10, 100, 1000, 10000])
>>> slice = numpy.copy(num_arr[1:3])
>>> slice[0] = 1
>>> num_arr
array([0, 5, 10, 100, 1000, 10000])
```

Unlike with a view, the original array has not changed. You can now operate on `slice` as a separate object because its data is no longer being shared with `num_arr`. Keep in mind that, as compared to views, copies of large datasets may lead to performance issues.

You can also use `copy()` as a method of a NumPy array object. The following code does the same thing as the previous example:

```
>>> slice = num_arr[1:3].copy()
>>> slice[0] = 1
>>> num_arr
array([0, 5, 10, 100, 1000, 10000])
```

The `numpy.reshape()` Function

The `numpy.reshape()` function enables you to change an array's shape to the value you specify. This is a good way to clean up data that may have been formatted incorrectly, especially if you need to preserve some tabular relationship between the array items. It takes the array to reshape as the `a` argument, as well a tuple of the new shape for the `newshape` argument. For example:

```
x = numpy.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 10, 11, 12]])

numpy.reshape(x, (4, 3))
```

The array goes from (3, 4) to (4, 3):

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [10, 11, 12]])
```

By reshaping an array, you can also change its dimensions. The following example changes a one-dimensional array into a two-dimensional array (and uses object method syntax):

```
>>> x = numpy.arange(0, 100, 12)
>>> x
array([0, 12, 24, 36, 48, 60, 72, 84, 96])
>>> x.reshape((3, 3))
>>> x
array([[0, 12, 24],
       [36, 48, 60],
       [72, 84, 96]])
```

So, reshaping can be helpful when you need to take raw, one-dimensional data and make it more meaningful in a tabular form. However, you must make sure that your existing data is compatible with the new shape you're trying to fit it into. In the previous example, trying to fit the data into shape (2, 3) won't work, because there is too much data to fit into that shape.



Note: The `reshape()` function typically returns a view, rather than a copy.

The `numpy.resize()` Function

The `numpy.resize()` function is similar to `numpy.reshape()`, with a couple of key differences. One difference is that `resize()` changes the actual shape of the array in place, while `reshape()` just returns a temporary transformation.

The other difference is that `resize()` works when the new shape is not directly compatible with the existing shape. If the shape you're resizing to is smaller than the length of the initial array, then the excess values will not appear in the resulting array. If the shape you're resizing to is larger than the initial array, then the initial array's values will start repeating:

```
>>> numpy.resize(x, (4, 3))
array([[ 0, 12, 24],
       [36, 48, 60],
       [72, 84, 96],
       [ 0, 12, 24]])
```

However, if you use object method syntax, zeros will be added instead:

```
>>> x.resize(4, 3)
>>> x
array([[ 0, 12, 24],
       [36, 48, 60],
       [72, 84, 96],
       [ 0,  0,  0]])
```

The `numpy.ravel()` Function

The `numpy.ravel()` function "flattens" an array into a single dimension, while also ensuring that its memory block is contiguous. A contiguous memory block is usually faster to read from and write to. However, this also means that `ravel()` is not guaranteed to provide a view, and may end up providing a copy if that is required to maintain a contiguous block. Contrast that with using `reshape()` to flatten an array, which will return a view even if the memory block ends up being non-contiguous.

The following example flattens a two-dimensional array:

```
x = numpy.array([[1, 2, 3],
                [4, 5, 6]])
```

```
numpy.ravel(x)
```

This returns:

```
array([1, 2, 3, 4, 5, 6])
```

To do the same thing using object method syntax:

```
x.ravel()
```

Flattening a multi-dimensional array might come in handy when you need to feed the array as input to a tool that does not support multi-dimensional input. It can also help streamline array operations, like indexing and slicing, when the data doesn't need to be in multiple dimensions.



Note: An alternative function is `numpy.flatten()`, which always returns a copy of the array instead of a view.



Note: Unless specified, most of the functions discussed in this topic will return views.

The `numpy.flip()` Function

The `numpy.flip()` function reverses the order of items in an array while maintaining the array's shape. Take the following example:

```
x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

```
numpy.flip(x)
```

This results in:

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

In this case, the order of both the rows and the columns is reversed. However, you can flip the array by a specific dimension by providing a value to the `axis` argument. In the following example, only the rows are flipped:

```
>>> numpy.flip(x, 0) # Flip axis 0 (rows).
array([[7, 8, 9],
       [4, 5, 6],
       [1, 2, 3]])
```

In this next example, only the columns are flipped:

```
>>> numpy.flip(x, 1) # Flip axis 1 (columns).
array([[3, 2, 1],
       [6, 5, 4],
       [9, 8, 7]])
```

You might want to flip an array if, for example, it contains data that can be used to construct an image; by flipping the array, you can reorient the image horizontally or vertically. Another potential use case is if you have two datasets that hold related data, but they're ordered differently: one is in descending order, the other in ascending. By flipping one array, you make both arrays compatible and easier to work with.

The `numpy.transpose()` Function

Another way to reorder data is by using the `numpy.transpose()` function. By default, if an array has two dimensions, data in row 0 is shifted to column 0; data in row 1 is shifted to column 1; and so on. For example:

```
x = numpy.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
numpy.transpose(x)
```

The resulting view is:

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

In other words, the array's axes have been permuted (reordered).

The `numpy.concatenate()` Function

There may be situations where you want to combine arrays. You might have multiple sources of raw data, or you might have deliberately split an array so that you can operate on only certain parts, then combine them back into one. To combine two or more arrays into one, you can use the `numpy.concatenate()` function and provide a tuple of the arrays. Concatenating one-dimensional arrays simply adds the data in the order you provide:

```
x = numpy.array([1, 2, 3])
y = numpy.array([4, 5, 6])
numpy.concatenate((x, y))
```

This results in:

```
array([1, 2, 3, 4, 5, 6])
```

When you concatenate multi-dimensional arrays, the default behavior is to combine the data along the first axis (index 0):

```
>>> x = numpy.array([[1, 2, 3],
...                   [4, 5, 6]])
>>> y = numpy.array([[7, 8, 9],
...                   [10, 11, 12]])
>>> numpy.concatenate((x, y))
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [10, 11, 12]])
```

You can also concatenate along the columns axis:

```
>>> numpy.concatenate((x, y), axis = 1)
array([[1, 2, 3, 7, 8, 9],
       [4, 5, 6, 10, 11, 12]])
```

The `numpy.append()` Function

A related function is `numpy.append()`. It does essentially the same thing as `concatenate()`, except it takes two arguments, rather than a single tuple: `arr` is the array you're appending the values to, and `values` are the values you're appending to the array (e.g., another array). Also, unlike `concatenate()`, `append()` automatically flattens an array unless you specify the `axis` to perform the operation on:

```
>>> x = numpy.array([[1, 2, 3],
...                   [4, 5, 6]])
>>> y = numpy.array([[7, 8, 9],
...                   [10, 11, 12]])
>>> numpy.append(x, y)
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

Stacking Functions

You can simplify concatenation syntax by using one of the following stacking functions, all of which take a tuple of the arrays to combine.

Function	Description
<code>numpy.vstack()</code>	Combines arrays along the vertical axis (i.e., stacks rows). Equivalent to <code>concatenate()</code> with <code>axis = 0</code> .
<code>numpy.hstack()</code>	Combines arrays along the horizontal axis (i.e., stacks columns). Equivalent to <code>concatenate()</code> with <code>axis = 1</code> .
<code>numpy.dstack()</code>	Combines arrays along the third axis. Equivalent to <code>concatenate()</code> with <code>axis = 2</code> .

The `numpy.stack()` Function

The `numpy.stack()` function behaves somewhat differently. Whereas `concatenate()` and its equivalent stacking functions combine arrays along an existing axis, `stack()` combines them along a new axis. For example, you can combine two one-dimensional arrays into a single two-dimensional array:

```
>>> x = numpy.array([1, 2, 3])
>>> y = numpy.array([4, 5, 6])
>>> numpy.stack((x, y))
array([[1, 2, 3],
       [4, 5, 6]])
```

Or, by column:

```
>>> numpy.stack((x, y), axis = 1)
array([[1, 4],
       [2, 5],
       [3, 6]])
```

The `numpy.split()` Function

Just as you can combine arrays, you can likewise split them. The `numpy.split()` function is the primary way to do this. Aside from providing the array you want to split as an argument, you must also specify *how* you want to split the array. This is done through the `indices_or_sections` argument. If you supply an integer, that integer is the number of subarrays that the function will return, with the data split evenly (if it can be):

```
x = numpy.array([1, 2, 3, 4, 5, 6])
numpy.split(x, 2)
```

This returns a Python list, where each index is a subarray:

```
[array([1, 2, 3]), array([4, 5, 6])]
```

If the data cannot be split evenly, Python will throw an error.

You can also split based on specific indices. In the following example, the first split occurs at index 1, then the second split occurs at index 4:

```
>>> numpy.split(x, (1, 4))
[array([1]), array([2, 3, 4]), array([5, 6])]
```

Additional Splitting Functions

If you have arrays of multiple dimensions, you can specify the axis to split on by using the `axis` argument in `numpy.split()`. However, like with concatenation, there are alternative functions you can use to do this.

Function	Description
<code>numpy.vsplit()</code>	Splits arrays along the vertical axis (i.e., splits rows). Equivalent to <code>split()</code> with <code>axis = 0</code> .
<code>numpy.hsplit()</code>	Splits arrays along the horizontal axis (i.e., splits columns). Equivalent to <code>split()</code> with <code>axis = 1</code> .
<code>numpy.dsplit()</code>	Splits arrays along the third axis. Equivalent to <code>split()</code> with <code>axis = 2</code> .

The `numpy.array_split()` Function

The `numpy.array_split()` function does the same thing as `numpy.split()`, except it works when an even split isn't possible:

```
>>> x = numpy.array([1, 2, 3, 4, 5, 6, 7])
>>> numpy.array_split(x, 3)
[array([1, 2, 3]), array([4, 5]), array([6, 7])]
```

The `numpy.sort()` Function

Anyone who's ever looked at a spreadsheet knows how important sorting can be to analyzing data. It gives you the information that's most important to you up front, and it also enables you to process your data in a specific order. There are several ways to sort a NumPy array, the most fundamental of which is, of course, `numpy.sort()`. Here's a simple example:

```
x = numpy.array([4, 3, 5, 1, 2])
numpy.sort(x)
```

The result is:

```
array([1, 2, 3, 4, 5])
```

If you sort a multi-dimensional array, it will, by default, be sorted by the last axis. So, in a two-dimensional array, it will sort by columns:

```
>>> x = numpy.array([[5, 3, 2, 8],
...                  [2, 9, 4, 1],
...                  [8, 7, 5, 3]])
>>> numpy.sort(x)
array([[2, 3, 5, 8],
       [1, 2, 4, 9],
       [3, 5, 7, 8]])
```

To sort by rows, use the `axis` argument:

```
>>> numpy.sort(x, axis = 0)
array([[2, 3, 2, 1],
       [5, 7, 4, 3],
       [8, 9, 5, 8]])
```

Note that `sort()` returns a *copy* of the array, not a view. There's a good reason for this, and you can see it in the previous examples. The relationship between each cell is not maintained; all cells in a row do not stay together, for instance. A copy is created, so there's no impact on the integrity of the original array.

Sorting Algorithms

In any context, not just NumPy, there are many ways a group of numbers can be sorted. Each algorithm performs its sorting logic differently, and some are faster than others in certain situations. You can actually tell `sort()` to use a specific algorithm through the `kind` argument. The default, '`'quicksort'`', tends to be faster than the others.

The `numpy.argsort()` Function

The `numpy.argsort()` function is essentially the same as `sort()`, with one key difference—it returns the indices of each cell in the array, ordered from smallest value to largest. For example:

```
x = numpy.array([4, 3, 5, 1, 2])
numpy.argsort(x)
```

The result is:

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

Index 3 holds the smallest value (1), so it is first; index 4 holds the next-largest value (2), so it is second; and so on.

Recall that `sort()` does not maintain the structure of the data, which is why it returns a copy. You can actually use the `argsort()` function, along with a bit of slicing and indexing, to maintain this structure:

```
>>> x = numpy.array([[5, 3, 2, 8],
...                  [2, 9, 4, 1],
...                  [8, 7, 5, 3]])
>>> i = numpy.argsort(x[:, 0])
>>> x[i]
array([[2, 9, 4, 1],
       [5, 3, 2, 8],
       [8, 7, 5, 3]])
```

The array input passed to `argsort()` is `[:, 0]`. This slices all rows and indexes the first column. So, `argsort()` will perform the sort only on that first column. Since it returns indices, you can plug the result of the `argsort()` back into the array to get the fully sorted array with its structure intact.

The `numpy.partition()` Function

Another way to sort a NumPy array is by partitioning low and high values according to some specified number of values. The `numpy.partition()` function does just that. It takes a `kth` argument that specifies how many of the lowest values should be placed at the beginning of the array, with all other values placed to the right. There is no guaranteed order within each partition, however.

In the following example, `kth` is 3:

```
x = numpy.array([8, 9, 3, 4, 1, 6, 5])
numpy.partition(x, 3)
```

The result is:

```
array([1, 4, 3, 9, 6, 8])
```

The three lowest numbers (1, 3, and 4) are placed into a partition and sent to the beginning of the array. Every remaining number is placed to the right. There is no defined order within each partition (e.g., 4 comes before 3).

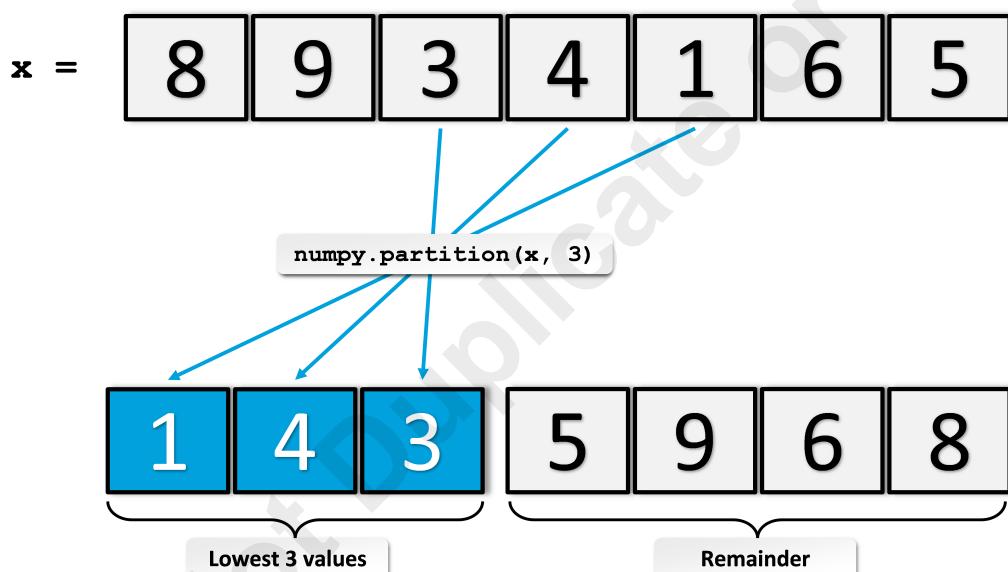


Figure 3-1: A visualization of the partitioning example.

Searching Functions

Searching functions enable you to find and alter certain values within NumPy arrays. This can be useful for identifying anomalous data points or for bringing outliers to the forefront. There are several functions that you can use to search for data, some of which are described in the following table.

The example outputs work with the following array:

```
x = numpy.array([[0, 9, 3],
                [2, 1, 7],
                [8, 0, 5]])
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

Function	Description and Example
<code>numpy.argmax(a, axis)</code>	Returns the indices of the maximum values in an array/axis. <pre>>>> numpy.argmax(x) 1</pre>
<code>numpy.argmin(a, axis)</code>	Returns the indices of the minimum values in an array/axis. <pre>>>> numpy.argmin(x, 0) array([0, 2, 0])</pre>
<code>numpy.nonzero(a)</code>	Returns the indices of all non-zero items in an array. The result is a tuple that includes each axis and the location of non-zero elements along that axis. <pre>>>> numpy.nonzero(x) (array([0, 0, 1, 1, 1, 2, 2]), array([1, 2, 0, 1, 2, 0, 2]))</pre>
<code>numpy.argwhere(a)</code>	Returns the indices of all non-zero items in an array. The result is grouped by elements in a NumPy array. <pre>>>> numpy.argwhere(x) array([[0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 2]])</pre>

Guidelines for Manipulating Data in NumPy Arrays



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are manipulating data in NumPy arrays.

Manipulate Data in NumPy Arrays

When manipulating data in NumPy arrays:

- Be aware of the difference between a view of and a copy of an array.
- Recognize when a view is created instead of a copy.
 - For example, slicing creates a view, while some functions create copies.
 - Use the `copy()` function when you want a copy and not a view.
 - Use functions like `reshape()` and `ravel()` to change an array's shape.
 - Use functions like `flip()` and `transpose()` to change the orientation of data in an array.
 - Use `concatenate()` and the associated stacking functions to append more data to an array, or to combine arrays.
 - Use `split()` and the associated splitting functions to divide an array into multiple parts.
 - Use `sort()` to sort the values in an array.
 - Use `argsort()` to retrieve the indices of each value as they are sorted in an array.
 - Use to sort while maintaining record integrity.
 - Use searching functions to retrieve maximum, minimum, and non-zero items in an array.

ACTIVITY 3-1

Manipulating Data in NumPy Arrays

Data Files

/home/student/DSTIP/NumPy/Manipulating Arrays.ipynb
 /home/student/DSTIP/NumPy/data/customer_ratings.npy
 /home/student/DSTIP/NumPy/data/unit_tax_total.csv
 /home/student/DSTIP/NumPy/data/revenue_cogs.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You're starting to get more data into your NumPy environment. In addition to the existing pricing data and customer ratings, you've also been given a data file with records on revenue and cost of goods (COGS) for each of the customer transactions you've been working with. You want to add this data to your primary array so that you can see all of what you have so far in a single dataset. However, you recognize that it might be easier to work with each attribute of the transaction by placing them in their own arrays. To get a more readable perspective into your data, you'll also transform the structure of the array so that each transaction is a row, and each attribute of the transaction is a column.

Lastly, you want to see how customer ratings relate to how much each is spent on each transaction. Do unsatisfied customers spend less on their purchases? Do satisfied customers lead to higher sales, or is there no significant correlation? So, you'll combine the ratings with the rest of the financial data, then sort this combined array by customer ratings. Manipulating the data in these ways can help reveal insights that would otherwise be difficult to attain.

1. In Jupyter Notebook, open the **DSTIP/NumPy/Manipulating Arrays.ipynb** file.
2. Import the relevant software libraries and load the datasets.
 - a) View the cell titled **Import software libraries and load the datasets**, and examine the code listing below it.
 On line 13, an extra data file called `revenue_cogs` is being loaded.
 - b) Run the code cell.
3. Concatenate prices with `revenue_cogs`.
 - a) Scroll down and view the cell titled **Concatenate prices with revenue_cogs**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 | revenue_cogs
```

- c) Run the code cell.

- d) Examine the output.

```
array([[522.83, 76.4, 324.31, 465.76, ],
       [604.17, 597.73, 413.04, 735.6, ],
       [72.52, 164.52, 57.92, 102.04, ],
       [234.75, 431.9, 713.8, 562.32, ],
       [482.51, 435.66, 164.01, 80.6, ],
       [430.2, 263.94, 66.4, 172.8, ],
       [265.89, 420.72, 33.52, 175.34, ],
       [441.8, 224.01, 470.65, 702.63, ],
       [670.24, 193.16, 397.68, 68.12, ],
       [212.1, 547.02, 420.26, 240.06, ]])
```

- This array holds the revenue and cost of goods (COGS) for each customer transaction.
- The array is in two dimensions, much like `prices`.
- Revenue is in row 1, and COGS is in row 2.

- e) Select the next code cell, and then type the following:

```
1 print('Shape of revenue_cogs: {}'.format(revenue_cogs.shape))
2 print('Shape of prices: {}'.format(prices.shape))
```

- f) Run the code cell.
g) Examine the output.

```
Shape of revenue_cogs: (2, 100)
Shape of prices: (3, 100)
```

Both arrays have the same number of columns, so it should be fairly easy to concatenate them without needing to alter their shapes.

- h) Select the next code cell, and then type the following:

```
1 finances = np.vstack((prices, revenue_cogs))
2 print('First 5 columns of stacked array:\n{}'.format(finances[:, :5]))
3 print('\nShape of finances: {}'.format(finances.shape))
```

Line 1 stacks the two arrays vertically so that rows from `revenue_cogs` are added to `prices`.

- i) Run the code cell.
j) Examine the output.

```
First 5 columns of stacked array:
[[ 74.69 15.28 46.33 58.22 86.31]
 [ 26.14  3.82 16.22 23.29 30.21]
 [548.97 80.22 340.53 489.05 634.38]
 [522.83 76.4 324.31 465.76 604.17]
 [500.24 73.21 321.12 430.98 578.9 ]]

Shape of finances: (5, 100)
```

- The new `finances` array has the rows from both constituent arrays stacked on top of each other. Rows from `revenue_cogs` have been appended to the bottom of `prices`.
- The new shape shows that the number of columns has remained the same, but there are now 5 total rows.

4. Transpose `finances` so it's in a more readable format.

- a) Scroll down and view the cell titled **Transpose `finances` so it's in a more readable format**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 finances = finances.transpose()
2 print('First 10 rows of finances:\n {}'.format(finances[:10, :]))
3 print('\nShape of finances: {}'.format(finances.shape))
```

It's usually more common to represent a dataset where the observations (each customer transaction) are rows, and attributes or features of those observations (the pricing information) are columns. Transposing the array essentially swaps the rows and columns while maintaining the relationship between cells.

- c) Run the code cell.
d) Examine the output.

```
First 10 rows of finances:
[[ 74.69  26.14 548.97 522.83 500.24]
 [ 15.28   3.82 80.22  76.4   73.21]
 [ 46.33  16.22 340.53 324.31 321.12]
 [ 58.22  23.29 489.05 465.76 430.98]
 [ 86.31  30.21 634.38 604.17 578.9 ]
 [ 85.39  29.89 627.62 597.73 585.04]
 [ 68.84  20.65 433.69 413.04 395.42]
 [ 73.56  36.78 772.38 735.6   702.08]
 [ 36.26   3.63 76.15  72.52  69.8 ]
 [ 54.84   8.23 172.75 164.52 156.08]]
```

```
Shape of finances: (100, 5)
```

As intended, the result of the transpose is that each transaction is a row, and each pricing attribute is a column. The shape of the array confirms this. The columns are now in the following order:

1. Unit price
2. Tax price
3. Total price
4. Revenue
5. COGS

5. Split finances into multiple arrays.

- a) Scroll down and view the cell titled **Split finances into multiple arrays**, then select the code cell below it.
b) In the code cell, type the following:

```
1 split = np.hsplit(finances, (2, 4))
```

- c) Run the code cell.
6. Before you run the next code cell, answer the following question.

How many arrays has `finances` been split into, given the code cell you just ran?

- 2
- 3
- 4
- 5

7. Continue the splitting operation.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

- a) Select the next code cell, and then type the following:

```
1 print('Number of arrays after split: {}'.format(len(split)))
```

- b) Run the cell.
c) Examine the output.

Number of arrays after split: 3.

- d) Select the next code cell, and then type the following:

```
1 print('Unit and tax prices:\n{}'.format(split[0][:5, :]))
2 print('\nTotal prices and revenue:\n{}'.format(split[1][:5, :]))
3 print('\nCOGS:\n{}'.format(split[2][:5, :]))
```

- A splitting operation returns a Python list, with each split array as an element of that list. The previous cell confirmed that there were three split arrays.
 - `split` is being indexed/sliced twice for each line—the first time selects the desired split array from the list, and the second time slices that split array. For example, line 1 retrieves the split array at index 0, then slices the first five rows (and all columns) from that array.
- e) Run the code cell.
f) Examine the output.

```
Unit and tax prices:
[[74.69 26.14]
 [15.28  3.82]
 [46.33 16.22]
 [58.22 23.29]
 [86.31 30.21]]

Total prices and revenue:
[[548.97 522.83]
 [ 80.22  76.4 ]
 [340.53 324.31]
 [489.05 465.76]
 [634.38 604.17]]

COGS:
[[500.24]
 [ 73.21]
 [321.12]
 [430.98]
 [578.9 ]]
```

The first five rows from each split array are displayed. Although concatenating arrays can help you manage that data all in one object, splitting arrays can make related portion of the dataset easier to work with, especially since there are no axis labels.

8. Make the split arrays copies instead of views.

- a) Scroll down and view the cell titled **Make the split arrays copies instead of views**, then select the code cell below it.

- b) In the code cell, type the following:

```

1 def view_copy_test(split_arr, new_price):
2     print('First unit price in finances (before): {}.' \
3           .format(finances[0, 0]))
4
5     # Use split to change first unit price.
6     split_arr[0, 0] = new_price
7
8     print('First unit price in finances (after): {}.' \
9           .format(finances[0, 0]))
10
11 view_copy_test(split[0], 75.02)

```

- The function defined on lines 1 through 9, when called, will test if the initial `finances` array is updated as a result of updating any of the `split` arrays.
 - The function takes the `split` array as the `split_arr` argument and an updated price as `new_price`.
 - On lines 2 through 3, the function checks the `finances` value in row 0, column 0.
 - On line 6, the `split` array is assigned a new price for this same index.
 - On lines 8 through 9, the `finances` value is checked again to see if it has changed.
 - Line 11 calls the function the first time, passing in the first `split` array and a new price.
- c) Run the code cell.
d) Examine the output.

```

First unit price in finances (before): 74.69.
First unit price in finances (after): 75.02.

```

- Despite the fact that only the `split` array was explicitly changed, the `finances` array was also updated with the new value. This is because the `split` operation took a view of the `finances` array, not a copy. Changes to the view will always apply changes to the array it is viewing.
- e) Select the next code cell, and then type the following:

```

1 unit_and_tax = split[0].copy()
2 total_and_rev = split[1].copy()
3 cogs = split[2].copy()
4 print('Split variables defined.')

```

- These variable assignments generate copies of the `split` arrays by using the `copy()` function.
- f) Run the code cell.
g) Select the next code cell, and then type the following:

```

1 # Set price back to what it was.
2 finances[0, 0] = 74.69
3
4 view_copy_test(unit_and_tax, 75.02)

```

- This time, you're calling the test function on a copy of a `split` array, not a view.
- h) Run the code cell.

- i) Examine the output.

```
First unit price in finances (before): 74.69.
First unit price in finances (after): 74.69.
```

As expected, changing the copy did not change the initial `finances` array.

9. What is the primary advantage of using a view over using a copy?

10. Append the `ratings` array to `finances`.

- a) Scroll down and view the cell titled **Append the `ratings` array to `finances`**, then select the code cell below it.
 b) In the code cell, type the following:

```
1 print('Shape of ratings: {}'.format(ratings.shape))
2 print('Shape of finances: {}'.format(finances.shape))
```

You want to append the `ratings` array to `finances` as a new column, but you need to check each array's shape to see if they're compatible.

- c) Run the code cell.
 d) Examine the output.

```
Shape of ratings: (100,)
Shape of finances: (100, 5)
```

Both arrays have 100 rows, but the `ratings` array is one dimensional and `finance` is two dimensional. This will pose a problem, as you can't use the standard stacking or concatenation functions to append a 1-D array to a 2-D array like this.

- e) Select the next code cell, then type the following:

```
1 ratings = ratings.reshape((100, 1))
2 finances_and_ratings = np.hstack((finances, ratings))
3 finances_and_ratings[:5, :]
```

- Line 1 reshapes the `ratings` array to give it an extra dimension, making it compatible for stacking.
 - Line 2 stacks `finances` horizontally onto `ratings`.
- f) Run the code cell.
 g) Examine the output.

```
array([[ 74.69,   26.14,  548.97,  522.83,  500.24,    9.1 ],
       [ 15.28,    3.82,   80.22,   76.4 ,   73.21,    9.6 ],
       [ 46.33,   16.22,  340.53,  324.31,  321.12,   7.4 ],
       [ 58.22,   23.29,  489.05,  465.76,  430.98,   8.4 ],
       [ 86.31,   30.21,  634.38,  604.17,  578.9 ,   5.3 ]])
```

The `ratings` array values now show up as a new column at the end of `finances_and_ratings`.

11. Sort columns and rows in `finances_and_ratings`.

- a) Scroll down and view the cell titled **Sort columns and rows in finances_and_ratings**, then select the code cell below it.
- b) In the code cell, type the following:

```

1 s = slice(0, 5) # Slice first 5 rows.
2 col_sort = np.sort(finances_and_ratings[s, :])
3 row_sort = np.sort(finances_and_ratings[s, :], axis = 0)
4 print('Unsorted:\n{}'.format(finances_and_ratings[s, :]))
5 print('\nSorted by columns:\n{}'.format(col_sort[s, :]))
6 print('\nSorted by rows:\n{}'.format(row_sort[s, :]))

```

- To keep the output simple, you'll sort just the first five rows.
 - Line 2 sorts the array by columns.
 - Line 3 sorts the array by rows.
- c) Run the code cell.
- d) Examine the output.

```

Unsorted:
[[ 74.69  26.14 548.97 522.83 500.24   9.1 ]
 [ 15.28   3.82  80.22  76.4    73.21   9.6 ]
 [ 46.33  16.22 340.53 324.31 321.12   7.4 ]
 [ 58.22  23.29 489.05 465.76 430.98   8.4 ]
 [ 86.31  30.21 634.38 604.17 578.9    5.3 ]]

Sorted by columns:
[[ 9.1    26.14 74.69 500.24 522.83 548.97]
 [ 3.82   9.6   15.28 73.21 76.4   80.22]
 [ 7.4    16.22 46.33 321.12 324.31 340.53]
 [ 8.4    23.29 489.05 465.76 430.98 489.05]
 [ 5.3    30.21 86.31 578.9   604.17 634.38]]

Sorted by rows:
[[ 15.28   3.82  80.22  76.4    73.21   5.3 ]
 [ 46.33  16.22 340.53 324.31 321.12   7.4 ]
 [ 58.22  23.29 489.05 465.76 430.98   8.4 ]
 [ 74.69  26.14 548.97 522.83 500.24   9.1 ]
 [ 86.31  30.21 634.38 604.17 578.9   9.6 ]]

```

- The column sort has reordered all column values in each row from lowest to highest.
- The row sort has reordered all the row values in each column from lowest to highest.
- While sorting an array with one axis is usually safe, sorting an array with multiple rows or columns may negatively affect the relational nature of a dataset. For example, in the column sort, the first column includes both customer ratings and tax prices. In the row sort, two values from the same transaction are in different rows.

12. Sort finances_and_ratings by customer ratings.

- a) Scroll down and view the cell titled **Sort finances_and_ratings by customer ratings**, then select the code cell below it.
- b) In the code cell, type the following:

```

1 smart_sort = np.argsort(finances_and_ratings[:, 5])
2 smart_sort

```

- Using `argsort()` can help account for the data integrity issue you just experienced. It retrieves the indices of the sorted values.
 - Line 1 sorts only on column 5 (ratings).
- c) Run the code cell.

- d) Examine the output.

```
array([85, 72, 97, 5, 47, 69, 22, 19, 15, 31, 10, 16, 37, 20, 30, 92, 21,
       96, 33, 32, 4, 58, 75, 80, 42, 88, 14, 77, 6, 56, 65, 71, 9, 24,
       79, 53, 89, 41, 68, 81, 54, 52, 57, 78, 84, 82, 94, 66, 26, 11, 35,
       17, 36, 61, 12, 90, 44, 8, 95, 87, 99, 29, 2, 34, 38, 64, 43, 39,
       27, 49, 40, 7, 48, 13, 55, 91, 83, 46, 3, 51, 25, 63, 76, 93, 18,
       73, 0, 98, 59, 74, 45, 50, 28, 1, 70, 67, 23, 86, 60, 62])
```

The result is an array where the indices of `finances_and_ratings` are sorted according to the values they hold. For example, index 85 is first, meaning that it contains the lowest customer rating.

- e) Select the next code cell, and then type the following:

```
1 sorted_arr = finances_and_ratings[smart_sort]
2 sorted_arr[:5, :]
```

Line 2 uses the `smart_sort` indices from the previous cell as a fancy index into `finances_and_ratings`. This retrieves each row in order of lowest to highest rating value.

- f) Run the code cell.
g) Examine the output.

```
array([[ 83.06,  29.07, 610.49, 581.42, 564.36,   4. ],
       [ 48.52,   7.28, 152.84, 145.56, 138.12,   4. ],
       [ 12.45,   3.73,  78.44,  74.7 ,  72.06,  4.1 ],
       [ 85.39,  29.89, 627.62, 597.73, 585.04,  4.1 ],
       [ 20.01,    9. , 189.09, 180.09, 175.49,  4.1 ]])
```

The ratings column is now in order, but it's difficult to tell whether or not each rating stayed with its associated prices. You could display the entire array and look for the row yourself, but there are several ways to make this easier.

- h) Select the next code cell, and then type the following:

```
1 print('Row index of lowest rating (first occurrence): {}'.format(np.argmin(finances_and_ratings[:, 5])))
```

This finds the index of the minimum value in the unsorted array. You can use this to compare the first row in the sorted array to its equivalent row in the unsorted array.

- i) Run the code cell.
j) Examine the output.

```
Row index of lowest rating (first occurrence): 72.
```

- k) Select the next code cell, and then type the following:

```
1 print('Row from initial array:\n {}' .format(finances_and_ratings[72, :]))
2 print('\nRow from sorted array:\n {}' .format(sorted_arr[1, :]))
```

There are two rows that have the minimum rating value; row 1 in the sorted array is the one that matches up with row 72 in the unsorted array.

- l) Run the code cell.

-
- m) Examine the output.

```
Row from initial array:  
[ 48.52    7.28 152.84 145.56 138.12    4. ]  
  
Row from sorted array:  
[ 48.52    7.28 152.84 145.56 138.12    4. ]
```

As intended, this smarter sorting operation kept values in the same row together.

13. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel→Shutdown**.
 - b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
 - c) Close the **Manipulating Arrays** tab in Firefox, but keep a tab open to **DSTIP/NumPy** in the file hierarchy.
-

TOPIC B

Modify Data in NumPy Arrays

You've manipulated the order and shape of the data inside of NumPy arrays, and now it's time to change that data itself. This is an important task that applies to many different phases of the data science process.

The `numpy.insert()` Function

When you accumulate data that you want to add to an existing dataset, it won't always make sense to just add it to the end. If the data has some natural order, for example, you'll want to insert it into a specific place. So, the `numpy.insert()` function enables you to specify where you want to add the new data. This is done through the second argument, `obj`, which takes the index before which you want to insert the data:

```
x1 = numpy.array([1, 2, 3, 7, 8, 9])
x2 = numpy.array([4, 5, 6])

numpy.insert(x1, 3, x2)
```

So, because the second array is inserted before index 3, the result is:

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

Just like with `append()`, the `insert()` function creates a flattened copy of the original array unless you specify an axis. The following example inserts a new array into an existing two-dimensional array:

```
>>> x1 = numpy.array([[1, 2, 3],
...                   [7, 8, 9]])
>>> x2 = numpy.array([4, 5, 6])
>>> numpy.insert(x1, 1, x2, axis = 0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Because `obj` is 1, the new array is inserted at that index for the specified axis—in this case, the row axis.

The `numpy.delete()` Function

There may be times when you need to remove anomalous or incorrect data to minimize the effect of noise or other issues with the overall dataset. In these cases, you can get a copy of a NumPy array with the offending data removed. This is possible by calling the `numpy.delete()` function. With this function, you specify the indices of the item(s) you want to remove in the `obj` argument:

```
x = numpy.array([1, 2, 3, 9, 4, 5, 6])
numpy.delete(x, 3)
```

Which returns:

```
array([1, 2, 3, 4, 5, 6])
```

Or, provide multiple indices in a tuple format:

```
>>> numpy.delete(x, (3, 6))
array([1, 2, 3, 4, 5])
```

Like with `insert()`, the array is flattened unless you specify an axis from which to remove data. In the following example, the entire second row is being deleted:

```
>>> x = numpy.array([[1, 2, 3],
...                  [9, 8, 7],
...                  [4, 5, 6]])
>>> numpy.delete(x, 1, axis = 0)
array([[1, 2, 3],
       [4, 5, 6]])
```

Universal Functions

As discussed previously, one of NumPy's most powerful features is vectorization—a method of optimizing how operations are applied to an entire array. The mathematical functions that support vectorization are called *universal functions*. There are many such functions, and they cover several different types of mathematical operations, including but not limited to:

- Arithmetic operations
- Exponential and logarithmic operations
- Trigonometric operations
- Comparison operations
- Logical operations

When it comes to syntax, you can use a universal function without calling that function directly. For example, the plus (+) operator, when applied to NumPy arrays, acts as a wrapper to the `numpy.add()` universal function.



Note: The universal functions discussed in this topic will typically return copies, rather than views.

Arithmetic Functions and Operators

The following table lists some of the most common universal functions that you can use in arithmetic. As you'll see, the operations are applied to each element in the array.

The example outputs work with the following array:

```
x = numpy.array([1, 2, 3, 4, 5])
```



Note: The examples show both ways to reach the same result; you need to do only one.

Function	Wrapper Operator and Example
<code>numpy.add(x1, x2)</code>	Operator: + <code>>>> numpy.add(x, 3)</code> <code>>>> x + 3</code> <code>array([4, 5, 6, 7, 8])</code>
<code>numpy.subtract(x1, x2)</code>	Operator: - <code>>>> numpy.subtract(x, 3)</code> <code>>>> x - 3</code> <code>array([-2, -1, 0, 1, 2])</code>
<code>numpy.multiply(x1, x2)</code>	Operator: * <code>>>> numpy.multiply(x, 3)</code> <code>>>> x * 3</code> <code>array([3, 6, 9, 12, 15])</code>

Function	Wrapper Operator and Example
<code>numpy.divide(x1, x2)</code>	Operator: / <pre>>>> numpy.divide(x, 5) >>> x / 5 array([0.2, 0.4, 0.6, 0.8, 1.])</pre>
<code>numpy.floor_divide(x1, x2)</code>	Operator: // <pre>>>> numpy.floor_divide(x, 5) >>> x // 5 array([0, 0, 0, 0, 1])</pre>
<code>numpy.mod(x1, x2)</code>	Operator: % <pre>>>> numpy.mod(x, 5) >>> x % 5 array([1, 2, 3, 4, 0])</pre>
<code>numpy.power(x1, x2)</code>	Operator: ** <pre>>>> numpy.power(x, 3) >>> x ** 3 array([1, 8, 27, 64, 125])</pre>

Broadcasting

In the examples in the previous table, the arithmetic operations were being performed on scalar values. You can also just as easily operate on two arrays:

```
>>> x = numpy.array([1, 2, 3, 4, 5])
>>> y = numpy.array([3, 3, 3, 3, 3])
>>> x + y
array([4, 5, 6, 7, 8])
```

Notice how this is the exact same as `x + 3`, except that the scalar is now an array (`y`) of the same shape as `x`. In fact, this is exactly what NumPy does when arithmetic operations are performed on arrays—it "stretches" or "pads" a scalar value out until it meets the dimensions and shape of the largest array in the calculation. The padding just repeats the values that are already in the array being padded. This process is called **broadcasting**. Broadcasting is useful because it enables NumPy to perform arithmetic operations on arrays of different shapes.

Observe what happens when `y` no longer has the same shape as `x`:

```
>>> y = numpy.array([3, 3, 3])
>>> x + y
ValueError: operands could not be broadcast together with shapes (5,) (3,)
```

Broadcasting must abide by a set of rules in order to work properly.

Broadcasting Rules

You must be mindful of the rules of broadcasting when attempting to operate on arrays of different shapes and dimensions. Otherwise, Python may throw an error, or you may end up with unintended results. The rules are as follows:

- When two arrays have a different number of dimensions, the array with fewer dimensions has its shape padded with 1s at the beginning.

- In the following example, `y` has fewer dimensions than `x`:

```
>>> x = numpy.array([[1, 2]])
>>> y = numpy.array([3, 4])
>>> x.shape
```

```
(1, 2)
>>> y.shape
(2,)
```

So, the shape of `y` will be padded with a 1 at the beginning so that it becomes shape `(1, 2)`. This makes it the same shape as `x`. The result of this broadcasting when applied to addition is:

```
>>> x + y
array([[4, 6]])
```

2. The size of each dimension in the output is equal to the largest size of that dimension in the inputs.

- In the following example, `z` has three dimensions:

```
>>> z = numpy.array([[[5, 6]]])
>>> x.shape
>>> y.shape
>>> z.shape
(1, 2)
(2,)
(1, 1, 2)
```

Recall that `y` is broadcast with 1s padding (due to Rule #1). The same will happen to `x` so that it matches the shape of `z`. Ultimately, the largest size of each dimension is `(1, 1, 2)`. This is confirmed like so:

```
>>> x + y + z
array([[[ 9, 12]]])
>>> (x + y + z).shape
(1, 1, 2)
```

3. If the arrays have the same number of dimensions, but the sizes of a dimension do not match, the array whose dimension is size 1 will be stretched to match the array with a larger dimension.

- In the following example, `x` and `y` now have different shapes:

```
>>> x = numpy.array([[1,
...                   [2]])
>>> y = numpy.array([3, 4])
>>> x.shape
>>> y.shape
(2, 1)
(2,)
```

First, Rule #1 applies to turn `y` into a two-dimensional array with shape `(1, 2)`. The shapes now look like this:

```
(2, 1)
(1, 2)
```

The sizes for each dimension do not match, but there is a 1 in both cases, so that 1 stretches to become a 2. Therefore, the shape of both arrays becomes `(2, 2)`:

```
>>> x + y
array([[4, 5],
       [5, 6]])
>>> (x + y).shape
(2, 2)
```

4. If Rule #3 applies *except* that no array has a dimension size of 1, then broadcasting will not work and Python will throw an error.

- This is why the example mentioned earlier resulted in an error. Here's another example:

```
>>> x = numpy.array([[1, 2],
...                   [3, 4]])
>>> y = numpy.array([[5, 6, 7],
```

```

...
[8, 9, 10]])

>>> x.shape
>>> y.shape
(2, 2)
(2, 3)

```

Since these arrays have the same dimensions, the padding in Rule #1 is not necessary. Although they have the same number of rows (2), they have a different number of columns, and neither array has 1 column. Trying to add these arrays will not work:

```

>>> x + y
ValueError: operands could not be broadcast together with shapes (2, 2)
(2, 3)

```



Note: The data in a one-dimensional array is broadcast as axis 1. In other words, arrays of size (5, 3) and (5,) will not broadcast because the first array's column size (3) does not match the second array (5). If the second array were (3,), broadcasting would work.

Exponential and Logarithmic Functions

There are more universal functions than those that perform arithmetic. The following table lists some of the most common functions used to perform exponential and logarithmic operations on NumPy arrays. All of these functions take argument *x* as the input.

The example outputs work with the following array:

```
x = numpy.array([1, 2, 3])
```

Function	Description and Example
<code>numpy.exp()</code>	Returns the exponential function of each array value. <pre>>>> numpy.exp(x) array([2.7182, 7.3890, 20.0855])</pre>
<code>numpy.expm1()</code>	Returns the exponential function of each array value minus 1. <pre>>>> numpy.expm1(x) array([1.7182, 6.3890, 19.0855])</pre>
<code>numpy.exp2()</code>	Returns 2 raised to the power of each array value. <pre>>>> numpy.exp2(x) array([2., 4., 8.])</pre>
<code>numpy.log()</code>	Returns the natural logarithm of each array value. <pre>>>> numpy.log(x) array([0., 0.6931, 1.0986])</pre>
<code>numpy.log10()</code>	Returns the base-10 logarithm of each array value. <pre>>>> numpy.log10(x) array([0., 0.3010, 0.4771])</pre>
<code>numpy.log2()</code>	Returns the base-2 logarithm of each array value. <pre>>>> numpy.log2(x) array([0., 1., 1.5849])</pre>
<code>numpy.log1p()</code>	Returns the natural logarithm plus 1 of each array value. <pre>>>> numpy.log1p(x) array([0.6931, 1.0986, 1.3862])</pre>

Additional Mathematical Functions

The following table lists some additional mathematical functions you may want to apply to your data, including trigonometric functions.

The examples here work with the following arrays:

```
x = numpy.array([1, 2, 3])
y = numpy.array([4, 5, 6])
```

Function	Description and Example
<code>numpy.sin(x)</code>	Returns the sine of each value in an array, where each value equates to radians. <pre>>>> numpy.sin(x) array([0.8414, 0.9092, -0.1411])</pre>
<code>numpy.cos(x)</code>	Returns the cosine of each value in an array, where each value equates to radians. <pre>>>> numpy.cos(x) array([0.5403, -0.4161, -0.9899])</pre>
<code>numpy.tan(x)</code>	Returns the tangent of each value in an array, where each value equates to radians. <pre>>>> numpy.tan(x) array([1.5574, -2.1850, 0.1425])</pre>
<code>numpy.hypot(x1, x2)</code>	Returns the hypotenuse of a right triangle, where each value across the same axis in both arrays corresponds to a "leg" of the triangle. <pre>>>> numpy.hypot(x, y) array([4.1231, 5.3851, 6.7082])</pre>
<code>numpy.absolute(x)</code>	Returns the absolute number of each value in an array. <pre>>>> numpy.absolute(x) array([1, 2, 3])</pre>
<code>numpy.square(x)</code>	Returns the square of each value in an array. <pre>>>> numpy.square(x) array([1, 4, 9])</pre>
<code>numpy.sqrt(x)</code>	Returns the square root of each value in an array. <pre>>>> numpy.sqrt(y) array([2., 2.2360, 2.4494])</pre>
<code>numpy.around(a, decimals)</code>	Returns rounded numbers of each value in an array, where <code>decimals</code> is an integer that specifies the number of places to round to. <pre>>>> hypot = numpy.hypot(x, y) >>> numpy.around(hypot, 2) array([4.12, 5.39, 6.71])</pre>



Note: In addition to the functions provided directly by NumPy, you can also leverage SciPy functions to perform specialized mathematical operations.

Comparison Functions and Operators

Like with regular Python, you can test the relationship between items in a NumPy array by using comparison operators. As these are universal functions, the comparisons are performed element by element, so NumPy returns an array of `True` or `False` values for each array item depending on the conditional logic of the expression. These are also referred to as Boolean arrays.

Like arithmetic functions in NumPy, comparison functions have equivalent wrapper operators that map to regular Python comparison operators. The following table lists the comparison functions, all of which take `x1` and `x2` as input arguments. The example outputs work with the following array:

```
x = numpy.array([1, 2, 3, 4, 5])
```

Function	Wrapper Operator and Example
<code>numpy.equal()</code>	Operator: <code>==</code> <code>>>> numpy.equal(x, 3)</code> <code>>>> x == 3</code> <code>array([False, False, True, False, False])</code>
<code>numpy.not_equal()</code>	Operator: <code>!=</code> <code>>>> numpy.not_equal(x, 3)</code> <code>>>> x != 3</code> <code>array([True, True, False, True, True])</code>
<code>numpy.less()</code>	Operator: <code><</code> <code>>>> numpy.less(x, 3)</code> <code>>>> x < 3</code> <code>array([True, True, False, False, False])</code>
<code>numpy.less_equal()</code>	Operator: <code><=</code> <code>>>> numpy.less_equal(x, 3)</code> <code>>>> x <= 3</code> <code>array([True, True, True, False, False])</code>
<code>numpy.greater()</code>	Operator: <code>></code> <code>>>> numpy.greater(x, 3)</code> <code>>>> x > 3</code> <code>array([False, False, False, True, True])</code>
<code>numpy.greater_equal()</code>	Operator: <code>>=</code> <code>>>> numpy.greater_equal(x, 3)</code> <code>>>> x >= 3</code> <code>array([False, False, True, True, True])</code>

Comparing Arrays

Just as you can perform arithmetic on multiple arrays, rather than just an array and a scalar, you can do the same with comparison operators. For example:

```
>>> x = numpy.array([8, 9, 3])
>>> y = numpy.array([5, 2, 9])
>>> x > y
array([True, True, False])
```

Note that the same broadcasting rules apply if your arrays are of different shapes.

Logical Functions and Operators

Logical functions—which connect multiple values together so that they can be evaluated—also work on NumPy arrays. They return Boolean arrays, much like comparison functions. And, like comparison functions, they have wrapping operators you can use instead of the function call.

The following table lists the universal logical functions. The example outputs work with the following array:

```
x = numpy.array([1, 2, 3, 4, 5])
```

Function	Wrapping Operator and Example
<code>numpy.logical_and(x1, x2)</code>	Operator: & <pre>>>> numpy.logical_and(x > 3, x < 5) >>> (x > 3) & (x < 5) array([False, False, False, True, False])</pre>
<code>numpy.logical_or(x1, x2)</code>	Operator: <pre>>>> numpy.logical_or(x > 3, x < 1) >>> (x > 3) (x < 1) array([False, False, False, True, True])</pre>
<code>numpy.logical_xor(x1, x2)</code>	Operator: ^ <pre>>>> numpy.logical_xor(x > 3, x > 1) >>> (x > 3) ^ (x > 1) array([False, True, True, False, False])</pre>
<code>numpy.logical_not(x)</code>	No operator <pre>>>> numpy.logical_not(x < 3) array([False, False, True, True, True])</pre>



Caution: The equivalent wrapping operators are bitwise operators. Using built-in Python keywords like and and or will evaluate the entire array, not each element.

Evaluating Arrays

Just like comparison functions, you can apply multiple arrays to a logical function:

```
>>> x = numpy.array([8, 9, 3])
>>> y = numpy.array([5, 2, 9])
>>> (x < y) & (x >= 3)
array([False, False, True])
```

Again, different shapes of arrays will be subject to broadcasting rules.

Masking

The Boolean arrays returned by comparison and logical functions are very useful for both analyzing and modifying data. Let's say you have a dataset of sales figures by sales rep, where each value is in thousands of dollars:

```
sales = numpy.array([[35, 27, 41, 56, 31],
                    [38, 21, 39, 999, 26],
                    [999, 29, 46, 38, 34]])
```

You may have noticed an anomaly in this array: most values are between \$20,000 and \$60,000, but a couple are \$999,000—completely unrealistic. So, how do you identify these anomalies? You can do so by applying a mask. **Masking** is the bitwise process of selecting a subset of an array to keep, while discarding the rest.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

In practical terms, this means applying a comparison and/or logical expression as an index to an array. Instead of an array of Booleans, masking returns the actual values from the array that return True. For example:

```
>>> sales[sales > 100]
array([999, 999])
```

As you can see, the index to the array is just a bitwise expression. What results is every value for which this expression is True. You can also combine multiple comparison and/or logical operators to create more complex masking:

```
>>> sales[(sales < 100) & (sales > 40)]
array([41, 56, 46])
```

So, now that you can select anomalous values, you want to change them to be more realistic. Here's an example:

```
>>> sales[sales > 100] = 40
>>> sales
array([[35, 27, 41, 56, 31],
       [38, 21, 39, 40, 26],
       [40, 29, 46, 38, 34]])
```

The numpy.where() Function

An alternative to array masking is the `numpy.where()` function. It returns array items based on the condition you provide. The function takes three arguments: `condition`, `x`, and `y`. Where `condition` is True, `x` will be returned; where False, `y` will be returned. The following example does essentially the same thing as the masking example: sales above (\$100,000 will be changed to \$40,000; otherwise, the sales values will remain as is).

```
>>> numpy.where(sales > 100, 40, sales)
array([[35, 27, 41, 56, 31],
       [38, 21, 39, 40, 26],
       [40, 29, 46, 38, 34]])
```

String Modification Functions

NumPy provides the `numpy.char` module for vectorized operations on string-based arrays. These are all based on the string methods that are part of Python's standard library.

The example outputs work with the following array:

```
x = numpy.array(['nAme', 'QuEst', 'cOLOr'])
```

	Note: In the table, the <code>numpy</code> namespace is omitted for brevity.
Function	Description and Example
<code>char.add(x1, x2)</code>	Returns a concatenation of two string elements in arrays. <pre>>>> char.add(x, '?') array(['nAme?', 'QuEst?', 'cOLOr?'])</pre>
<code>char.multiply(a, i)</code>	Returns a multiple concatenation of string elements in arrays, where <code>i</code> is an integer specifying the number of times to multiply. <pre>>>> char.multiply(x, 2) array(['nAmenAme', 'QuEstQuEst', 'cOL0rc0L0r'])</pre>

Function	Description and Example
<code>char.replace(a, old, new)</code>	Returns string elements in an array that replace occurrences of old with new. <pre>>>> char.replace(x, '0', 'o') array(['nAme', 'QuEST', 'coLor'])</pre>
<code>char.upper(a)</code>	Returns all string elements in an array turned to uppercase. <pre>>>> char.upper(x) array(['NAME', 'QUEST', 'COLOR'])</pre>
<code>char.lower(a)</code>	Returns all string elements in an array turned to lowercase. <pre>>>> char.lower(x) array(['name', 'quest', 'color'])</pre>
<code>char.capitalize(a)</code>	Returns all string elements in an array with their first character capitalized. <pre>>>> char.capitalize(x) array(['Name', 'Quest', 'COLOR'])</pre>
<code>char.title(a)</code>	Returns all string elements in an array with title casing applied. <pre>>>> char.title(x) array(['Name', 'Quest', 'COLOR'])</pre>
<code>char.swapcase(a)</code>	Returns all string elements in an array with their casing swapped. <pre>>>> char.swapcase(x) array(['NaME', 'qUESt', 'cOLOR'])</pre>

Comparison Operations

The `numpy.char` module also provides comparison functions and operators that are essentially the same as their equivalent universal functions. However, the difference is that any trailing whitespace is stripped from the end of the string before the comparison is made.

Guidelines for Modifying Data in NumPy Arrays

Follow these guidelines when you are modifying data in NumPy arrays.

Modify Data in NumPy Arrays

When modifying data in NumPy arrays:

- Use the `insert()` function to add data to an array at some place other than at the end.
- Use the `delete()` function to remove unwanted data from an array.
- Use universal functions to perform mathematical operations on arrays.
 - Use to perform arithmetic, trigonometry, etc.
 - Use wrapper operators instead of the equivalent function calls, when available.
- When operating on arrays, ensure they are broadcastable.
 - Review the rules of broadcasting to ensure the arrays are actually compatible.
 - Python will throw an error if the arrays are not broadcastable.
- Use comparison operators and logical operators to evaluate desired conditions as either true or false.
- Use masking to retrieve and/or change the values that match a condition.

ACTIVITY 3–2

Modifying Data in NumPy Arrays

Data Files

/home/student/DSTIP/NumPy/Modifying Arrays.ipynb
 /home/student/DSTIP/NumPy/data/unit_and_tax.csv
 /home/student/DSTIP/NumPy/data/total_and_rev.csv
 /home/student/DSTIP/NumPy/data/cogs.csv
 /home/student/DSTIP/NumPy/data/quantity.csv

Before You Begin

Jupyter Notebook is open.

Scenario

As part of the data analysis team at GCE, you've been given several tasks that involve modifying some of the data that you've already seen, as well as some new data that's starting to come in. In particular, you need to do the following:

- The quantities sold for each transaction is now available for loading into NumPy. Rather than append it to an array, you want to insert it after unit price and before tax price, as this aligns more closely with how total price is calculated (i.e., unit price \times quantity + tax price).
- Sales tax will soon increase from 5% to 6%, so you want to adjust your current sales data to see how the prices might change in the future.
- A team member has informed you that the data collection process has encountered a few errors, and that some transactions are marked with mistaken quantities. You want to identify these erroneous transactions and remove them from the data.
- You want to identify transactions that lead to losses, as this may influence the organization's pricing decisions.
- Lastly, you want identify major transactions—those that include high quantities and/or involve a high COGS. You'll identify the gross profits for each of these major transactions to see if they are meeting the company's goals.

1. In Jupyter Notebook, open the **DSTIP/NumPy/Modifying Arrays.ipynb** file.
2. Import the relevant software libraries and load the datasets.
 - a) View the cell titled **Import software libraries and load the datasets**, and examine the code listing below it.

On lines 10 through 17, the following datasets are loaded as arrays:

 - `unit_and_tax` —Unit price and tax price.
 - `total_and_rev` —Total price and revenue.
 - `cogs` —COGS.
 - `quantity` —Quantity sold.
 - b) Run the code cell.
3. Insert `quantity` between unit price and tax price.
 - a) Scroll down and view the cell titled **Insert quantity between unit price and tax price**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 print('Quantity array:\n {}'.format(quantity))
2 print('\nRow example from unit and tax prices:\n {}' \
3       .format(unit_and_tax[0, :]))
```

- c) Run the code cell.
d) Examine the output.

```
Quantity array:
[ 7.  5.  7.  8.  7.  7.  6. 10.  2.  3.  4.  4.  5. 10. 10.  6.  7.  6.
 3.  2.  5.  3.  2.  5.  3.  8.  1.  2.  5.  9.  5.  9.  8.  2.  4.  1.
 5.  9.  8.  8.  1.  2.  6.  8. -1.  4.  9.  9.  6. 10.  7.  5.  4.  1.
 2.  8.  2.  0. 10.  6.  3.  6.  9. 10.  4. 10.  2.  6. 10.  1. 10. 10.
 3.  6. -1. 10.  9.  9. 10.  5.  6.  3.  8.  4.  5.  7.  5.  0.  1.  6.
 4. 10.  7.  9.  1.  1. 10.  6.  3.  6.]
```

Row example from unit and tax prices:
[75.02 26.14]

The `quantity` array is one dimensional. You want to place this as a column between the unit price and the tax price, so it would become the new column 1.

- e) Select the next code cell, and then type the following:

```
1 unit_and_tax = np.insert(unit_and_tax, 1, quantity, axis = 1)
2 unit_and_tax[:10, :]
```

Line 1 inserts the `quantity` array along axis 1 (columns), before the current column 1.

- f) Run the code cell.
g) Examine the output.

```
array([[75.02,  7. , 26.14],
       [15.28,  5. ,  3.82],
       [46.33,  7. , 16.22],
       [58.22,  8. , 23.29],
       [86.31,  7. , 30.21],
       [85.39,  7. , 29.89],
       [68.84,  6. , 20.65],
       [73.56, 10. , 36.78],
       [36.26,  2. ,  3.63],
       [54.84,  3. ,  8.23]])
```

The purchase quantities are now shown in the middle of the array.

4. Adjust the pricing for the new tax rate.

- a) Scroll down and view the cell titled **Adjust the pricing for the new tax rate**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 new_tax_rate = .06
2 new_tax_prices = unit_and_tax[:, 0] * unit_and_tax[:, 1] * new_tax_rate
3 new_tax_prices
```

Line 2 performs some arithmetic: each unit price in the array is multiplied by each quantity in the array, then each result is multiplied by the new tax rate. Using wrapper operators like * and + leverages their universal function equivalents to perform these calculations efficiently on each item in the array.

- c) Run the code cell.
d) Examine the output.

```
array([31.5084, 4.584, 19.4586, 27.9456, 36.2502, 35.8638, 24.7824,
       44.136, 4.3512, 9.8712, 3.4752, 6.1224, 14.085, 25.914,
       42.828, 33.7392, 28.9506, 26.1396, 9.8406, 4.836, 25.812,
       15.8364, 3.984, 10.368, 15.9534, 25.2432, 2.0112, 10.5204,
       26.508, 13.4406, 28.239, 42.1578, 40.2144, 11.5896, 23.8608,
       4.0872, 18.786, 32.8752, 26.3616, 14.4576, 5.2032, 6.7332,
       24.8832, 47.376, -0.9222, 22.5504, 30.6126, 10.8054, 6.8148,
       49.578, 38.388, 13.377, 4.2888, 0.9258, 1.9392, 41.2704,
       5.3208, 0, 43.41, 11.0196, 4.4532, 20.0628, 29.7378,
       9.486, 18.1776, 9.522, 4.0164, 35.1396, 47.262, 1.0998,
       53.688, 37.272, 8.7336, 27.3276, -4.4802, 24.99, 26.4816,
       10.8054, 46.986, 6.114, 35.7084, 17.4024, 9.24, 19.2864,
       14.673, 34.8852, 22.956, 0, 2.5482, 27.7164, 11.3712,
       26.916, 9.2316, 34.7544, 5.385, 5.8296, 52.722, 4.482,
       9.495, 29.772])
```

The one-dimensional array returned here holds the updated tax prices. You'll just replace the existing tax prices with this data.



Note: Remember, NumPy has no `Decimal` data type, and some precision is lost by performing calculations on floats. As mentioned earlier, you could multiply the prices by 100 and cast them as integers as a workaround.

- e) Select the next code cell, and then type the following:

```
1 unit_and_tax[:, 2] = new_tax_prices
2 unit_and_tax[:10, :]
```

- f) Run the code cell.
g) Examine the output.

```
array([[75.02, 7., 31.5084],
       [15.28, 5., 4.584],
       [46.33, 7., 19.4586],
       [58.22, 8., 27.9456],
       [86.31, 7., 36.2502],
       [85.39, 7., 35.8638],
       [68.84, 6., 24.7824],
       [73.56, 10., 44.136],
       [36.26, 2., 4.3512],
       [54.84, 3., 9.8712]])
```

The tax prices (the last column) have been updated with the new data. Since the updated tax prices also affect the total prices, you'll need to update the totals as well.

- h) Select the next code cell, and then type the following:

```
1 total_and_rev[:10, :]
```

- i) Run the code cell.
j) Examine the output.

```
array([[548.97, 522.83],
       [ 80.22,  76.4 ],
       [340.53, 324.31],
       [489.05, 465.76],
       [634.38, 604.17],
       [627.62, 597.73],
       [433.69, 413.04],
       [772.38, 735.6 ],
       [ 76.15,  72.52],
       [172.75, 164.52]])
```

In this array, the first column shows the total prices (unit price × quantity + tax price), and the second column shows revenue (unit price × quantity).

- k) Select the next code cell, and then type the following:

```
1 total_and_rev[:, 0] = unit_and_tax[:, 0] * \
2                               unit_and_tax[:, 1] + unit_and_tax[:, 2]
3 total_and_rev[:10, :]
```

Lines 1 and 2 update the total price using the formula mentioned in the previous step. Revenue isn't updated, as it's not affected by the tax increase.

- l) Run the code cell.
m) Examine the output.

```
array([[556.6484, 522.83 ],
       [ 80.984 ,  76.4 ],
       [343.7686, 324.31 ],
       [493.7056, 465.76 ],
       [640.4202, 604.17 ],
       [633.5938, 597.73 ],
       [437.8224, 413.04 ],
       [779.736 , 735.6 ],
       [ 76.8712,  72.52 ],
       [174.3912, 164.52 ]])
```

The total prices (first column) have been updated to reflect the tax increase.

5. Identify and remove anomalous entries.

- a) Scroll down and view the cell titled **Identify and remove anomalous entries**, then select the code cell below it.
b) In the code cell, type the following:

```
1 quant_err = unit_and_tax[:, 1] < 1
2 quant_err
```

- This code starts by actually identifying which quantity entries are less than 1 (e.g., a zero or negative quantity) in `unit_and_tax`.
- The code uses the less than (`<`) comparison operator.

- c) Run the code cell.
- d) Examine the output.

```
array([False, False, False, False, False, False, False, False,
       False, False, False, True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, True, False, False, False, False, False, False,
       False, False, False, False, False, False, True, False, False,
       False, False, False, False, False, False, False, False, False])
```

- The result of the comparison operation is a Boolean array that matches the size of the input (in this case, the single-column slice of `unit_and_tax`).
- Each value in the input that does *not* match the condition is given a value of `False`.
- Each value in the input that *does* match the condition is given a value of `True`.
- The prevalence of `False` values indicates that most entries are not anomalous; however, if you scan the Boolean array, you can see that some values did return `True` (i.e., they have a mistaken quantity).

- e) Select the next code cell, and then type the following:

```
1 unit_and_tax[quant_err]
```

By passing in the Boolean array to the data array of interest—a process called masking—you can identify what rows match the anomalous condition.

- f) Run the code cell.
- g) Examine the output.

```
array([[15.37 , -1.    , -0.9222],
       [89.6  ,  0.    ,  0.    ],
       [74.67 , -1.    , -4.4802],
       [49.38 ,  0.    ,  0.    ]])
```

There are four rows that have a mistaken quantity, as confirmed by the second column. You'll delete these rows, but before you can do that, you need to identify where they are in the `unit_and_tax` array.

- h) Select the next code cell, and then type the following:

```
1 err_ind = np.where(quant_err)
2 err_ind
```

When you supply the condition to the `where()` function and don't provide any other arguments, it returns the indices of where the condition is `True`.

- i) Run the code cell.
- j) Examine the output.

```
(array([44, 57, 74, 87]),)
```

You'll use these row indices to delete the anomalous rows from the array.

k) Select the next code cell, and then type the following:

```
1 unit_and_tax = np.delete(unit_and_tax, err_ind, axis = 0)
2 total_and_rev = np.delete(total_and_rev, err_ind, axis = 0)
3 cogs = np.delete(cogs, err_ind, axis = 0)
4 print(unit_and_tax[:, 1] < 1)
5 print('\nRows remaining: {}'.format(unit_and_tax.shape[0]))
```

- Lines 1 through 3 call `delete()` using the error indices you just retrieved, deleting the matches along the row axis (axis 0).
 - You'll delete these anomalous rows across all of your arrays to ensure that they stay in sync.
 - Line 4 confirms that the rows were deleted by trying to identify the condition again.
- l) Run the code cell.
m) Examine the output.

```
[False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False
 False False False False False False False False False False]
```

Rows remaining: 96.

- The Boolean array shows no `True` values, so the offending rows were deleted.
- All four offending rows were removed, so there are 96 rows left out of the original 100.

6. Identify losses.

- a) Scroll down and view the cell titled **Identify losses**, then select the code cell below it.
b) In the code cell, type the following:

```
1 # See if arrays are compatible for broadcasting.
2 print('Shape of total_and_rev array: {}' \
3       .format(total_and_rev.shape))
4 print('Shape of cogs array: {}'.format(cogs.shape))
```

Your objective here is to compare `total_and_rev` with `cogs`. Before you can perform an operation using multiple arrays, you must ensure they have the same shape, and if they don't, that they are still compatible for broadcasting.

- c) Run the code cell.
d) Examine the output.

```
Shape of total_and_rev array: (96, 2)
Shape of cogs array: (96,)
```

7. If you tried to compare these two arrays as is, what would happen?

- The arrays will be broadcast successfully since both arrays have the same dimensions.
- The arrays will be broadcast successfully since the cogs array will be padded with 1 to match the size of total_and_rev.
- Broadcasting will fail since both arrays have different sizes in the same axis, and neither of those sizes is 1.
- Broadcasting will fail since, even after being padded with 1, cogs will not have the same dimensions as total_and_rev.

8. Continue identifying losses.

- a) Select the next code cell, and then type the following:

```
1 rev = total_and_rev[:, 1]
2 print('Shape of rev array: {}'.format(rev.shape))
```

- Although both arrays will not broadcast in their default form, you can slice total_and_rev so that its shape is the same as cogs.
 - Line 1 does the slicing and extracts just the revenue. This is all you need to calculate the losses, as total price does not factor into it.
- b) Run the cell.
c) Examine the output.

Shape of rev array: (96,)

Both rev and cogs are the same shape, so there's no need to rely on broadcasting.

- d) Select the next code cell, and then type the following:

```
1 loss_arr = rev < cogs
2 loss_arr
```

Line 1 identifies where in the array revenue is less than the COGS, which will lead to a loss.

- e) Run the code cell.
f) Examine the output.

```
array([False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False])
```

The Boolean array shows a few instances where this condition is True.

g) Select the next code cell, and then type the following:

```
1 loss_amount = rev[loss_arr] - cogs[loss_arr]
2 print('Revenue: {}'.format(rev[loss_arr]))
3 print('COGS:   {}'.format(cogs[loss_arr]))
4 print('Loss:    {}'.format(loss_amount))
```

- Line 1 calculates the actual loss values by subtracting COGS from revenue.
 - Lines 2 and 3 use the `loss_arr` mask to identify the revenue and COGS values matching the condition.
- h) Run the code cell.
i) Examine the output.

```
Revenue: [441.8 240.96 789.6 ]
COGS:   [443.54 241.91 789.76]
Loss:    [-1.74 -0.95 -0.16]
```

The transactions leading to a loss are shown, as are the actual loss values.

9. Identify high-quantity or high-COGS purchases, and determine their profits.

- a) Scroll down and view the cell titled **Identify high-quantity or high-COGS purchases and determine profit**, then select the code cell below it.
b) In the code cell, type the following:

```
1 unit_and_quant = unit_and_tax[:, :2]
2 high = (cogs >= 700) | (unit_and_tax[:, 1] >= 10)
3 stack = (unit_and_quant[high],
4           cogs[high].reshape(len(cogs[high]), 1))
5 high_arr = np.hstack(stack)
6 high_arr
```

- Line 1 slices unit price and quantity out of the main array to make it easier to work with these values.
 - Line 2 uses both comparison and logical operators to construct the following condition: return all transactions where the COGS is equal to or above \$700, *or* quantity is equal to or above 10.
 - Lines 3 and 4 define a tuple to use in stacking arrays.
 - Line 3 masks the `unit_and_quant` array with the condition.
 - Line 4 masks the `cogs` array with the condition, then reshapes it so that it can be appended to `unit_and_quant` as the last column.
 - Line 5 does the concatenation so that you can see unit price, quantity, and COGS in context for all transactions meeting the condition.
- c) Run the code cell.

- d) Examine the output.

```
array([[ 73.56,  10. ,  702.08],
       [ 43.19,  10. ,  420.76],
       [ 71.38,  10. ,  675.18],
       [ 98.7 ,   8. ,  789.76],
       [ 82.63,  10. ,  800.45],
       [ 72.35,  10. ,  686.28],
       [ 15.81,  10. ,  154.87],
       [ 15.87,  10. ,  155.48],
       [ 78.77,  10. ,  762.98],
       [ 89.48,  10. ,  850.89],
       [ 62.12,  10. ,  593.16],
       [ 41.65,  10. ,  407.62],
       [ 78.31,  10. ,  749.4 ],
       [ 44.86,  10. ,  427.73],
       [ 87.87,  10. ,  848.4 ]])
```

Some insights you can glean from this result include:

- Only one purchase of 8 or fewer items led to a COGS over \$700.
- Several purchases of 10 items led to a COGS below \$700.
- No purchases exceeded 10 items.

- e) Select the next code cell, and then type the following:

```
1 gross_income = rev[high] - cogs[high]
2 gross_income = gross_income.reshape(len(gross_income), 1)
3 np.hstack((high_arr, gross_income))
```

- Line 1 generated the gross income by subtracting COGS from revenue, but only for transactions meeting the previously defined condition.
- Line 2 reshapes `gross_income` so that it can be appended to the previous array.
- Line 3 performs the stacking operation.

- f) Run the code cell.

- g) Examine the output.

```
array([[ 73.56,  10. ,  702.08,  33.52],
       [ 43.19,  10. ,  420.76,  11.14],
       [ 71.38,  10. ,  675.18,  38.62],
       [ 98.7 ,   8. ,  789.76, -0.16],
       [ 82.63,  10. ,  800.45,  25.85],
       [ 72.35,  10. ,  686.28,  37.22],
       [ 15.81,  10. ,  154.87,  3.23],
       [ 15.87,  10. ,  155.48,  3.22],
       [ 78.77,  10. ,  762.98,  24.72],
       [ 89.48,  10. ,  850.89,  43.91],
       [ 62.12,  10. ,  593.16,  28.04],
       [ 41.65,  10. ,  407.62,  8.88],
       [ 78.31,  10. ,  749.4 ,  33.7 ],
       [ 44.86,  10. ,  427.73,  20.87],
       [ 87.87,  10. ,  848.4 ,  30.3 ]])
```

Here you can see unit price, quantity, COGS, and gross income for all transactions meeting the condition. Some insights you can glean from this result include:

- The transaction with a quantity of 8 is one of the transactions that led to a loss.
- Most of these major transactions led to gross profits exceeding \$20 per transaction.
- A couple of the high-quantity, low-COGS transactions led to smaller gross profits.

10. Shut down this Jupyter Notebook kernel.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

-
- a) From the menu, select **Kernel**→**Shutdown**.
 - b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
 - c) Close the **Modifying Arrays** tab in Firefox, but keep a tab open to the file hierarchy in Jupyter Notebook.
-

Do Not Duplicate or Distribute

Summary

In this lesson, you manipulated and modified data in NumPy arrays. Being able to shape data, as well as change its values, will make the data easier to work with and help you extract valuable insights.

What types of universal functions do you think you'll use the most in your own work with NumPy arrays?

Do you foresee views of arrays being a problem, as opposed to copies? Why or why not?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

4

Managing and Analyzing Data with pandas

Lesson Time: 2 hours, 30 minutes

Lesson Introduction

While NumPy serves as the foundation of your data science tasks, you may instead work directly with the more user-friendly library called pandas, which builds on NumPy. Or, you may find it beneficial to work with both. In either case, as with NumPy, you'll want to begin by managing your data within pandas structures and then analyze that data for useful insights.

Lesson Objectives

In this lesson, you will:

- Create `Series` and `DataFrame` objects using the pandas library.
- Load existing datasets into pandas `DataFrames`, and save `DataFrames` as files.
- Analyze data in pandas `DataFrames`.
- Slice and filter data in pandas `DataFrames`.

TOPIC A

Create Series and DataFrames

You'll start by writing code to construct the two main objects provided by pandas—the `Series` and the `DataFrame`.

Limitations of NumPy

NumPy is extremely powerful, and it serves as the basis for many of the Python® data science tools that are used by professionals. However, it does have some limitations. These limitations are not necessarily flaws, as developers choose to focus on the most important parts of a project and ensure the highest level of performance, rather than continually adding new features. So, NumPy might simply be the wrong tool for certain kinds of jobs.

In any case, NumPy has two main limitations:

- One array can contain only one data type.
- Arrays don't support labeling, like column names.

If your data is heterogeneous (i.e., it has mixed data types), a NumPy array will not be sufficient. Likewise, it can be difficult to analyze values in large, complex NumPy arrays, since there is no clear labeling of what the data is referring to. The pandas library addresses these issues.

pandas Data Structures

The pandas library leverages NumPy, and can be thought of as an attempt to enhance the usability of the NumPy array object. Its data structures actually build on top of `ndarray`, so working with these structures will be somewhat familiar, with a few key differences. There are two primary data structures in pandas:

- `Series`
- `DataFrame`

Both objects differ from NumPy arrays in that they can contain heterogeneous data and can contain labels for easy reference. This makes them both better suited for data cleanup and "munging" processes than NumPy arrays, which tend to be better for operating on numerical data that has already been prepared.



Note: Data munging, also called data wrangling, is a technique that transforms data from one format to another to make it easier to analyze and process.

The Series Object

The `Series` object is essentially just a one-dimensional NumPy array with the ability to specify axis labels. It, therefore, has at least two components: an `ndarray` of values, and an `Index` object that holds the axis labels. Optionally, the entire `Series` may also be given a name as a unique identifier. Although one dimensional, `Series` objects are often represented in tabular form, with the indices in the first column and the associated values in the second column.

Index	Value
0	1971
1	1975
2	1979
3	1983

Figure 4–1: A representation of a simple Series object.

The indices in the preceding figure are the default: start with 0 and add one for every new value. But, pandas structures are flexible, and you can actually specify the format the indices take. This can be very useful when you go to index data for which the default index format is not very meaningful. For example, in the following figure, the year values of Monty Python movies are indexed by the (truncated) names of the movies themselves.

Index	Value
And Now	1971
Holy Grail	1975
Life of Brian	1979
Meaning of Life	1983

Figure 4–2: A Series with custom axis labels.

Interestingly, the ability to tie one item (i.e., a key) to another item (i.e., a value) is very similar to a standard Python dictionary. Aside from the greater level of flexibility offered by the pandas library, Series objects also offer better performance than dictionaries, much like how NumPy arrays perform better than lists.

Series Creation

You can call `pandas.Series()` to create a `Series` object. Using the parameter defaults, assigning it values is pretty much the same as a one-dimensional NumPy array:

```
pandas.Series([1, 'two', 3, 4.5])
```

The resulting object is displayed as follows:

```
0      1
1    two
2      3
3    4.5
dtype: object
```

So, this uses the 0-start index format as the axis labels, with each index referring to a value. Python also printed the `Series` object's type—in this case, a string object. The type rules are similar to NumPy; mixed types are automatically converted according to a hierarchy, where strings take precedence over numbers, floats take precedence over integers, etc.

If you want to specify your own axis labels, you can pass a list into the `index` argument:

```
>>> titles = ['And Now', 'Holy Grail', 'Life of Brian', 'Meaning of Life']
>>> pandas.Series([1971, 1975, 1979, 1983], index = titles)
And Now      1971
Holy Grail   1975
Life of Brian 1979
Meaning of Life 1983
dtype: int64
```

The DataFrame Object

The `Series` object is powerful, but the most widely used data structure in pandas is the `DataFrame` object. This is because a `DataFrame` object is essentially a two-dimensional version of a `Series`. The structure of a `DataFrame` closely resembles a table in a spreadsheet: there are rows and columns, along with axis labels for both. Naturally, this makes `DataFrames` a logical choice for representing tabular data.

A `DataFrame` is essentially just multiple `Series` objects, and as such, each `DataFrame` has indices and values. Because they have two dimensions, `DataFrames` also have column labels you can specify.

The diagram illustrates a `DataFrame` object with four rows and five columns. The columns are labeled "Year", "Director", "Running Time", and "IMDb Score". The rows are labeled "And Now", "Holy Grail", "Life of Brian", and "Meaning of Life". A bracket on the left is labeled "Row Labels" and spans all four rows. A bracket at the top is labeled "Column Labels" and spans all four columns. Arrows point from the labels to their corresponding cells in the table. The table data is as follows:

	Year	Director	Running Time	IMDb Score
And Now	1971	MacNaughton	88	7.6
Holy Grail	1975	Gilliam and Jones	92	8.2
Life of Brian	1979	Jones	94	8.1
Meaning of Life	1983	Jones	90	7.6

Figure 4–3: A representation of a simple `DataFrame` object.

When it comes to performance, two-dimensional NumPy arrays consume less memory than equivalent DataFrames. Likewise, NumPy arrays are typically processed faster when the number of rows is under 50,000. Between 50,000 and 500,000 rows, the performance varies depending on the type of operation you're performing. However, with datasets that exceed 500,000 rows, the DataFrame object usually performs better than a NumPy array. Keep this in mind when processing time is a major factor in your projects.

DataFrame Creation

There are several ways to create a DataFrame object through a `pandas.DataFrame()` call. If you provide only values, both the row and column labels will start with 0. In the following example, the input values are part of a two-dimensional NumPy array:

```
data = numpy.array([[1971, 'MacNaughton'],
                   [1975, 'Gilliam and Jones'],
                   [1979, 'Jones'],
                   [1983, 'Jones']])

pandas.DataFrame(data)
```

The resulting DataFrame is structured as:

	0	1
0	1971	MacNaughton
1	1975	Gilliam and Jones
2	1979	Jones
3	1983	Jones

If you want to specify row and column names, you can provide them as lists through the `index` and `columns` arguments, respectively:

```
>>> titles = ['And Now', 'Holy Grail', 'Life of Brian', 'Meaning of Life']
>>> pandas.DataFrame(data, index = titles, columns = ['Year', 'Director'])

      Year        Director
And Now    1971      MacNaughton
Holy Grail 1975  Gilliam and Jones
Life of Brian 1979          Jones
Meaning of Life 1983          Jones
```

Creating DataFrames with Series and Dictionaries

Aside from providing NumPy arrays and specifying labels as arguments, you can also construct DataFrames using both Series and Python dictionaries. For example:

```
>>> year = pandas.Series([1971, 1975, 1979, 1983], index = titles)
>>> director = pandas.Series(['MacNaughton', 'Gilliam and Jones',
...                           'Jones', 'Jones'], index = titles)
>>> pandas.DataFrame({'Year': year, 'Director': director})

      Year        Director
And Now    1971      MacNaughton
Holy Grail 1975  Gilliam and Jones
Life of Brian 1979          Jones
Meaning of Life 1983          Jones
```

During DataFrame creation, you pass the column names as dictionary keys and the associated Series as their values.

Series and DataFrame Attributes

Like with NumPy arrays, Series and DataFrame objects that you construct all have attributes. You can retrieve these attributes on an object to learn more about that object. The following table

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

describes several of the most common attributes. Each example output assumes that `x` is a `DataFrame` defined as follows:

```
data = numpy.array([[0, '00'],
                   [157, '9D'],
                   [220, 'DC']])
colors = ['Red', 'Green', 'Blue']
x = pandas.DataFrame(data, index = colors, columns = ['RGB', 'Hex'])
```

Attribute	Description and Example
<code>x.ndim</code>	Returns the number of dimensions in the <code>DataFrame</code> . Example output: 2
<code>x.shape</code>	Returns the shape of the <code>DataFrame</code> , i.e., number of rows by number of columns. Example output: (3, 2)
<code>x.size</code>	Returns the total number of items in the <code>DataFrame</code> . Example output: 6
<code>x.index</code>	Returns the row labels in an <code>Index</code> object. Example output: <code>Index(['Red', 'Green', 'Blue'], dtype='object')</code>
<code>x.columns</code>	Returns the column labels in an <code>Index</code> object. Example output: <code>Index(['RGB', 'Hex'], dtype='object')</code>
<code>x.dtypes</code>	Returns the type of data in the <code>DataFrame</code> . Example output: RGB object Hex object dtype: object

Data Type Conversion

Although a `DataFrame` can hold multiple data types, it does not do this on a cell-by-cell level. Instead, data types are defined for each column. Like with NumPy or a `pandas Series`, if a column has mixed data types, the values are automatically converted to the data type that best accommodates all values. A mix of integers and floats will be converted to floats; a mix of floats and strings will be converted to strings; and so on.

The `DataFrame.head()` and `DataFrame.tail()` Functions

Most datasets are large enough that it's not feasible to print every record. A common practice for getting a quick look at the format of your `DataFrame` is to print a limited number of records. The `DataFrame.head()` function outputs the first five rows of a `DataFrame`, along with all of its columns:

```
>>> x.head()
   Temperature  ExhaustVacuum  AmbientPressure  RelativeHumidity
0            8.34          40.77        1010.84           90.01
1           23.64          58.49        1011.40           74.20
2           29.74          56.90        1007.15           41.91
```

3	19.07	49.69	1007.22	76.79
4	11.80	40.66	1017.13	97.20



Note: As with NumPy, most pandas functions can be called directly, or they can be called as object methods, like in the preceding example.

You can override the default by providing the number of rows you want to see as the `n` argument.

The `DataFrame.tail()` function prints the *last* five rows in the dataset:

```
>>> x.tail()
   Temperature  ExhaustVacuum  AmbientPressure  RelativeHumidity
9563      15.12          48.92        1011.80            72.93
9564      33.41          77.95        1010.30            59.72
9565      15.99          43.34        1014.20            78.66
9566      17.65          59.87        1018.58            94.65
9567      23.68          51.30        1011.86            71.24
```

As with the `DataFrame.head()` function, you can pass a different number of rows into the `DataFrame.tail()` function.

Guidelines for Creating Series and DataFrames



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are creating `Series` and `DataFrames` in pandas.

Create Series and DataFrames

When creating `Series` and `DataFrames`:

- Consider using pandas when working with heterogeneous data.
- Consider using pandas when cleaning data.
- Prefer pandas over NumPy when the number of rows exceeds 500,000.
- Use the `Series` object for simple, one-dimensional data.
- Use the `DataFrame` object for two-dimensional data, particularly data in a tabular format.
- Use attributes to retrieve metadata about pandas structures.
- Consider that a column of a `DataFrame` must hold one type of data.
 - As with NumPy, the same data type conversion takes place in mixed data.
 - Each column in a `DataFrame` can contain a different data type.
- Use the `head()` and `tail()` functions to get a quick overview of the structure of your data.

ACTIVITY 4-1

Creating Series and DataFrames

Data Files

/home/student/DSTIP/pandas/Creating DataFrames.ipynb
 /home/student/DSTIP/pandas/data/initial_sales.csv
 /home/student/DSTIP/pandas/data/initial_ratings.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've been working with Greene City Emporium's sales data through NumPy thus far, but it's becoming difficult to keep track of all the rows and columns and what each means. You also anticipate getting more information about each transaction that isn't necessarily numeric, like where the transaction occurred and what product department it occurred in. You could create a separate array for each data type, but you would like to have one dataset from which to perform your analyses. NumPy will fall short of meeting this need.

To overcome these shortcomings, you'll begin transferring your data to pandas data structures. For now, you'll import the customer ratings as a one-dimensional `Series`, but you'll eventually want to incorporate the data into an overall `DataFrame`. Since you have a two-dimensional array of pricing and sales data, you'll create that `DataFrame` and assign it some meaningful labels.

1. In Jupyter Notebook, open the **DSTIP/pandas/Creating DataFrames.ipynb** file.
2. Import the relevant software libraries and load the datasets.
 - a) View the cell titled **Import software libraries and load the datasets**, and examine the code listing below it.
 Lines 12 through 15 are loading CSV files as NumPy arrays.
 - b) Run the code cell.
 - c) Verify that the version of Python is displayed, as are the versions of NumPy and pandas that were imported.

Libraries used in this project:
 - Python 3.7.6 (default, Jan 8 2020, 19:59:22)
 [GCC 7.3.0]
 - NumPy 1.18.1
 - pandas 1.0.1

Loaded datasets.

3. Create a `Series` from a 1-D NumPy array.
 - a) Scroll down and view the cell titled **Create a Series from a 1-D NumPy array**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 print(ratings_arr)
2 print('\nShape of ratings_arr: {}'.format(ratings_arr.shape))
```

- c) Run the code cell.
d) Examine the output.

```
[ 9.1  9.6  7.4  8.4  5.3  4.1  5.8  8.   7.2  5.9  4.5  6.8  7.1  8.2
 5.7  4.5  4.6  6.9  8.6  4.4  4.8  5.1  4.4  9.9  6.   8.5  6.7  7.7
 9.6  7.4  4.8  4.5  5.1  5.1  7.5  6.8  7.   4.7  7.6  7.7  7.9  6.3
 5.6  7.6  9.5  8.4  4.1  8.1  7.9  9.5  8.5  6.5  6.1  6.5  8.2  5.8
 6.6  9.3  10.   7.   10.   8.6  7.6  5.8  6.7  9.9  6.4  4.3  9.6  5.9
 4.   8.7  9.4  5.4  5.7  6.6  6.   5.5  6.4  6.6  8.3  6.6  4.   9.9
 7.3  5.7  7.1  8.2  5.1  8.6  6.6  7.2  5.1  4.1  9.3  7.4]
```

Shape of ratings_arr: (96,)

This is a one-dimensional NumPy array of customer ratings. To make it easier to work with, you'll transform it into a `Series` object.

- e) Select the next code cell, and then type the following:

```
1 ratings_s = pd.Series(ratings_arr)
2 ratings_s
```

- f) Run the code cell.
g) Examine the output.

```
0    9.1
1    9.6
2    7.4
3    8.4
4    5.3
...
91   7.2
92   5.1
93   4.1
94   9.3
95   7.4
Length: 96, dtype: float64
```

- pandas used `ratings_arr` as the data with which to create the `Series`.
- The `Series` is ordered by a visible row index and has a more tabular appearance than the NumPy array on which it was based.

- h) Select the next code cell, and then type the following:

```
1 print('Series shape:  {}'.format(ratings_s.shape))
2 print('Series indices: {}'.format(ratings_s.index))
```

- i) Run the code cell.

- j) Examine the output.

```
Series shape: (96,)
Series indices: RangeIndex(start=0, stop=96, step=1)
```

- The Series has the same shape as the NumPy array on which it was based.
- The row indices in the series are a RangeIndex object, which starts at 0 and ends at the total number of rows (96 in this case), with a step value of 1. You could give the row indices more meaningful labels, but at this point, you don't have the relevant information. So, you'll leave the rows with the default indices for now.

4. Create a DataFrame from a 2-D NumPy array.

- Scroll down and view the cell titled **Create a DataFrame from a 2-D NumPy array**, then select the code cell below it.
- In the code cell, type the following:

```
1 print(sales_arr[:10, :])
2 print('\nShape of sales_arr: {}'.format(sales_arr.shape))
```

- Run the code cell.
- Examine the output.

```
[[ 74.69   7.    26.14 548.97 522.83 500.24]
 [ 15.28   5.    3.82  80.22 76.4   73.21]
 [ 46.33   7.    16.22 340.53 324.31 321.12]
 [ 58.22   8.    23.29 489.05 465.76 430.98]
 [ 86.31   7.    30.21 634.38 604.17 578.9 ]
 [ 85.39   7.    29.89 627.62 597.73 585.04]
 [ 68.84   6.    20.65 433.69 413.04 395.42]
 [ 73.56  10.    36.78 772.38 735.6   702.08]
 [ 36.26   2.    3.63  76.15 72.52 69.8 ]
 [ 54.84   3.    8.23 172.75 164.52 156.08]]
```

```
Shape of sales_arr: (96, 6)
```

The first 10 rows of the NumPy array are displayed, as is its shape. From left to right, the columns are:

- Unit price
- Quantity
- Tax price
- Total price
- Revenue
- COGS

- Select the next code cell, and then type the following:

```
1 cols = ['UnitPrice', 'Quantity', 'Tax', 'TotalPrice', 'Revenue', 'COGS']
2 sales_df = pd.DataFrame(sales_arr, columns = cols)
3 sales_df
```

The cols list is being provided to label the columns in the DataFrame. The labels are in order from left to right.

- Run the code cell.

g) Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
0	74.69	7.0	26.14	548.97	522.83	500.24
1	15.28	5.0	3.82	80.22	76.40	73.21
2	46.33	7.0	16.22	340.53	324.31	321.12
3	58.22	8.0	23.29	489.05	465.76	430.98
4	86.31	7.0	30.21	634.38	604.17	578.90
...
91	97.16	1.0	4.86	102.02	97.16	92.68
92	87.87	10.0	43.94	922.64	878.70	848.40
93	12.45	6.0	3.74	78.44	74.70	72.06
94	52.75	3.0	7.91	166.16	158.25	152.99
95	82.70	6.0	24.81	521.01	496.20	477.84

96 rows × 6 columns

- A `DataFrame` was created that more closely resembles a table or spreadsheet than the NumPy array on which it was based. It'll be much easier to keep track of the context of each data point thanks to these labels.
- The column labels are included at the top of the `DataFrame`, in bold.
- As with the `Series`, the rows indices are to the left of the table data.



Note: As with NumPy, pandas has no `Decimal` data type. For demonstration purposes, the prices will be kept as floats, but in a real-world situation, you should consider a workaround.

h) Select the next code cell, and then type the following:

```
1 print('DataFrame shape:  {}'.format(sales_df.shape))
2 print('DataFrame size:   {}'.format(sales_df.size))
3 print('DataFrame indices: {}'.format(sales_df.index))
```

- i) Run the code cell.
j) Examine the output.

```
DataFrame shape:  (96, 6)
DataFrame size:   576
DataFrame indices: RangeIndex(start=0, stop=96, step=1)
```

- The shape of the `DataFrame` matches the NumPy array on which it was based. There are 96 rows and 6 columns.
- There are 576 total data points in the `DataFrame` (96×6).
- Once again, you haven't yet defined a custom row index labeling scheme, so the `DataFrame` uses the default `RangeIndex` format.

k) Select the next code cell, and then type the following:

```
1 sales_df.dtypes
```

- l) Run the code cell.

- m) Examine the output.

```
UnitPrice      float64
Quantity       float64
Tax            float64
TotalPrice     float64
Revenue        float64
COGS           float64
dtype: object
```

- Each column label is printed, along with its data type. In a `DataFrame`, each column holds a single data type, but a `DataFrame` can be made up of multiple columns with different data types.
- In this case, all columns are floats.
- The output is actually a `Series` object, where each column from the `DataFrame` is a row index, and the data type is the data point.

5. Get a quick look at `DataFrame` records.

- Scroll down and view the cell titled **Get a quick look at DataFrame records**, then select the code cell below it.
- In the code cell, type the following:

```
1 sales_df.head(10)
```

- Run the code cell.
- Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
0	74.69	7.0	26.14	548.97	522.83	500.24
1	15.28	5.0	3.82	80.22	76.40	73.21
2	46.33	7.0	16.22	340.53	324.31	321.12
3	58.22	8.0	23.29	489.05	465.76	430.98
4	86.31	7.0	30.21	634.38	604.17	578.90
5	85.39	7.0	29.89	627.62	597.73	585.04
6	68.84	6.0	20.65	433.69	413.04	395.42
7	73.56	10.0	36.78	772.38	735.60	702.08
8	36.26	2.0	3.63	76.15	72.52	69.80
9	54.84	3.0	8.23	172.75	164.52	156.08

The first 10 records in the `DataFrame` are printed. This function is handy for checking the structure of your data, as well as for seeing an example of its values.

- Select the next code cell, and then type the following:

```
1 sales_df.tail(10)
```

- Run the code cell.

- g) Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
86	47.38	4.0	9.48	199.00	189.52	182.38
87	44.86	10.0	22.43	471.03	448.60	427.73
88	21.98	7.0	7.69	161.55	153.86	149.21
89	64.36	9.0	28.96	608.20	579.24	561.34
90	89.75	1.0	4.49	94.24	89.75	84.76
91	97.16	1.0	4.86	102.02	97.16	92.68
92	87.87	10.0	43.94	922.64	878.70	848.40
93	12.45	6.0	3.74	78.44	74.70	72.06
94	52.75	3.0	7.91	166.16	158.25	152.99
95	82.70	6.0	24.81	521.01	496.20	477.84

The last 10 records in the DataFrame are printed.

6. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel→Shutdown**.
- b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
- c) Close the **Creating DataFrames** tab in Firefox, but keep a tab open to **DSTIP/pandas/** in the file hierarchy.

TOPIC B

Load and Save pandas Data

Just as with NumPy, you'll probably have an external source of data you'll want to load into pandas. Likewise, you may want to share pandas structures through files.

Loading Data into a DataFrame

The pandas library provides multiple ways for you to get your existing data into a `DataFrame`. Like with a NumPy array, you could load the data manually using standard Python, but this is slow and tedious. Instead, you should leverage the functions that pandas provides to not only load the data into your Python environment, but also to convert that data into a `DataFrame`-compatible format.

Both binary and text file types are supported by pandas. Depending on how these files are formatted, pandas will automatically populate `DataFrame` attributes like row names and column names. This can save you a lot of trouble, but you can also tweak how pandas parses the dataset if the defaults are not to your liking.

The `pandas.read_csv()` Function

The `pandas.read_csv()` function is a common way to load data into a `DataFrame`. It parses comma-separated value (CSV) files. As the name suggests, each value in this type of text file is delineated by a comma so that the parser knows when one value ends and another begins. Because of their simple formatting, CSV files are a popular way to store public datasets. Let's say you have a `colors.csv` file with the following contents:

```
,RGB,Hex
Red,0,00
Green,157,9D
Blue,220,DC
```

Notice how the file includes column labels in the first row, as well as row labels in the first column. These will come into play when you load the file.

The `read_csv()` function has many arguments that you can leverage to tweak the parsing process, but the only required argument is `filepath_or_buffer`, which is where you'll provide the location of your CSV file as a string. The same rules for absolute vs. relative paths apply here as they did with NumPy. For example:

```
colors_df = pandas.read_csv('colors.csv', index_col = 0)
```

That's it—the data is loaded into a `DataFrame`. Look what happens when you print the `DataFrame`:

```
>>> colors_df
      RGB Hex
Red    0   00
Green  157  9D
Blue   220  DC
```

The parser automatically took the first row of the CSV file and made those the column names. The `index_col` argument specifies what column has the values you want to use as row labels. If you don't use this argument, the row labels will start at 0 and count up.

Additional Parameters

Aside from `filepath_or_buffer` and `index_col`, here are a few other parameters you might leverage when loading a CSV file:

- `dtype` —Specify the data type to use.

- `comments` —Specify what character(s) are considered the start of comments; the information on these comment lines will not be loaded.
- `header` —Specify which row includes the column labels, instead of the default (row 0).
- `skiprows` —Specify the first `n` rows to skip when loading the file.
- `usecols` —Specify which column(s) to read from the file.
- `sep` —Specify the delimiter to use instead of the default (comma).
- `squeeze` —Set to `True` to convert a single-column file to a `Series` instead of a `DataFrame`.

The `pandas.read_html()` Function

Another type of text file you might want to load into a `DataFrame` is an HTML file. Scraping web pages can reveal data that might be useful in a data science project. Python has excellent HTML parsing libraries, so pandas can easily take an HTML-formatted table and convert it into a `DataFrame`. For example, the file `colors.html` includes the same data as the CSV file, but stores that data in an HTML `<table>` element:

```
<!DOCTYPE html>
<html>
<body>

<table>
    <tr>
        <th></th>
        <th>RGB</th>
        <th>Hex</th>
    </tr>
    <tr>
        <td>Red</td>
        <td>0</td>
        <td>00</td>
    </tr>
    <tr>
        <td>Green</td>
        <td>157</td>
        <td>9D</td>
    </tr>
    <tr>
        <td>Blue</td>
        <td>220</td>
        <td>DC</td>
    </tr>
</table>

</body>
</html>
```

To read this file, you can supply the path and the column index for the row labels:

```
>>> colors = pandas.read_html('colors.html', index_col = 0)
>>> colors
[   RGB  Hex
 Red      0  00
 Green    157  9D
 Blue     220  DC]
```

This function actually returns a list of `DataFrame` objects, since each table is parsed as its own `DataFrame`. In this case, there's only one table, so you can just index the first list item to get the `DataFrame`:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

```
>>> colors_df = colors[0]
>>> colors_df
   RGB Hex
Red    0   00
Green  157  9D
Blue   220  DC
```

Additional Parameters

The `io` parameter supports more than just a file path; you can also supply it with a URL of a live web page. Some additional parameters include:

- `header` —Specify which row includes the column labels instead of the default (row 0).
- `skiprows` —Specify the first n rows to skip when loading the file.
- `flavor` —Specify the parsing engine to use.
- `attrs` —Specify the particular tables to parse based on their attributes.

Additional Loading Functions

There are many functions provided by pandas for loading data into a `DataFrame`. The following table describes several of the most common.

<i>Function</i>	<i>Description</i>
<code>pandas.read_table()</code>	Essentially the same as <code>read_csv()</code> , except <code>sep</code> is set to <code>\t</code> (tab) by default.
<code>pandas.read_fwf()</code>	Reads files that have fixed column widths instead of having columns delimited by some character. You must supply the column widths as an argument.
<code>pandas.read_clipboard()</code>	Reads text from the operating system's clipboard and then calls <code>read_csv()</code> with a <code>sep</code> of <code>s+</code> (one or more whitespace characters).
<code>pandas.read_excel()</code>	Reads files that are in a Microsoft Excel-compatible format. Older formats like XLS are supported, as are newer XML-based formats like XLSX and ODF. You can read from an entire workbook or from specific worksheets.
<code>pandas.read_json()</code>	Reads files in the JavaScript Object Notation (JSON) format, which is common for storing data in many different environments.
<code>pandas.read_sql()</code>	Reads Structured Query Language (SQL) databases or queries.
<code>pandas.read_pickle()</code>	Reads files saved in Python's pickle format.

Saving DataFrame Data

Unsurprisingly, pandas gives you plenty of ways to save `DataFrame` objects as actual files. You can store these files for later use, or share them with other members of the project. The fact that so many different formats are supported gives you the flexibility to capture data in whatever format(s) are best for your environment. For example, if you have sales records in an SQL database, you can load that data into a `DataFrame`, operate on the data as needed, then save that data in an SQL format so that it can be stored using the same database infrastructure.

The DataFrame.to_csv() Function

The `DataFrame.to_csv()` function takes a `DataFrame` object and converts it to a text file where each value is separated by commas (or whatever delimiter you specify). In other words, this is the reverse of `pandas.read_csv()`. So, let's say you have a `DataFrame` object named `colors_df`:

	RGB	Hex
Red	0	00
Green	157	9D
Blue	220	DC

To save this as a CSV file, supply the `path_or_buf` argument with a location:

```
colors_df.to_csv('colors.csv')
```

Opening the file reveals the following contents:

```
,RGB,Hex
Red,0,00
Green,157,9D
Blue,220,DC
```

Additional Parameters

Some additional parameters include:

- `columns` —Specify which columns to write.
- `header` —Specify whether or not to write column names.
- `index` —Specify whether or not to write row names.
- `sep` —Specify the delimiter to use instead of the default (comma).
- `compression` —Specify which file compression method to use.

The DataFrame.to_html() Function

Saving a `DataFrame` as an HTML file with table markup is done through the `DataFrame.to_html()` function. The following example assumes you still have the same `colors_df` object, and pass in the desired file location as the `buf` argument:

```
colors_df.to_html('colors.html')
```

If you opened this file in a web browser, it would render as a simple table. Opening the file in a text editor, you can see the markup it produced:

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>RGB</th>
      <th>Hex</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>Red</th>
      <td>0</td>
      <td>00</td>
    </tr>
    <tr>
      <th>Green</th>
      <td>157</td>
      <td>9D</td>
    </tr>
  </tbody>
</table>
```

```

<tr>
    <th>Blue</th>
    <td>220</td>
    <td>DC</td>
</tr>
</tbody>
</table>

```

The function formatted the table with a few attributes to make it more visually pleasing. You can alter some of this behavior through parameters, however.

Additional Parameters

Some additional parameters include:

- `columns` —Specify which columns to write.
- `header` —Specify whether or not to write column names.
- `index` —Specify whether or not to write row names.
- `justify` —Specify how to justify column labels.
- `bold_rows` —Specify whether or not to bold rows on output.
- `classes` —Specify Cascading Style Sheets (CSS) classes to use for formatting the output.
- `escape` —Specify whether or not to convert certain characters to a format that is safer for HTML.
- `border` —Specify the border width of the table.

Additional Saving Functions

Most of the formats that pandas supports for loading data are also supported for saving data. The following table describes the equivalent saving functions.

Function	Description
<code>pandas.to_clipboard()</code>	Writes a DataFrame to the operating system's clipboard using <code>to_csv()</code> with a <code>sep</code> of <code>\t</code> (tab).
<code>pandas.to_excel()</code>	Writes a DataFrame to a single-worksheet XLSX file.
<code>pandas.to_json()</code>	Writes a DataFrame to JSON format.
<code>pandas.to_sql()</code>	Writes a DataFrame to an SQL database.
<code>pandas.to_pickle()</code>	Writes a DataFrame to a Python pickle file.



Note: As of pandas 1.0.0, there are no `to_table()` or `to_fwf()` functions.

Guidelines for Loading and Saving pandas Data

Follow these guidelines when you are loading and saving pandas data.

Load and Save pandas Data

When loading and saving pandas data:

- Use `read_csv()` to load data from a text file.
- Use `read_html()` to load data in HTML tables.
- Consider using other load functions that are appropriate for loading specific sources and formats of data.
- Use `to_csv()` to save pandas data in a text file.

- Use `to_html()` to save pandas data in HTML markup.
- Consider using other save functions that are appropriate for saving data in a specific format.

Do Not Duplicate or Distribute

ACTIVITY 4–2

Loading and Saving DataFrame Data

Data Files

/home/student/DSTIP/pandas/Loading and Saving DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've been given more data that was generated from in-store transactions. The data is in a CSV file format, and it contains records of new transactions, as well as additional attributes for each transaction. You'll load these new records and attributes into a DataFrame that you can build on later. This will eventually become the main data object that will hold the store transaction records.

In addition, one of your junior team members is new to programming and is more comfortable analyzing and manipulating data in Microsoft Excel. So, you'll save a copy of the DataFrame as an Excel file for this team member.

1. Examine the data file.

- From the Linux desktop, open the file manager.
- Navigate to `/home/student/DSTIP/pandas/data`.
- Double-click `stores_data.csv` to open it in a text editor.
- Observe how the data is structured in this file.
 - This is an expanded version of the store transaction data you've been working with thus far.
 - There are several more columns of data for each transaction, which you'll learn about soon.
 - The column heading labels are in the CSV file itself. When you load the file, pandas will be able to use this metadata to label the columns in the DataFrame.
- Close the file when you're done.

2. Switch back to Jupyter Notebook, then open the `DSTIP/pandas/Loading and Saving DataFrames.ipynb` file.

3. Import the relevant software libraries.

- View the cell titled **Import software libraries**, and examine the code listing below it.
- Run the code cell.
- Verify that the library versions are returned.

4. Load a CSV file as a DataFrame.

- Scroll down and view the cell titled **Load a CSV file as a DataFrame**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 stores_df = pd.read_csv('data/stores_data.csv', index_col = 0)
2 stores_df.head()
```

The `index_col = 0` argument tells pandas to use the first column as the row index labels.

- c) Run the code cell.
d) Examine the output.

InvoiceID	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Date	Revenue	COGS
OLI-CLO-005	Olinger	Member	Male	Clothing	48.71	1	2.44	51.15	3/26/2019	48.71	45.96
OLI-CLO-006	Olinger	Normal	Male	Clothing	78.55	9	35.35	742.30	3/1/2019	706.95	690.66
OLI-ELE-009	Olinger	Normal	Female	Electronics	23.07	9	10.38	218.01	2/1/2019	207.63	200.86
CAR-FBV-006	Carbon Creek	Normal	Male	Food and beverages	58.26	6	17.48	367.04	3/28/2019	349.56	336.31
GRC-HBE-006	Greene City	Normal	Male	Health and beauty	30.35	7	10.62	223.07	3/19/2019	212.45	206.67

All of the columns from the CSV file were pulled in as column labels for the `DataFrame`, *except* `InvoiceID`, which was pulled in as the name of the row indices. The columns are as follows:

- `City` —The city the store branch is in.
- `CustomerType` —Customers marked as `Member` are part of the GCE rewards club. `Normal` customers are not.
- `Gender` —The gender of the customer who made the purchase.
- `ProductLine` —The category of product that the customer purchased.
- `UnitPrice`, `Quantity`, `Tax`, and `TotalPrice` —The same sales figures you worked with earlier.
- `Date` —The date of the purchase.
- `Revenue` and `COGS` —Same as before.

- e) Select the next code cell, and then type the following:

```
1 print('DataFrame shape: {}'.format(stores_df.shape))
2 print('DataFrame size: {}'.format(stores_df.size))
```

- f) Run the code cell.
g) Examine the output.

```
DataFrame shape: (200, 11)
DataFrame size: 2200
```

- The `DataFrame` is 200 rows by 11 columns.
- There are 2,200 items in the `DataFrame`.

- h) Select the next code cell, and then type the following:

```
1 stores_df.dtypes
```

- i) Run the code cell.

- j) Examine the output.

```

City          object
CustomerType  object
Gender         object
ProductLine   object
UnitPrice     float64
Quantity       int64
Tax            float64
TotalPrice    float64
Date           object
Revenue        float64
COGS           float64
dtype: object

```

- This demonstrates one of the strengths of a `DataFrame`—its ability to hold multiple data types, where there is one type per column.
- Columns like `City` and `ProductLine` are object-based data types because they are represented as strings, rather than numbers.
- Columns like `UnitPrice` and `Revenue` are floats, as these include prices with two decimal places.
- `Quantity` was cast as an integer because it contains only whole numbers.

- k) Select the next code cell, and then type the following:

```
1 stores_df.index
```

- l) Run the code cell.
m) Examine the output.

```

Index(['OLI-CLO-005', 'OLI-CLO-006', 'OLI-ELE-009', 'CAR-FBV-006',
       'GRC-HBE-006', 'CAR-ELE-006', 'OLI-CLO-007', 'CAR-STR-009',
       'OLI-FBV-008', 'OLI-ELE-010',
       ...
       'GRC-ELE-014', 'OLI-ELE-017', 'CAR-ELE-016', 'CAR-FBV-015',
       'GRC-HBE-014', 'OLI-ELE-018', 'OLI-ELE-019', 'CAR-HML-025',
       'CAR-HML-026', 'OLI-HML-014'],
      dtype='object', name='InvoiceID', length=200)

```

Instead of the default `RangeIndex`, the `DataFrame` was created with a row index of custom labels. In this case, the row index labels (and the overall name) are pulled from the first column of the CSV file (`InvoiceID`). Each transaction has its own invoice ID, so that acts as a type of "key" that you can use to refer to a row (rather than a number between 0 and 200).

5. Save the DataFrame as a Microsoft Excel file.

- a) Scroll down and view the cell titled **Save the DataFrame as a Microsoft Excel file**, and then select the code cell below it.

- b) In the code cell, type the following:

```
1 stores_df.to_excel('stores_data.xlsx', sheet_name = 'Stores Data',
2                     freeze_panes = (1, 1))
3 print('DataFrame has been saved.')
```

- Lines 1 and 2 use the `to_excel()` method on the `DataFrame` to save the file in a format compatible with Microsoft Excel.
 - In particular, the file is being saved as an XLSX spreadsheet, which is part of the Office Open XML format. This type of file can be read by many spreadsheet programs, not just Excel.
 - The `sheet_name` argument gives the worksheet the data is saved to a name.
 - The `freeze_panes` argument specifies where to start "freezing" rows and/or columns of data so that they stay on screen as you scroll. The position `(1, 1)` tells pandas to freeze the first row and the first column.
- c) Run the code cell.
d) Examine the output.

DataFrame has been saved.

6. Examine the file you saved.

- a) From the top panel on the desktop, select the **data - File Manager** tab to open it.
- b) Navigate back one directory so you're in `/home/student/DSTIP/pandas/`.
- c) Double-click `stores_data.xlsx`.
The file automatically opens in LibreOffice Calc, an open-source alternative to Microsoft Excel.
- d) Observe how the data is structured in this file.
 - The data was given table-style formatting, with column names and row indices in bold.
 - There are borders separating the frozen row and column panes.
- e) Scroll up and down, as well as left and right, to verify that the relevant panes are frozen.
- f) Close the file when you're done.

7. Shut down this Jupyter Notebook kernel.

- a) Return to Jupyter Notebook.
- b) From the menu, select **Kernel->Shutdown**.
- c) In the **Shutdown kernel?** dialog box, select **Shutdown**.
- d) Close the **Loading and Saving DataFrames** tab in Firefox, but keep a tab open to `DSTIP/pandas/` in the file hierarchy.

TOPIC C

Analyze Data in DataFrames

Your analysis of data can now begin in earnest. Just like NumPy, there are many ways to approach analysis of a pandas DataFrame.

DataFrame Indexing

Indexing a DataFrame is somewhat similar to indexing a NumPy array, but there are key differences. Let's say you have a dataset of student grades that you've imported into a DataFrame. A truncated version is as follows:

	Math	English	Science	History
Parker	A	B	A-	C+
Baldwin	C	D	A-	B-
Duncan	A	A-	F	B+
Cain	C	C	B+	B

If this were a NumPy array, and you wanted to retrieve Duncan's English grade, you would probably index like so:

```
grades[2, 2]
```

If you try this on a DataFrame, however, it won't work. You need a different method of accessing specific items. The pandas library offers several, in fact.

Indexing Operator

The simplest way to index a DataFrame is by directly referencing the name of the column that holds the data you're looking for. The indexing operator syntax still uses the familiar brackets, but with a name instead of an integer location:

```
grades['Math']
```

This references the Math column, so that entire column is returned:

```
Parker      A
Baldwin    C
Duncan     A
Cain       C
Name: Math, dtype: object
```

If this output format looks familiar, it's because indexing a DataFrame in this way actually returns a Series object. The row labels are preserved, as you would expect in a Series.

You can also supply multiple column labels in a list—a form of fancy indexing—to return a DataFrame:

```
>>> grades[['Math', 'English']]
   Math English
Parker      A      B
Baldwin    C      D
Duncan     A     A-
Cain       C      C
```

There is a limitation to this indexing method, however. If you wanted to return just a single value instead of a whole column, you might think to follow the pattern and do something like:

```
grades['Baldwin', 'Math']
```

But this won't work; Python will return an error. You also can't index this way by providing the row name only. Thankfully, pandas provides other ways to index data.

Attribute Syntax

If your column names are strings, you can also index a DataFrame like you would retrieve an attribute. For example:

```
>>> grades.Math
Parker      A
Baldwin     C
Duncan      A
Cain        C
Name: Math, dtype: object
```

Note that this will not work if your column names aren't strings, or if your column names happen to be the same as existing attributes (e.g., `grades.values`).

The DataFrame.loc() Function

The `DataFrame.loc()` function is an alternative indexing method that enables you to access groups of values in rows or columns, or as single values. In particular, it supports label-based inputs. The following example uses `loc()` to return the Baldwin row:

```
>>> grades.loc['Baldwin']
Math      C
English   D
Science   A-
History  B-
Name: Baldwin, dtype: object
```

The result is a `Series`, but the data has pivoted from the indexing operator example—The DataFrame column labels are now row labels, and the values that were in a row are now in a column.

You can also index multiple rows and return a DataFrame by supplying the row labels as a list:

```
>>> grades.loc[['Baldwin', 'Duncan']]
          Math English Science History
Baldwin    C      D     A-     B-
Duncan    A      A-      F     B+
```

To get even more granular, you can select specific rows *and* columns by supplying each axis as a separate list:

```
>>> grades.loc[['Baldwin', 'Duncan'], ['Math', 'English']]
          Math English
Baldwin    C      D
Duncan    A      A-
```

And, finally, to get as granular as possible, supply both the row label and column label of the value you're looking for in a single list:

```
>>> grades.loc['Baldwin', 'Math']
'C'
```

The DataFrame.iloc() Function

The `DataFrame.iloc()` function is essentially the same as `loc()`, except that it takes integer-based indices as input instead of labels. Although labeling is one of pandas' strengths, you may prefer to use standard integer-based indexing in some cases. The following example returns a single row as a `Series`:

```
>>> grades.iloc[1]
Math          C
English        D
Science        A-
History        B-
Name: Baldwin, dtype: object
```

The following example returns multiple rows as a DataFrame:

```
>>> grades.iloc[[1, 2]]
   Math English Science History
Baldwin      C       D     A-      B-
Duncan      A       A-      F      B+
```

To select specific rows and columns:

```
>>> grades.iloc[[1, 2], [0, 1]]
   Math English
Baldwin      C       D
Duncan      A       A-
```

And, finally, to select a specific value:

```
>>> grades.iloc[1, 0]
'C'
```

The DataFrame.idxmin() and DataFrame.idxmax() Functions

The `DataFrame.idxmin()` and `DataFrame.idxmax()` functions return the index of the smallest and largest values in an axis of a DataFrame, respectively. If the smallest or largest value occurs multiple times, the index where it occurs first is returned. For this function to work, the values in the DataFrame must be of a numeric data type. In the following example, `grades_num` is just the DataFrame from before, but its values have been converted to integers (out of 100):

	Math	English	Science	History
Parker	94	83	91	79
Baldwin	74	65	91	82
Duncan	96	90	58	89
Cain	75	74	89	85

Now, let's say you want to return the names of the students who scored the lowest in each subject:

```
>>> grades_num.idxmin()
Math      Baldwin
English    Baldwin
Science    Duncan
History    Parker
```

The result is a Series where the column labels (academic subject) are now row labels, and the row labels (student names) are now column values.

If you want to return the students who scored the highest in each subject, use `idxmax()`:

```
>>> grades_num.idxmax()
Math      Duncan
English    Duncan
Science    Parker
History    Duncan
```

By default, both functions operate on axis 0 (rows). You can use 1 for the `axis` argument to operate on columns:

```
>>> grades_num.idxmax(1)
Parker      Math
Baldwin    Science
```

Duncan	Math
Cain	Science

So, this example returned each student as the row label, and each subject as the column value. This Series shows which subject the student had the highest score in.



Note: By default, both functions skip missing values, but you can change this behavior by specifying `False` for the `skipna` argument.

Reindexing

Reindexing is the process of changing the order and/or labels of `DataFrame` indices. Recall that NumPy has functions like `sort()` that rearrange values in an array, but they do not maintain the relationship between the cells across an axis. Reindexing a `DataFrame`, however, does.

Looking at either `DataFrame` that holds students' grades, you can see that the students are not ordered alphabetically. If you wanted to change this—or reorder the rows for any other reason—you can use the `DataFrame.reindex()` function. This function takes a list, where each item is an index label placed in the desired order:

```
new_index = ['Baldwin', 'Cain', 'Duncan', 'Parker']
grades_num.reindex(new_index)
```

The result of this reindexing maintains each student's grades for each subject:

	Math	English	Science	History
Baldwin	74	65	91	82
Cain	75	74	89	85
Duncan	96	90	58	89
Parker	94	83	91	79

Note that if your new index list is missing a label, the resulting `DataFrame` simply drops that row. Also, if your new index list adds a label that doesn't already exist, the resulting `DataFrame` shows `NaN` (null) for that label's values by default. You can change this behavior by providing a value for the `fill_value` function.

You can also reindex columns by providing 1 for the `axis` argument:

```
>>> new_index = ['English', 'History', 'Math', 'Science']
>>> grades_num.reindex(new_index, axis = 1)
      English   History   Math   Science
Parker        83        79       94       91
Baldwin       65        82       74       91
Duncan        90        89       96       58
Cain          74        85       75       89
```



Note: `DataFrame.reindex()` returns a copy of the `DataFrame`, not a view.

The `DataFrame.corr()` Function

You've learned some of the fundamental indexing operations in pandas to help you select subsets of data, but like a NumPy array, you'll also want to generate some statistical information about your `DataFrame`. For instance, the `DataFrame.corr()` function applies a correlation metric to each column. By default, the function uses the Pearson correlation coefficient (PCC) to determine the linear correlation between each column and every other column. The PCC returns a value between -1 and 1. For example:

```
>>> grades_num.corr()
      Math   English   Science   History
Math    1.000000  0.930591 -0.609921  0.142964
```

```
English 0.930591 1.000000 -0.735211 0.409431
Science -0.609921 -0.735211 1.000000 -0.846808
History 0.142964 0.409431 -0.846808 1.000000
```

The result is a DataFrame. PCC values closer to 1 indicate a strong positive correlation; PCC values close to -1 indicate a strong negative correlation; and PCC values close to 0 indicate a weak correlation. So, from this particular dataset, you can draw a few conclusions:

- You can ignore the left-to-right diagonal, as this is just comparing a column with itself. A column will always have a perfect correlation with itself.
- The correlation between Math and English is around 0.93. Since this is close to 1, it indicates a strong positive correlation. In other words, students who score well in Math class are likely to score well in English class; and vice versa—students scoring poorly in one will probably score poorly in the other.
- The correlation between Science and History is around -0.85 . Since this is close to -1 , it indicates a strong negative correlation. In other words, students who score well in Science class are likely to score poorly in History class; and vice versa—students scoring poorly in Science will probably score well in History.
- The correlation between History and Math is around 0.14. Since this is close to 0, it indicates a weak correlation. In other words, there is no apparent relationship between students' performance in History class and Math class.



Note: This dataset is artificially generated and should not be used to draw any realistic conclusions.

The DataFrame.describe() Function

The `DataFrame.describe()` function gives you a dashboard view into the statistics of your DataFrame. It returns a DataFrame containing the following descriptive statistics for each column:

- Number of rows
- Mean
- Standard deviation
- Minimum and maximum values
- 25th, 50th, and 75th percentile (or as specified in the `percentiles` argument)

For example:

```
>>> grades_num.describe().round(2)
      Math   English   Science   History
count  4.00      4.00      4.00      4.00
mean  84.75     78.00     82.25     83.75
std   11.87     10.86     16.19      4.27
min   74.00     65.00     58.00     79.00
25%  74.75     71.75     81.25     81.25
50%  84.50     78.50     90.00     83.50
75%  94.50     84.75     91.00     86.00
max   96.00     90.00     91.00     89.00
```

If your DataFrame has non-numeric types (e.g., strings), the statistics displayed are:

- Number of rows
- Number of unique values
- Most common value
- Frequency of most common value

For example:

```
>>> grades.describe()
      Math   English   Science   History
```

```

count      4        4        4        4
unique     2        4        3        4
top         C        C       A-
freq        2        1        2        1

```

Mixed Data Types

If your DataFrame has both numeric and non-numeric data types, the `describe()` function, by default, returns only numeric statistics where relevant. You can force it to return both numeric and non-numeric statistics by supplying `include = 'all'` as an argument.

DataFrame Statistical Summary Functions

The `describe()` function is nice for getting a quick overview of your data, but you may want to return individual statistics instead of an entire DataFrame full of them. Also, there are plenty of statistical measures that aren't included in the `describe()` output. The following table lists some of the most common of these statistical functions.

Each function has, at minimum, an `axis` argument. The default value (0) tells the function to summarize by rows, while 1 tells it to summarize by columns. The results are placed in a Series. The example outputs in the table work with the `grades_num` DataFrame defined earlier.

Function	Description and Example
<code>DataFrame.min()</code>	Returns the lowest value in the axis. <pre>>>> grades_num.min() Math 74 English 65 Science 58 History 79</pre>
<code>DataFrame.max()</code>	Returns the highest value in the axis. <pre>>>> grades_num['English'].max() 90</pre>
<code>DataFrame.mean()</code>	Returns the arithmetic mean of values in the axis. <pre>>>> grades_num['Math'].mean() 84.75</pre>
<code>DataFrame.median()</code>	Returns the median of values in the axis. <pre>>>> grades_num['History'].median() 83.5</pre>
<code>DataFrame.mode()</code>	Returns the mode of values in the axis. <pre>>>> grades_num['Science'].mode() 0 91</pre>
<code>DataFrame.std()</code>	Returns the standard deviation of values in the axis. <pre>>>> grades_num['English'].std() 10.8627</pre>
<code>DataFrame.var()</code>	Returns the variance of values in the axis. <pre>>>> grades_num['Math'].var() 140.9166</pre>
<code>DataFrame.sum()</code>	Returns the sum of all values in the axis. <pre>>>> grades_num['History'].sum() 335</pre>

SciPy and statsmodels Integration

As with NumPy arrays, both SciPy and statsmodels support operations on a pandas DataFrame. These libraries can go above and beyond the statistical functions provided by pandas. For example, the following code calculates the geometric mean of the Math column by using SciPy:

```
>>> from scipy import stats
>>> stats.gmean(grades_num['Math']).round(2)
84.12
```

Summarizing an Entire DataFrame

The `describe()` function and all of the other summary functions listed in this section return statistics for each column. If you want to summarize an entire DataFrame, you need to flatten it into a Series first:

```
>>> grades_num_series = pandas.Series(grades_num.values.ravel())
>>> grades_num_series.max()
96
```

The DataFrame.apply() Function

The `DataFrame.apply()` function applies an existing function to all values in a DataFrame along a certain axis. You would typically use this to perform some calculation on the data that isn't natively supported by pandas. For example, pandas doesn't have a function to calculate geometric mean, but SciPy does. You could call the SciPy function directly on the DataFrame, but this would return a NumPy array of the values:

```
>>> stats.gmean(grades_num).round(2)
array([84.12, 77.42, 80.86, 83.67])
```

Instead, you could pass this SciPy function to `DataFrame.apply()`, which will use it to operate over all values in the DataFrame. In this case, it returns a Series:

```
>>> grades_num.apply(stats.gmean).round(2)
Math      84.12
English   77.42
Science   80.86
History   83.67
```

The `apply()` function is also useful when you've defined your own custom functions. For example, the following function performs a bunch of arbitrary arithmetic:

```
def my_func(data, user_choice):
    return ((data - 100 / 21 + 8 * 6 ** 2) * user_choice).round()
```

When you apply this to your DataFrame, you can specify any positional arguments to your custom function as a tuple to `args`. In this case, 10 is being passed as `user_choice`:

```
>>> grades_num.apply(my_func, args = (10,))
   Math  English  Science  History
Parker    3772.0    3662.0    3742.0    3622.0
Baldwin   3572.0    3482.0    3742.0    3652.0
Duncan    3792.0    3732.0    3412.0    3722.0
Cain      3582.0    3572.0    3722.0    3682.0
```

Missing Data

The pandas library is particularly good at helping you manage missing data. After all, datasets—especially large ones—are rarely in a completely pristine state when you start a project. You'll likely encounter data values that are corrupt, incorrect, or simply missing. Being able to identify and handle these values is an important step of the data cleaning process.

It's important to understand how NumPy handles missing values, as NumPy is the foundation pandas is built on. NumPy leverages Python's `None` type object as one of its missing value types. If a NumPy array has at least one value equal to `None`, the entire array is converted to a Python object type. This means you cannot perform most mathematical functions on the data because it is considered non-numeric:

```
>>> numpy.array([1, 2, None]).dtype
dtype('O')
```

The other way NumPy handles missing data is by leveraging its own `NaN` specification. This is actually a float, so you can still preserve numeric data:

```
>>> numpy.array([1, 2, numpy.nan]).dtype
dtype('float64')
```

However, any attempt at operating on this data will always produce a `NaN` value unless you use specific functions that ignore `NaN` values:

```
>>> x = numpy.array([1, 2, numpy.nan])
>>> x.max()
nan
>>> numpy.nanmax(x)
2.0
```

In pandas, both `None` and `NaN` types are consolidated into `NaN`:

```
>>> df = pandas.DataFrame([[1, None], [2, numpy.nan]])
>>> df
   0    1
0  1  NaN
1  2  NaN
```

By default, most pandas functions ignore `NaN` values:

```
>>> df.max()
0    2.0
1    NaN
dtype: float64
```

`NaN` values are usually not explicitly defined when you pull the dataset into pandas. For example, in a CSV file, if a particular cell is empty (i.e., there's a comma delimiter, but no value between those commas), then pandas converts that to `NaN`.

The NA Value

The `NA` value is an alternative missing-data value that is currently in an "experimental" state as of pandas 1.0.0. This value is meant to be more flexible than `NaN`, as it can be cast to any data type, rather than being limited to a float.

The DataFrame.isna() and DataFrame.notna() Functions

Although pandas can handle missing values gracefully, you might still want to handle them differently. For example, it's common to perform some calculation on the dataset that reveals a pattern (like the mean) and then fill in missing values with that estimate. But, before you can do this, you need to identify if any `NaN` values exist and where they are located. This is what `DataFrame.isna()` does:

```
>>> df = pandas.DataFrame([[1, None], [2, numpy.nan]])
>>> df.isna()
   0    1
0  False  True
1  False  True
```

As you can see, the function returned a DataFrame of Booleans, where each value has been replaced by either `True` or `False`. If the value is `NaN`, it is replaced with `True`; if not, it is replaced with `False`.

The opposite function is `DataFrame.notna()`, which returns `True` when a value is some type other than `NaN`, and `False` when a value is `NaN`:

```
>>> df.notna()
      0      1
0  True  False
1  True  False
```

The DataFrame.any() Function

The `DataFrame.any()` function returns whether or not each column has at least one `True` value. You can use this in a variety of ways, including with a DataFrame of Booleans generated by the `isna()` or `notna()` functions:

```
>>> df.isna().any()
0    False
1     True
```

In this case, column 0 has no missing values, but column 1 does.

Guidelines for Analyzing Data in DataFrames

Follow these guidelines when you are analyzing data in DataFrames.

Analyze Data in DataFrames

When analyzing data in DataFrames:

- Use the indexing operator to index a DataFrame by column labels.
- Use `loc()` to index a DataFrame by row and/or column labels.
- Use `iloc()` to index a DataFrame by row and/or column integer indices.
- Use `idxmin()` and `idxmax()` to return the indices of the smallest and largest values along an axis.
- Use reindexing to change the order of row and/or column labels.
- Use `corr()` to generate a correlation matrix for all numeric variables.
- Use `describe()` to retrieve summary statistics about numeric or categorical variables.
- Use individual summary functions to get more statistical information about a DataFrame.
- Use `apply()` to apply an existing function (e.g., a custom function) to all values in a DataFrame.
- Be aware that pandas consolidates missing data into a single type, `NaN`.
 - This is a float value.
 - NA is currently experimental.
- Use `isna()` and `notna()` to identify missing values.

ACTIVITY 4–3

Analyzing Data in DataFrames

Data Files

/home/student/DSTIP/pandas/Analyzing DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data_more.csv

Before You Begin

Jupyter Notebook is open.

Scenario

More store transaction data has come in, and you're ready to begin the analysis process in earnest. The DataFrame has become too large for you to examine it all at once, so you'll need to retrieve what you're looking for through indexing. You'll examine specific rows and columns to get a better sense of how each transaction and its attributes differ from one another. You'll also get more granular and extract specific values that might be of interest. While you're indexing, it might be beneficial to move the Date column to the beginning of the DataFrame, as it's more common to place date values in a transactional record up front, rather than toward the middle or end of the dataset.

Aside from extracting certain points of data, you'll also apply statistical summary functions to the data to get a better sense of the data's spread, central tendency, and more. For example, you want to identify the transaction that led to the highest revenue for GCE—learning more about the other attributes of this transaction might help the company focus its marketing and operational efforts. You also suspect that the dataset might have arrived with a few missing values, so you'll try to identify them if they exist.

1. In Jupyter Notebook, open the **DSTIP/pandas/Analyzing DataFrames.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 On lines 12 and 13, the **stores_data_more.csv** file is being loaded. This is the same file as before, but with more rows of transactions added.
 - b) Run the code cell.
3. Index DataFrame columns.
 - a) Scroll down and view the cell titled **Index DataFrame columns**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 print('DataFrame shape: {}'.format(stores_df.shape))
2 stores_df.head()
```

 - c) Run the code cell.

- d) Examine the output.

DataFrame shape: (900, 11)											
InvoiceID	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Date	Revenue	COGS
OLI-CLO-005	Olinger	Member	Male	Clothing	48.71	1.0	2.44	51.15	3/26/2019	48.71	45.96
OLI-CLO-006	Olinger	Normal	Male	Clothing	78.55	9.0	35.35	742.30	3/1/2019	706.95	690.66
OLI-ELE-009	Olinger	Normal	Female	Electronics	23.07	9.0	10.38	218.01	2/1/2019	207.63	200.86
CAR-FBV-006	Carbon Creek	Normal	Male	Food and beverages	58.26	6.0	17.48	367.04	3/28/2019	349.56	336.31
GRC-HBE-006	Greene City	Normal	Male	Health and beauty	30.35	7.0	10.62	223.07	3/19/2019	212.45	206.67

More data has been added to the dataset, and there are now 900 rows (transactions).

- e) Select the next code cell, and then type the following:

```
1 stores_df['UnitPrice'].head()
```

This uses the indexing operator to index a single column—`UnitPrice`.

- f) Run the code cell.
g) Examine the output.

```
InvoiceID
OLI-CLO-005    48.71
OLI-CLO-006    78.55
OLI-ELE-009    23.07
CAR-FBV-006    58.26
GRC-HBE-006    30.35
Name: UnitPrice, dtype: float64
```

- A `Series` object is returned.
 - `InvoiceID` is still being used for the row indices.
 - The data in the `Series` corresponds to each value in `UnitPrice` that matches the appropriate index.
 - The unit prices of the first five transactions are displayed.
- h) Select the next code cell, and then type the following:

```
1 ind_prices = ['UnitPrice', 'Quantity', 'Tax', 'TotalPrice']
2 stores_df[ind_prices].head()
```

This code supplies a list of column names to use in fancy indexing.

- i) Run the code cell.

- j) Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice
InvoiceID				
OLI-CLO-005	48.71	1.0	2.44	51.15
OLI-CLO-006	78.55	9.0	35.35	742.30
OLI-ELE-009	23.07	9.0	10.38	218.01
CAR-FBV-006	58.26	6.0	17.48	367.04
GRC-HBE-006	30.35	7.0	10.62	223.07

- A DataFrame was returned.
- All of the columns specified in the fancy indexing operation were returned.
- The values in each of these columns were returned for the first five transactions.

4. Index DataFrame rows and columns.

- Scroll down and view the cell titled **Index DataFrame rows and columns**, then select the code cell below it.
- In the code cell, type the following:

```
1 stores_df.loc['CAR-HML-032']
```

This code supplies a specific row index label to the `loc()` function.

- Run the code cell.
- Examine the output.

City	Carbon Creek
CustomerType	Normal
Gender	Female
ProductLine	Home and lifestyle
UnitPrice	56.53
Quantity	4
Tax	11.31
TotalPrice	237.43
Date	3/4/2019
Revenue	226.12
COGS	220.85
Name:	CAR-HML-032, dtype: object

- This Series contains all of the values for a specific transaction, where each attribute is now a row index, and the data corresponds to the value for that attribute in the specific transaction.
 - Indexing by row can help you retrieve the values for one or more records.
- Select the next code cell, and then type the following:

```
1 stores_df.loc[['CAR-HML-032', 'CAR-HML-033']]
```

This uses fancy indexing to return multiple rows.

- Run the code cell.

- g) Examine the output.

InvoiceID	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Date	Revenue	COGS
CAR-HML-032	Carbon Creek	Normal	Female	Home and lifestyle	56.53	4.0	11.31	237.43	3/4/2019	226.12	220.85
CAR-HML-033	Carbon Creek	Normal	Male	Home and lifestyle	50.93	8.0	20.37	427.81	3/22/2019	407.44	392.65

A DataFrame was returned; this is essentially a truncated version of the entire stores_df DataFrame, showing only the two rows specified.

- h) Select the next code cell, and then type the following:

```
1 stores_df.loc[['CAR-HML-032', 'CAR-HML-033'], ['UnitPrice', 'Quantity']]
```

This time, the code indexes by both rows and columns. This is done through a list of lists, where the indices to the left of the comma are the rows, and the indices to the right of the comma are columns.

- i) Run the code cell.
j) Examine the output.

InvoiceID	UnitPrice	Quantity
CAR-HML-032	56.53	4.0
CAR-HML-033	50.93	8.0

- Only the two rows and the two columns specified are returned.
 - You may want to index both rows and columns in order to focus on the data you're looking for, while leaving out irrelevant data.
- k) Select the next code cell, and then type the following:

```
1 stores_df.loc['CAR-HML-032', 'Revenue']
```

This provides a single row and a single column as an index.

- l) Run the code cell.
m) Examine the output.

```
226.12
```

- The revenue for the specific transaction is returned.
- This type of granular indexing can help you narrow your focus even further by retrieving a single value, rather than a Series or DataFrame.

5. Reindex the DataFrame.

- a) Scroll down and view the cell titled **Reindex the DataFrame**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 new_index = ['Date', 'City', 'CustomerType', 'Gender',
2               'ProductLine', 'UnitPrice', 'Quantity',
3               'Tax', 'TotalPrice', 'Revenue', 'COGS']
4 stores_df = stores_df.reindex(new_index, axis = 1)
5 stores_df.head()
```

Line 4 rearranges the columns according to the order of `new_index`. The only difference is that `Date` has been moved to the beginning.

- c) Run the code cell.
d) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
OLI-CLO-005	3/26/2019	Olinger	Member	Male	Clothing	48.71	1.0	2.44	51.15	48.71	45.96
OLI-CLO-006	3/1/2019	Olinger	Normal	Male	Clothing	78.55	9.0	35.35	742.30	706.95	690.66
OLI-ELE-009	2/1/2019	Olinger	Normal	Female	Electronics	23.07	9.0	10.38	218.01	207.63	200.86
CAR-FBV-006	3/28/2019	Carbon Creek	Normal	Male	Food and beverages	58.26	6.0	17.48	367.04	349.56	336.31
GRC-HBE-006	3/19/2019	Greene City	Normal	Male	Health and beauty	30.35	7.0	10.62	223.07	212.45	206.67

The `Date` column is now first. This can make it easier to analyze a transaction in the context of time.

6. If you wanted to index the `DataFrame` by numerical indices instead of labels, how would you retrieve the value in the fifth row, third column?

- `stores_df.loc[4, 2]`
- `stores_df.iloc[4, 2]`
- `stores_df[4, 2]`
- `stores_df[[4, 2]]`

7. Summarize statistics about the store data.

- a) Scroll down and view the cell titled **Summarize statistics about the store data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 stores_df.describe().round(2)
```

- c) Run the code cell.

- d) Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
count	900.00	898.00	900.00	899.00	896.00	896.00
mean	55.31	5.47	15.15	318.18	303.45	291.77
std	26.44	2.94	11.66	245.01	233.29	224.26
min	10.08	-1.00	0.51	10.68	10.17	9.75
25%	32.42	3.00	5.83	122.50	118.50	113.04
50%	54.53	5.00	11.96	250.71	239.88	229.57
75%	77.48	8.00	21.99	461.87	439.71	424.52
max	99.96	10.00	49.65	1042.65	993.00	955.90

The `describe()` function printed many common summary statistics about each numeric column of the data. Notice how the non-numeric columns were left out by default. Some potential observations include:

- The `count` for each column isn't the same, which suggests that there are some missing values.
- The `mean` price paid in total is \$318.18.
- The `mean` quantity of each transaction is 5.47.
- The `min` quantity is -1.00, which indicates that there is an error in the data somewhere.
- The `max` quantity is 10.00.

- e) Select the next code cell, and then type the following:

```
1 mode = int(stores_df['Quantity'].mode())
2 print('Most frequent item quantity: {}'.format(mode))
```

- The `describe()` function doesn't return the mode, which may be more beneficial than the mean for assessing the typical item quantity of a purchase.
- Line 1 uses indexing to take the mode of just the `Quantity` column.
- The value of `mode` will also be converted to an integer to make it easier to read.

- f) Run the code cell.

- g) Examine the output.

Most frequent item quantity: 10.

- h) Select the next code cell, and then type the following:

```
1 categorical_cols = ['City', 'CustomerType', 'Gender', 'ProductLine']
2 stores_df[categorical_cols].describe()
```

- Line 1 creates a list of all the columns holding non-numeric categorical values.
- Line 2 calls `describe()` to get summary statistics about these variables.

- i) Run the code cell.

- j) Examine the output.

	City	CustomerType	Gender	ProductLine
count	900	900	898	900
unique	3	2	2	6
top	Carbon Creek	Member	Female	Clothing
freq	305	451	451	166

Summary statistics that are more relevant to categorical variables were returned. Some potential observations include:

- The `Gender` column is missing two values.
- There are three different cities and six different product lines (`unique`).
- Members are more common than non-members; more shoppers buy from Carbon Creek than the other branches; most shoppers are female; and more shoppers buy clothing than any other product line (`top`).
- The `freq` stat indicates the total count of each value in `top`.

8. Retrieve information about the highest revenue purchase.

- a) Scroll down and view the cell titled **Retrieve information about the highest revenue purchase**, then select the code cell below it.
- b) In the code cell, type the following:

```

1 high_inv = stores_df['Revenue'].idxmax()
2 high_row = stores_df.loc[high_inv, stores_df.columns]
3
4 print('Invoice {} on {} led to the highest revenue: ${:.2f}.' \
5       .format(high_inv, high_row['Date'], high_row['Revenue']))
6 print('{} {} items were purchased.' \
7       .format(int(high_row['Quantity']), high_row['ProductLine']))

```

- Line 1 uses `idxmax()` to retrieve the row index that has the highest value in the `Revenue` column.
 - Line 2 constructs a new `DataFrame` with just the row index identified in line 1, but all columns.
 - Lines 4 through 7 print values from this new `DataFrame`.
- c) Run the code cell.
- d) Examine the output.

```

Invoice OLI-CL0-023 on 2/15/2019 led to the highest revenue: $993.00.
10 Clothing items were purchased.

```

By combining a summary statistic with indexing, you extracted specific information about a record of note.

9. Identify correlations between the numeric variables.

- a) Scroll down and view the cell titled **Identify correlations between the numeric variables**, then select the code cell below it.
- b) In the code cell, type the following:

```
1 stores_df.corr().round(3)
```

- c) Run the code cell.

- d) Examine the output.

	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
UnitPrice	1.000	-0.001	0.630	0.630	0.630	0.629
Quantity	-0.001	1.000	0.697	0.697	0.696	0.696
Tax	0.630	0.697	1.000	1.000	1.000	1.000
TotalPrice	0.630	0.697	1.000	1.000	1.000	1.000
Revenue	0.630	0.696	1.000	1.000	1.000	1.000
COGS	0.629	0.696	1.000	1.000	1.000	1.000

A correlation matrix of all numeric variables was returned. Some potential observations include:

- `Quantity` does not appear to correlate with `UnitPrice` (-0.00). In the real world, you could intuitively conclude that the larger a price is per item, the more likely someone is to buy fewer of that item. However, this dataset (which was artificially generated) does not support that conclusion.
- `Quantity` has a strong positive correlation with `TotalPrice` (0.70). This makes sense—as you buy more of something, the total price tends to increase. However, since there are many different unit prices (which, along with quantity, is part of the total price calculation), the correlation is not 100%.
- `Tax`, on the other hand, correlates perfectly with `TotalPrice` (1.00). If the tax price rises, the total price will rise necessarily.
- Similar conclusions apply to the other price columns like `Revenue` and `COGS`.

10. Identify missing values.

- a) Scroll down and view the cell titled **Identify missing values**, then select the code cell below it.
 b) In the code cell, type the following:

```
1 stores_df.isna().head()
```

- c) Run the code cell.
 d) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
OLI-CLO-005	False	False		False	False	False	False	False	False	False	False
OLI-CLO-006	False	False		False	False	False	False	False	False	False	False
OLI-ELE-009	False	False		False	False	False	False	False	False	False	False
CAR-FBV-006	False	False		False	False	False	False	False	False	False	False
GRC-HBE-006	False	False		False	False	False	False	False	False	False	False

A DataFrame of Booleans was returned. Every `NaN` value is `True`, whereas every other value is `False`. Because of the size of the DataFrame, it's not immediately clear from the first five rows how many `NaN` values there are or where they're located.

- e) Select the next code cell, and then type the following:

```
1 stores_df.isna().any()
```

- f) Run the code cell.

- g) Examine the output.

```
Date      False
City     False
CustomerType  False
Gender    True
ProductLine  False
UnitPrice  False
Quantity   True
Tax       False
TotalPrice  True
Revenue   True
COGS     True
dtype: bool
```

The output shows which columns have NaN values (True) and which columns do not (False).

- h) Select the next code cell, and then type the following:

```
1 stores_df.isna().sum()
```

- i) Run the code cell.
j) Examine the output.

```
Date      0
City      0
CustomerType  0
Gender    2
ProductLine  0
UnitPrice  0
Quantity   2
Tax       0
TotalPrice  1
Revenue   4
COGS     4
dtype: int64
```

The output shows the total number of NaN values in each column.

11.What data type will a column adopt when any of its values are NaN?

- Boolean
- String object
- Integer
- Float

12.Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel→Shutdown**.
- b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
- c) Close the **Analyzing DataFrames** tab in Firefox, but keep a tab open to **DSTIP/pandas/** in the file hierarchy.

TOPIC D

Slice and Filter Data in DataFrames

The ability to filter DataFrame data by using indexing and slicing is worth special attention, as it can help focus your analysis efforts on the most relevant subsets of data.

DataFrame Slicing with the Indexing Operator

The syntax for slicing a DataFrame is essentially the same as it is for a NumPy array—you separate the beginning and end of the slice by a colon. You can slice by using the indexing operator, the `loc()` function, or the `iloc()` function. So, this means you can slice by axis label or by the integer-style index. The following example returns to `grades_num` and uses the indexing operator to slice by row labels:

```
>>> grades_num['Baldwin':'Cain']
      Math  English  Science  History
Baldwin     74       65      91      82
Duncan      96       90      58      89
Cain        75       74      89      85
```

Recall that indexing using the indexing operator requires a column label, not a row label. With slicing, however, you supply row labels (or integer indices).

One caveat when you are using the indexing operator is that the end value of the slice can be inclusive or exclusive, depending on how you do the slice. The previous example was inclusive; the `Cain` row was included. However, consider what happens when you do the same slice, only replacing labels with integer indices:

```
>>> grades_num[1:3]
      Math  English  Science  History
Baldwin     74       65      91      82
Duncan      96       90      58      89
```

The `Cain` row doesn't appear because this slice is exclusive. So, to recap:

- Slicing by label is inclusive.
- Slicing by integer index is exclusive.

DataFrame Slicing with Functions

The `DataFrame.loc()` and `DataFrame.iloc()` methods are alternative ways of slicing a DataFrame, just like how they're alternatives to indexing. One advantage of using these functions instead of the indexing operator is that each function supports either inclusive (`loc()`) or exclusive (`iloc()`) slicing behavior, not both. So, you're less likely to make mistakes. For example:

```
>>> grades_num.loc['Baldwin':'Cain'] # Inclusive
      Math  English  Science  History
Baldwin     74       65      91      82
Duncan      96       90      58      89
Cain        75       74      89      85
>>> grades_num.iloc[1:3] # Exclusive
      Math  English  Science  History
Baldwin     74       65      91      82
Duncan      96       90      58      89
```

The other advantage is that these functions enable you to slice by both row and column. So, if you want to show the scores for Baldwin, Duncan, and Cain, but only for Math, English, and Science:

```
>>> grades_num.loc['Baldwin':'Cain', 'Math':'Science']
   Math  English  Science
Baldwin      74       65       91
Duncan      96       90       58
Cain        75       74       89
```

You can accomplish the same thing by using an integer-style index:

```
grades_num.iloc[1:4, :3]
```

DataFrame Comparison Functions and Operators

The DataFrame module includes comparison functions that are similar to those in NumPy. And, like NumPy, each function has a wrapper operator you can use to simplify the syntax. Instead of returning an array of Booleans, each of these functions returns a DataFrame of Booleans.

The following table lists each function and its operator. You can specify the axis that the operation will work on; otherwise, it will default to working on columns. The example outputs in the table work with a sliced version of `grades_num` for the sake of brevity:

```
sliced = grades_num.loc['Baldwin':'Duncan', 'Math':'English']
```

<i>Function</i>	<i>Wrapper Operator and Example</i>
<code>DataFrame.eq()</code>	Operator: <code>==</code> <pre>>>> sliced.eq(90) >>> sliced == 90 Math English Baldwin False False Duncan False True</pre>
<code>DataFrame.ne()</code>	Operator: <code>!=</code> <pre>>>> sliced.ne(90) >>> sliced != 90 Math English Baldwin True True Duncan True False</pre>
<code>DataFrame.lt()</code>	Operator: <code><</code> <pre>>>> sliced.lt(90) >>> sliced < 90 Math English Baldwin True True Duncan False False</pre>
<code>DataFrame.le()</code>	Operator: <code><=</code> <pre>>>> sliced.le(90) >>> sliced <= 90 Math English Baldwin True True Duncan False True</pre>
<code>DataFrame.gt()</code>	Operator: <code>></code> <pre>>>> sliced.gt(90) >>> sliced > 90 Math English Baldwin False False Duncan True False</pre>

Function	Wrapper Operator and Example									
DataFrame.ge()	<p>Operator: <code>>=</code></p> <pre>>>> sliced.ge(90) >>> sliced >= 90</pre> <table> <thead> <tr> <th></th> <th>Math</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>False</td> <td>False</td> </tr> <tr> <td>Duncan</td> <td>True</td> <td>True</td> </tr> </tbody> </table>		Math	English	Baldwin	False	False	Duncan	True	True
	Math	English								
Baldwin	False	False								
Duncan	True	True								

DataFrame Logical Operators

Unlike NumPy, pandas does not explicitly provide functions for performing logical operations. You must instead use the operator characters themselves. The following table lists each one. The output examples also use the `sliced` object from the previous table.

Operator	Example									
<code>&</code> (AND)	<pre>>>> (sliced > 90) & (sliced < 94)</pre> <table> <thead> <tr> <th></th> <th>Math</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>False</td> <td>False</td> </tr> <tr> <td>Duncan</td> <td>False</td> <td>False</td> </tr> </tbody> </table>		Math	English	Baldwin	False	False	Duncan	False	False
	Math	English								
Baldwin	False	False								
Duncan	False	False								
<code> </code> (OR)	<pre>>>> (sliced > 90) (sliced < 94)</pre> <table> <thead> <tr> <th></th> <th>Math</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>True</td> <td>True</td> </tr> <tr> <td>Duncan</td> <td>True</td> <td>True</td> </tr> </tbody> </table>		Math	English	Baldwin	True	True	Duncan	True	True
	Math	English								
Baldwin	True	True								
Duncan	True	True								
<code>^</code> (XOR)	<pre>>>> (sliced >= 90) ^ (sliced > 93)</pre> <table> <thead> <tr> <th></th> <th>Math</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>False</td> <td>False</td> </tr> <tr> <td>Duncan</td> <td>False</td> <td>True</td> </tr> </tbody> </table>		Math	English	Baldwin	False	False	Duncan	False	True
	Math	English								
Baldwin	False	False								
Duncan	False	True								

DataFrame Filtering with Masks

As mentioned, using comparison and logical operators on a DataFrame produces a DataFrame of Boolean values. Just like with a NumPy array of Booleans, you can use masking techniques to retrieve only the data that matches one or more conditions. In other words, you can filter a DataFrame to your liking.

Let's say you want to return only the students who scored below the mean for Math class. That way, you can identify which students may need additional help. You can start by generating a DataFrame of Booleans for all columns:

```
>>> grades_num < grades_num.mean()
          Math   English   Science   History
Parker    False     False     False      True
Baldwin   True      True     False      True
Duncan   False     False     True     False
Cain      True      True     False     False
```

But, rather than getting the Boolean values, you can apply a mask to an indexing operation to get only the values that return `True`. The mask is simply a bitwise operation. In the following example, the mask indexes just the Math column to get a DataFrame of all students whose Math scores are below the average:

```
>>> grades_num[(grades_num['Math'] < grades_num['Math'].mean())]
          Math   English   Science   History
```

```
Baldwin    74      65      91      82
Cain       75      74      89      85
```

So, Baldwin and Cain have scored below the average in Math class. Let's say you want to return only the Math column, since those are the only scores you're checking at the moment. You can do a bit more indexing to get a Series:

```
>>> grades_num.loc[(grades_num['Math'] < grades_num['Math'].mean()), 'Math']
Baldwin    74
Cain       75
```

Or, to get both Math and English columns in a DataFrame, you can do a bit of slicing:

```
>>> grades_num.loc[(grades_num['Math'] < grades_num['Math'].mean()),
...                  'Math':'English']
   Math  English
Baldwin    74      65
Cain       75      74
```

Ultimately, by combining indexing, slicing, and masking, you can filter your data to reveal exactly what you're looking for.

Guidelines for Slicing and Filtering Data in DataFrames

Follow these guidelines when you are slicing and filtering data in DataFrames.

Slice and Filter Data in DataFrames

When slicing and filtering data in DataFrames:

- When slicing with the indexing operator, keep in mind that it slices by rows and not by columns.
- Use `loc()` and `iloc()` to slice by rows and/or columns.
 - `loc()` slice range is inclusive.
 - `iloc()` slice range is exclusive.
- Use comparison operators and logical operators to evaluate desired conditions as either true or false.
- Use masking to filter and/or change the values that match a condition.

ACTIVITY 4–4

Slicing and Filtering Data in DataFrames

Data Files

/home/student/DSTIP/pandas/Slicing and Filtering DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data_reindex.csv

Before You Begin

Jupyter Notebook is open.

Scenario

Indexing is a good way to dig into the data, but you can only do so much by specifying *where* to look. You need to tell pandas *what* data to return based on the parameters you provide. So, you'll incorporate slicing and conditional logic to filter your data, giving you more insight into the store transactions that have taken place.

There are also a couple of specific tasks you need to accomplish based on some deals that GCE ran for the period of time represented by the dataset:

- GCE is running a deal on clothing in its Carbon Creek branch only, where rewards members who purchase an item of clothing worth \$75 or more will be given a store credit on their next purchase. You want to see how much revenue has been generated by customers who have qualified for this deal.
- Customers who purchase three food, beverage, sporting goods, or travel products will receive a fourth item for half the price. The company wants to gauge the success of this promotion, and in particular, it wants to identify which store has generated the most transactions in which the customer didn't purchase that third item. This will likely influence future deals for that branch.

1. In Jupyter Notebook, open the **DSTIP/pandas/Slicing and Filtering DataFrames.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 On lines 12 and 13, the **stores_data_reindex.csv** file is being loaded. This is the reindexed DataFrame from the previous activity.
 - b) Run the code cell.
3. Slice DataFrame rows and columns.
 - a) Scroll down and view the cell titled **Slice DataFrame rows and columns**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 stores_df[99:104]
```

- When you use the indexing operator, slicing takes row values.
- This code uses integer indices to slice five rows in the range specified.
- c) Run the code cell.

- d) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
OLI-FBV-015	2/17/2019	Olinger	Member	Female	Food and beverages	71.39	5.0	17.85	374.80	356.95	347.79
OLI-STR-011	1/29/2019	Olinger	Member	Female	Sports and travel	19.15	6.0	5.75	120.65	114.90	112.03
GRC-ELE-007	3/15/2019	Greene City	Member	Female	Electronics	57.49	4.0	11.50	241.46	229.96	222.84
OLI-ELE-012	1/14/2019	Olinger	Normal	Male	Electronics	61.41	7.0	21.49	451.36	429.87	421.35
GRC-HBE-009	2/6/2019	Greene City	Member	Male	Health and beauty	25.90	10.0	12.95	271.95	259.00	245.24

- e) Select the next code cell, and then type the following:

```
1 stores_df.iloc[99:104, 5:9]
```

This code uses `iloc()` to slice on both rows (left of the comma) and columns (right of the comma).

- f) Run the code cell.
g) Examine the output.

InvoiceID	UnitPrice	Quantity	Tax	TotalPrice
OLI-FBV-015	71.39	5.0	17.85	374.80
OLI-STR-011	19.15	6.0	5.75	120.65
GRC-ELE-007	57.49	4.0	11.50	241.46
OLI-ELE-012	61.41	7.0	21.49	451.36
GRC-HBE-009	25.90	10.0	12.95	271.95

The same rows are returned, but only the specified columns.

- h) Select the next code cell, and then type the following:

```
1 stores_df.loc['OLI-FBV-015':'GRC-ELE-007', 'City':'ProductLine']
```

This code uses `loc()` to slice by row and column labels. Note that, because the row labels in this dataset aren't sorted in any particular order, slicing by row labels may not be as feasible.

- i) Run the code cell.
j) Examine the output.

InvoiceID	City	CustomerType	Gender	ProductLine
OLI-FBV-015	Olinger	Member	Female	Food and beverages
OLI-STR-011	Olinger	Member	Female	Sports and travel
GRC-ELE-007	Greene City	Member	Female	Electronics

4. Identify which records have missing values.

- a) Scroll down and view the cell titled **Identify which records have missing values**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 rows = stores_df.isna().any(axis = 1)
2 cols = stores_df.isna().any()
3 stores_df.loc[rows, cols]
```

Earlier, you had identified the existence of missing values, but you hadn't yet pinpointed which transactions had those values. That's what this code block does:

- Line 1 retrieves any row that has a `NaN` value in it.
 - Line 2 does the same, but for columns.
 - Line 3 uses the labels of both the missing rows and missing columns to index the `DataFrame`.
- c) Run the code cell.
d) Examine the output.

	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-STR-027	Male	NaN	81.40	77.52	73.26
CAR-HBE-025	Male	1.0	26.25	NaN	NaN
CAR-CLO-015	Male	4.0	156.03	NaN	NaN
GRC-HBE-029	NaN	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	NaN	279.18	264.93
OLI-HBE-038	NaN	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	NaN	NaN
CAR-FBV-054	Male	NaN	279.38	266.08	256.98
CAR-ELE-060	Male	2.0	121.86	NaN	NaN

Only the relevant information is printed—each row that has at least one missing value, and each column that has at least one missing value. In other words, you've filtered the dataset to focus on erroneous data. You'll handle these errors later.

5. Filter the store data to find records matching certain conditions.

- a) Scroll down and view the cell titled **Filter the store data to find records matching certain conditions**, then select the code cell below it.
- b) In the code cell, type the following:

```
1 stores_df['Revenue'] > 800
```

This simple condition checks to see if there are any revenue figures that are above \$800.

- c) Run the code cell.

- d) Examine the output.

```
InvoiceID
OLI-CLO-005    False
OLI-CLO-006    False
OLI-ELE-009    False
CAR-FBV-006    False
GRC-HBE-006    False
...
OLI-HBE-051    False
GRC-HML-050    True
CAR-FBV-058    False
CAR-HML-064    False
CAR-CLO-051    False
Name: Revenue, Length: 900, dtype: bool
```

- A Series of Booleans is returned.
- For each value in `Revenue`, the value is either less than 800 (`False`) or greater than 800 (`True`).
- You can use Boolean structures like this to mask a `DataFrame`, much like how you can mask a NumPy array.

- e) Select the next code cell, and then type the following:

```
1 stores_df[stores_df['Revenue'] > 800]
```

This supplies the condition as an index into the `DataFrame`.

- f) Run the code cell.
g) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
CAR-ELE-006	1/12/2019	Carbon Creek	Member	Male	Electronics	88.67	10.0	44.34	931.04	886.70	847.41
OLI-ELE-010	3/3/2019	Olinger	Member	Male	Electronics	81.97	10.0	40.99	860.69	819.70	797.66
GRC-STR-007	3/9/2019	Greene City	Member	Male	Sports and travel	99.96	9.0	44.98	944.62	899.64	880.06
GRC-STR-008	2/8/2019	Greene City	Normal	Female	Sports and travel	90.28	9.0	40.63	853.15	812.52	784.24
OLI-STR-007	1/23/2019	Olinger	Member	Female	Sports and travel	89.80	10.0	44.90	942.90	898.00	855.14

Every transaction whose revenue exceeds \$800 is printed.

- h) Select the next code cell, and then type the following:

```
1 cond = (stores_df['City'] == 'Carbon Creek') \
2   & (stores_df['Revenue'] > 800)
3
4 stores_df[cond]
```

- Lines 1 and 2 define a more complex condition that uses both comparison and logical operators to filter the data even further.
 - The city in which the transaction took place must be Carbon Creek, and it must have exceeded \$800 in revenue.
 - Line 4 masks the `DataFrame` using this condition.
- i) Run the code cell.

- j) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
CAR-ELE-006	1/12/2019	Carbon Creek	Member	Male	Electronics	88.67	10.0	44.34	931.04	886.70	847.41
CAR-CLO-005	2/8/2019	Carbon Creek	Normal	Male	Clothing	98.98	10.0	49.49	1039.29	989.80	955.90
CAR-HML-034	3/8/2019	Carbon Creek	Member	Female	Home and lifestyle	90.65	10.0	45.33	951.83	906.50	869.76
CAR-HML-036	1/15/2019	Carbon Creek	Member	Female	Home and lifestyle	89.21	9.0	40.14	843.03	802.89	778.28
CAR-STR-034	2/17/2019	Carbon Creek	Normal	Male	Sports and travel	98.09	9.0	44.14	926.95	882.81	840.33
CAR-FBV-028	1/17/2019	Carbon Creek	Normal	Female	Food and beverages	81.21	10.0	40.61	852.71	812.10	782.54
CAR-HML-050	3/20/2019	Carbon Creek	Normal	Male	Home and lifestyle	93.96	9.0	42.28	887.92	845.64	804.60
CAR-FBV-055	2/19/2019	Carbon Creek	Member	Female	Food and beverages	98.66	9.0	44.40	932.34	887.94	852.84
CAR-STR-059	3/14/2019	Carbon Creek	Member	Female	Sports and travel	97.48	9.0	43.87	921.19	877.32	858.61

All transactions matching this condition were printed.

6. Identify purchases that qualify for the clothing deal.

- Scroll down and view the cell titled **Identify purchases that qualify for the clothing deal**, then select the code cell below it.
- In the code cell, type the following:

```

1 cond = (stores_df['City'] == 'Carbon Creek') \
2   & (stores_df['CustomerType'] == 'Member') \
3   & (stores_df['ProductLine'] == 'Clothing') \
4   & (stores_df['UnitPrice'] >= 75)
5
6 stores_df[cond]

```

This condition has four components, all of which must be met:

- The city must be Carbon Creek.
 - The customer must be a rewards club member.
 - The product must be clothing.
 - The price of each clothing item must be greater than or equal to \$75.
- Run the code cell.
 - Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
CAR-CLO-013	1/14/2019	Carbon Creek	Member	Female	Clothing	96.70	5.0	24.18	507.68	483.50	468.58
CAR-CLO-018	1/24/2019	Carbon Creek	Member	Male	Clothing	86.68	8.0	34.67	728.11	693.44	666.82
CAR-CLO-025	1/18/2019	Carbon Creek	Member	Female	Clothing	88.15	3.0	13.22	277.67	264.45	250.19
CAR-CLO-051	2/18/2019	Carbon Creek	Member	Female	Clothing	88.34	7.0	30.92	649.30	618.38	594.50

- Select the next code cell, and then type the following:

```

1 print('Revenue generated for qualifying purchases: ${}.')
2 .format(stores_df[cond]['Revenue'].sum().round(2))

```

The condition indexes `stores_df`, the output of which will be indexed by the `Revenue` column. So, only that column's data for the qualifying transactions will be included in the summation.

- Run the code cell.

- g) Examine the output.

```
Revenue generated for qualifying purchases: $2059.77.
```

7. Find the branch that is underperforming during the food and travel deal.

- a) Scroll down and view the cell titled **Find the branch that is underperforming during the food and travel deal**, then select the code cell below it.
 b) In the code cell, type the following:

```
1 cond = ((stores_df['ProductLine'] == 'Food and beverage') \
2     | (stores_df['ProductLine'] == 'Sports and travel')) \
3     & (stores_df['Quantity'] <= 2)
4
5 cities = stores_df[cond]['City']
6 cities
```

The logic of this condition is a little more complex:

- Lines 1 and 2 specify that the product must either be a food and beverage product *or* a sports and travel product.
 - Line 3 specifies that, if the product type requirement is met, the quantity of products sold must also be two or fewer.
 - Line 5 indexes using the condition, and then indexes that output by city so that only the city is returned.
- c) Run the code cell.
 d) Examine the output.

InvoiceID	
GRC-STR-006	Greene City
CAR-STR-013	Carbon Creek
OLI-STR-010	Olinger
OLI-STR-012	Olinger
CAR-STR-017	Carbon Creek
OLI-STR-013	Olinger
GRC-STR-015	Greene City
CAR-STR-021	Carbon Creek
GRC-STR-020	Greene City

A Series of cities meeting the condition is returned.

- e) Select the next code cell, and then type the following:

```
1 print('Underperforming branch: {}'.format(cities.mode()[0]))
```

This takes the mode of the Series to find out which city is represented the most from the condition.

- f) Run the code cell.
 g) Examine the output.

```
Underperforming branch: Greene City.
```

8. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel→Shutdown**.
 b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

- c) Close the **Slicing and Filtering DataFrames** tab in Firefox, but keep a tab open to **DSTIP/pandas/** in the file hierarchy.
-

Do Not Duplicate or Distribute

Summary

In this lesson, you loaded data into pandas structures like the `Series` and the `DataFrame`, and then analyzed that data. You also saved the data in shareable files. While NumPy still forms the backbone of data in memory, pandas will likely be your go-to library for working with that data.

For your own job, what kinds of data might you load into a `DataFrame`?

What is your preferred way of indexing and slicing a `DataFrame`?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

5

Transforming and Visualizing Data with pandas

Lesson Time: 2 hours, 30 minutes

Lesson Introduction

The pandas library provides many tools for changing data to meet your needs. It also provides basic plotting functionality for the analysis and/or presentation of data. In this lesson, you'll transform and visualize your data in multiple ways.

Lesson Objectives

In this lesson, you will:

- Manipulate data in pandas `DataFrames`.
- Modify data in pandas `DataFrames`.
- Plot data in `DataFrames` using pandas.

TOPIC A

Manipulate Data in DataFrames

As with NumPy arrays, you'll likely need to change the shape and orientation of your DataFrame objects so that they better serve your current purposes.

The DataFrame.copy() Function

The DataFrame.copy() function creates a full copy of a DataFrame in memory, as opposed to returning a view. The pandas library relies on NumPy to decide when to return a view versus a copy. Typically, you can expect a view when you index or slice the DataFrame using the methods discussed earlier. For example, let's say Parker has been caught cheating on his Math test, and you want to nullify his grade by using a new DataFrame object:

```
>>> grades_new = grades_num
>>> grades_new.loc['Parker', 'Math'] = numpy.nan
>>> grades_new.loc['Parker', 'Math']
nan
```

Retrieve the original DataFrame to see that it's been updated as well:

```
>>> grades_num.loc['Parker', 'Math']
nan
```

So, if you want to prevent this from happening, use the copy() function like so:

```
>>> grades_new = grades_num.copy()
>>> grades_new.loc['Parker', 'Math'] = numpy.nan
>>> grades_new.loc['Parker', 'Math']
nan
>>> grades_num.loc['Parker', 'Math']
94
```

The DataFrame.append() Function

You can use the DataFrame.append() function to concatenate one DataFrame with another. It appends the DataFrame being called (other) to the DataFrame doing the calling, by rows. If a column in other does not exist in the caller, that column is added to the caller.

For example, you have a DataFrame with one more student's grades:

```
grades_rivera = pandas.DataFrame([[74, 79, 89, 91, 83]],
                                 index = ['Rivera'],
                                 columns = ['Math', 'English',
                                            'Science', 'History', 'Business'])
```

You need to move that student into the main grade_num dataset. To do so:

```
>>> grades_num.append(grades_rivera, sort = False)
      Math   English   Science   History   Business
Parker     94        83        91        79       NaN
Baldwin    74        65        91        82       NaN
Duncan     96        90        58        89       NaN
Cain       75        74        89        85       NaN
Rivera     74        79        89        91      83.0
```

Notice that Rivera has been added to the bottom of the main DataFrame and that the Business column has also been added. Because none of the other students have scores for this class yet, their

scores are NaN. Also, the `sort` argument tells the function not to sort the columns if they are not aligned; otherwise, the columns will be reordered alphabetically.



Note: The `append()` function creates a copy, not a view.

Appending Data via Indexing

You can also append data by using an indexing function like `loc()` on a non-existent index. Therefore, you could add the Rivera row like so:

```
grades_num.loc['Rivera'] = [74, 79, 89, 91, 83]
```

DataFrame.append() Shortcomings

The `append()` function works great if you want to add each row in the second DataFrame as a new row in the first. But, what if each DataFrame shares some of the same row indices, but different column values? In the following example, a second DataFrame has Art class scores for Parker and Cain, but those students are already in the main `grades_num` DataFrame:

```
>>> grades_pc_art = pandas.DataFrame([[83],
...                                     [92]],
...                                     index = ['Parker', 'Cain'],
...                                     columns = ['Art'])
>>> grades_pc_art
   Art
Parker    83
Cain      92
```

Ideally, adding this data to `grades_num` would result in just one new column with the relevant scores for Parker and Cain. But, if you used `append()`, you would get duplicate rows:

```
>>> grades_num.append(grades_pc_art, sort = False)
   Math  English  Science  History  Business  Art
Parker     94.0     83.0     91.0     79.0      NaN  NaN
Baldwin    74.0     65.0     91.0     82.0      NaN  NaN
Duncan     96.0     90.0     58.0     89.0      NaN  NaN
Cain       75.0     74.0     89.0     85.0      NaN  NaN
Rivera     74.0     79.0     89.0     91.0     83.0  NaN
Parker     NaN       NaN       NaN       NaN      NaN  83.0
Cain       NaN       NaN       NaN       NaN      NaN  92.0
```

A different approach is needed.

The DataFrame.join() Function

The `DataFrame.join()` function can help combine data based on row indices. Any rows that are labeled the same between each DataFrame will be subsumed into the DataFrame making the call. The following example uses the same data as before, but uses `join()` instead of `append()`:

```
>>> grades_num.join(grades_pc_art)
   Math  English  Science  History  Business  Art
Parker     94        83        91        79        NaN  83.0
Baldwin    74        65        91        82        NaN  NaN
Duncan     96        90        58        89        NaN  NaN
Cain       75        74        89        85        NaN  92.0
Rivera     74        79        89        91     83.0  NaN
```

The rows are no longer duplicated, and the correct scores for Art class have been combined with the Parker and Cain rows.

The `join()` function has several arguments that you can use to configure how the join works. Keep in mind that the `DataFrame` making the call is also commonly referred to as the "first" or "left" `DataFrame`, while the `DataFrame` being called is also referred to as the "second" or "right" `DataFrame`.

- `on`—Specify the column or row index name(s) to perform the join on. For example, there might be a key column in the first `DataFrame` whose values match the row indices of the second `DataFrame`. By specifying `on = 'Key'`, the data will be subsumed into the appropriate row.
- `how`—Specify how the join is performed:
 - '`left`' to use the row index of the first `DataFrame`, or the column if specified through `on`. This is the default.
 - '`right`' to use the row index of the second `DataFrame`.
 - '`outer`' to perform an outer join; i.e., a join in which unmatched rows or columns are preserved.
 - '`inner`' to perform an inner join; i.e., a join in which unmatched rows or columns are not preserved.
- `lsuffix` and `rsuffix`—Specify a suffix for the name of any overlapping columns.

Key Example

The following is an example of using the `on` argument with a key value:

```
>>> left
     A    Key
0   a0    k0
1   a2    k1
2   a3    k1
>>> right
     B    C
k0   b0    c0
k1   b1    c1
```

Notice how the values in the `Key` column for `left` match the row indices in `right`. To join on this column:

```
>>> left.join(right, on = 'Key')
     A  Key    B    C
0   a0    k0   b0   c0
1   a2    k1   b1   c1
2   a3    k1   b1   c1
```

The DataFrame.merge() Function

The `DataFrame.merge()` function is similar to `DataFrame.join()`—in fact, both call the underlying `pandas.merge()` function. However, `DataFrame.merge()` combines on columns by default, rather than on row indices. You can alter this behavior to be more like `join()`, however. Essentially, `join()` is meant to be a convenience function to help reduce typing, while `merge()` is considered a general-use function.

The DataFrame.pivot() Function

The `DataFrame.pivot()` function reshapes a `DataFrame` based on column values. If you've ever used a pivot table in Microsoft® Excel®, this function may look familiar. There are three arguments that you use to pivot the data:

- `index`—The unique values of the column you provide here will be transformed into the row indices of the new `DataFrame`.
- `columns`—The unique values of the column you provide here will be transformed into the columns of the new `DataFrame`.

- `values` —The values of the column(s) you provide here will populate the cells of the new DataFrame.

Consider the following DataFrame called `sales_df`:

	Name	Product	Units	Sales
0	Jones	Prod1	150	9804.45
1	Jones	Prod2	46	5425.32
2	Jones	Prod3	176	7834.89
3	Perez	Prod1	112	7320.66
4	Perez	Prod2	57	6722.58
5	Perez	Prod3	143	6365.85

Let's say you want to get a quick look at how many units each sales rep sold for each product. In a much larger dataset, eyeballing this information could be difficult. Pivoting the data makes things easier:

```
>>> sales_df.pivot(index = 'Name', columns = 'Product', values = 'Units')
Product  Prod1  Prod2  Prod3
Name
Jones      150     46    176
Perez      112     57    143
```

Each unique value in the `Name` column became its own row; each unique value in the `Product` column became its own column; and the cell data is based on the values in the `Units` column.



Note: The function will throw an error if there are duplicate `columns` values for the same `index`.

The DataFrame.pivot_table() Function

If there are duplicate values, you can still pivot the data by using the `DataFrame.pivot_table()` function. This applies an aggregation function of your choosing on the duplicate data. The `index`, `columns`, and `values` arguments are the same, with `aggfunc` taking the aggregation function. The default function is `numpy.mean`.

The DataFrame.transpose() Function

The `DataFrame.transpose()` function permutes data by flipping the rows and columns of a DataFrame. This enables you to manipulate the orientation of a DataFrame without fully pivoting it. For example, in `grades_num`, you might want to make each subject a row, and make each student a column. The performance in each subject can be easier to examine in this way.

```
>>> grades_num.transpose()
          Parker  Baldwin  Duncan  Cain  Rivera
Math        94.0     74.0    96.0   75.0    74.0
English     83.0     65.0    90.0   74.0    79.0
Science     91.0     91.0    58.0   89.0    89.0
History     79.0     82.0    89.0   85.0    91.0
Business    NaN       NaN     NaN    NaN    83.0
Art         83.0     NaN     NaN    92.0    NaN
```

The DataFrame.sort_values() Function

There are two main ways of sorting a DataFrame. The first involves sorting by values along a specified axis. The `DataFrame.sort_values()` function can take one or more labels as input to the `by` argument, which determines what rows or columns by which the function will sort. For example, let's say you want to sort the `Math` column by its values:

```
grades_num.sort_values(by = ['Math'])
```

The result is:

	Math	English	Science	History	Business	Art
Baldwin	74	65	91	82	NaN	NaN
Rivera	74	79	89	91	83.0	NaN
Cain	75	74	89	85	NaN	92.0
Parker	94	83	91	79	NaN	83.0
Duncan	96	90	58	89	NaN	NaN

Notice how the values in the `Math` column are now displayed in ascending order. Also, unlike a NumPy array, the relationships between rows and columns was preserved; in this case, the rows were reindexed in order to keep the scores associated with the right student.

To sort on multiple columns, you simply add more labels to the input list:

	Math	English	Science	History	Business	Art
Rivera	74	79	89	91	83.0	NaN
Baldwin	74	65	91	82	NaN	NaN
Cain	75	74	89	85	NaN	92.0
Parker	94	83	91	79	NaN	83.0
Duncan	96	90	58	89	NaN	NaN

Baldwin and Rivera have the same Math score, but the latter has a lower Science score than the former, so their indices have switched.

To sort a row, you need to override the default `axis` value (0) and make it a 1:

	History	English	Art	Science	Math	Business
Parker	79	83	83.0	91	94	NaN
Baldwin	82	65	NaN	91	74	NaN
Duncan	89	90	NaN	58	96	NaN
Cain	85	74	92.0	89	75	NaN
Rivera	91	79	NaN	89	74	83.0

The order of columns changed in order to accommodate the value sorting.

Additional Parameters

Some additional parameters include:

- `ascending` —Specify whether the sort should be ascending (`True`) or descending (`False`).
- `inplace` —Specify whether or not to change the `DataFrame` object in place after the sort.
- `kind` —Specify the sorting algorithm to use.
- `na_position` —Specify whether `NaN` values should be placed first or last.

The `DataFrame.sort_index()` Function

The other way to sort a `DataFrame` is by indices. You can use `DataFrame.sort_index()` to sort either the row or column labels. For example, to sort row labels, specify an `axis` of 0:

	Math	English	Science	History	Business	Art
Baldwin	74	65	91	82	NaN	NaN
Cain	75	74	89	85	NaN	92.0
Duncan	96	90	58	89	NaN	NaN
Parker	94	83	91	79	NaN	83.0
Rivera	74	79	89	91	83.0	NaN

The `DataFrame` has been reindexed to sort students (rows) by ascending alphabetical order.

To sort by column labels in descending order:

```
>>> grades_num.sort_index(axis = 1, ascending = False)
   Science  Math  History  English  Business  Art
Parker      91     94       79      83       NaN  83.0
Baldwin     91     74       82      65       NaN  NaN
Duncan      58     96       89      90       NaN  NaN
Cain         89     75       85      74       NaN  92.0
Rivera      89     74       91      79      83.0  NaN
```

Grouping

In pandas, **grouping** is the process that encompasses the following three steps:

- **Split** up data into groups.
- **Apply** a function to each individual group.
- **Combine** the results into a single output.

The purpose of grouping is to facilitate the analysis and manipulation of data within related portions of the overall dataset. One of the most common types of functions used during the "apply" step is a statistical summary function. For example, consider the `sales_df` DataFrame:

	Name	Product	Units	Sales
0	Jones	Prod1	150	9804.45
1	Jones	Prod2	46	5425.32
2	Jones	Prod3	176	7834.89
3	Perez	Prod1	112	7320.66
4	Perez	Prod2	57	6722.58
5	Perez	Prod3	143	6365.85

You want to quickly be able to get the total sales and units sold for each rep. You could certainly use indexing and slicing, or even some masking, to accomplish this:

```
>>> sales_df.loc[:2, 'Units':'Sales'].sum()
Units      372.00
Sales    23064.66
>>> sales_df.loc[3:5, 'Units':'Sales'].sum()
Units      312.00
Sales    20409.09
```

But grouping is an easier and cleaner way to do this.

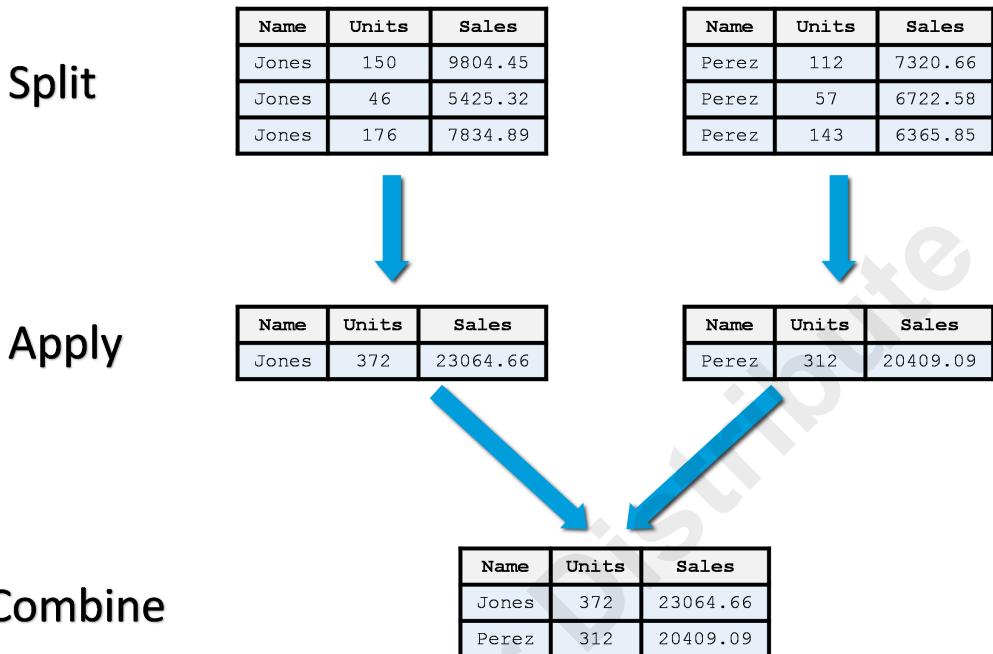


Figure 5–1: The grouping process applied to sales_df.

The DataFrame.groupby() Function

The DataFrame.groupby() function leverages pandas' GroupBy object, which supports the three-step grouping process. It is named after the GROUP BY statement in SQL, as it provides similar functionality. The function essentially takes a key, or a group of keys, which it uses to determine the split. A key is usually a column that includes repeated values that identify a potential group. In the case of sales_df, the Name column is a good key to use, as it includes multiple instances of the same sales rep. In other words, you can group by rep:

```
sales_df.groupby('Name')
```

This returns a GroupBy object that performs the split. In order to actually apply a function and return a combined output, just call the function you want to use on this object. Getting the total figures for sales_df is simple:

```
>>> sales_df.groupby('Name').sum()
      Units      Sales
Name
Jones    372  23064.66
Perez    312  20409.09
```

You now have a complete DataFrame where each rep's figures are totaled. If all you care about are the totals for one or more columns and not others, you can use indexing and slicing on the group. To get just the total units sold:

```
>>> sales_df.groupby('Name')['Units'].sum()
Name
Jones    372
Perez    312
```

Summation is just one example of the power of grouping in pandas, however; there are other types of functions you can apply to data, like mathematical transformation functions and filtering functions.



Note: Grouping and pivoting can both be used to retrieve similar results. However, grouping is a little more flexible when it comes to outputting the results.

Multi-Indexing

The example `DataFrames` you've seen so far have used a single index—e.g., the name of the student in `grades_num`. However, some datasets are more complicated, and instead have a hierarchy of indices. For example, let's say the `grades_num` dataset also keeps track of each students' grades per year. You could add rows to the `DataFrame` like so:

	Math	English	...
Parker	2019	94	83
Parker	2020	92	87
Baldwin	2019	74	65
Baldwin	2020	81	66
	...		

The problem with this approach, other than being a little messy visually, is that it won't be easy to extract all grades from 2019, or all grades from 2020. Those years are, after all, part of the same index as the names. This is where multi-indexing comes into play. In pandas, [multi-indexing](#) gives a `Series` or `DataFrame` multiple row indices at different levels to support easy manipulation and modification of hierarchical datasets. The `MultiIndex` object supports this feature.

To populate a `DataFrame` with multiple indices, you can supply a list of lists to the `index` argument during creation. The first list will be the first row index, the second list will be the second row index, and so on. There's no real limit to the number of index levels that you can include in a multi-indexed `DataFrame`. Here's an example of solving the student grades per year problem with a multi-indexed object:

```
>>> data = numpy.array([[94, 83],
...                     [92, 87],
...                     [74, 65],
...                     [81, 66]])
>>> mult_ind = [['Parker', 'Parker', 'Baldwin', 'Baldwin'],
...               [2019, 2020, 2019, 2020]]
>>> grades_num_yr = pandas.DataFrame(data, index = mult_ind,
...                                      columns = ['Math', 'English'])
>>> grades_num_yr
      Math   English
Parker 2019     94      83
          2020     92      87
Baldwin 2019     74      65
          2020     81      66
```

Now, the `DataFrame` has two indices: one for student name, one for year.



Note: Just as you can create multiple levels of row indices, you can also create multiple levels of columns by using the same list of lists approach.

Level Names

You can label each level of a multi-indexed `DataFrame` to give those levels more meaning. Do this by assigning a list of labels to the `index.names` property for the `DataFrame`:

```
>>> grades_num_yr.index.names = ['Name', 'Year']
>>> grades_num_yr
      Math   English
Name  Year
Parker 2019     94      83
```

2020	92	87
Baldwin	2019	74
		65

Level names are also helpful because they can be passed to some functions as arguments. For example, both of the sorting functions discussed earlier take a `level` argument that you can use to further refine your sort. The following example sorts the DataFrame based on the `Year` index:

```
>>> grades_num_yr.sort_index(level = 'Year')
      Math   English
Name   Year
Baldwin 2019    74      65
Parker  2019    94      83
Baldwin 2020    81      66
Parker  2020    92      87
```

Indexing and Slicing MultiIndex Objects

When you work with multiple indices, you can use the index operator like you normally would on specific columns:

```
>>> grades_num_yr['English']
Parker  2019    83
        2020    87
Baldwin 2019    65
        2020    66
```

In this case, indexing a single column returned a Series with multiple indices.

However, if you want to index and/or slice on rows using a function like `loc()`, you'll need to supply a tuple of each row index as part of the operation. For example, if you want to retrieve all grades for Parker in the year 2020:

```
>>> grades_num_yr.loc[('Parker', 2020), :]
Math      92
English   87
```

Notice how the first index (name) is in the first spot in the tuple, and the second index (year) is in the second spot.

What if you just want to retrieve grades in 2020 for all students? Unfortunately, using the slice operator (`:`) won't work inside a tuple, so you'll need to use a different approach. The `pandas.IndexSlice` module was created to help with this; it essentially takes the place of the tuple. For example:

```
>>> grades_num_yr.loc[pandas.IndexSlice[:, 2020], :]
      Math   English
Name   Year
Parker 2020    92      87
Baldwin 2020    81      66
```

In other words, the `IndexSlice` module is taking the place of the row slice, and its own indexing tells pandas to slice all of the first row index (names) while only slicing 2020 in the second row index (year). Then, the last part of the expression tells pandas to slice on all columns.

Retrieving Level Labels

To retrieve the row labels for a specific level of a multi-indexed DataFrame, you can use the `get_level_values()` function on the indices. The only argument this function takes is the integer position of the level or its name. For example, to retrieve only the year values:

```
>>> grades_num_yr.index.get_level_values('Year')
Int64Index([2019, 2020, 2019, 2020], dtype='int64', name='Year')
```

The DataFrame.stack() and DataFrame.unstack() Functions

The stacking functions in pandas transform a DataFrame from single-indexed to multi-indexed along an axis, and vice versa. For example, let's say the `grades_num_yr` DataFrame looks like this:

	Math		English	
	2019	2020	2019	2020
Parker	94	92	83	87
Baldwin	74	81	65	66

There are multiple columns levels, but reshaping this data to multi-index the rows might be more desirable. Calling the `DataFrame.stack()` function makes this easy. It either returns a Series (if there is just one level of columns) or a DataFrame (if there are multiple levels of columns). For example, you can get something similar to the DataFrame shown earlier:

	Math	English
Baldwin	2019	74
	2020	66
Parker	2019	83
	2020	87

The data remains the same, but it has essentially pivoted along its axes. Like pivoting a single-indexed DataFrame, stacking a multi-indexed DataFrame can make analysis of specific data points easier.

To pivot back to multiple column levels, use the `DataFrame.unstack()` function:

	English		Math	
	2019	2020	2019	2020
Baldwin	65	66	74	81
Parker	83	87	94	92

Both functions automatically sort the index labels.

Guidelines for Manipulating Data in DataFrames



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are manipulating data in DataFrames.

Manipulate Data in DataFrames

When manipulating data in DataFrames:

- Remember that pandas relies on NumPy to decide when to return a view vs. a copy.
- Use the `copy()` function when you want a copy and not a view.
- Use the `append()` function to concatenate DataFrames.
 - Keep in mind this adds new rows by default, even if rows have repeated values.
- Use `join()` to merge rows with similar values.
 - Specify the join approach through the `how` argument.
- Use the `pivot()` function to reshape and summarize data based on column values.
 - Use `pivot_table()` if there are duplicate columns values for the same index.
- Use the `transpose()` function to permute data.
- Use the `sort_values()` function to sort values in a DataFrame.
- Use the `sort_index()` function to sort row or column indices in a DataFrame.
- Use grouping to summarize related portions of the overall dataset.
 - Use the `groupby()` function with the label whose unique values you want to group.
 - Call a statistical summary function on the grouping operation.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

- Consider multi-indexing a DataFrame to make hierarchical datasets easier to work with.
- When indexing or slicing a DataFrame on multiple levels, supply a tuple of the levels as part of the index/slice operation.
 - Use IndexSlice to slice on the levels themselves.
- Use the stack() and unstack() functions to transform a DataFrame from single-indexed to multi-indexed, and vice versa.

Do Not Duplicate or Distribute

ACTIVITY 5-1

Manipulating Data in DataFrames

Data Files

/home/student/DSTIP/pandas/Manipulating DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data_reindex.csv
 /home/student/DSTIP/pandas/data/initial_invoices.csv
 /home/student/DSTIP/pandas/data/ratings_more.csv

Before You Begin

Jupyter Notebook is open.

Scenario

Your analysis of the GCE store data is yielding useful results so far, but the data itself is not yet in an acceptable state. You need to incorporate the initial invoice data you worked on earlier into the larger DataFrame so that all of the transactions are in one place. You also need to add customer ratings for each transaction to fill out the dataset.

Once you're done filling out the dataset, you can begin rearranging the data itself in multiple ways. Sorting the data can make it easier to read and even index, especially since the invoice IDs—the row indices—seem to follow a pattern. You'll also pivot and group the data to extract some insights, such as how each store is performing relative to the others; how the sales of each product line compare; how male and female customers rate their transactions; and more.

1. In Jupyter Notebook, open the **DSTIP/pandas/Manipulating DataFrames.ipynb** file.
2. Import the relevant software libraries, and load the datasets.
 - a) View the cell titled **Import software libraries and load the datasets**, and examine the code listing below it.
 In addition to **stores_data_reindex.csv**, there are two more data files being loaded:
 - **initial_invoices.csv**, which contains the initial 96 invoice records that you worked with using NumPy and eventually imported as a DataFrame. All attributes except CustomerRating are present.
 - **ratings_more.csv**, which contains customer ratings for all 996 invoices.
 - b) Run the code cell.
3. Append the initial invoices to **stores_df**.
 - a) Scroll down and view the cell titled **Append the initial invoices to stores_df**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 initial_df
```

- c) Run the code cell.

- d) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
CAR-HBE-001	1/5/2019	Carbon Creek	Member	Female	Health and beauty	74.69	7	26.14	548.97	522.83	500.24
OLI-ELE-001	3/8/2019	Olinger	Normal	Female	Electronics	15.28	5	3.82	80.22	76.40	73.21
CAR-HML-001	3/3/2019	Carbon Creek	Normal	Male	Home and lifestyle	46.33	7	16.22	340.53	324.31	321.12
CAR-HBE-002	1/27/2019	Carbon Creek	Member	Male	Health and beauty	58.22	8	23.29	489.05	465.76	430.98
CAR-STR-001	2/8/2019	Carbon Creek	Normal	Male	Sports and travel	86.31	7	30.21	634.38	604.17	578.90
...
CAR-ELE-005	3/8/2019	Carbon Creek	Normal	Male	Electronics	97.16	1	4.86	102.02	97.16	92.68
GRC-HBE-005	3/29/2019	Greene City	Normal	Male	Health and beauty	87.87	10	43.94	922.64	878.70	848.40
OLI-ELE-008	2/9/2019	Olinger	Normal	Female	Electronics	12.45	6	3.74	78.44	74.70	72.06
CAR-FBV-005	3/23/2019	Carbon Creek	Normal	Male	Food and beverages	52.75	3	7.91	166.16	158.25	152.99
GRC-HML-006	3/5/2019	Greene City	Normal	Male	Home and lifestyle	82.70	6	24.81	521.01	496.20	477.84

96 rows × 11 columns

The records in this DataFrame are unique. You'll append them to the main DataFrame.

- e) Select the next code cell, and then type the following:

```

1 print('Number of rows BEFORE append: {}'.format(stores_df.shape[0]))
2
3 stores_df = stores_df.append(initial_df, sort = False)
4
5 print('Number of rows AFTER append: {}'.format(stores_df.shape[0]))

```

Line 3 uses `append()` to add the initial invoices to the bottom of the main DataFrame.

- f) Run the code cell.
g) Examine the output.

```

Number of rows BEFORE append: 900.
Number of rows AFTER append: 996.

```

All 96 initial records were added to the main DataFrame.

- h) Select the next code cell, and then type the following:

```

1 stores_df.tail()

```

- i) Run the code cell.
j) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS
CAR-ELE-005	3/8/2019	Carbon Creek	Normal	Male	Electronics	97.16	1.00	4.86	102.02	97.16	92.68
GRC-HBE-005	3/29/2019	Greene City	Normal	Male	Health and beauty	87.87	10.00	43.94	922.64	878.70	848.40
OLI-ELE-008	2/9/2019	Olinger	Normal	Female	Electronics	12.45	6.00	3.74	78.44	74.70	72.06
CAR-FBV-005	3/23/2019	Carbon Creek	Normal	Male	Food and beverages	52.75	3.00	7.91	166.16	158.25	152.99
GRC-HML-006	3/5/2019	Greene City	Normal	Male	Home and lifestyle	82.70	6.00	24.81	521.01	496.20	477.84

This shows some of the initial invoices.

4. Merge the customer ratings into `stores_df`.

- a) Scroll down and view the cell titled **Merge the customer ratings into `stores_df`**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 | ratings_df
```

- c) Run the code cell.
d) Examine the output.

CustomerRating	
InvoiceID	
GRC-FBV-004	4.00
OLI-STR-004	4.00
GRC-HBE-010	4.00
GRC-HBE-016	4.00
GRC-ELE-019	4.00
...	...
OLI-STR-002	10.00
GRC-STR-005	10.00
GRC-STR-011	10.00
CAR-HBE-020	10.00
GRC-ELE-041	10.00

996 rows × 1 columns

- This DataFrame has every invoice ID as an index, along with its associated rating.
- The DataFrame is sorted by rating.

5. Why would indexing a new column for `stores_df` and setting it equal to the `CustomerRating` values in `ratings_df` not be ideal?

6. Why would appending `ratings_df` to `stores_df` not be ideal?

7. Continue merging the customer ratings into `stores_df`.

- a) Select the next code cell, and then type the following:

```
1 | print('Number of rows BEFORE merge: {}'.format(stores_df.shape[0]))
2 |
3 | stores_df = stores_df.join(ratings_df)
4 |
5 | print('Number of rows AFTER merge: {}'.format(stores_df.shape[0]))
```

- Line 3 does a join where `stores_df` is on the "left" and `ratings_df` is on the "right."
 - Since `on` and `how` are not specified, the default behavior is to join on the row indices of the `stores_df` DataFrame.
- b) Run the code cell.

- c) Examine the output.

```
Number of rows BEFORE merge: 996.
Number of rows AFTER merge: 996.
```

As intended, no more rows were added, as the invoice numbers from `ratings_df` already exist in the main DataFrame.

- d) Select the next code cell, and then type the following:

```
1 stores_df.tail()
```

- e) Run the code cell.
f) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS	CustomerRating
CAR-ELE-005	3/8/2019	Carbon Creek	Normal	Male	Electronics	97.16	1.00	4.86	102.02	97.16	92.68	7.20
GRC-HBE-005	3/29/2019	Greene City	Normal	Male	Health and beauty	87.87	10.00	43.94	922.64	878.70	848.40	5.10
OLI-ELE-008	2/9/2019	Olinger	Normal	Female	Electronics	12.45	6.00	3.74	78.44	74.70	72.06	4.10
CAR-FBV-005	3/23/2019	Carbon Creek	Normal	Male	Food and beverages	52.75	3.00	7.91	166.16	158.25	152.99	9.30
GRC-HML-006	3/5/2019	Greene City	Normal	Male	Home and lifestyle	82.70	6.00	24.81	521.01	496.20	477.84	7.40

- The `CustomerRating` column was added.
- Each invoice row has its own customer rating.
- The ratings are assigned to their proper row index (invoice ID), rather than being sorted in ascending order like they were in `ratings_df`.

8. Sort the store data.

- a) Scroll down and view the cell titled **Sort the store data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 stores_df.sort_index(axis = 0, inplace = True)
2 stores_df
```

To begin, this sorts the row indices. Since you're sorting indices instead of values, `sort_index()` is used. The `inplace` argument specifies that the DataFrame will be changed as a result of this sort.

- c) Run the code cell.

- d) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS	CustomerRating
CAR-CLO-001	3/10/2019	Carbon Creek	Normal	Female	Clothing	87.67	2.00	8.77	184.11	175.34	170.12	7.70
CAR-CLO-002	1/12/2019	Carbon Creek	Member	Female	Clothing	20.01	9.00	9.00	189.09	180.09	174.55	5.70
CAR-CLO-003	2/10/2019	Carbon Creek	Member	Female	Clothing	30.14	10.00	15.07	316.47	301.40	284.76	9.20
CAR-CLO-004	1/29/2019	Carbon Creek	Normal	Male	Clothing	83.24	9.00	37.46	786.62	749.16	709.60	7.40
CAR-CLO-005	2/8/2019	Carbon Creek	Normal	Male	Clothing	98.98	10.00	49.49	1,039.29	989.80	955.90	8.70
...
OLI-STR-041	1/1/2019	Olinger	Member	Female	Sports and travel	29.22	6.00	8.77	184.09	175.32	168.27	5.00
OLI-STR-042	1/30/2019	Olinger	Normal	Female	Sports and travel	22.38	1.00	1.12	23.50	22.38	21.53	8.60
OLI-STR-043	3/14/2019	Olinger	Member	Male	Sports and travel	42.85	1.00	2.14	44.99	42.85	41.93	9.30
OLI-STR-044	1/10/2019	Olinger	Normal	Female	Sports and travel	83.14	7.00	29.10	611.08	581.98	569.84	6.60
OLI-STR-045	3/14/2019	Olinger	Member	Female	Sports and travel	35.22	6.00	10.57	221.89	211.32	203.65	6.50

996 rows × 12 columns

The DataFrame is now sorted in order of the invoice IDs.

- e) Select the next code cell, and then type the following:

```
1 stores_df.sort_values(by = ['City', 'ProductLine', 'CustomerType'])
```

Now that the DataFrame is sorted by row indices, you can sort values from the specified columns. However, rather than change the DataFrame in place, you'll just get a temporary look at the sort.

- f) Run the code cell.
g) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS	CustomerRating
CAR-CLO-002	1/12/2019	Carbon Creek	Member	Female	Clothing	20.01	9.00	9.00	189.09	180.09	174.55	5.70
CAR-CLO-003	2/10/2019	Carbon Creek	Member	Female	Clothing	30.14	10.00	15.07	316.47	301.40	284.76	9.20
CAR-CLO-007	1/23/2019	Carbon Creek	Member	Male	Clothing	17.94	5.00	4.49	94.19	89.70	85.88	6.80
CAR-CLO-013	1/14/2019	Carbon Creek	Member	Female	Clothing	96.70	5.00	24.18	507.68	483.50	468.58	7.00
CAR-CLO-014	2/2/2019	Carbon Creek	Member	Male	Clothing	43.13	10.00	21.57	452.87	431.30	419.05	5.50
...
OLI-STR-027	3/2/2019	Olinger	Normal	Female	Sports and travel	73.98	7.00	25.89	543.75	517.86	495.93	4.10
OLI-STR-028	2/17/2019	Olinger	Normal	Female	Sports and travel	46.66	9.00	21.00	440.94	419.94	409.27	5.30
OLI-STR-032	2/21/2019	Olinger	Normal	Female	Sports and travel	98.80	2.00	9.88	207.48	197.60	187.39	7.70
OLI-STR-042	1/30/2019	Olinger	Normal	Female	Sports and travel	22.38	1.00	1.12	23.50	22.38	21.53	8.60
OLI-STR-044	1/10/2019	Olinger	Normal	Female	Sports and travel	83.14	7.00	29.10	611.08	581.98	569.84	6.60

996 rows × 12 columns

- The output is sorted by City, then by ProductLine, then by CustomerType.
- The store branch and the product department are actually incorporated in the invoice ID. The inclusion of CustomerType changes the sort so that members are given priority.

- h) Select the next code cell, and then type the following:

```
1 stores_df.sort_values(by = ['CustomerRating'], ascending = False)
```

- i) Run the code cell.

- j) Examine the output.

InvoiceID	Date	City	CustomerType	Gender	ProductLine	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS	CustomerRating
GRC-STR-005	2/3/2019	Greene City	Member	Female	Sports and travel	55.07	9.00	24.78	520.41	495.63	469.75	10.00
OLI-STR-002	2/15/2019	Olinger	Member	Female	Sports and travel	24.74	3.00	3.71	77.93	74.22	70.11	10.00
GRC-STR-011	3/27/2019	Greene City	Normal	Male	Sports and travel	93.39	6.00	28.02	588.36	560.34	538.89	10.00
CAR-HBE-020	2/20/2019	Carbon Creek	Normal	Female	Health and beauty	32.32	10.00	16.16	339.36	323.20	306.21	10.00
GRC-ELE-041	2/25/2019	Greene City	Normal	Female	Electronics	52.79	10.00	26.40	554.30	527.90	502.63	10.00
...

The DataFrame output has been sorted by customer rating, where the transactions with the highest rating are at the top.

9. Pivot the data so you can analyze it from different perspectives.

- Scroll down and view the cell titled **Pivot the data so you can analyze it from different perspectives**, then select the code cell below it.
- In the code cell, type the following:

```
1 stores_df.pivot_table(index = 'Gender',
2                         columns = 'ProductLine',
3                         values = 'Quantity').round(2)
```

This code block will pivot the DataFrame using:

- Each Gender as a row.
 - Each ProductLine as a column.
 - The mean of Quantity values for each matchup. Mean is the default aggregation function for pivot_table().
- Run the code cell.
 - Examine the output.

ProductLine	Clothing	Electronics	Food and beverages	Health and beauty	Home and lifestyle	Sports and travel
Gender						
Female	5.52	5.81	5.60	5.35	6.26	5.64
Male	4.48	5.62	5.23	5.86	5.10	5.47

- A multi-indexed DataFrame was returned.
 - You can use this pivot to compare the average number of items from each department that each gender purchases.
 - Example observation: In this dataset, female customers seem to have purchased a slightly higher number of electronics items on average.
- Select the next code cell, and then type the following:

```
1 stores_df.pivot_table(index = 'ProductLine',
2                         columns = 'City',
3                         values = 'Revenue',
4                         aggfunc = np.sum)
```

This generates a new pivot, with:

- Each ProductLine as a row.
- Each City as a column.
- The sum of all Revenue for each matchup.

- f) Run the code cell.
- g) Examine the output.

City	Carbon Creek	Greene City	Olinger
ProductLine			
Clothing	15,406.17	15,631.73	20,092.04
Electronics	17,328.81	16,239.47	18,065.69
Food and beverages	16,345.81	14,490.37	22,635.10
Health and beauty	11,972.86	18,567.26	15,793.38
Home and lifestyle	20,626.21	16,713.49	12,711.78
Sports and travel	18,450.19	19,036.38	15,011.36

- You can use this pivot to compare how much revenue each product line generated for each individual store branch.
- Example observation: In this dataset, Olinger's strongest revenue generator was the food and beverages department.

10. Why will using the standard `pivot()` function not work on this `DataFrame`?

11. Use grouping to summarize categories of data.

- a) Scroll down and view the cell titled **Use grouping to summarize categories of data**, then select the code cell below it.
- b) In the code cell, type the following:

```
1 stores_df.groupby('City').sum()
```

- c) Run the code cell.
- d) Examine the output.

City	UnitPrice	Quantity	Tax	TotalPrice	Revenue	COGS	CustomerRating
Carbon Creek	18,553.14	1,842.00	5,021.18	105,440.89	100,130.05	96,226.04	2,383.80
Greene City	18,401.89	1,814.00	5,034.26	105,712.96	100,678.70	96,824.58	2,257.50
Olinger	18,503.35	1,810.00	5,241.72	109,780.01	104,309.35	100,287.37	2,304.10

- The DataFrame was grouped by City, so each unique city became a row.
- The sum of all numeric values for each unique city are returned.
- Taking the sum of customer ratings is perhaps not as useful as taking the mean.
- Non-numeric values were excluded automatically, as `sum()` does not apply to them.
- Example observation: Olinger generated the most revenue out of the three cities, but also the highest COGS.

- e) Select the next code cell, and then type the following:

```
1 stores_df.groupby('City')[['Revenue', 'COGS']].sum()
```

- f) Run the code cell.
g) Examine the output.

	Revenue	COGS
City		
Carbon Creek	100,130.05	96,226.04
Greene City	100,678.70	96,824.58
Olinger	104,309.35	100,287.37

The grouping operation is essentially the same as before, but the result is indexed to focus on only the columns of interest (Revenue and COGS).

- h) Select the next code cell, and then type the following:

```
1 stores_df.groupby('Gender')['CustomerRating'].mean()
```

- i) Run the code cell.
j) Examine the output.

Gender	
Female	6.96
Male	6.97
Name:	CustomerRating, dtype: float64

- The mean of all customer ratings for each gender is returned as a Series.
- Both male and female customers gave similar ratings.

12. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel→Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Manipulating DataFrames** tab in Firefox, but keep a tab open to **DSTIP/pandas/** in the file hierarchy.

TOPIC B

Modify Data in DataFrames

The pandas library is particularly useful for preparing and cleaning data. In this topic, you'll modify your data in various ways so that it's in a better state than when you started.

The DataFrame.rename() Function

Suppose that the labels you assign to rows and columns are incorrect or otherwise need to change. To change these labels, you can use the `DataFrame.rename()` function. There are two ways to use this function, both of which involve supplying a dictionary of the desired changes. First, you can supply a dictionary to the `mapper` argument, where the key is the current label and the value is the new label. Then, you specify the `axis` to apply it to. For example:

```
new_names = {'Math': 'Calculus', 'Science': 'Physics'}
grades_num.rename(mapper = new_names, axis = 1)
```

This results in the two column names being changed:

	Calculus	English	Physics	History	Business	Art
Parker	94	83	91.0	79	NaN	83.0
Baldwin	74	65	91.0	82	NaN	NaN
Duncan	96	90	NaN	89	NaN	NaN
Cain	75	74	89.0	85	NaN	92.0
Rivera	74	79	89.0	91	83.0	NaN

Alternatively, you can feed the dictionary to either the `index` (row labels) or `columns` argument, or both. The following example does the same thing as the previous example:

```
grades_num.rename(columns = new_names)
```



Note: You can also use the `level` argument to rename specific multi-index labels.

The DataFrame.fillna() Function

Recall that the `grades_num` DataFrame at this point has many `NaN` values. In a larger dataset, you could use the `DataFrame.isna()` function to identify null values that aren't easily visible. What if you want to change these null values, though? In this hypothetical case, the `NaN` values represent a group project grade that hasn't been entered into the system yet. For both Business and Art class, students participate in group projects, and, therefore, members of the group receive the same grade. So, you'll adjust the DataFrame accordingly.

The `DataFrame.fillna()` function enables you to quickly replace all `NaN` values with the value you specify, or with a dictionary of values. You can also specify the axis along which to fill these values. The following code replaces all instances of `NaN` in `grades_num` with the appropriate group score for that subject:

```
grades_num.fillna({'Business': 84, 'Art': 91}).astype('int64')
```

In the input dictionary, the key is the column whose `NaN` cells will be replaced, and the value is what they will be replaced with. The `astype()` function ensures that the new scores are cast as integers, rather than inheriting the float property of `NaN`. The result is as follows:

	Calculus	English	Physics	History	Business	Art
Parker	94	83	91	79	84	83
Baldwin	74	65	91	82	84	91
Duncan	96	90	58	89	84	91

Cain	75	74	89	85	84	92
Rivera	74	79	89	91	83	91

The DataFrame.where() Function

The `DataFrame.where()` function replaces values in a `DataFrame` when a given condition is false. There are two main arguments to consider: `cond`, which is the condition you want to evaluate; and `other`, which is the value that any cells not matching `cond` will be changed to. For example, let's say you want to change all failing grades (under 60) to a 60. You can do the following:

```
grades_num.where(grades_num > 59, 60)
```

Because the `where()` function applies the new value only when the condition is `False`, the `grades_num > 59` condition means that any values 60 or higher will be left as is. All values less than 60 will be changed. Here's the result:

	Calculus	English	Physics	History	Business	Art
Parker	94	83	91	79	84	83
Baldwin	74	65	91	82	84	91
Duncan	96	90	60	89	84	91
Cain	75	74	89	85	84	92
Rivera	74	79	89	91	83	91

Duncan's Physics score, which used to be 58, has been changed to 60.

You could, of course, accomplish the same thing by using a mask:

```
grades_num[grades_num < 60] = 60
```

So, it ultimately comes down to preference. Note that the mask changes data in place, but the `where()` function does not.

DataFrame Arithmetic Functions and Operators

The following table lists some common arithmetic functions you can use to modify data in a `DataFrame`. Each function takes an `other` argument to use in the operation, which can either be a scalar, sequence, or pandas object like another `DataFrame`. You can optionally specify an `axis` to operate on. Each function also has an equivalent wrapper operator.

The example outputs in the table work with a sliced version of `grades_num` for the sake of brevity:
`sliced = grades_num.loc['Baldwin':'Duncan', 'Calculus':'English']`

Function	Wrapper Operator and Example									
<code>DataFrame.add()</code>	Operator: + <pre>>>> sliced.add(3) >>> sliced + 3</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>77</td> <td>68</td> </tr> <tr> <td>Duncan</td> <td>99</td> <td>93</td> </tr> </tbody> </table>		Calculus	English	Baldwin	77	68	Duncan	99	93
	Calculus	English								
Baldwin	77	68								
Duncan	99	93								
<code>DataFrame.sub()</code>	Operator: - <pre>>>> sliced.sub(3) >>> sliced - 3</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>71</td> <td>62</td> </tr> <tr> <td>Duncan</td> <td>93</td> <td>87</td> </tr> </tbody> </table>		Calculus	English	Baldwin	71	62	Duncan	93	87
	Calculus	English								
Baldwin	71	62								
Duncan	93	87								

Function	Wrapper Operator and Example									
<code>DataFrame.mul()</code>	Operator: * <pre>>>> sliced.mul(2) >>> sliced * 2</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>148</td> <td>130</td> </tr> <tr> <td>Duncan</td> <td>192</td> <td>180</td> </tr> </tbody> </table>		Calculus	English	Baldwin	148	130	Duncan	192	180
	Calculus	English								
Baldwin	148	130								
Duncan	192	180								
<code>DataFrame.div()</code>	Operator: / <pre>>>> sliced.div(3) >>> sliced / 3</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>24.666667</td> <td>21.666667</td> </tr> <tr> <td>Duncan</td> <td>32.000000</td> <td>30.000000</td> </tr> </tbody> </table>		Calculus	English	Baldwin	24.666667	21.666667	Duncan	32.000000	30.000000
	Calculus	English								
Baldwin	24.666667	21.666667								
Duncan	32.000000	30.000000								
<code>DataFrame.floordiv()</code>	Operator: // <pre>>>> sliced.floordiv(3) >>> sliced // 3</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>24</td> <td>21</td> </tr> <tr> <td>Duncan</td> <td>32</td> <td>30</td> </tr> </tbody> </table>		Calculus	English	Baldwin	24	21	Duncan	32	30
	Calculus	English								
Baldwin	24	21								
Duncan	32	30								
<code>DataFrame.mod()</code>	Operator: % <pre>>>> sliced.mod(3) >>> sliced % 3</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>2</td> <td>2</td> </tr> <tr> <td>Duncan</td> <td>0</td> <td>0</td> </tr> </tbody> </table>		Calculus	English	Baldwin	2	2	Duncan	0	0
	Calculus	English								
Baldwin	2	2								
Duncan	0	0								
<code>DataFrame.power()</code>	Operator: ** <pre>>>> sliced.pow(2) >>> sliced ** 2</pre> <table> <thead> <tr> <th></th> <th>Calculus</th> <th>English</th> </tr> </thead> <tbody> <tr> <td>Baldwin</td> <td>5476</td> <td>4225</td> </tr> <tr> <td>Duncan</td> <td>9216</td> <td>8100</td> </tr> </tbody> </table>		Calculus	English	Baldwin	5476	4225	Duncan	9216	8100
	Calculus	English								
Baldwin	5476	4225								
Duncan	9216	8100								

Operating on Mixed Data Types

Sometimes the datasets you're working with will contain both numeric and non-numeric data. Consider `grades_num`, but in this case, some of the scores are marked with the string `N/A` to indicate that the student hasn't taken the course, or `P` to indicate the score is still pending:

	Calculus	English	Physics	History	Business	Art
Parker	94	83	N/A	79	84	83
Baldwin	74	65	91	82	84	91
Duncan	96	90	60	89	P	91
Cain	75	N/A	89	85	84	92
Rivera	74	79	89	91	83	91

Now, let's say you're grading on a curve and want to raise the scores accordingly. You'll use a square root method for the curve. If you only cared about regrading History class, this wouldn't be a problem:

```
>>> grades_num['History'] = (10 * grades_num['History'] ** .5).astype('int64')
>>> grades_num
   Calculus English Physics History Business Art
Parker      94       83      N/A      88       84     83
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

Baldwin	74	65	91	90	84	91
Duncan	96	90	60	94	P	91
Cain	75	N/A	89	92	84	92
Rivera	74	79	89	95	83	91

As long as the row or column you're operating on has all numeric types, pandas can perform a numeric operation. But let's say you wanted to apply this curve to the entire DataFrame:

```
>>> grades_num = (grades_num ** .5 * 10).astype('int64')
...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

Because some non-numeric data is included in this operation, it won't work. Remember that the data type of a column is converted to the data type that best accommodates the column's values—in this case, any column with an N/A or P value is converted to a string type. So, you need to convert this data to a numeric format first.

The `pandas.to_numeric()` Function

To operate on non-numeric data, you can use the `pandas.to_numeric()` function to convert it to a numeric type. By default, the function tries a direct conversion, so the string 8 turns into the integer 8. Obviously, this isn't going to work in all cases—the N/A and P strings in `grades_num` have no numeric equivalent. When these instances happen, pandas must decide how to handle them. You can specify this handling logic through the `errors` argument, which takes one of three values:

- '`'raise'`' —The Python interpreter will throw an exception, and the conversion will be unsuccessful. This is the default behavior.
- '`'coerce'`' —All values that cannot be directly converted will instead be converted to NaN.
- '`'ignore'`' —All values that cannot be directly converted will remain as they are.

The `to_numeric()` function works with most types of objects *except* a DataFrame. If you want to use it with a DataFrame, call the `DataFrame.apply()` function with `to_numeric()` as the main argument. The following example does this for `grades_num`, then does the grade-curving arithmetic:

```
>>> grades_num = grades_num.apply(pandas.to_numeric, errors = 'coerce')
>>> grades_num ** .5 * 10
      Calculus   English   Physics   History   Business   Art
Parker    96.953597  91.104336       NaN  93.808315  91.651514  91.104336
Baldwin   86.023253  80.622577  95.393920  94.868330  91.651514  95.393920
Duncan    97.979590  94.868330  77.459667  96.953597       NaN  95.393920
Cain      86.602540       NaN  94.339811  95.916630  91.651514  95.916630
Rivera    86.023253  88.881944  94.339811  97.467943  91.104336  95.393920
```

If you wanted to convert these figures to integers, you would need to handle the NaN values in some way first (e.g., using `fillna(0)`). In newer versions of pandas, you could also set them to the NA value so that the null value is of an integer type.

The `pandas.to_datetime()` Function

When you pull date and time data into a DataFrame from an external source, it might come in as a string object. You can convert this string into a datetime object so it's easier to operate on or perform calculations on. Like with `to_numeric()`, if the date or time string is already in a compatible format, it can be converted easily. The `pandas.to_datetime()` function is quite smart at parsing the different ways to represent date and time. For example:

```
>>> dates = pandas.Series(['1/30/2020', '2020/1/30',
                           '1-30-2020', 'January 30, 2020'])
>>> pandas.to_datetime(dates)
0    2020-01-30
1    2020-01-30
2    2020-01-30
```

```
3    2020-01-30
dtype: datetime64[ns]
```

Referencing Date and Time

When your data is in the datetime format, you can easily reference specific components of the date or time. The following attributes and functions are some examples of how to do this:

- `dt.year`, `dt.month`, `dt.week`, `dt.day` —Return the integer-based year, month, week, and day, respectively.
- `dt.hour`, `dt.minute`, `dt.second` —Return the integer-based hour, minute, and second, respectively.
- `dt.month_name()`, `dt.day_name()` —Return the string-based names of months and days, respectively.

Arithmetic on Multiple DataFrames

Aside from the scalar values used in the previous examples, you can also perform arithmetic on two or more DataFrame objects. Let's say you have a summarized version of `sales_df`:

	Units	Sales
Jones	372	23064.66
Perez	312	20409.09

Now, you have another DataFrame called `new_sales_df` with more sales figures:

	Units	Sales
Jones	76	4712.23
Perez	98	6419.01

This DataFrame has the same shape and index names as the original DataFrame, so pandas can easily add the appropriate cell values together and return a single DataFrame:

```
>>> sales_df + new_sales_df
      Units      Sales
Jones    448  27776.89
Perez    410  26828.10
```

However, what if the shapes are mismatched? This time, `new_sales_df` has one extra row and one extra column:

	Units	Sales	Clients
Jones	76	4712.23	65
Perez	98	6419.01	41
Stevens	43	2537.64	12

pandas still operates on data that exists in the same place in all DataFrame objects, but not on data that does not exist in all objects. It also causes the resulting DataFrame to take on the shape of the larger object, sorts the columns in alphabetical order, and marks the missing values as NaN:

```
>>> sales_df + new_sales_df
      Clients      Sales      Units
Jones      Nan  27776.89  448.0
Perez      Nan  26828.10  410.0
Stevens    Nan       NaN     NaN
```

You can then handle these missing values as your needs dictate.

Fill Values

You can get a little more flexibility out of the operation by using the function itself (e.g., `add()`), rather than the wrapper operator. The `fill_value` argument fills any missing values with whatever value you provide. This fills in the *inputs before the operation*, not the resulting DataFrame after the operation. For example:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

```
>>> sales_df.add(new_sales_df, fill_value = 0)
      Clients   Sales  Units
Jones       65.0  27776.89  448.0
Perez       41.0  26828.10  410.0
Stevens     12.0   2537.64   43.0
```

Notice how the `Clients` column and the `Stevens` row all have their values from `sales_df_new`. This is because all of those numbers were added to 0, as that is what the missing values in `sales_df` were filled in with.

The DataFrame.drop() Function

You can remove entire rows or columns by using the `DataFrame.drop()` function. Like with `rename()`, there are two ways to use the arguments in this function. This first example drops the `Business` and `Art` columns from `grades_num` by supplying these as `labels` and specifying `axis=1`:

```
>>> grades_num.drop(labels = ['Business', 'Art'], axis = 1)
      Calculus  English  Physics  History
Parker        96      91.0      NaN      93
Baldwin       86      80.0      95.0      94
Duncan        97      94.0      77.0      96
Cain          86      NaN      94.0      95
Rivera        86      88.0      94.0      97
```

Alternatively, supply the labels via the `index` or `columns` arguments, depending on whether you want to drop rows or columns:

```
grades_num.drop(columns = ['Business', 'Art'])
```



Note: If you're working with a multi-indexed DataFrame, you can also drop rows or columns through the `level` argument.

Guidelines for Modifying Data in DataFrames

Follow these guidelines when you are modifying data in DataFrames.

Modify Data in DataFrames

When modifying data in DataFrames:

- Use the `rename()` function to change row and/or column labels.
- Use the `fillna()` function to fill all missing values with the value(s) you specify.
- Use the `where()` function to replace values when a given condition is false.
- Use arithmetic operators to perform calculations on DataFrame data.
- Be aware that you cannot operate on an entire DataFrame or group of columns where at least one column has a non-numeric type.
 - Use the `to_numeric()` function to convert that column to a numeric type first.
- Use the `to_datetime()` function to make date and time values easier to work with.
- Use the `drop()` function to remove unwanted rows or columns.

ACTIVITY 5–2

Modifying Data in DataFrames

Data Files

/home/student/DSTIP/pandas/Modifying DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data_full.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've analyzed and shaped your data to get it into a more useful form, but you're not done yet. You still need to perform some cleaning tasks, including:

- Some of the column labels could stand to be renamed to something more accurate.
- The Date column is currently formatted with string types. It'll be much easier to work with dates if they're in a datetime format.
- There are several missing values in different columns. This is likely due to an error in how the data was transmitted to you. Thankfully, your colleagues have provided you with some of the missing values—you just need to plug them in to the DataFrame.
- Some of the missing values were lost: revenue and COGS. Revenue is easy enough to calculate, given the available data (total sales – tax price), but COGS is not. You'll need to provide a mathematically generated estimate of these missing COGS values—a process called imputation.
- Similar to what you encountered earlier, a few of the transactions have erroneous quantities. You've decided to just drop these rows from the DataFrame.

In addition to these cleaning procedures, you also want to enhance the DataFrame by adding a new column for gross income. This can be generated from the available data once all the missing values are filled in.

1. In Jupyter Notebook, open the DSTIP/pandas/Modifying DataFrames.ipynb file.
2. Import the relevant software libraries and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 The stores_data_full.csv file has the full 996 invoice records and all current columns.
 - b) Run the code cell.
3. Rename some of the columns.
 - a) Scroll down and view the cell titled **Rename some of the columns**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 | stores_df.columns
```

- c) Run the code cell.

- d) Examine the output.

```
Index(['Date', 'City', 'CustomerType', 'Gender', 'ProductLine', 'UnitPrice',
       'Quantity', 'Tax', 'TotalPrice', 'Revenue', 'COGS', 'CustomerRating'],
      dtype='object')
```

- This is a list of all column labels in the DataFrame.
 - Branch might be a more relevant descriptor than City.
 - Tax should be TaxPrice so it's not confused with the tax rate.
- e) Select the next code cell, and then type the following:

```
1 new_cols = {'City': 'Branch', 'Tax': 'TaxPrice'}
2 stores_df = stores_df.rename(columns = new_cols)
3 stores_df.head()
```

- Line 1 creates a dictionary that will be used to rename the columns, where the key is the current column label and the value is the new column label.
 - Line 2 performs the rename operation.
- f) Run the code cell.
g) Examine the output.

InvoiceID	Date	Branch	CustomerType	Gender	ProductLine	UnitPrice	Quantity	TaxPrice	TotalPrice	Revenue	COGS	CustomerRating
CAR-CLO-001	3/10/2019	Carbon Creek	Normal	Female	Clothing	87.67	2.0	8.77	184.11	175.34	170.12	7.7
CAR-CLO-002	1/12/2019	Carbon Creek	Member	Female	Clothing	20.01	9.0	9.00	189.09	180.09	174.55	5.7
CAR-CLO-003	2/10/2019	Carbon Creek	Member	Female	Clothing	30.14	10.0	15.07	316.47	301.40	284.76	9.2
CAR-CLO-004	1/29/2019	Carbon Creek	Normal	Male	Clothing	83.24	9.0	37.46	786.62	749.16	709.60	7.4
CAR-CLO-005	2/8/2019	Carbon Creek	Normal	Male	Clothing	98.98	10.0	49.49	1039.29	989.80	955.90	8.7

As desired, City is now Branch and Tax is now TaxPrice.

4. Convert the Date column to datetime format.

- Scroll down and view the cell titled **Convert the Date column to datetime format**, then select the code cell below it.
- In the code cell, type the following:

```
1 converted_dates = pd.to_datetime(stores_df['Date'])
2 stores_df['Date'] = converted_dates
3 stores_df.head()
```

- Line 1 converts all of the string objects in Date to the datetime format. The to_datetime() function parses the strings automatically.
 - Line 2 replaces the string values with the new datetime values in the DataFrame.
- c) Run the code cell.

- d) Examine the output.

InvoiceID	Date	Branch	CustomerType	Gender	ProductLine	UnitPrice	Quantity	TaxPrice	TotalPrice	Revenue	COGS	CustomerRating
CAR-CLO-001	2019-03-10	Carbon Creek	Normal	Female	Clothing	87.67	2.0	8.77	184.11	175.34	170.12	7.7
CAR-CLO-002	2019-01-12	Carbon Creek	Member	Female	Clothing	20.01	9.0	9.00	189.09	180.09	174.55	5.7
CAR-CLO-003	2019-02-10	Carbon Creek	Member	Female	Clothing	30.14	10.0	15.07	316.47	301.40	284.76	9.2
CAR-CLO-004	2019-01-29	Carbon Creek	Normal	Male	Clothing	83.24	9.0	37.46	786.62	749.16	709.60	7.4
CAR-CLO-005	2019-02-08	Carbon Creek	Normal	Male	Clothing	98.98	10.0	49.49	1039.29	989.80	955.90	8.7

- The values in the `Date` column have been reformatted.
 - In datetime format, the year is shown first, then the month, then the day.
- e) Select the next code cell, and then type the following:

```
1 feb_cond = stores_df['Date'].dt.month_name() == 'February'
2 stores_df[feb_cond].head()
```

- This code uses a mask to index only the transactions that took place in February.
 - The `dt.month_name()` function enables you to supply the month as a string, which the datetime format can interpret. You could also supply the number of the month as an integer by using `dt.month`.
- f) Run the code cell.
g) Examine the output.

InvoiceID	Date	Branch	CustomerType	Gender	ProductLine	UnitPrice	Quantity	TaxPrice	TotalPrice	Revenue	COGS	CustomerRating
CAR-CLO-003	2019-02-10	Carbon Creek	Member	Female	Clothing	30.14	10.0	15.07	316.47	301.40	284.76	9.2
CAR-CLO-005	2019-02-08	Carbon Creek	Normal	Male	Clothing	98.98	10.0	49.49	1039.29	989.80	955.90	8.7
CAR-CLO-010	2019-02-03	Carbon Creek	Normal	Male	Clothing	77.02	5.0	19.26	404.36	385.10	377.39	5.5
CAR-CLO-012	2019-02-27	Carbon Creek	Normal	Female	Clothing	77.93	9.0	35.07	736.44	701.37	674.23	7.6
CAR-CLO-014	2019-02-02	Carbon Creek	Member	Male	Clothing	43.13	10.0	21.57	452.87	431.30	419.05	5.5

Only transactions from February are printed.

5. Handle missing values.

- a) Scroll down and view the cell titled **Handle missing values**, and examine the code listing below it.
This is code you've run before; it returns all rows and columns that have missing values.
- b) Run the code cell.

- c) Examine the output.

	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-CLO-015	Male	4.0	156.03	NaN	NaN
CAR-ELE-060	Male	2.0	121.86	NaN	NaN
CAR-FBV-054	Male	NaN	279.38	266.08	256.98
CAR-HBE-025	Male	1.0	26.25	NaN	NaN
CAR-STR-027	Male	NaN	81.40	77.52	73.26
GRC-HBE-029	NaN	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	NaN	279.18	264.93
OLI-HBE-038	NaN	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	NaN	NaN

The missing values are as follows:

- Two transactions in which the Gender of the customer is missing.
- Two transactions in which the purchase Quantity is missing.
- One transaction in which the TotalPrice is missing.
- Four transactions in which both the Revenue and COGS are missing.

- d) Select the next code cell, and then type the following:

```
1 fill_vals = {'Gender': 'Male', 'TotalPrice': 293.14}
2 stores_df = stores_df.fillna(fill_vals)
3 stores_df.loc[rows, cols]
```

- Line 1 creates a dictionary of the fill values that have been provided for you. For each key (column), there is a value to replace NaN in that column.
- Line 2 uses fillna() to fill the DataFrame according to the dictionary.

- e) Run the code cell.
f) Examine the output.

	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-CLO-015	Male	4.0	156.03	NaN	NaN
CAR-ELE-060	Male	2.0	121.86	NaN	NaN
CAR-FBV-054	Male	NaN	279.38	266.08	256.98
CAR-HBE-025	Male	1.0	26.25	NaN	NaN
CAR-STR-027	Male	NaN	81.40	77.52	73.26
GRC-HBE-029	Male	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	293.14	279.18	264.93
OLI-HBE-038	Male	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	NaN	NaN

- The missing Gender values have been replaced with Male.
- The missing TotalPrice value has been replaced with 293.14.

g) Select the next code cell, and then type the following:

```
1 quant_fill = pd.Series([4, 3], index = ['CAR-FBV-054', 'CAR-STR-027'])
2 stores_df['Quantity'] = stores_df['Quantity'].fillna(quant_fill)
3 stores_df.loc[rows, cols]
```

- Line 1 creates a Series that maps the indices that have missing Quantity values to their new values. You can use a Series or DataFrame to replace missing values with multiple replacement values in a single column, rather than applying the same replacement value for all missing values in a column. In this case, each of the two rows has a different missing quantity.
 - Line 2 uses the Series to fill the missing items in the Quantity column.
- h) Run the code cell.
i) Examine the output.

	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-CLO-015	Male	4.0	156.03	NaN	NaN
CAR-ELE-060	Male	2.0	121.86	NaN	NaN
CAR-FBV-054	Male	4.0	279.38	266.08	256.98
CAR-HBE-025	Male	1.0	26.25	NaN	NaN
CAR-STR-027	Male	3.0	81.40	77.52	73.26
GRC-HBE-029	Male	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	293.14	279.18	264.93
OLI-HBE-038	Male	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	NaN	NaN

The first missing quantity was changed to 4.0, and the second missing quantity was changed to 3.0.

- j) Select the next code cell, and then type the following:

```
1 revenue = stores_df['TotalPrice'] - stores_df['TaxPrice']
2 stores_df['Revenue'] = stores_df['Revenue'].fillna(revenue)
3 stores_df.loc[rows, cols]
```

- Line 1 subtracts all tax prices from their associated total prices in each row to generate a Series that holds revenue values.
 - Line 2 fills missing revenue values with those generated in the previous line.
- k) Run the code cell.

- I) Examine the output.

	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-CLO-015	Male	4.0	156.03	148.60	NaN
CAR-ELE-060	Male	2.0	121.86	116.06	NaN
CAR-FBV-054	Male	4.0	279.38	266.08	256.98
CAR-HBE-025	Male	1.0	26.25	25.00	NaN
CAR-STR-027	Male	3.0	81.40	77.52	73.26
GRC-HBE-029	Male	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	293.14	279.18	264.93
OLI-HBE-038	Male	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	522.08	NaN

The revenue values are now filled in.

6. Use arithmetic to impute missing COGS values.

- Scroll down and view the cell titled **Use arithmetic to impute missing COGS values**, then select the code cell below it.
- In the code cell, type the following:

```

1 # Average percentage decrease from revenue to COGS.
2 perc_decr = ((stores_df['Revenue'] - stores_df['COGS']) \
3               / (stores_df['Revenue'])) .mean()
4
5 perc_decr

```

- Unlike with revenue, COGS cannot be easily derived from the other attributes in the dataset. You'll need to impute the missing COGS values.
 - Lines 2 and 3 calculate the average percentage decrease from revenue to COGS for existing values.
 - COGS values are subtracted from revenue, the result of which is divided by revenue to get the percentages. Then, the mean of all percentages is taken.
- Run the code cell.
 - Examine the output.

```
0.03862743423943755
```

On average, the COGS for a transaction is about 4% less than its revenue. You can use this figure to generate an estimation for the missing COGS values, as every transaction now has a revenue value.

- e) Select the next code cell, and then type the following:

```
1 ind = ['CAR-CLO-015', 'CAR-ELE-060', 'CAR-HBE-025', 'OLI-HML-039']
2
3 impute_vals = round(stores_df.loc[ind, 'Revenue'] / (1 + perc_decr), 2)
4 impute_vals
```

- Line 1 creates a list of indices that have missing COGS values.
 - Line 3 indexes the relevant rows on their Revenue values, then divides that revenue by the percentage decrease plus 1. The result is rounded to two decimal places.
- f) Run the code cell.
g) Examine the output.

InvoiceID	
CAR-CLO-015	143.07
CAR-ELE-060	111.74
CAR-HBE-025	24.07
OLI-HML-039	502.66
Name:	Revenue, dtype: float64

- A Series was returned with the imputed COGS values.
h) Select the next code cell, and then type the following:

```
1 stores_df['COGS'] = stores_df['COGS'].fillna(impute_vals)
2 stores_df.loc[rows, cols]
```

This code fills in the missing COGS values in the DataFrame with the imputed values.



Note: This is not the only way to impute values, nor is it necessarily the optimal way for this scenario.

- i) Run the code cell.
j) Examine the output.

InvoiceID	Gender	Quantity	TotalPrice	Revenue	COGS
InvoiceID					
CAR-CLO-015	Male	4.0	156.03	148.60	143.07
CAR-ELE-060	Male	2.0	121.86	116.06	111.74
CAR-FBV-054	Male	4.0	279.38	266.08	256.98
CAR-HBE-025	Male	1.0	26.25	25.00	24.07
CAR-STR-027	Male	3.0	81.40	77.52	73.26
GRC-HBE-029	Male	5.0	288.02	274.30	264.65
OLI-HBE-025	Male	6.0	293.14	279.18	264.93
OLI-HBE-038	Male	5.0	57.70	54.95	52.02
OLI-HML-039	Male	8.0	548.18	522.08	502.66

All missing values have been filled in.

7. Create a new gross income column.

- a) Scroll down and view the cell titled **Create a new gross income column**, then select the code cell below it.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

- b) In the code cell, type the following:

```
1 stores_df['GrossIncome'] = stores_df['Revenue'] - stores_df['COGS']
2 stores_df.iloc[:10, -8:]
```

Line 1 indexes a new column by subtracting all COGS values from revenue (gross income).

- c) Run the code cell.
d) Examine the output.

	UnitPrice	Quantity	TaxPrice	TotalPrice	Revenue	COGS	CustomerRating	GrossIncome
InvoiceID								
CAR-CLO-001	87.67	2.0	8.77	184.11	175.34	170.12	7.7	5.22
CAR-CLO-002	20.01	9.0	9.00	189.09	180.09	174.55	5.7	5.54
CAR-CLO-003	30.14	10.0	15.07	316.47	301.40	284.76	9.2	16.64
CAR-CLO-004	83.24	9.0	37.46	786.62	749.16	709.60	7.4	39.56
CAR-CLO-005	98.98	10.0	49.49	1039.29	989.80	955.90	8.7	33.90
CAR-CLO-006	89.69	1.0	4.48	94.17	89.69	87.36	4.9	2.33
CAR-CLO-007	17.94	5.0	4.49	94.19	89.70	85.88	6.8	3.82
CAR-CLO-008	81.91	2.0	8.19	172.01	163.82	156.37	7.8	7.45
CAR-CLO-009	61.77	5.0	15.44	324.29	308.85	292.80	6.7	16.05
CAR-CLO-010	77.02	5.0	19.26	404.36	385.10	377.39	5.5	7.71

GrossIncome was appended as a new column.

8. Identify and drop rows with erroneous quantities.

- a) Scroll down and view the cell titled **Identify and drop rows with erroneous quantities**, then select the code cell below it.
b) In the code cell, type the following:

```
1 stores_df[stores_df['Quantity'] < 1]
```

- c) Run the code cell.
d) Examine the output.

	Date	Branch	CustomerType	Gender	ProductLine	UnitPrice	Quantity	TaxPrice	TotalPrice	Revenue	COGS	CustomerRating	GrossIncome
InvoiceID													
OLI-FBV-049	2019-03-30	Olinger	Member	Female	Food and beverages	72.52	0.0	29.01	609.17	580.16	559.08	4.0	21.08
OLI-FBV-056	2019-02-02	Olinger	Member	Female	Food and beverages	38.42	-1.0	1.92	40.34	38.42	37.32	8.6	1.10

All transactions in which the quantity is less than 1 are displayed.

- e) Select the next code cell, and then type the following:

```

1 print('Number of rows BEFORE drop: {}'.format(stores_df.shape[0]))
2
3 rows_drop = stores_df[stores_df['Quantity'] < 1].index
4 stores_df = stores_df.drop(index = rows_drop)
5
6 print('Number of rows AFTER drop: {}'.format(stores_df.shape[0]))

```

- Line 3 uses the mask from before to grab the indices of the two affected rows.
 - Line 4 drops these rows.
- f) Run the code cell.
g) Examine the output.

```
Number of rows BEFORE drop: 996.
Number of rows AFTER drop: 994.
```

Both affected rows have been dropped.

9. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel→Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Modifying DataFrames** tab in Firefox, but keep a tab open to **DSTIP/pandas/** in the file hierarchy.

TOPIC C

Plot DataFrame Data

While not primarily a visualization library, pandas does have some ability to generate plots based on the data in your `DataFrame`. So, before you take a deeper dive into the world of Python data visualization, you'll use pandas to create some rudimentary plots.

The `DataFrame.plot()` Function

`DataFrame.plot()` is provided by pandas as the function for plotting `DataFrame` objects. It enables you to visualize your data for more effective analysis, as well present that data to an audience. The function is actually an interface into a back-end plotting library. You can specify the back-end library it uses, but the default, and by far most popular, library is Matplotlib. So, you must have Matplotlib installed to use this functionality.

The most important arguments to this function are:

- `x`—Specify what data (e.g., a column in a `DataFrame`) to plot along the x-axis.
- `y`—Specify what data to plot along the y-axis.
- `kind`—Specify the type of plot you wish to generate. There are more than 10 possible options, most of which are discussed in this topic.

Not all plot types have an x-axis or a y-axis, so instead of using these arguments, you may just need to index or slice the data you want to use in the plot. The `plot()` function also has many arguments for customizing the look and feel of plots, but these will be discussed along with Matplotlib. For now, you'll see the different plots you can create and how to go about creating them from your data.

Scatter Plots

A *scatter plot* visually represents the relationship between two variables through the use of points on a graph. It is typically plotted in two dimensions, where the horizontal axis (the x-axis) corresponds to variable `x` and the vertical axis (the y-axis) corresponds to variable `y`. The `x` variable is also called the independent variable, as it may have an effect on `y`, also called the dependent variable. The dependent variable is the variable whose change you wish to study.

Scatter plots are useful in determining how two numeric variables correlate. In the context of a `DataFrame`, a variable usually corresponds to a column. So, as the values in one column increase, the values in another column may also increase—a positive correlation, showing the points going from bottom left to top right. Or, as the values in one column increase, the values in another column may decrease—a negative correlation, showing the points going from top left to bottom right. The other possibility is that there is no discernible pattern to how the points are spread, which indicates that there is no real correlation between the variables.

For the `DataFrame.plot()` function, the `kind = 'scatter'` argument generates a scatter plot. The following example uses a `DataFrame` of mock census data for a fictitious state (Richland). Each row represents an anonymized household. A row-truncated version of this `DataFrame` is as follows:

	City	Size	Income	Age
0	Olinger	1	43112.03	27
1	Greene City	5	78267.56	41
2	Olinger	1	37651.32	29
3	Carbon Creek	2	123714.98	45
4	Agerstown	4	83120.54	53
...				



Note: The `City` column is where the household resides; the `Size` column is the number of people in the household; and `Age` is the average age of working adults in the household.

In this scenario, you're interested in studying how household income is affected by age. So, the `Age` column is your x-axis, and `Income` is your y-axis:

```
census_df.plot(x = 'Age', y = 'Income', kind = 'scatter')
```

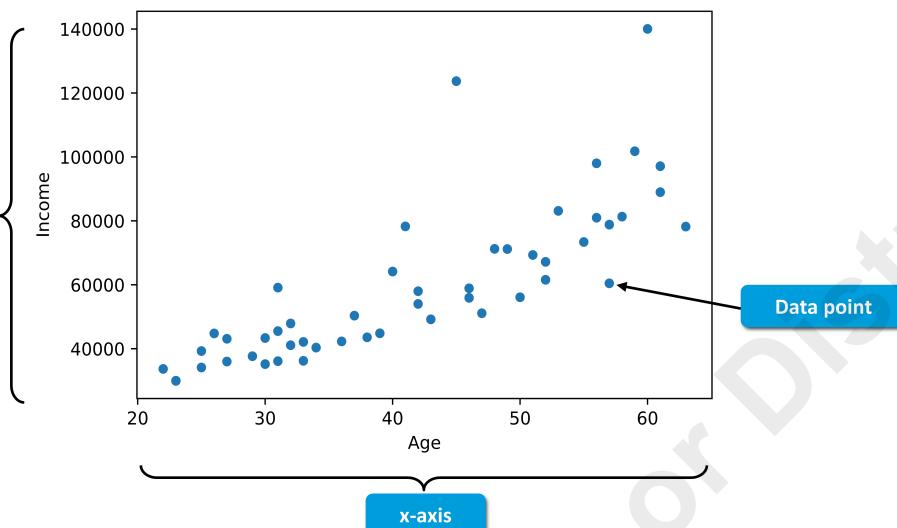


Figure 5-2: The generated scatter plot.

The scatter plot seems to indicate a positive correlation: as age increases, so does income.

Line Plots

A **line plot** is similar to a scatter plot, except that the points on the graph are connected by a series of straight lines. Line plots are most commonly used to visualize some trend over time, where time is along the x-axis and the dependent variable is along the y-axis. The lines are drawn from the first x-axis value to the next, then to the next, and so on. Therefore, it's common to sort the x-axis data first if it isn't already in order.

For the `DataFrame.plot()` function, the `kind = 'line'` argument generates a line plot. The following example uses a `DataFrame` of business revenue (in millions of dollars) over all 12 months of the year. A row-truncated version of this `DataFrame` is:

Month	Revenue	
0	Jan	2.1
1	Feb	3.2
2	Mar	2.4
3	Apr	2.0
4	May	2.9
...		

In this scenario, you're interested in studying how business revenue changes as the year progresses. So, the `'Month'` column is your x-axis, and `'Revenue'` is your y-axis:

```
revenue_df.plot(x = 'Month', y = 'Revenue', kind = 'line')
```

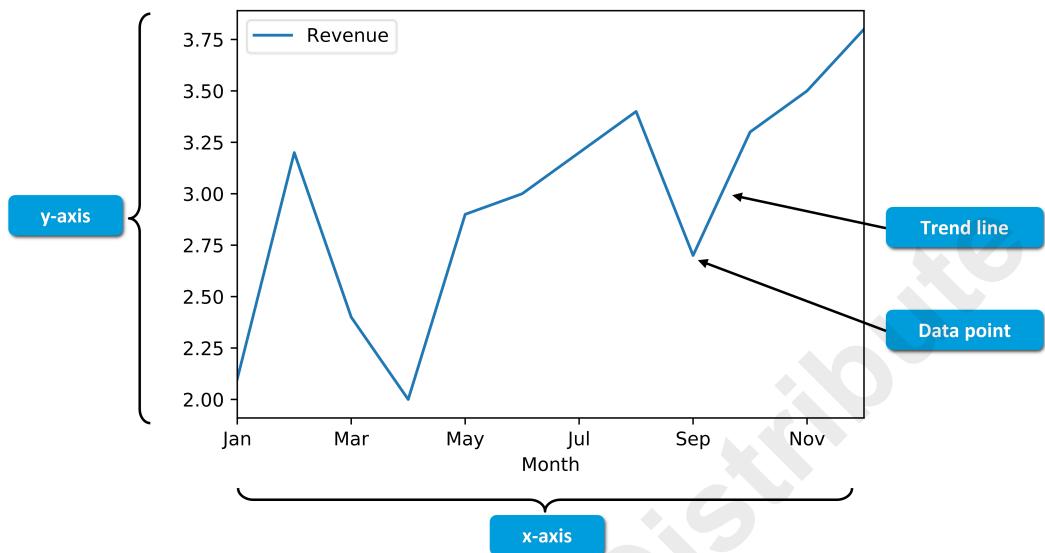


Figure 5–3: The generated line plot.

The line plot indicates that revenue fluctuated throughout the year, but the general trend appears to show an increase.

Area Plots

An **area plot** is a type of line plot in which the space below the line is filled in with some color or texture. Like a standard line plot, an area plot is typically used to represent change over time, but with an added emphasis on the general trend of the data, rather than the specific data points. They are also commonly used in a stacked format, in which the same measurements are compared across different contexts. For example, you could create an area plot in which revenue per month is shown, but with each area stack representing a different year.

For the `DataFrame.plot()` function, the `kind = 'area'` argument generates an area plot. The same `revenue_df` DataFrame is being used in this example, and the axes are the same:

```
revenue_df.plot(x = 'Month', y = 'Revenue', kind = 'area')
```

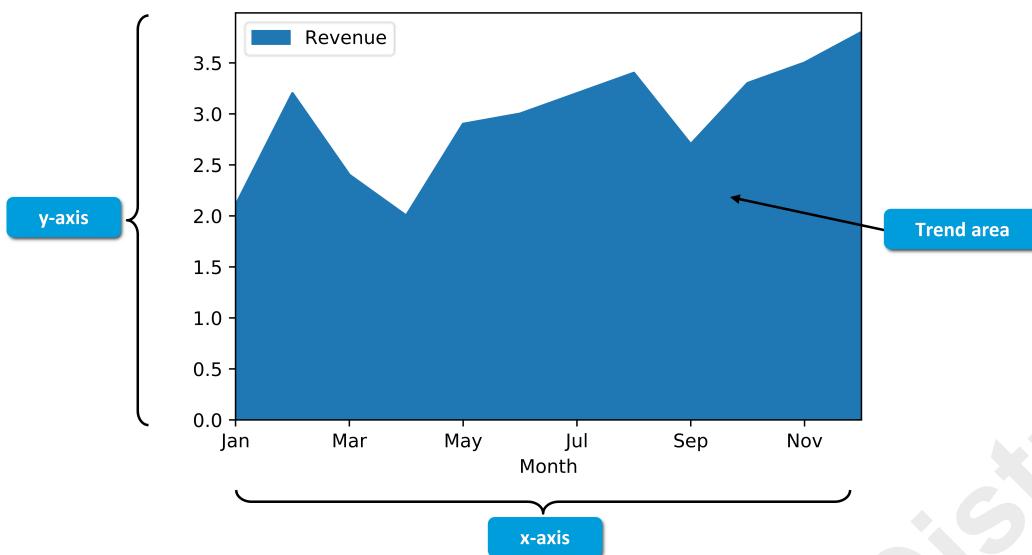


Figure 5-4: The generated area plot.

The area plot underscores the overall upward trend of revenue, without focusing on the specific amounts at each month. Note that the line looks different here than it did in the line plot, as the default behavior of this area plotting function is to start the y-axis at 0, rather than at the lowest value in the dataset.

Bar Charts

A **bar chart** represents the proportional measurement of categorical values by using either horizontal or vertical bars. In a vertical bar chart, the categorical values lie along the horizontal axis, and the measurement of each categorical value lies along the vertical axis. The opposite is true for a horizontal bar chart. Categorical variables are discrete, in that they have a limited number of possible values, and that there is a noticeable gap between each category. As the name suggests, they often represent categories of things, like countries, organizations, years, products, people, and so on. The purpose of a bar chart is to compare the measurement of each thing with other things in that category.

For the `DataFrame.plot()` function, the `kind = 'bar'` argument generates a vertical bar chart, and `kind = 'barh'` generates a horizontal bar chart. In the following scenario, which uses `census_df`, you're interested in comparing the mean income of residents based on their city of residence. Therefore, the `City` column represents the category, and each unique city is a categorical value. The mean income is the measurement that will appear on the opposite axis.

However, you need to prepare the data somewhat before generating the chart. If you just created the chart like in the previous examples, you would have a bar for every single row of data. Instead, you should group the data together based on the resident's city, then take the mean of that city's income:

```
census_df.groupby('City')['Income'].mean().plot(kind = 'barh')
```

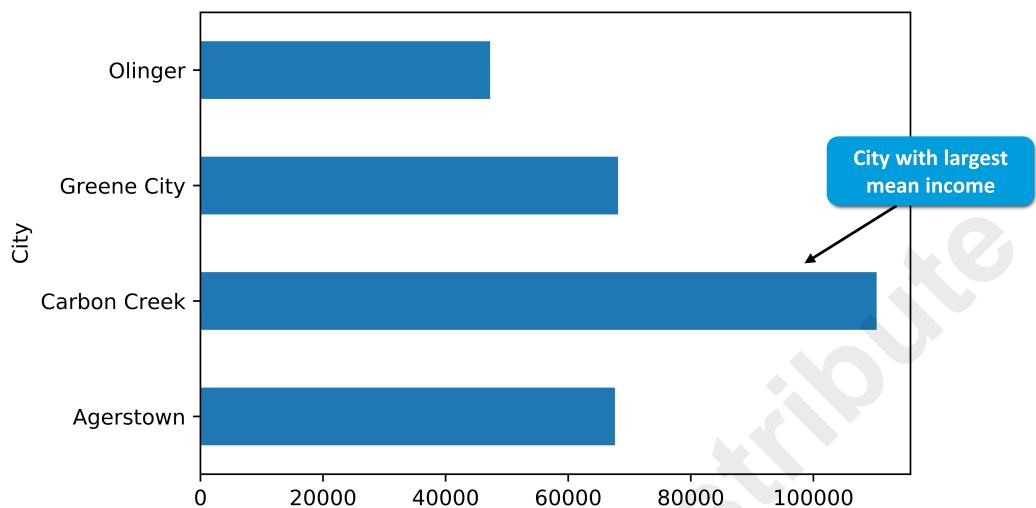


Figure 5–5: The generated bar chart.

Looking at this chart, it appears the residents of Carbon Creek are the most wealthy among the four cities, while the residents of Olinger are the least wealthy.

Histograms

A **histogram** is similar in appearance to a bar chart; the main difference is that a histogram compares different frequencies of one continuous numeric variable, rather than comparing a measurement of multiple values in a discrete categorical variable. In other words, it represents the distribution of a continuous variable. A continuous variable can extend infinitely and has no defined gaps between each value. Instead, the variable is placed into multiple "bins" that divide the entire range of the variable along the horizontal axis. The vertical axis shows the frequency of observations in each bin. So, a continuous variable can be something like price, revenue, velocity, time, and so on. The histogram will show how many observations fit into each bin.

For the `DataFrame.plot()` function, the `kind = 'hist'` argument generates a histogram. In the following scenario, which uses `census_df`, you're interested in seeing the overall spread of income values for the state's households. Therefore, the `Income` column has the continuous variable of interest. The `plot()` function will divide the variable into 10 bins by default, though you can change this behavior through the `bins` argument.

```
census_df['Income'].plot(kind = 'hist')
```

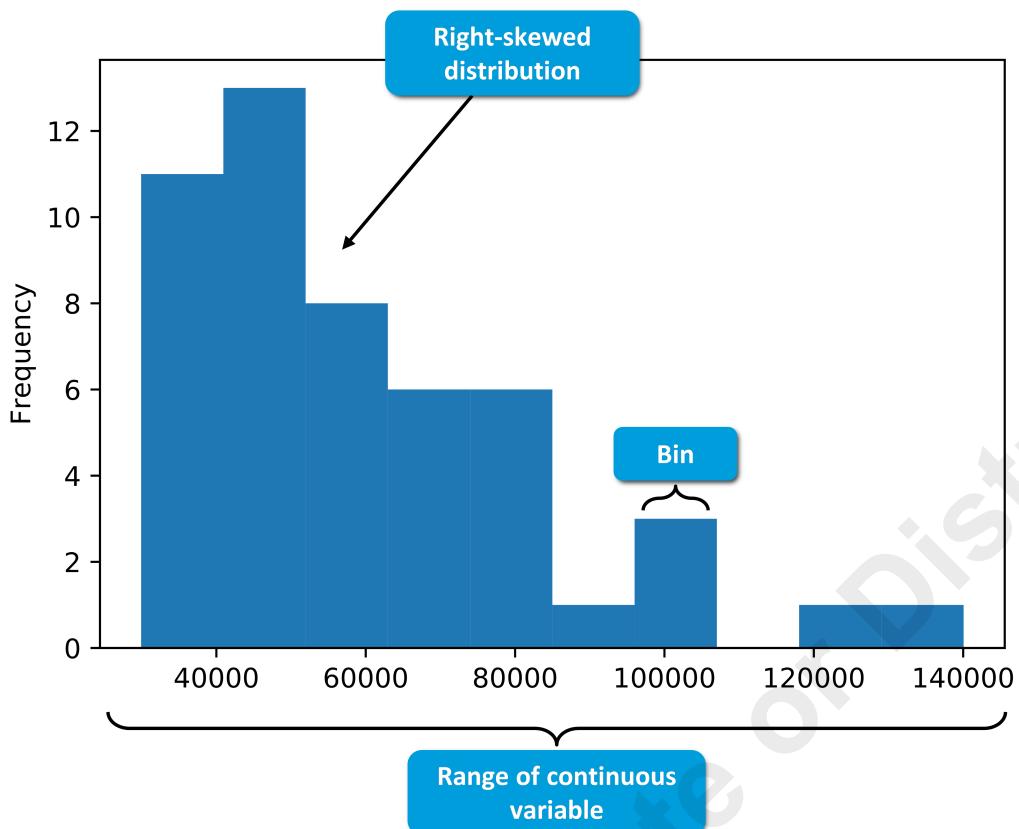


Figure 5-6: The generated histogram.

Household income appears to skew to the right, suggesting that there are more people who make between \$20,000–60,000 than people who make higher amounts.

Box Plots

A **box plot**, also called a box-and-whisker plot, is another method of displaying the distribution of numerical data. It visually represents five different summary statistics about a variable. Either the horizontal or vertical axis shows the range of the numeric variable, and the summary statistics are drawn to match the appropriate values along that range. The statistics are:

- **Median (Q2)**—A line is drawn at the median value, also referred to as the second **quartile**.
- **Q1**—A box is drawn before the median to represent the first quartile range. The first quartile range corresponds to the 25th percentile of data.
- **Q3**—A box is drawn after the median to represent the third quartile range. The third quartile range corresponds to the 75th percentile of data.
- **Minimum**—A line is drawn at the minimum value. This is the lowest value in the data *excluding outliers*.
- **Maximum**—A line is drawn at the maximum value. This is the highest value in the data *excluding outliers*.

For the `DataFrame.plot()` function, the `kind = 'box'` argument generates a box plot. In the following scenario, you're interested in seeing the spread of household income again, but this time you want to see the summary statistics and any data points that are considered outliers.

```
census_df['Income'].plot(kind = 'box')
```

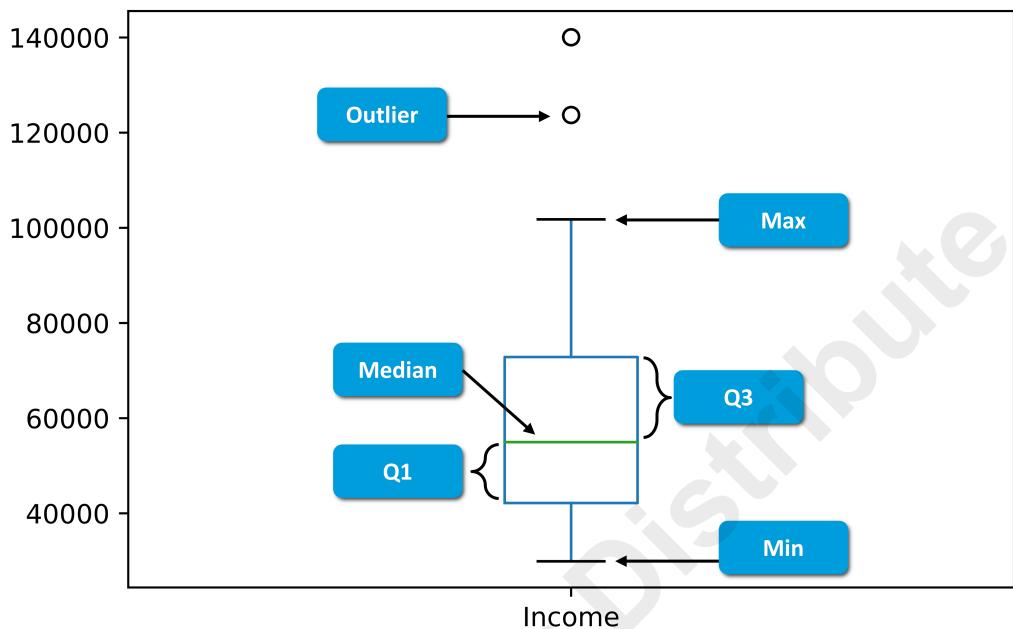


Figure 5–7: The generated box plot.

Like the histogram, this confirms that the majority of income values fall within a lower range of the distribution. You can also see some of the outliers that appear at higher income values.

Pie Charts

A **pie chart** represents the numerical proportion of some categorical variable in the form of a circle, where each slice corresponds to a categorical value's proportion. In other words, the chart depicts each "slice" of the pie as a percentage of the overall measurement that a group includes. Pie charts are commonly used in the business world to compare the performance or prevalence of some aspects of the business, though they are less suited to performing a true statistical analysis. This is because it is difficult for people to truly tell the difference between the relative size of each slice.

For the `DataFrame.plot()` function, the `kind = 'pie'` argument generates a pie chart. The default behavior is to take a column `y` as the numerical measurement, with the row indices as the categorical variable. However, in the following scenario, you want to visualize the proportion of household sizes by getting the totals of each unique size. So, each unique size is a categorical value, and the total number of households is the measurement:

```
census_df.groupby('Size')['Size'].sum().plot(kind = 'pie')
```

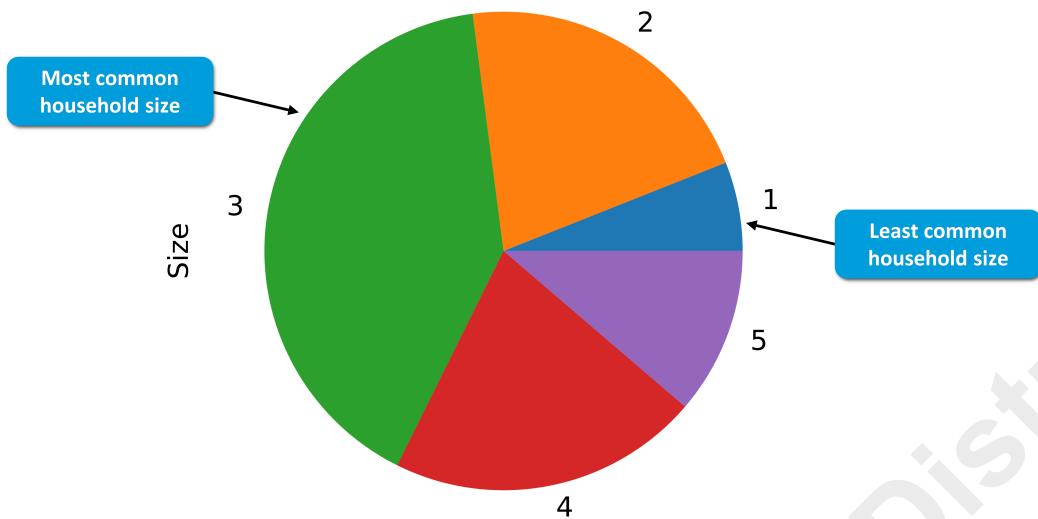


Figure 5-8: The generated pie chart.

It looks like a three-person household is the most common type of household in this dataset, with single-person households being the least common. The two-person and four-person household slices seem roughly the same size, though it is difficult to tell which is larger.

Guidelines for Plotting DataFrame Data

Follow these guidelines when you are plotting DataFrame data.

Plot DataFrame Data

When plotting DataFrame data:

- Consider using the `plot()` function directly from pandas to construct quick, simple plots from DataFrame data.
- Use scatter plots to show the relationship between two numeric variables.
- Use line plots to show the relationship between two variables, where a line is drawn from one point to the next.
 - Use to show trends over time.
- Use area plots to emphasize the trend of data in a line plot, rather than specific data points.
- Use bar charts to show how categorical variables are proportional to each other according to some numeric measurement.
 - Consider grouping the categorical variables first.
- Use histograms to show the distribution of a continuous numeric variable by using bins.
- Use box plots to show the distribution of a continuous numeric variable by using summary statistics.
- Avoid pie charts whenever possible, as it is difficult to compare the relative size of each "slice."
 - If you must use a pie chart, consider supplementing it with a similar plot like a bar chart.

ACTIVITY 5–3

Plotting DataFrame Data

Data Files

/home/student/DSTIP/pandas/Plotting DataFrames.ipynb
 /home/student/DSTIP/pandas/data/stores_data_full_clean.csv

Before You Begin

Jupyter Notebook is open.

Scenario

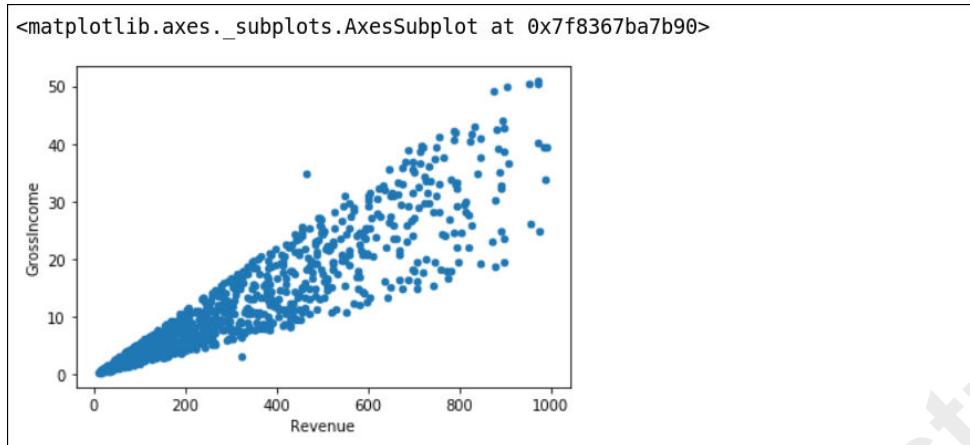
You've wrangled the GCE store data to a point where it's in a good state for further analysis. However, looking at summary statistics and other numerical metadata doesn't always make analysis any easier. You can greatly enhance your ability to interpret and make decisions about data if it's presented visually. Visuals can also help non-technical audiences or those without a data science background understand the simplified ideas you're trying to communicate about the data. So, you'll start producing some basic plots using the pandas library. For now, these plots should suffice for getting you started.

1. In Jupyter Notebook, open the **DSTIP/pandas/Plotting DataFrames.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 - The `stores_data_full_clean.csv` file has all the modifications you made in the previous activity.
 - On line 17, the `Date` column is being converted to datetime format. This does not happen automatically when you are loading text files.
 - b) Run the code cell.
3. Generate a scatter plot comparing revenue and gross income.
 - a) Scroll down and view the cell titled **Generate a scatter plot comparing revenue and gross income**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 stores_df.plot(x = 'Revenue', y = 'GrossIncome', kind = 'scatter')
```

- Values from `Revenue` will be on the x-axis.
 - Values from `GrossIncome` will be on the y-axis.
- c) Run the code cell.

- d) Examine the output.



- Each data point represents a row, so in this case, each transaction is a data point.
- Each point is plotted according to its values for revenue and gross income. For example, points with high revenue and high income are closer to the top right of the graph.
- This graph shows a positive correlation between revenue and gross income—as revenue increases, gross income tends to increase. This makes sense, as gross income is derived in part from revenue.
- The graph also shows that many values seem to cluster toward the low end of both revenue and income, which gives some hint about the distribution of those values.

4. Generate a line plot of revenue over each day in January.

- Scroll down and view the cell titled **Generate a line plot of revenue over each day in January**, then select the code cell below it.
- In the code cell, type the following:

```

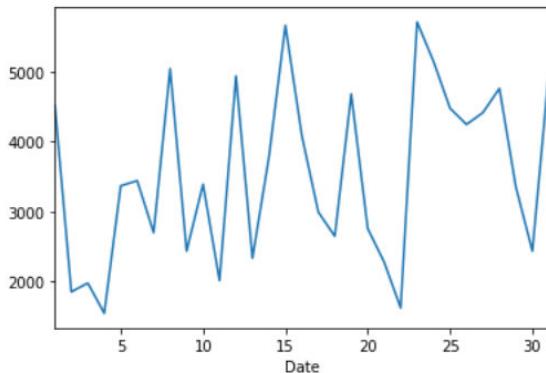
1 # Return January only.
2 jan_only = stores_df[stores_df['Date'].dt.month == 1]
3
4 days = jan_only['Date'].dt.day
5 days_rev = stores_df.groupby(days)[['Revenue']].sum()
6
7 days_rev.plot(kind = 'line')

```

- Line 2 uses a mask to create a new DataFrame of only the transactions that occurred in January.
- Line 4 gets all of the days in January on which a transaction occurred.
- Line 5 uses grouping to count the total revenue for each day in January. This returns a Series where each day is a row index and each day's total revenue is a value.
- Line 7 automatically plots the days in January on the x-axis, and the total revenue on the y-axis. You don't need to supply x and y arguments because plot() is being called on a Series (i.e., there are only two variables to choose from).
- Run the code cell.

- d) Examine the output.

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f833b47f8d0>
```



- You can see that revenue seems to fluctuate from day to day.
- There is not necessarily an increasing or decreasing trend at this level of time. A macro analysis of time (e.g., revenue over months or years) might reveal a steadier pattern.

5. Generate a bar plot of each branch's total revenue.

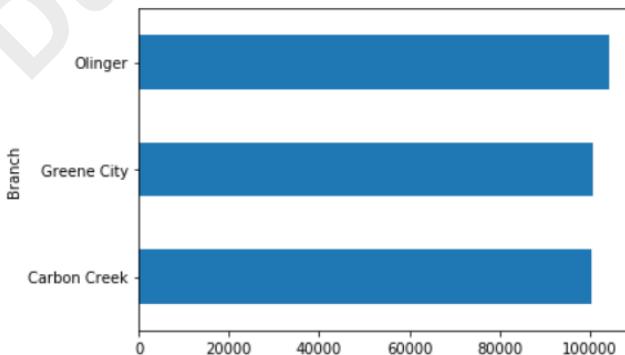
- Scroll down and view the cell titled **Generate a bar plot of each branch's total revenue**, then select the code cell below it.
- In the code cell, type the following:

```
1 branch_rev = stores_df.groupby('Branch')[['Revenue']].sum()
2 branch_rev.plot(kind = 'barh')
```

Line 1 using grouping to count the total revenue generated by each store branch.

- Run the code cell.
- Examine the output.

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4f0a7bed10>
```



- The bar chart shows the measurement of total revenue for each individual branch.
- Each branch generated a similar amount of revenue, though Olinger appears to have generated the most.

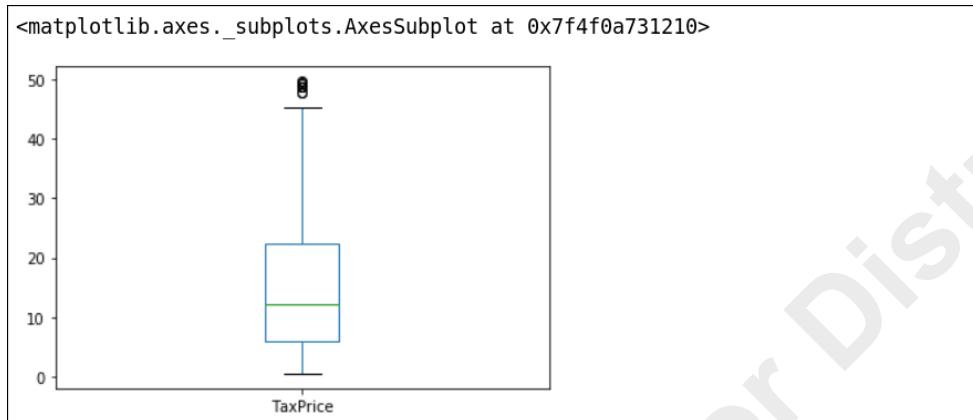
6. Generate a box plot showing the distribution of tax prices.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

- a) Scroll down and view the cell titled **Generate a box plot showing the distribution of tax prices**, then select the code cell below it.
- b) In the code cell, type the following:

```
1 | stores_df['TaxPrice'].plot(kind = 'box')
```

- c) Run the code cell.
- d) Examine the output.



- The box plot shows the distribution of tax price values.
- The median tax price seems to be around \$13.
- The Q1 range seems to be from around \$7 to \$13.
- The Q2 range seems to be from around \$13 to \$23.
- The minimum value is around \$1.
- The maximum value is around \$46.
- There are a few outliers that go beyond the maximum, but they seem to cut off around \$50.

7. Shut down this Jupyter Notebook kernel.

- a) From the menu, select **Kernel→Shutdown**.
- b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
- c) Close the **Plotting DataFrames** tab in Firefox, but keep a tab open to the file hierarchy in Jupyter Notebook.

Summary

In this lesson, you manipulated and modified `DataFrame` data, as well as generated some basic visualizations of that data. Compared to NumPy, pandas will likely make it easier for you to clean, prepare, and process data for its eventual presentation, as well as other downstream tasks that are part of your data science pipeline.

Considering your own data, why might you pivot or group that data?

Considering your own data, what are some reasons why might you drop a record from a `DataFrame`?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

6

Visualizing Data with Matplotlib and Seaborn

Lesson Time: 3 hours, 30 minutes

Lesson Introduction

Although you did some simple plotting with pandas directly, you'll likely need to get more detailed with your visualizations. Matplotlib is the most common plotting library in Python®, and you'll use it to generate visualizations that help you tell a story with your data. Likewise, you'll use the Seaborn library, which is built on Matplotlib, to help you streamline your plotting efforts.

Lesson Objectives

In this lesson, you will:

- Create simple line plots with Matplotlib and save them as files.
- Create subplots in Matplotlib.
- Create different types of plots that are available in Matplotlib.
- Format plots in Matplotlib.
- Use Seaborn to streamline plot creation and customization.

TOPIC A

Create and Save Simple Line Plots

To begin, you'll generate simple plots so that you can get used to the different ways to use Matplotlib.

The `pyplot.plot()` Function

The `matplotlib.pyplot` module is the main interface for plotting data by using Matplotlib. The interface can take data in multiple forms, including NumPy arrays and pandas `DataFrame` objects. While pandas can use Matplotlib as a back end, using Matplotlib directly gives you more control over your data visualization processes. You can use the `pyplot.plot()` function to plot the data.

For the sake of brevity, most programmers use a selective import alias of `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt
```

By default, the `plot()` function plots data on an x-axis and a y-axis. You supply the data for each axis as arguments.

```
plt.plot(revenue_df['Month'], revenue_df['Revenue'])
```

With no other arguments given, this produces a line plot:

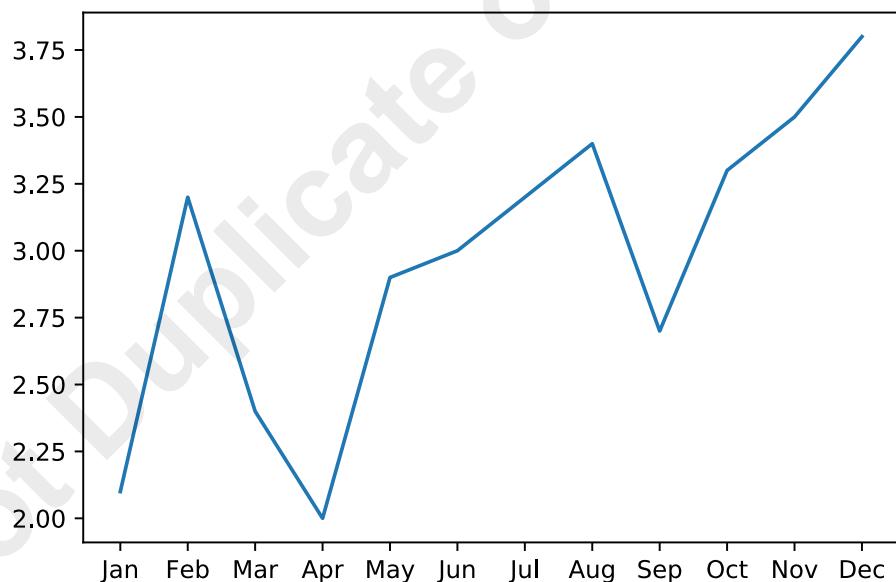


Figure 6–1: The generated line plot.

Showing Plots

In Jupyter® Notebook, using the `plot()` function automatically displays the plot in the output cell as a PNG image. You don't have to do anything else to get the plot to show. However, this behavior differs if you're executing a Python script or using an interactive shell like IPython. For such cases, Matplotlib provides a `pyplot.show()` function.

- When executing a script, the plots do not show unless you explicitly call the `show()` function. You would usually put this call at the end of the script, which opens up one or more system windows with your plots inside.
- When working with a shell, use `show()` whenever you want to open a window to display the plots you've already defined.
- When working with IPython specifically, you can also use the `%matplotlib` magic command to automatically open a window with your plots, without having to use `show()`.

Stateful Interface

Matplotlib actually provides two ways to define and generate plots: the stateful interface and the object-oriented (stateless) interface. The stateful interface, as the name implies, maintains a memory of the plots that have been defined. For example, if you call `pyplot.plot()`, the stateful interface keeps track of the elements that make up the plot: the overall figure itself, its axes, any decorations or other stylistic components, etc.

Using the stateful interface is simple, and it works best when your plots are simple. You use the stateful interface when you call `pyplot.plot()` directly, like in the previous example:

```
plt.plot(revenue_df['Month'], revenue_df['Revenue'])
```

There are some limitations to the stateful interface, however. If you call `pyplot.plot()` twice, the interface keeps track of only one plot—the "active" plot. If you want to change some aspect of the active plot, like adjusting its size, you can do so easily. However, if you want to change the plot that isn't active in the stateful interface, it can be quite complicated. Therefore, it's best to limit your use of the stateful interface to simple, individual plots.



Note: The stateful interface is intended to mimic the plotting format of MATLAB, a proprietary programming language.

Object-Oriented Interface

The alternative approach to plotting in Matplotlib is to use the object-oriented interface. This interface is stateless; it does not keep track of the "active" plot elements like the stateful interface does. Instead, you create objects that represent the plot and then use these objects to both generate and modify the plot, as needed.

Compared to the stateful approach, the object-oriented approach requires a little more code up front. You need to first construct objects of the `Figure` and `Axes` classes. Objects constructed from methods of these classes will have the necessary information to generate a plot. Consider the following example, which constructs the same plot as the stateful example earlier, but this time using the object-oriented interface:

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(revenue_df['Month'], revenue_df['Revenue'])
```

There are essentially three steps to this process:

- The `fig` object uses the `plt.figure()` method to construct an instance of the `Figure` class.
- The `ax` object uses the `fig` object from the previous line to construct an instance of the `Axes` class. The method being used here is `add_axes()`, which takes the dimensions of the axes.
- The `ax` object calls `plot()` to actually generate the plot.

So, rather than using two objects at once (one for `Figure` and one for `Axes`) to generate or modify the plot, you just need to use `ax`. When it comes time to modify this plot, you simply need to reference `ax`. Any other plots you create in this manner will exist as their own separate objects, so there's no need to worry about which plot is the "active" one. Ultimately, you should prefer to use the object-oriented interface over the stateful interface in most cases, especially when you have multiple, complex plots that will need to be modified.



Note: This course will prefer the object-oriented interface over the stateful interface in its examples, though both are shown in some cases.

The pyplot.savefig() Function

You can use the `pyplot.savefig()` function to save a plot as a file on the system. This is particularly useful if you need to share the plots with another person or application, as Jupyter Notebook or whatever other programming interface you're using may not be the ideal environment for demonstrating visuals. While you could technically use your browser to save the PNG file generated in Jupyter Notebook, the `savefig()` function gives you more formatting options. You can specify arguments like `dpi`, `quality`, and more. You can also generate the image in a specific file format through the `format` argument; alternatively, the function will infer which format you want based on the file name extension you provide. The following example saves a statefully created image to a vector format (SVG) in the current directory:

```
plt.savefig('revenue_df.svg')
```

To save a plot that was created using the object-oriented interface, call `savefig()` on the object that was created from `Figure`:

```
fig.savefig('revenue_df.svg')
```



Note: If you save a vector image and parts of the figure are cut off, add the `bbox_inches = 'tight'` argument.

Supported File Types

To get a list of all file types that `savefig()` supports on your system, enter the following code:

```
fig.canvas.get_supported_filetypes()
```

Guidelines for Creating and Saving Simple Line Plots



Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when you are creating and saving simple line plots in Matplotlib.

Create and Save Simple Line Plots

When creating and saving simple line plots:

- Use the `plot()` function to generate a simple line plot.
- Consider use the stateful interface for quick and simple plotting.
- Prefer to use the object-oriented interface for most scenarios.
- When executing from a script or shell, use the `pyplot.show()` function to actually show the plots.
- When using IPython, use the `%matplotlib` magic command to automatically open a window with the plots.
- Use the `savefig()` function to save a plot as a file.
 - Provide a relative or absolute path to the location where you want to save the file.
 - Provide the desired extension in the file name to automatically save the image as that file type.

ACTIVITY 6–1

Creating and Saving Simple Line Plots

Data Files

/home/student/DSTIP/Matplotlib/Creating Simple Plots.ipynb
 /home/student/DSTIP/Matplotlib/data/daily_revenue_by_month.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've gotten a feel for pandas' interface into Matplotlib, and you've explored some different types of plots, but you plan on eventually building more complex visuals that you can tweak to your liking. These visuals will not only help you and your fellow data scientists analyze the store data from multiple perspectives, but they'll also help you present your findings and conclusions to GCE's business managers in a way that they'll respond to. To begin with, you'll generate simple line graphs that show the ups and downs of revenue across entire months. You may be able to identify trends in how the stores experience lulls and growth periods in a 30-day window.

1. In Jupyter Notebook, open the **DSTIP/Matplotlib/Creating Simple Plots.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 Lines 15 and 16 load a CSV file of daily revenue per month.
 - b) Run the code cell.
 - c) Verify that the version of Python is displayed, as are the versions of NumPy, pandas, and Matplotlib that were imported.

```
Libraries used in this project:
- Python 3.7.6 (default, Jan 8 2020, 19:59:22)
[GCC 7.3.0]
- NumPy 1.18.1
- pandas 1.0.1
- Matplotlib 3.1.3
```

Loaded dataset.

3. Create line plots using the stateful interface.
 - a) Scroll down and view the cell titled **Create line plots using the stateful interface**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 month_rev.head(10)
```

- c) Run the code cell.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

- d) Examine the output.

Date	January	February	March
1	4,519.22	2,328.13	2,508.92
2	1,852.86	3,905.34	6,247.91
3	1,979.17	5,207.55	4,622.07
4	1,546.37	2,323.33	3,708.99
5	3,368.27	2,886.79	5,934.17
6	3,442.10	2,767.07	2,945.33
7	2,699.28	6,884.01	1,369.77
8	5,041.65	4,842.53	2,976.56
9	2,436.11	3,116.09	7,118.14
10	3,391.38	2,991.45	3,012.60

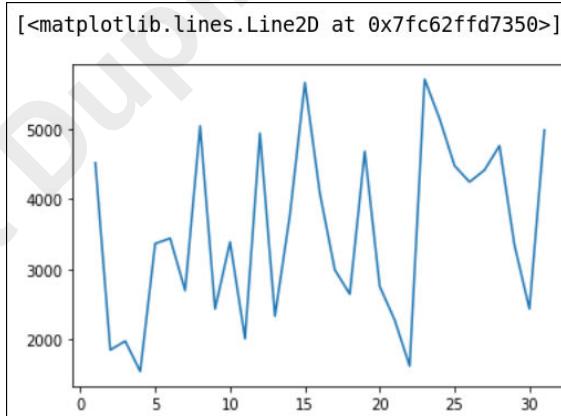
This DataFrame includes the total revenue for each day of the month for each month that was tracked in the GCE stores dataset.

- e) Select the next code cell, and then type the following:

```
1 plt.plot(month_rev.index, month_rev['January'])
```

This code uses the stateful interface to plot a simple line graph where:

- The row indices (the days) are on the x-axis.
 - The revenue figures for each day are on the y-axis.
- f) Run the code cell.
g) Examine the output.



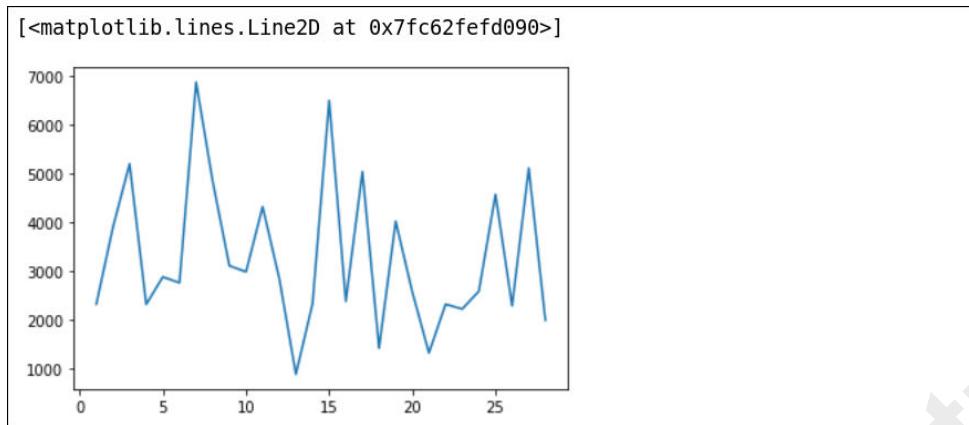
- h) Select the next code cell, and then type the following:

```
1 plt.plot(month_rev.index, month_rev['February'])
```

This code plots the same revenue figures per day, but for February.

- i) Run the code cell.

- j) Examine the output.



As you can see, generating these two plots is simple and can easily be done on one line. However, what if you wanted to change the January plot somehow? For instance, maybe you want to change the color of the plotting area. Because the January plot is no longer the "active" plot, you would need to write some code to "find" it, and then set the color. While doing so is possible, it can get messy and is prone to error. This is where the object-oriented approach comes into play.

4. Create line plots using the object-oriented interface.

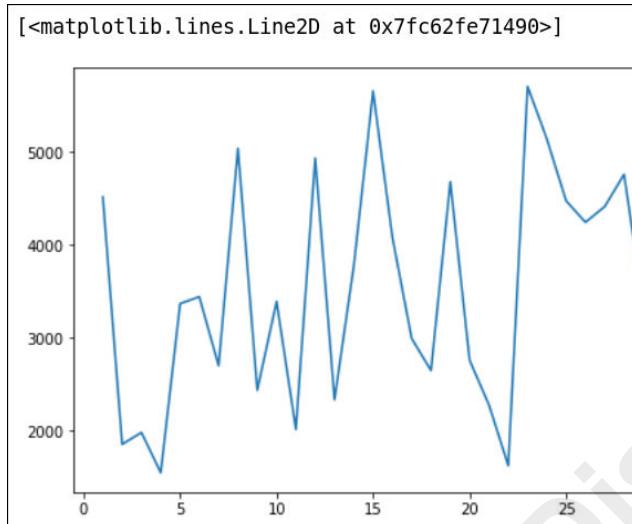
- Scroll down and view the cell titled **Create line plots using the object-oriented interface**, then select the code cell below it.
- In the code cell, type the following:

```
1 fig = plt.figure()
2 ax = fig.add_axes([0, 0, 1, 1])
3 ax.plot(month_rev.index, month_rev['January'])
```

This is one of several object-oriented approaches to plotting.

- Line 1 creates a `Figure` object.
 - Line 2 creates an `Axes` object by adding axes to the figure, given a set of coordinates.
 - Line 3 calls the `plot()` function on the `Axes` object, using the same parameters as before.
- Run the code cell.

- d) Examine the output.



The resulting plot is essentially the same as it was when you used the stateful interface. At this point, the biggest difference is that it took more lines of code to do the same thing. Although this is not the most common way to use the object-oriented interface, the other ways still require more code to just plot data than the stateful equivalent. Based on this alone, it might seem like the stateful interface is the better choice, but the object-oriented interface still has a key advantage.

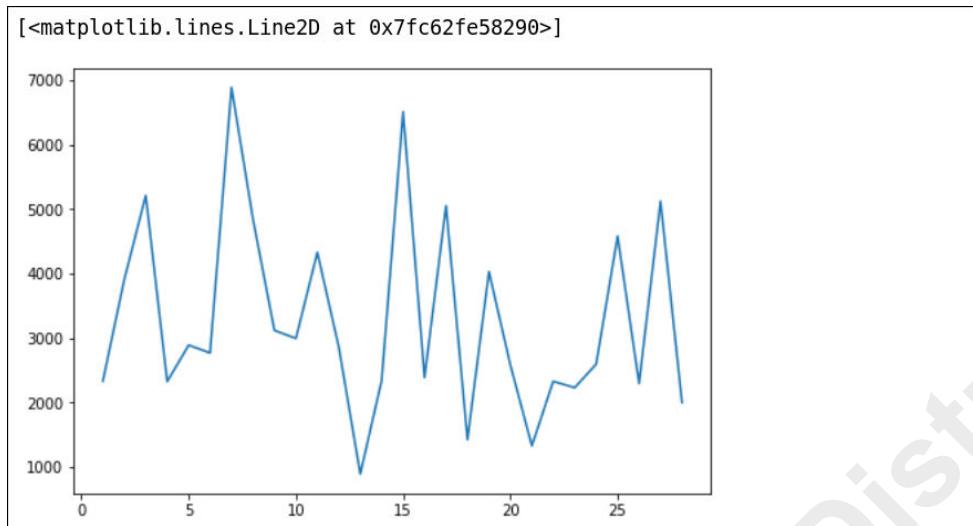
- e) Select the next code cell, and then type the following:

```
1 fig2 = plt.figure()  
2 ax2 = fig2.add_axes([0, 0, 1, 1])  
3 ax2.plot(month_rev.index, month_rev['February'])
```

This code creates a different `Figure` object with a different `Axes` object for the February revenue plot.

- f) Run the code cell.

- g) Examine the output.



Now that you have your objects, you can change the color of the January plot, even though it's not the "active" plot.

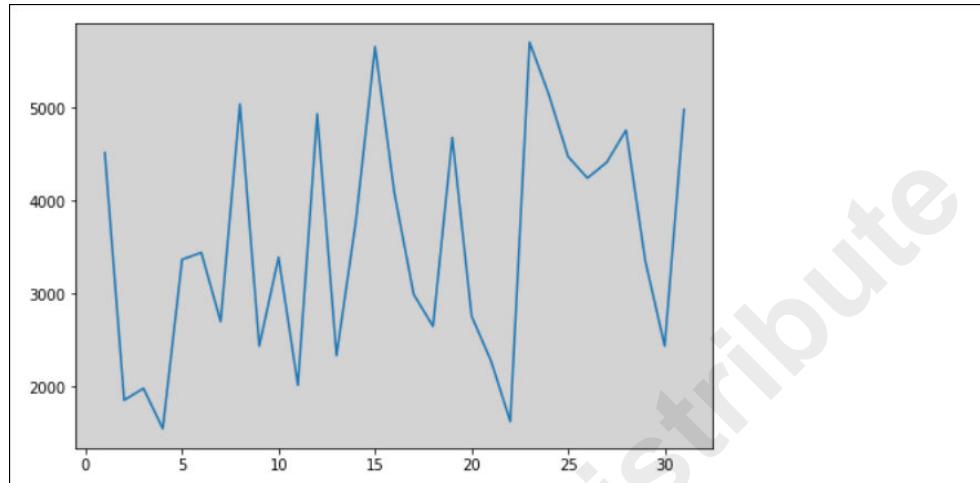
5. Update a plot using the object-oriented interface.

- Scroll down and view the cell titled **Update a plot using the object-oriented interface**, then select the code cell below it.
- In the code cell, type the following:

```
1 ax.set_facecolor('lightgrey')
2 fig # January
```

- Line 1 takes the `ax` object you created earlier—an `Axes` object for the January plot—and calls a method that sets a property. In this case, the property is the plot's face color, or the background color within the data portion of the plot.
 - Line 2 calls `fig`—the `Figure` object for the January plot—to actually show the plot in the output.
- c) Run the code cell.

- d) Examine the output.

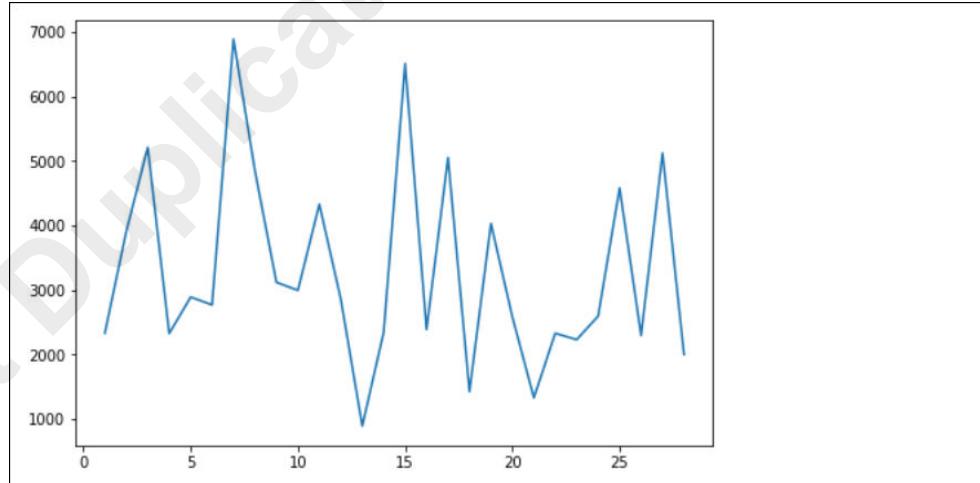


- e) Select the next code cell, and then type the following:

```
1 fig2 # February
```

This time, you're calling the `Figure` object for February.

- f) Run the code cell.
g) Examine the output.



As intended, the February plot's face color didn't change. You set the color property on only the January object. So, the object-oriented interface, although somewhat more verbose, can save you a lot of hassle when you construct more complex plots that require modification.

6. Save a plot as a file.

- a) Scroll down and view the cell titled **Save a plot as a file**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 fig.savefig('jan_revenue.svg', bbox_inches = 'tight')
2 print('Plot saved to file.')
```

The file will automatically be saved in SVG format because of the specified file extension.

- c) Run the code cell.
d) Examine the output.

```
Plot saved to file.
```

7. Open the saved image file.

- a) From the Linux desktop, open the file manager.
b) Navigate to `/home/student/DSTIP/Matplotlib/`.
c) Double-click `jan_revenue.svg` to open it.
d) Verify that the plot was saved correctly as an image.
e) Close the file when you're done.

8. Shut down this Jupyter Notebook kernel.

- a) Return to Jupyter Notebook.
b) From the menu, select **Kernel**–**Shutdown**.
c) In the **Shutdown kernel?** dialog box, select **Shutdown**.
d) Close the **Creating Simple Plots** tab in Firefox, but keep a tab open to **DSTIP/Matplotlib/** in the file hierarchy.

TOPIC B

Create Subplots

Now that you know how to generate individual plots in Matplotlib, you can start generating more advanced layouts. Although one plot by itself may be enough in some situations, you'll find that visualizing data from different perspectives can greatly enhance your ability to communicate results to your audience.

Modular Plotting

Plotting one graph after another might be sufficient in simple cases, but there will be times when you want to generate multiple plots within the same figure. This type of plotting is done modularly using multiple `plot()` calls, rather than a single `plot()` call with multiple sets of data. For example, let's say you have an income statement that tracks business revenue and expenses. The `income_df` object is a multi-indexed DataFrame where each row is a month, and each sub row is a year:

	Revenue	Expenses
Jan 2019	2.0	1.8
2020	2.1	1.8
Feb 2019	3.2	2.4
2020	3.2	2.5
...		

You want to be able to compare revenue performance between years. Rather than making one plot for 2019 and one plot for 2020, you can combine them into a single figure, where each year is given its own line. In the following code, the multi-indexed DataFrame is sliced for each year, then the first level of row indices (the month) is put on the x-axis, and revenue is put on the y-axis:

```
slice_2019 = income_df.loc[pandas.IndexSlice[:, 2019], :]
slice_2020 = income_df.loc[pandas.IndexSlice[:, 2020], :]
plt.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
plt.plot(slice_2020.index.get_level_values(0), slice_2020['Revenue'])
```

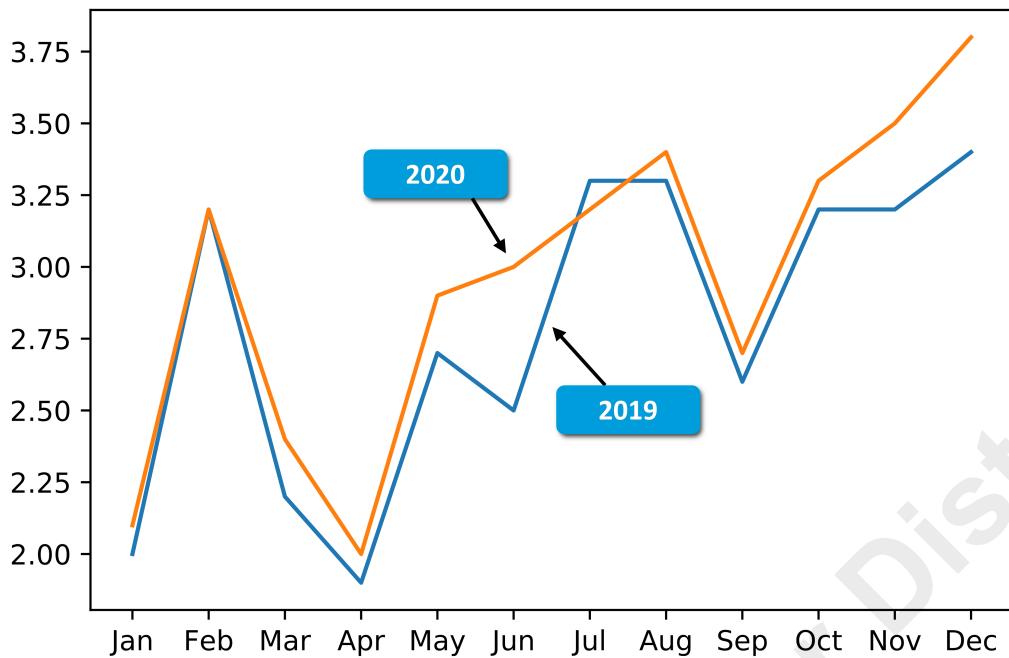


Figure 6-2: Notice that 2020 revenue largely exceeds 2019 revenue on a month-by-month basis.

You can also generate modular figures by using the object-oriented approach. The following generates a line graph comparing revenue to expenses over the same year:

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
ax.plot(slice_2019.index.get_level_values(0), slice_2019['Expenses'])
```

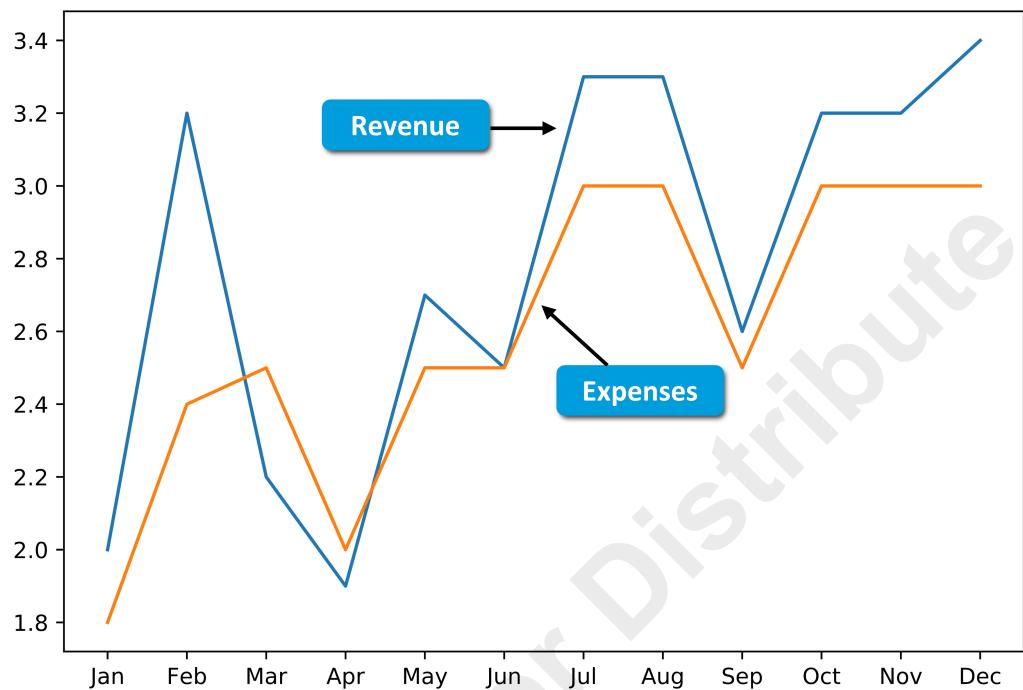


Figure 6–3: Notice that 2019 revenue largely exceeds 2019 expenses on a month-by-month basis.

Showing Plots Again in Jupyter Notebook

Using multiple `plot()` calls within the same Notebook document cell will automatically merge all plots into a single modular figure. However, if you call `pyplot.plot()` in a different cell later on by using the stateful interface, a new figure will be generated with only that plot data. This makes the new plot the "active" one. If you're using the object-oriented interface, adding more `ax.plot()` statements to later cells will plot the data on the same modular figure, but it won't be shown automatically. You need to call the `Figure` object you created initially to get Jupyter Notebook to display the figure again. For example:

```
fig = plt.figure() # Construct Figure object.
... # Additional plotting code.
```

Then, in a new cell:

```
fig
```

Subplotting

Modular plotting is great for combining like data into a single figure for analysis. However, there's only so much room you can get out of a single plot, and the major drawback of this approach is the risk of a figure becoming too cluttered or noisy. Consider that you're working with `income_df` and want to compare revenue between 2019 and 2020, but you also want to compare expenses between 2019 and 2020. You could certainly add four separate lines to a single figure, but this can quickly become confusing and hard to read. Also, some data is in different scales or different formats, so plotting it all on a single figure might not make sense.

Another approach is **subplotting**. In a subplotting scheme, you have multiple smaller plots with their own axes, and each one of these smaller plots forms the larger figure. So, using the previous example, you could have four line graph subplots:

1. Revenue for 2019

2. Revenue for 2020
3. Expenses for 2019
4. Expenses for 2020

Subplotting is not necessarily an alternative to modular plotting, as they can work together—in other words, you can make modular subplots. Instead of four subplots, you might have two: one for comparing revenue between 2019 and 2020; and one for comparing expenses between 2019 and 2020.

The Figure.add_axes() Function

The `Figure.add_axes()` function shown earlier is one way of creating subplots by using the object-oriented interface. You can call this function multiple times on individual `Axes` objects, where each call is a different subplot. The main argument this function takes is `rect`, which accepts a list of the following four values, in order:

1. The position of the left side of the plot.
2. The position of the bottom side of the plot.
3. The width of the plot.
4. The height of the plot.

Each of these values is a fraction of the overall figure size. The following example sets up two subplots of the same size, where `ax1` will be positioned to the left, and `ax2` will be positioned to the right:

```
fig = plt.figure()
ax1 = fig.add_axes([0, 1, 1, 1])
ax2 = fig.add_axes([1.1, 1, 1, 1])
```

Once you've added the sufficient number of axes, you need to plot them as desired. The following example plots two subplots, each one modular:

```
ax1.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
ax1.plot(slice_2020.index.get_level_values(0), slice_2020['Revenue'])
ax2.plot(slice_2019.index.get_level_values(0), slice_2019['Expenses'])
ax2.plot(slice_2020.index.get_level_values(0), slice_2020['Expenses'])
```



Figure 6-4: The generated figure with two subplots side by side.

Manually adding axes and adjusting their positions can be tedious, which is why `add_axes()` is less commonly used than functions like `add_subplot()` and `subplots()`.

The pyplot.axes() Function

The `pyplot.axes()` function is the stateful interface equivalent of `Figure.add_axes()`. For example:

```
plt.axes([0, 1, 1, 1])
plt.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
... # And so on.
```

The Figure.add_subplot() Function

The object-oriented `Figure.add_subplot()` function enables you to define an arrangement of subplots within an overall table, or grid. Rather than specify the exact positioning of each subplot, you instead pass this function three integers as arguments:

- The total number of rows of subplots.
- The total number of columns of subplots.
- The position of the specific subplot you're working with. Positions start with 1 at the top-left subplot, then run right until reaching the bottom-right subplot.

The `income_df` DataFrame has been expanded so that each expense is itemized:

	COGS	Payroll	Rent	Utilities	Maintenance	Interest	Taxes
Month Year							
Jan	2019	0.60	0.90	0.03	0.01	0.01	0.20
	2020	0.60	0.60	0.05	0.03	0.02	0.30
Feb	2019	0.65	1.40	0.05	0.03	0.02	0.20
	2020	0.65	1.15	0.05	0.03	0.02	0.40

Let's say you want to plot revenue, cost of goods sold (COGS), payroll, and utilities for 2019. That's four subplots, so you'll create a 2×2 grid. You can begin by defining each axis:

```
fig = plt.figure(figsize = (10, 6))
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)
```

Each axis refers to the subplot in a specific position. To plot each column, you can call `plot()` on each axis object as you normally would:

```
ax1.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
ax2.plot(slice_2019.index.get_level_values(0), slice_2019['COGS'])
ax3.plot(slice_2019.index.get_level_values(0), slice_2019['Payroll'])
ax4.plot(slice_2019.index.get_level_values(0), slice_2019['Utilities'])
```

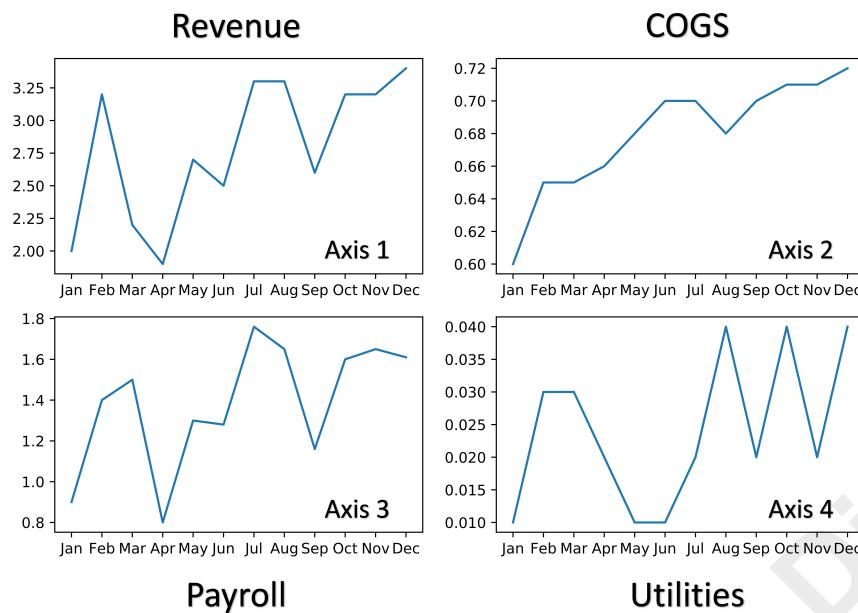


Figure 6-5: The generated subplot grid.

Although the `add_subplot()` function alleviates the need to add axes with specific dimensions, it can still get tedious. This is especially true if you intend to add many subplots.



Note: The `figsize` argument for creating a `Figure` object specifies the width and height of the entire figure. This may be necessary to make the resulting subplot grid look clean.

The `pyplot.subplot()` Function

The `pyplot.subplot()` function is the stateful interface equivalent of `Figure.add_subplot()`. For example:

```
plt.subplot(2, 2, 1)
plt.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
... # And so on.
```

The `Figure.subplots()` Function

To minimize the tediousness of defining each subplot axis separately, the `Figure.subplots()` function (note the plural) defines the overall grid and then returns a NumPy array for the `Axes` object. In other words, you don't need to assign a unique object to each individual subplot; the subplots are cells in the array. This time, you want to create a 3×2 subplot grid:

```
fig = plt.figure(figsize = (10, 8))
ax = fig.subplots(3, 2)
```

Because this is an array, you can access (and plot) each subplot through the same NumPy indexing you're familiar with. Remember that the subplot grid is in a tabular format, so the top-left subplot is $[0, 0]$; the subplot in the column to the right is $[0, 1]$; and so on.

```
ax[0, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
ax[0, 1].plot(slice_2019.index.get_level_values(0), slice_2019['COGS'])
ax[1, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Payroll'])
ax[1, 1].plot(slice_2019.index.get_level_values(0), slice_2019['Utilities'])
```

```
ax[2, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Interest'])
ax[2, 1].plot(slice_2019.index.get_level_values(0), slice_2019['Taxes'])
```

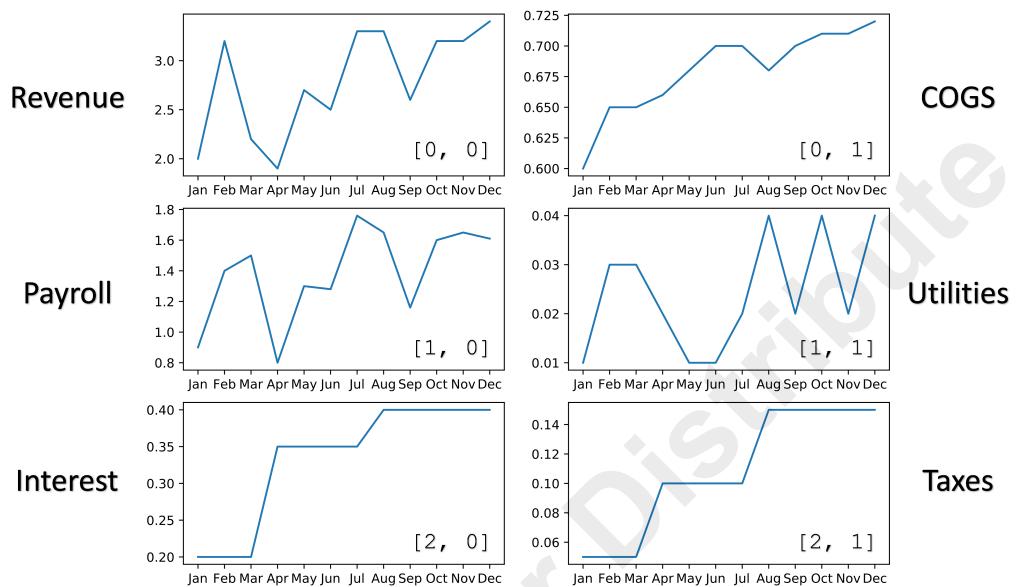


Figure 6-6: The generated subplot grid.

The `pyplot.subplots()` Function

The `pyplot.subplots()` function is the stateful interface equivalent of `Figure.subplots()`. For example:

```
plt.subplots(3, 2)
plt.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
... # And so on.
```

Single-Line Unpacking

You can define subplots on a single line by unpacking `plt.subplots()`. The first variable assignment will take on a `Figure` object, and the second will take on a NumPy array of `Axes` objects. For example:

```
fig, ax = plt.subplots(2, 2, figsize = (10, 8))
ax[0, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
... # And so on.
```

Shared Axes

Looking at the previous figure, you can see that the subplots show the rise and fall of each expense within its own context. If you were to compare one expense with another, it would be to compare how sharply each expense increases or decreases. But, what if you want to compare the overall spending value of each expense? For example, utility expenses seem to have spiked up and down throughout 2019, but the cost of utilities is on a much smaller scale than the cost of goods, which has a relatively steady increase. To compare each expense based on overall spending, you must place the subplots on a shared axis.

The `subplots()` function has two arguments for this: `sharex` and `sharey`. For each axis, you specify one of four options:

- 'none' —The specified axis will be independent for each subplot (the default behavior).
- 'all' —The specified axis will be shared between all subplots.

- 'row' —The specified axis will be shared between each row.
- 'col' —The specified axis will be shared between each column.

In the following example, all of the subplots in each column will have the same x-axis (months), whereas all of the subplots in each row will have the same y-axis (millions of dollars).

```
fig = plt.figure(figsize = (10, 6))
ax = fig.subplots(2, 2, sharex = 'col', sharey = 'row')

ax[0, 0].plot(slice_2019.index.get_level_values(0), slice_2019['COGS'])
ax[0, 1].plot(slice_2019.index.get_level_values(0), slice_2019['Utilities'])
ax[1, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Interest'])
ax[1, 1].plot(slice_2019.index.get_level_values(0), slice_2019['Taxes'])
```

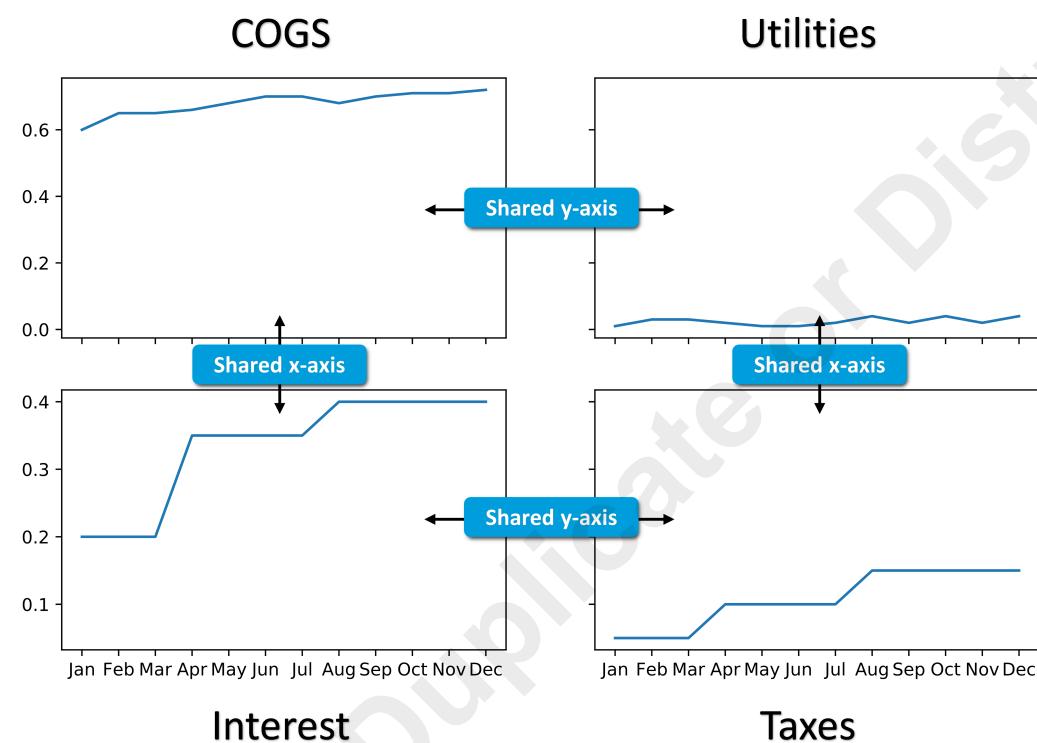


Figure 6-7: Each subplot in a row is plotted on the same monetary scale, making it easier to compare overall spending values.

Sharing axes also has the effect of cleaning up the figure labels. In the example, the x-axis tick labels appear only at the bottom row, and the y-axis tick labels appear only on the left. There's no need to repeat them, since they're being shared along their respective rows or columns.

	Note: When an axis with numerical values is shared between plots of different scales, the shared axis will adapt to the largest scale.
--	---

The GridSpec Object

The `GridSpec` object, provided by the `matplotlib.gridspec` module, enables more fine-tuned control over the layout of subplots on a grid. Rather than defining the plots, it defines the grid geometry, which you can manipulate as needed when it comes time to plot. You can create a `GridSpec` object through the object-oriented interface by calling the `add_gridspec()` function.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2019

This function takes the shape of the grid with some optional arguments that define layout spacing. In this example, the grid will be 2×2 :

```
fig = plt.figure(figsize = (13, 6))
grid = fig.add_gridspec(2, 2, width_ratios = [.6, .4])
```

The `width_ratios` argument defines the width of each column as a ratio; in this example, the first column will be somewhat wider than the second. You could also use the `height_ratios` argument to define row height. Additionally, you can pass values to `wspace` and `hspace` to control the width and height space between each subplot, expressed as a fraction of the overall axis width or height.

Once the overall grid is set up, you need to set up your subplots on the grid. The simplest way is to use the `add_subplot()` function for each desired subplot. In each call, pass the `GridSpec` object you created, indexing that object according to the grid spaces it should take up. In the following example, only three subplots are generated (despite the grid being 2×2). This is because the first subplot will take up two grid spaces.

```
ax1 = fig.add_subplot(grid[0:, 0])
ax2 = fig.add_subplot(grid[0, 1])
ax3 = fig.add_subplot(grid[1, 1])
```

Each `grid` argument is indexed by row and column. The first subplot, `ax1`, starts at row 0, but also uses a slice to state that it should span *all rows*. It will span only column 0, however. The other two subplots will occupy the first row, second column (`ax2`); and second row, second column (`ax3`).

Lastly, plot each subplot as you normally would:

```
ax1.plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
ax2.plot(slice_2019.index.get_level_values(0), slice_2019['Payroll'])
ax3.plot(slice_2019.index.get_level_values(0), slice_2019['COGS'])
```

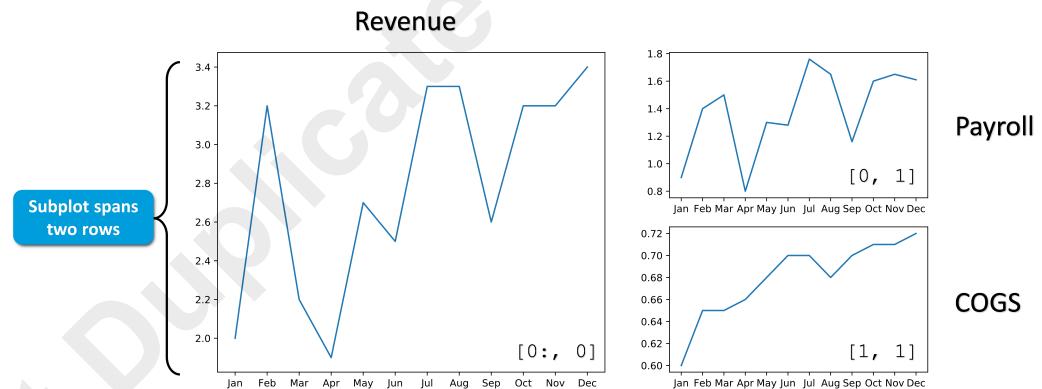


Figure 6-8: Revenue is given prominence in the grid, while COGS and payroll are supplementary.

There are many potential scenarios that could warrant a `GridSpec` instead of a standard subplot layout. For example, if you have a spread of numerical data across both x- and y-axes, you could make a scatter plot the main visualization, while also plotting smaller histograms for each axis to demonstrate the distribution of data in different ways.

Using GridSpec with the `subplots()` Function

To generate a `GridSpec` while using the `subplots()` function, you can pass a dictionary of `GridSpec` arguments to the `gridspec_kw` argument. The rest of the construction is similar to how you would normally use `subplots()`:

```
ax = fig.subplots(2, 2, gridspec_kw = {'width_ratios': [.6, .4]})
ax[0, 0].plot(slice_2019.index.get_level_values(0), slice_2019['Revenue'])
... # And so on.
```

However, it's important to note that you cannot use `subplots()` by itself to make a subplot span multiple columns or rows on the grid. To do this, you need to manually remove the axes you want the subplot to span, then use `add_subplot()` to add the subplot that should span those axes. You can then safely construct the rest of the subplots by using `subplots()`.

Guidelines for Creating Subplots

Follow these guidelines when you are creating subplots in Matplotlib.

Create Subplots

When creating subplots:

- Use modular plotting to compare multiple related data visualizations on the same graph.
- Avoid adding too many visualizations to a modular plot, which can create noise and make the plot hard to read.
- Use subplotting to compare multiple related data visualizations in self-contained plots.
- Consider subplotting when modular plotting gets too noisy.
- Combine both modular plotting and subplotting to get the most out of your figure layouts.
- Prefer to use the `Figure.subplots()` function in most cases when adding subplots.
- Consider plotting subplots with shared axes to reduce visual clutter or to change the scales of the plots.
- Construct a `GridSpec` object if you want to exercise more granular control over how subplots are laid out.
 - Prefer to use the `Figure.add_subplot()` function when adding subplots to a `GridSpec` object.

ACTIVITY 6–2

Creating Subplots

Data Files

/home/student/DSTIP/Matplotlib/Creating Subplots.ipynb
/home/student/DSTIP/Matplotlib/data/daily_accounting_by_month.csv

Before You Begin

Jupyter Notebook is open.

Scenario

While you begin to think of more ways to plot the GCE store data for analysis and presentation, you need to consider how those plots will be laid out. Rather than always show a series of individual, disconnected plots, some plots will benefit from being combined into an overall layout. That way, they can be viewed within the proper context, enriching their ability to reveal insights. So, you'll start by modularizing and combining your line graphs into multiple subplots that are part of a whole figure.

1. In Jupyter Notebook, open the **DSTIP/Matplotlib/Creating Subplots.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
Lines 15 and 16 load a CSV file of daily accounting figures for the first three months.
 - b) Run the code cell.
3. Generate a modular plot showing revenue for both January and February.
 - a) Scroll down and view the cell titled **Generate a modular plot showing revenue for both January and February**, then select the code cell below it.
 - b) In the code cell, type the following:

```
1 month_acc.head(10)
```

- c) Run the code cell.

d) Examine the output.

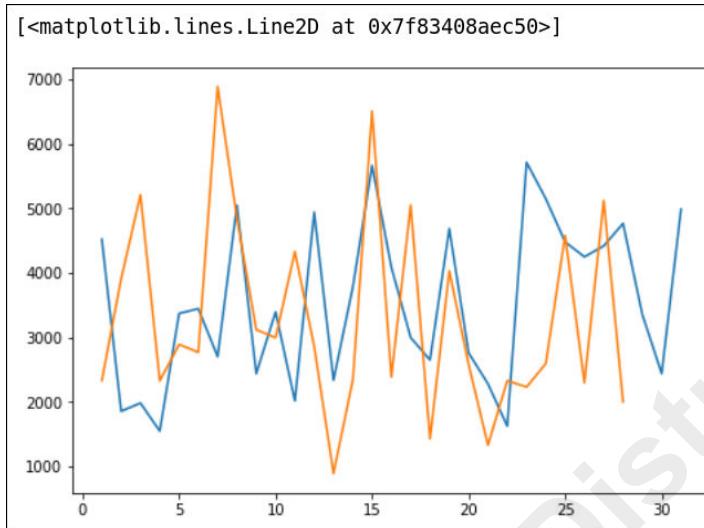
Date	Revenue			COGS			GrossIncome		
	January	February	March	January	February	March	January	February	March
	1	4,519.22	2,328.13	2,508.92	4,365.20	2,243.63	2,402.36	154.02	84.50
2	1,852.86	3,905.34	6,247.91	1,790.87	3,758.41	6,007.21	61.99	146.93	240.70
3	1,979.17	5,207.55	4,622.07	1,912.05	4,968.73	4,456.87	67.12	238.82	165.20
4	1,546.37	2,323.33	3,708.99	1,475.36	2,228.72	3,564.98	71.01	94.61	144.01
5	3,368.27	2,886.79	5,934.17	3,256.40	2,777.05	5,714.64	111.87	109.74	219.53
6	3,442.10	2,767.07	2,945.33	3,295.05	2,671.94	2,826.71	147.05	95.13	118.62
7	2,699.28	6,884.01	1,369.77	2,596.96	6,599.44	1,316.44	102.32	284.57	53.33
8	5,041.65	4,842.53	2,976.56	4,848.61	4,655.42	2,858.24	193.04	187.11	118.32
9	2,436.11	3,116.09	7,118.14	2,350.42	3,002.26	6,883.06	85.69	113.83	235.08
10	3,391.38	2,991.45	3,012.60	3,268.03	2,892.65	2,899.90	123.35	98.80	112.70

- This is a multi-indexed DataFrame based on `stores_df` that compiles the total revenue, COGS, and gross income for each day, across all three represented months.
 - The columns have multiple levels, where each account figure is the first level, and each month is the second level.
 - As before, each day is a row index.
- e) Select the next code cell, and then type the following:

```
1 fig = plt.figure()
2 ax = fig.add_axes([0, 0, 1, 1])
3 ax.plot(month_acc.index, month_acc.loc[:, ('Revenue', 'January')])
4 ax.plot(month_acc.index, month_acc.loc[:, ('Revenue', 'February')])
```

- Line 2 uses the `add_axes()` function to create an `Axes` object from a `Figure`.
 - Lines 3 and 4 both plot data on the same `Axes` object: one plot for January, one for February.
- f) Run the code cell.

- g) Examine the output.



- Because both `plot()` functions were called on the same `Axes` object, the data points show up on the same plot.
- The blue line is the revenue for January, and the orange line is the revenue for February.
- Both months seem to have a few days or spans of days that have similar peaks or valleys—day 15 and day 19, for example.
- Some days deviate significantly, such as day 28.
- Overall, February seems to have achieved higher highs and lower lows.
- Since February had 28 days in 2019, its line stops before January's.

4. Why would you want to avoid adding March to this modular plot?

5. Generate subplots comparing revenue across the months.

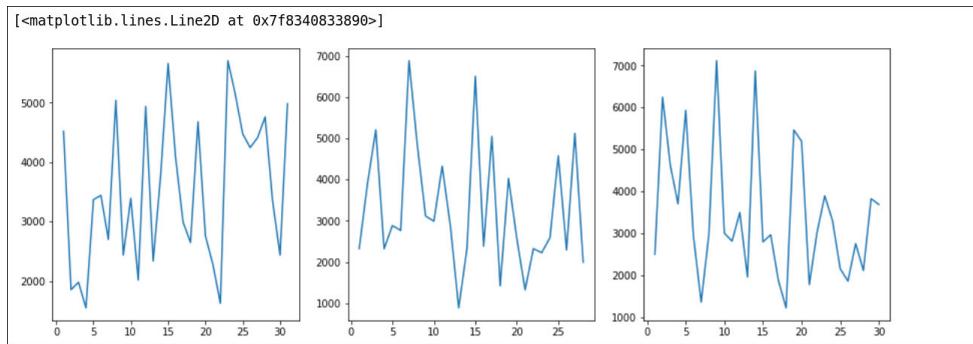
- Scroll down and view the cell titled **Generate subplots comparing revenue across the months**, then select the code cell below it.
- In the code cell, type the following:

```
1 fig, ax = plt.subplots(1, 3, figsize = (15, 5))
2 ax[0].plot(month_acc.index, month_acc.loc[:, ('Revenue', 'January')])
3 ax[1].plot(month_acc.index, month_acc.loc[:, ('Revenue', 'February')])
4 ax[2].plot(month_acc.index, month_acc.loc[:, ('Revenue', 'March')])
```

- Line 1 uses the `subplots()` function with single-line unpacking to create a grid of shape 1×3 .
- Lines 2 through 4 set up each subplot in the grid, where the first subplot is the revenue for January; the second, the revenue for February; and the third, the revenue for March.
- Note how each subplot is referenced in the `Axes` object by using its index. In this case, because there's only one row, you need to provide only a single index value.

- Run the code cell.

- d) Examine the output.



- Each month's revenue is arranged as a different subplot on a grid, in the order of January, then February, and then March.
- One example observation you could draw from this is that the revenue for the first week of January seemed to take a dip, while for February it held a little more steady.
- This might be easier to read than if you had placed every month on the same modular plot, but it isn't necessarily the best way to plot the data.

6. Generate subplots comparing performance for January vs. February.

- a) Scroll down and view the cell titled **Generate subplots comparing performance for January vs. February**, then select the code cell below it.
b) In the code cell, type the following:

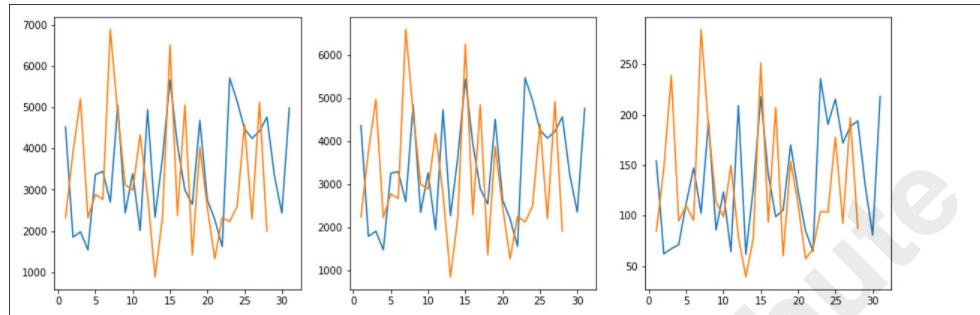
```

1 fig, ax = plt.subplots(1, 3, figsize = (15, 5))
2
3 measures = ['Revenue', 'COGS', 'GrossIncome']
4
5 col = 0
6
7 for measure in measures:
8     ax[col].plot(month_acc.index, month_acc.loc[:, (measure, 'January')])
9     ax[col].plot(month_acc.index, month_acc.loc[:, (measure, 'February')])
10    col += 1

```

- Lines 7 through 10 use a `for` loop to iterate over each accounting measurement and assign them to a subplot. You could plot each subplot individually line by line, but sometimes it's more concise to use a loop.
 - Line 8 plots the current subplot using the current measurement for January.
 - Line 9 plots the current subplot using the current measurement for February.
 - In other words, this loop will create modular subplots.
- c) Run the code cell.

- d) Examine the output.



- This leverages the advantages of both modular plotting and subplotting.
- From left to right, the measurements are: revenue, COGS, and gross income.
- This way, it's easier to compare each measurement in the context of month, rather than make each measurement for each month a separate subplot.
- One example observation you could make is that the trends for both revenue and COGS, for both January and February, are mostly consistent.

7. Generate subplots with shared axes.

- Scroll down and view the cell titled **Generate subplots with shared axes**, then select the code cell below it.
- In the code cell, type the following:

```

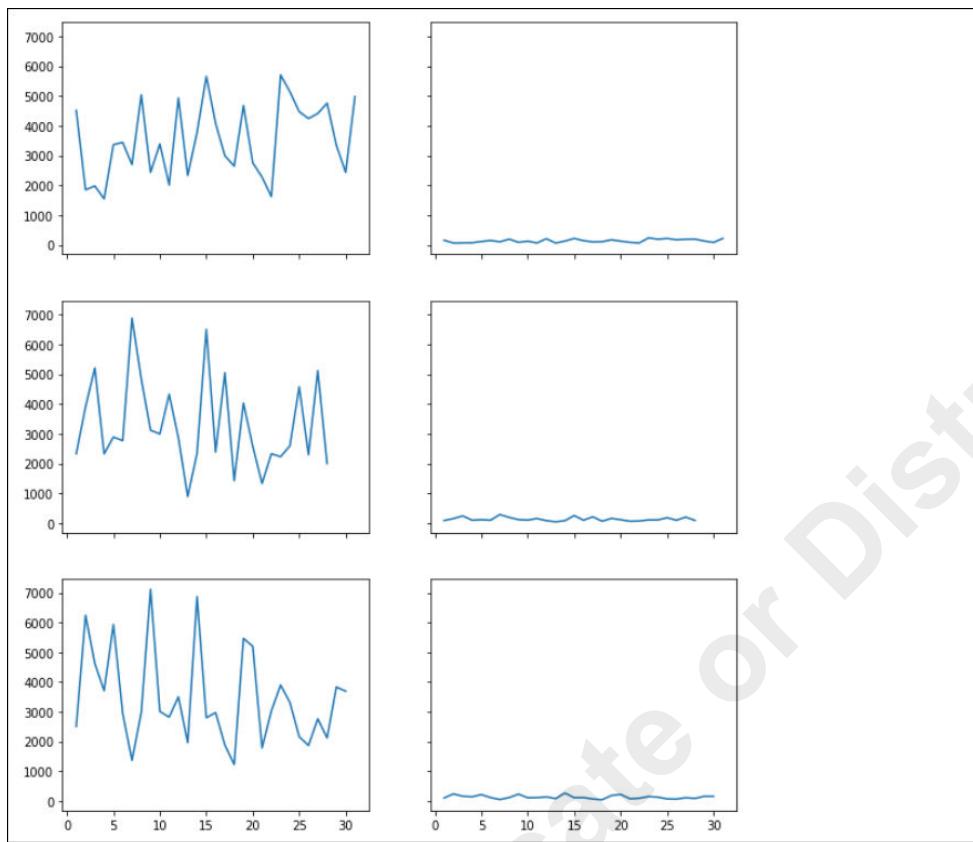
1 fig, ax = plt.subplots(3, 2, sharex = 'col', sharey = 'all',
2                         figsize = (10, 12))
3
4 months = ['January', 'February', 'March']
5 row = 0
6
7 for month in months:
8     ax[row, 0].plot(month_acc.index,
9                      month_acc.loc[:, ('Revenue', month)])
10    ax[row, 1].plot(month_acc.index,
11                      month_acc.loc[:, ('GrossIncome', month)])
12    row += 1

```

- Line 1 sets up a 3×2 Figure with shared axes.
- The x-axis will be shared per column, and the y-axis will be shared for all subplots.
- Lines 7 through 12 use a similar `for` loop as before, but iterate through months instead of measurements.
- Lines 8 and 9 plot the current row, column 0, using the current month's revenue.
- Lines 10 and 11 plot the current row, column 1, using the current month's gross income.

- c) Run the code cell.

- d) Examine the output.



- The result is a grid where the revenue for a month is in the left column, and the gross income for that month is in the right column.
- The rows are in order of month: January, February, and March.
- The bottom row of each column shows the day axes, but the rows above it do not. Since the months have more or less the same number of days, sharing the axes makes sense.
- The subplots all have the same y-axis scale: from 0 to 7,000. This means that the gross income plots, which never reach above 1,000, are shown with much less magnitude than revenue. You might want to do this if you're interested in comparing total sales figures, rather than comparing relative trends between each measurement. If the axes weren't shared, the gross income plots would look similar to revenue, which might give the false impression that income is in the same range as revenue.

8. Generate more complex subplots using a `GridSpec` object.

- a) Scroll down and view the cell titled **Generate more complex subplots using a `GridSpec` object**, then select the code cell below it.

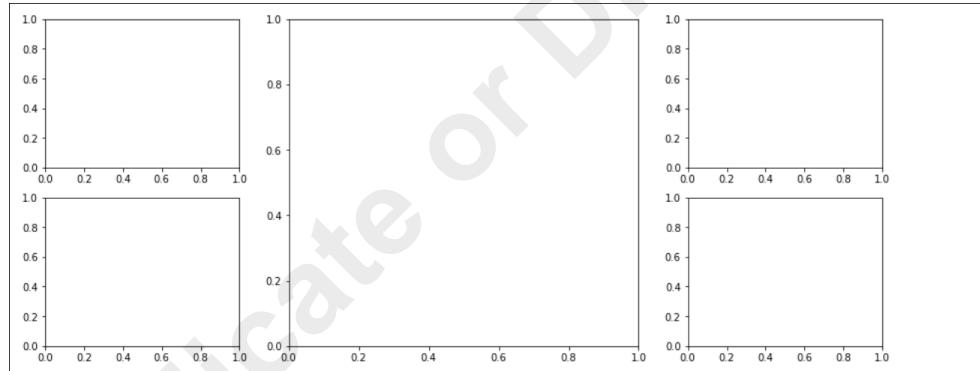
- b) In the code cell, type the following:

```

1 # Set up grid before plotting.
2 fig = plt.figure(figsize = (15, 6))
3 grid = fig.add_gridspec(2, 3, width_ratios = [.25, .45, .25])
4
5 ax_l1 = fig.add_subplot(grid[0, 0])
6 ax_l2 = fig.add_subplot(grid[1, 0])
7 ax_mid = fig.add_subplot(grid[:, 1])
8 ax_r1 = fig.add_subplot(grid[0, 2])
9 ax_r2 = fig.add_subplot(grid[1, 2])

```

- Line 3 adds a `GridSpec` object to the `Figure` created in line 2. It sets up the overall shape of the grid, as well as the relative widths of each column. In this case, the middle column will be the widest.
 - Lines 5 through 9 add subplots to the grid one by one. Note that `ax_mid` is using a slice to span the entire row of the middle column.
- c) Run the code cell.
d) Examine the output.



The grid itself is shown, though you haven't done any plotting yet. Notice how the middle subplot spans both rows and is made much more prominent than the others.

- e) Select the next code cell, and then type the following:

```

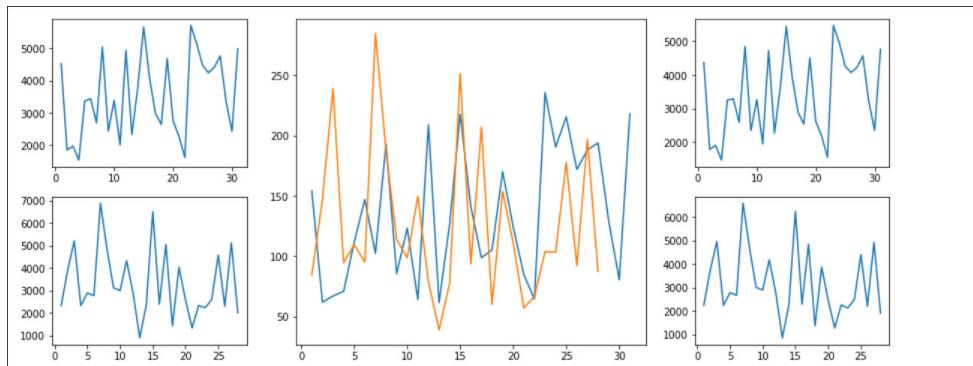
1 ax_l1.plot(month_acc.index, month_acc.loc[:, ('Revenue', 'January')])
2 ax_l2.plot(month_acc.index, month_acc.loc[:, ('Revenue', 'February')])
3 ax_mid.plot(month_acc.index, month_acc.loc[:, ('GrossIncome', 'January')])
4 ax_mid.plot(month_acc.index, month_acc.loc[:, ('GrossIncome', 'February')])
5 ax_r1.plot(month_acc.index, month_acc.loc[:, ('COGS', 'January')])
6 ax_r2.plot(month_acc.index, month_acc.loc[:, ('COGS', 'February')])
7 fig

```

This code uses the `Axes` objects you just created for the grid to plot the data.

- f) Run the code cell.

g) Examine the output.



- The plotted data has filled in the grid.
- The two smaller subplots on the left show revenue for January (top) and February (bottom).
- The two smaller subplots on the right show COGS for January (top) and February (bottom).
- The large middle plot shows the gross income for both January and February.
- This overall plot helps to tell the story of how gross income fluctuated over both months—which may be of most concern—while also showing the constituent measurements that led to that income, albeit in a more subdued way. How you shape and size your plots can have a significant impact on the message they communicate.

9. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel**–**Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Creating Subplots** tab in Firefox, but keep a tab open to **DSTIP/Matplotlib** in the file hierarchy.

TOPIC C

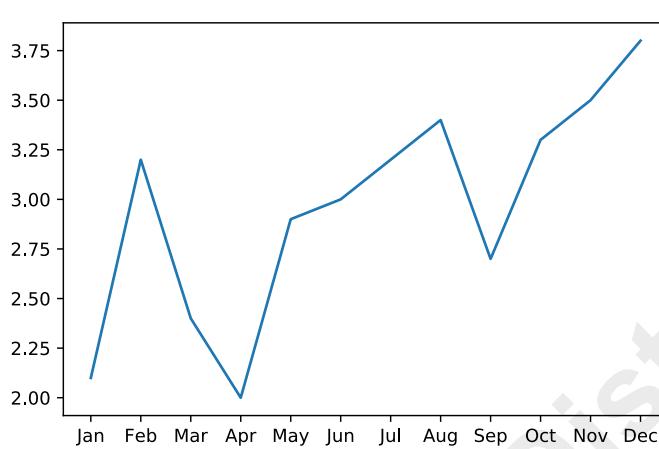
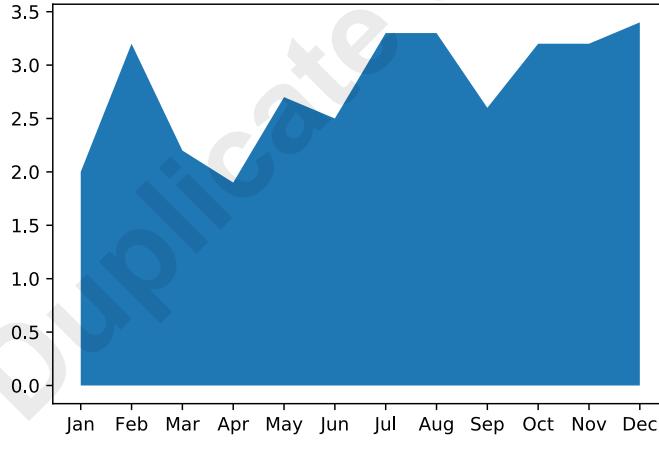
Create Common Types of Plots

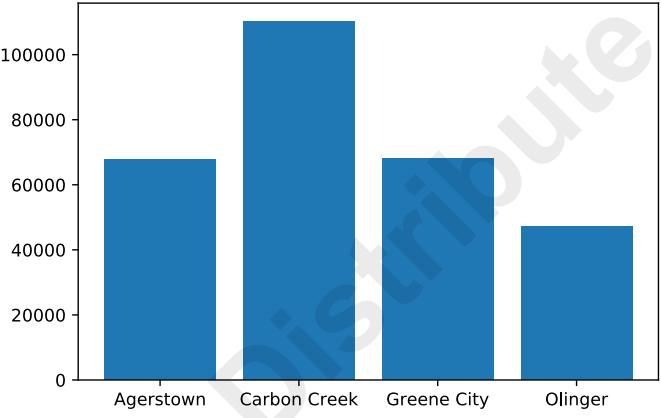
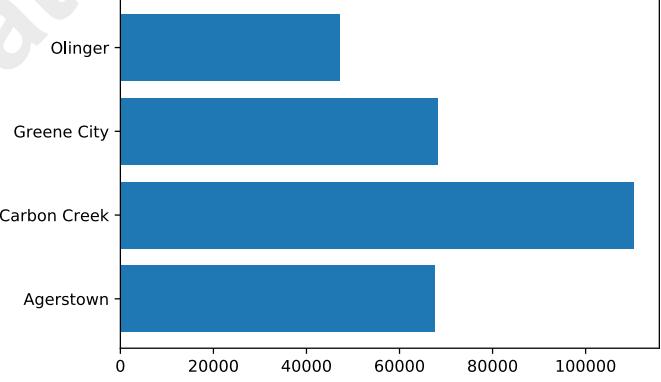
Now that you're comfortable laying out plots in different configurations, you can start generating specific types of plots. There are many plots to choose from; this topic will demonstrate the most common.

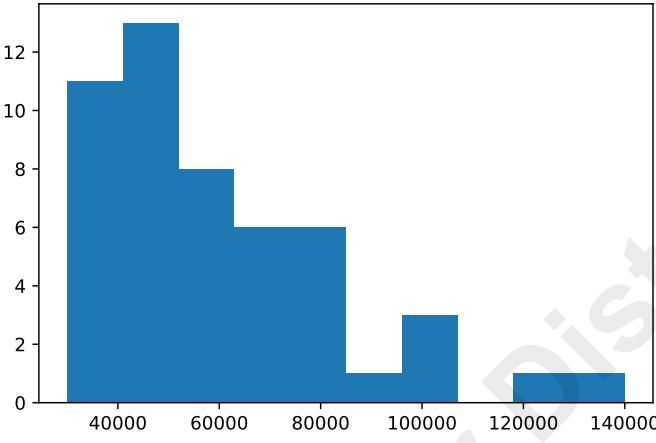
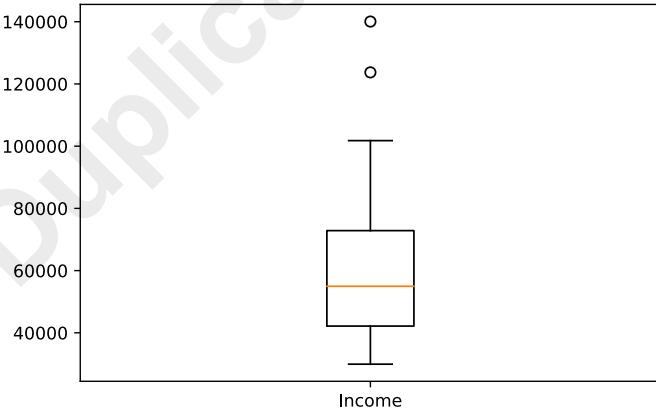
Common Plot Types

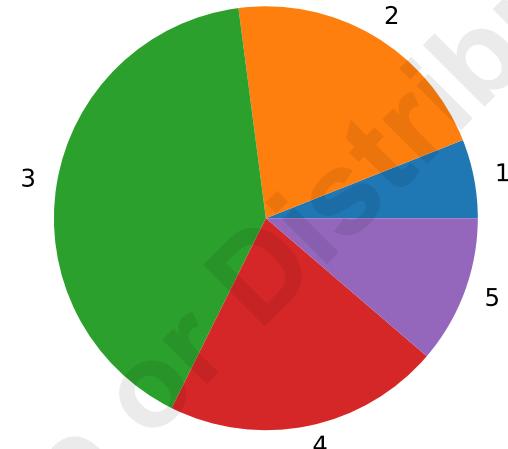
Recall that pandas offers the `DataFrame.plot()` function as an interface into a Matplotlib back end. So, all of those plot types and more are available for direct use within Matplotlib. The following table lists all of the plot types featured earlier, but it also describes how to use them from Matplotlib directly. You'll notice that Matplotlib provides more arguments for customizing each plot type as compared to `DataFrame.plot()`.

Plot Type Function	Optional Arguments of Note and Example
<code>pyplot.scatter(x, y)</code>	<ul style="list-style-type: none"> • <code>s</code> —Marker size. • <code>c</code> —Marker color. • <code>marker</code> —Marker style.

Plot Type Function	Optional Arguments of Note and Example																										
<code>pyplot.plot(x, y)</code>	<ul style="list-style-type: none"> • <code>fmt</code> —Formats color and/or appearance of lines and markers.  <table border="1"> <caption>Data for pyplot.plot(x, y) example</caption> <thead> <tr> <th>Month</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>Jan</td><td>2.1</td></tr> <tr><td>Feb</td><td>3.2</td></tr> <tr><td>Mar</td><td>2.4</td></tr> <tr><td>Apr</td><td>2.0</td></tr> <tr><td>May</td><td>2.9</td></tr> <tr><td>Jun</td><td>3.0</td></tr> <tr><td>Jul</td><td>3.2</td></tr> <tr><td>Aug</td><td>3.4</td></tr> <tr><td>Sep</td><td>2.7</td></tr> <tr><td>Oct</td><td>3.3</td></tr> <tr><td>Nov</td><td>3.4</td></tr> <tr><td>Dec</td><td>3.7</td></tr> </tbody> </table>	Month	Value	Jan	2.1	Feb	3.2	Mar	2.4	Apr	2.0	May	2.9	Jun	3.0	Jul	3.2	Aug	3.4	Sep	2.7	Oct	3.3	Nov	3.4	Dec	3.7
Month	Value																										
Jan	2.1																										
Feb	3.2																										
Mar	2.4																										
Apr	2.0																										
May	2.9																										
Jun	3.0																										
Jul	3.2																										
Aug	3.4																										
Sep	2.7																										
Oct	3.3																										
Nov	3.4																										
Dec	3.7																										
<code>pyplot.stackplot(x, y)</code>	<ul style="list-style-type: none"> • <code>labels</code> —Data labels. • <code>colors</code> —Area colors.  <table border="1"> <caption>Data for pyplot.stackplot(x, y) example</caption> <thead> <tr> <th>Month</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>Jan</td><td>2.1</td></tr> <tr><td>Feb</td><td>3.2</td></tr> <tr><td>Mar</td><td>2.4</td></tr> <tr><td>Apr</td><td>2.0</td></tr> <tr><td>May</td><td>2.9</td></tr> <tr><td>Jun</td><td>3.0</td></tr> <tr><td>Jul</td><td>3.2</td></tr> <tr><td>Aug</td><td>3.4</td></tr> <tr><td>Sep</td><td>2.7</td></tr> <tr><td>Oct</td><td>3.3</td></tr> <tr><td>Nov</td><td>3.4</td></tr> <tr><td>Dec</td><td>3.7</td></tr> </tbody> </table>	Month	Value	Jan	2.1	Feb	3.2	Mar	2.4	Apr	2.0	May	2.9	Jun	3.0	Jul	3.2	Aug	3.4	Sep	2.7	Oct	3.3	Nov	3.4	Dec	3.7
Month	Value																										
Jan	2.1																										
Feb	3.2																										
Mar	2.4																										
Apr	2.0																										
May	2.9																										
Jun	3.0																										
Jul	3.2																										
Aug	3.4																										
Sep	2.7																										
Oct	3.3																										
Nov	3.4																										
Dec	3.7																										

Plot Type Function	Optional Arguments of Note and Example										
<code>pyplot.bar(x, height)</code>	<ul style="list-style-type: none"> • <code>width</code> — Bar width. • <code>bottom</code> — The y-axis coordinates of the bars' bases. • <code>align</code> — The x-axis alignment of the bars.  <table border="1"> <thead> <tr> <th>City</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Agerstown</td> <td>~70,000</td> </tr> <tr> <td>Carbon Creek</td> <td>~110,000</td> </tr> <tr> <td>Greene City</td> <td>~70,000</td> </tr> <tr> <td>Olinger</td> <td>~50,000</td> </tr> </tbody> </table>	City	Value	Agerstown	~70,000	Carbon Creek	~110,000	Greene City	~70,000	Olinger	~50,000
City	Value										
Agerstown	~70,000										
Carbon Creek	~110,000										
Greene City	~70,000										
Olinger	~50,000										
<code>pyplot.bart(y, width)</code>	<ul style="list-style-type: none"> • <code>height</code> — Bar height. • <code>left</code> — The x-axis coordinates of the bars' left sides. • <code>align</code> — The y-axis alignment of the base.  <table border="1"> <thead> <tr> <th>City</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Olinger</td> <td>~45,000</td> </tr> <tr> <td>Greene City</td> <td>~65,000</td> </tr> <tr> <td>Carbon Creek</td> <td>~100,000</td> </tr> <tr> <td>Agerstown</td> <td>~65,000</td> </tr> </tbody> </table>	City	Value	Olinger	~45,000	Greene City	~65,000	Carbon Creek	~100,000	Agerstown	~65,000
City	Value										
Olinger	~45,000										
Greene City	~65,000										
Carbon Creek	~100,000										
Agerstown	~65,000										

Plot Type Function	Optional Arguments of Note and Example																								
<code>pyplot.hist(x)</code>	<ul style="list-style-type: none"> • <code>bins</code> —The number of bins. • <code>histtype</code> —The type of histogram. • <code>orientation</code> —Orients plot vertically or horizontally.  <table border="1"> <caption>Data for Histogram</caption> <thead> <tr> <th>Bin Range (Income)</th> <th>Frequency</th> </tr> </thead> <tbody> <tr><td>30,000 - 40,000</td><td>11</td></tr> <tr><td>40,000 - 50,000</td><td>12</td></tr> <tr><td>50,000 - 60,000</td><td>8</td></tr> <tr><td>60,000 - 70,000</td><td>6</td></tr> <tr><td>70,000 - 80,000</td><td>1</td></tr> <tr><td>80,000 - 90,000</td><td>3</td></tr> <tr><td>90,000 - 100,000</td><td>1</td></tr> <tr><td>100,000 - 110,000</td><td>1</td></tr> <tr><td>110,000 - 120,000</td><td>1</td></tr> <tr><td>120,000 - 130,000</td><td>1</td></tr> <tr><td>130,000 - 140,000</td><td>1</td></tr> </tbody> </table>	Bin Range (Income)	Frequency	30,000 - 40,000	11	40,000 - 50,000	12	50,000 - 60,000	8	60,000 - 70,000	6	70,000 - 80,000	1	80,000 - 90,000	3	90,000 - 100,000	1	100,000 - 110,000	1	110,000 - 120,000	1	120,000 - 130,000	1	130,000 - 140,000	1
Bin Range (Income)	Frequency																								
30,000 - 40,000	11																								
40,000 - 50,000	12																								
50,000 - 60,000	8																								
60,000 - 70,000	6																								
70,000 - 80,000	1																								
80,000 - 90,000	3																								
90,000 - 100,000	1																								
100,000 - 110,000	1																								
110,000 - 120,000	1																								
120,000 - 130,000	1																								
130,000 - 140,000	1																								
<code>pyplot.boxplot(x)</code>	<ul style="list-style-type: none"> • <code>notch</code> —Determines whether box plot will have notches representing the confidence interval. • <code>vert</code> —Orients plot vertically or horizontally. • <code>whis</code> —Determines length of whiskers beyond Q1 and Q3.  <table border="1"> <caption>Data for Box Plot</caption> <thead> <tr> <th>Statistic</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>Minimum</td><td>35,000</td></tr> <tr><td>Q1</td><td>45,000</td></tr> <tr><td>Median</td><td>52,500</td></tr> <tr><td>Q3</td><td>70,000</td></tr> <tr><td>Maximum</td><td>100,000</td></tr> <tr><td>Outliers</td><td>~125,000, ~140,000</td></tr> </tbody> </table>	Statistic	Value	Minimum	35,000	Q1	45,000	Median	52,500	Q3	70,000	Maximum	100,000	Outliers	~125,000, ~140,000										
Statistic	Value																								
Minimum	35,000																								
Q1	45,000																								
Median	52,500																								
Q3	70,000																								
Maximum	100,000																								
Outliers	~125,000, ~140,000																								

Plot Type Function	Optional Arguments of Note and Example
<code>pyplot.pie(x)</code>	<ul style="list-style-type: none"> • <code>explode</code> —Fraction of the radius by which each slice is offset. • <code>autopct</code> —Labels slices with their percentages. • <code>startangle</code> —Rotates chart counter-clockwise from x-axis. • <code>radius</code> —Chart radius. 

Violin Plots

A *violin plot* is a method of showing the distribution of a numerical value through probability density. It is similar to a box plot in that it can show a summary statistic like *interquartile range (IQR)*, which is the difference between the upper and lower quartiles. However, because it also plots the distribution of data, it can reveal more about where the data tends to fall within the range of values. The distribution is computed using a kernel density estimation (KDE), which differs from how a histogram distribution is computed. The distribution of a KDE is smoothed into curves, rather than represented as rectangular bins like a histogram. This helps alleviate some of the issues of a histogram, such as the difficulty of choosing a good bin size. A violin plot gets its name from the fact that the KDE distribution extends from both the sides of the range line, forming a shape that is reminiscent of a violin. The wider areas of the distribution indicate a greater probability of data samples being at that value, whereas thinner areas indicate lower probability.

The `pyplot.violinplot()` function generates violin plots. It takes one required argument: the data you wish to plot. By default, the plot draws a line vertically for the entire range of the data, as well as horizontal lines for both extremes. You can also specify that you want to show the mean and/or median for the distribution, and you can flip the plot so that it lies horizontally. Violin plots are most often used to compare distributions across similar data samples, so you can input an array of arrays to plot each data sample as its own violin.

In the following example, you have a dataset of average temperatures for three months over a period of 10 years:

```
>> aug_temp # Demonstrate one month.
array([84, 85, 93, 81, 80, 84, 75, 77, 79, 78])
```

You want to plot the distribution of each month's temperatures to see what temperature ranges are most probable for each month:

```
temps = [jun_temp, jul_temp, aug_temp] # Each month will have its own violin.
fig, ax = plt.subplots()
ax.violinplot(temps, showmeans = True)
```

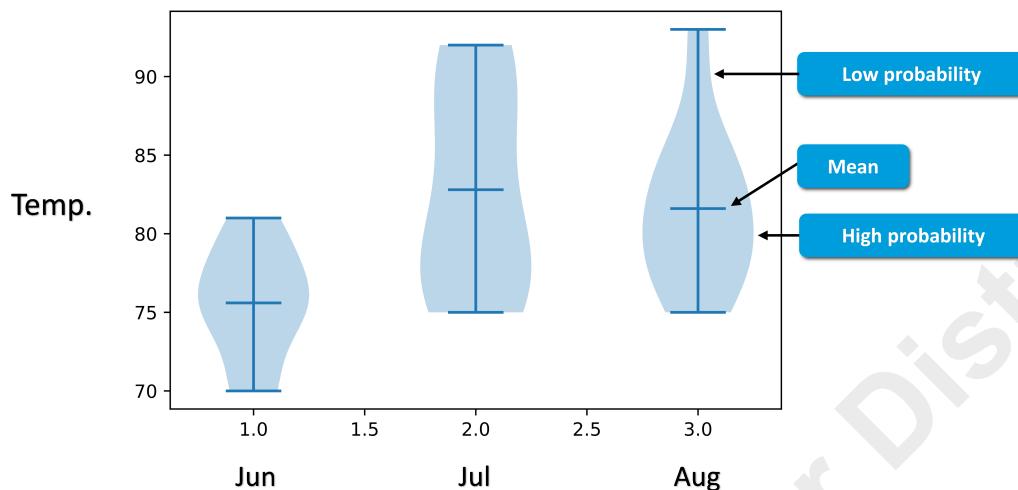


Figure 6–9: The generated violin plot. July seems to have a greater probability of having temperatures in the high 80s/low 90s than August.

Contour Plots

A [contour plot](#) represents three-dimensional data in a two-dimensional format by plotting contour lines. A contour line is a curve that connects x and y coordinates at a value for z (the third dimension). Ultimately, the contour plot demonstrates how z changes with respect to x and y . One real-world example of a contour plot is a topographic map, which can show the sloping of hills and valleys (z) where they occur on a two-dimensional plane (e.g., latitude and longitude). Contour plots are also useful for showing the density of data—in other words, the distribution of data across two variables. This essentially combines a scatter plot with a histogram, which can be helpful in large datasets where there are so many data points that interpreting a normal scatter plot becomes difficult. For example, a hardware manufacturer might create a contour plot of the usage hours for one brand of storage drive as compared to its incidence of bad sectors, then compare the contours to those of other storage drives.

The `pyplot.contour()` function generates contour plots. It takes x , y , and z coordinate values. The following example plots temperature anomalies on a map of the world. A temperature anomaly is the difference between the current temperature for a location and its long-term average. The latitude and longitude variables are both one-dimensional NumPy arrays; however, any coordinate arrays input to `contour()` must be in the form of a two-dimensional grid. To do this, you can transform the coordinates using `numpy.meshgrid()`:

```
lon_grid, lat_grid = numpy.meshgrid(lon, lat)
```

In this case, the temperature anomalies variable (`anom`) is already in the form of a two-dimensional array. While `lon_grid` and `lat_grid` will be plotted as x and y , respectively, `anom` will represent z . So, creating a contour map can be done like so:

```
fig, ax = plt.subplots()
ax.contour(lon_grid, lat_grid, anom)
```

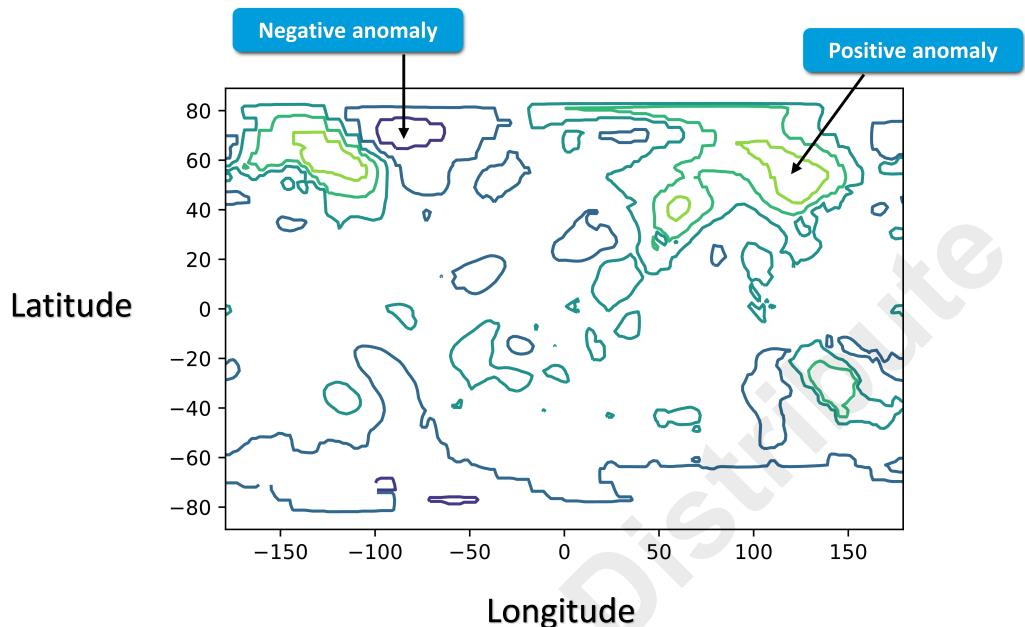


Figure 6–10: The generated contour plot.

The contour lines show the gradual changes in anomalous temperatures per region. In the default color scheme, neutral-colored lines indicate less significant anomalies (less difference between the current temperature and the long-term average); darker lines indicate negative anomalies (the current temperature is lower); and brighter lines indicate positive anomalies (the current temperature is higher).



Note: The `pyplot.contourf()` function does the same thing, except that it fills in the contours to create something more akin to a heatmap.

The `numpy.meshgrid()` Function

The `numpy.meshgrid()` function creates a grid using an array of `x` values and an array of `y` values. It's often used to generate the necessary two-dimensional grids to input into a plotting function like `contour()`. Here's a simple example:

```
>> x = numpy.array([1, 2, 3, 4])
>> y = numpy.array([5, 6, 7])
>> xx, yy = numpy.meshgrid(x, y)
>> print(xx)
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
>> print(yy)
[[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]
```

For the generated `xx` grid, the values in `x` are repeated as rows. The number of rows is equal to the number of values in `y`. Likewise, for the `yy` grid, the values in `y` are repeated as columns. The number of columns is equal to the number of values in `x`.

Quiver Plots

A [quiver plot](#) represents four-dimensional data in a two-dimensional format by plotting arrows. Each arrow is placed at the appropriate x and y location, and the direction of the arrow corresponds to dimensions u and v . Quiver plots are commonly used to show processes that have a direction or magnitude in addition to location. For example, quiver plots can make it easier to analyze the direction of wind at certain locations. Another example is plotting the intensity of sound waves as they travel through a medium.

The `pyplot.quiver()` function generates quiver plots. It takes the four coordinates just mentioned, where each coordinate can be either a one-dimensional or two-dimensional array. However, u and v are often transformed into two-dimensional grids. In the following example, the wind patterns around Lake Victoria (the world's largest tropical lake) are plotted with arrows that indicate the wind's direction. The longitude and latitude values are one-dimensional NumPy arrays, while the u and v values for the wind direction have already been placed in two-dimensional NumPy grids:

```
fig, ax = plt.subplots()
ax.quiver(lon, lat, wind_u, wind_v)
```

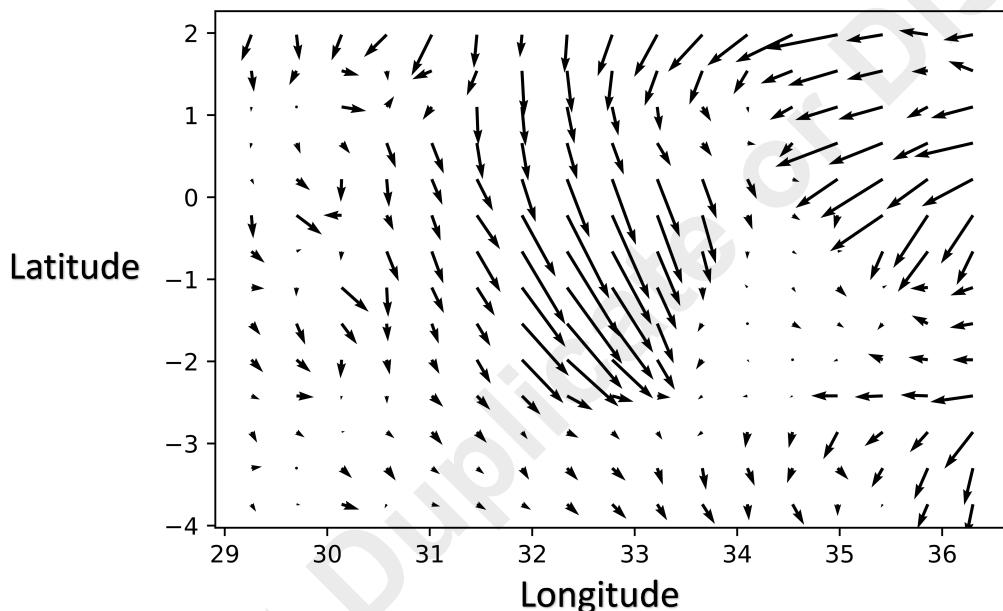


Figure 6–11: The generated quiver plot. The arrows show the wind going from north to south in the middle of the map, with some west-to-east movement.

Image Plots

An [image plot](#), as the name suggests, takes an array of data representing a bitmap image and places it on a plot. There are several reasons why you might plot an image: you might want to modify that image in some way by changing its properties, like converting a color image to grayscale; you might want to compare the properties of two or more images; you might want to give context to another plot by plotting the data on top of the image; and more.

The `pyplot.imshow()` function generates image plots. It takes a NumPy array of RGB or RGBA (alpha channel) values. The values in this array are floats that correspond to levels of red, green, blue, and (optionally) alpha at specific locations in the two-dimensional image. However, before you

use this function, you need to actually convert your image into a compatible array. Matplotlib's `image` module, with its `imread()` function, can do this for you:

```
>>> import matplotlib.image as mpimg
>>> img = mpimg.imread('map_image.png')
>>> img
array([[[0.80784315, 0.93333334, 0.80784315, 1.],
       [0.80784315, 0.93333334, 0.80784315, 1.],
       [0.80784315, 0.93333334, 0.80784315, 1.],
       ...], # Truncated.
```

The output shows the different RGBA values for the image in a NumPy array. Note that `image.imread()` supports only PNG-formatted images. If your image uses a different bitmap format, you must use a different library (like Pillow).

Now that your image is in an array, you can plot it using `pyplot.imshow()`:

```
fig, ax = plt.subplots(figsize = (8, 8))
ax.imshow(img)
```

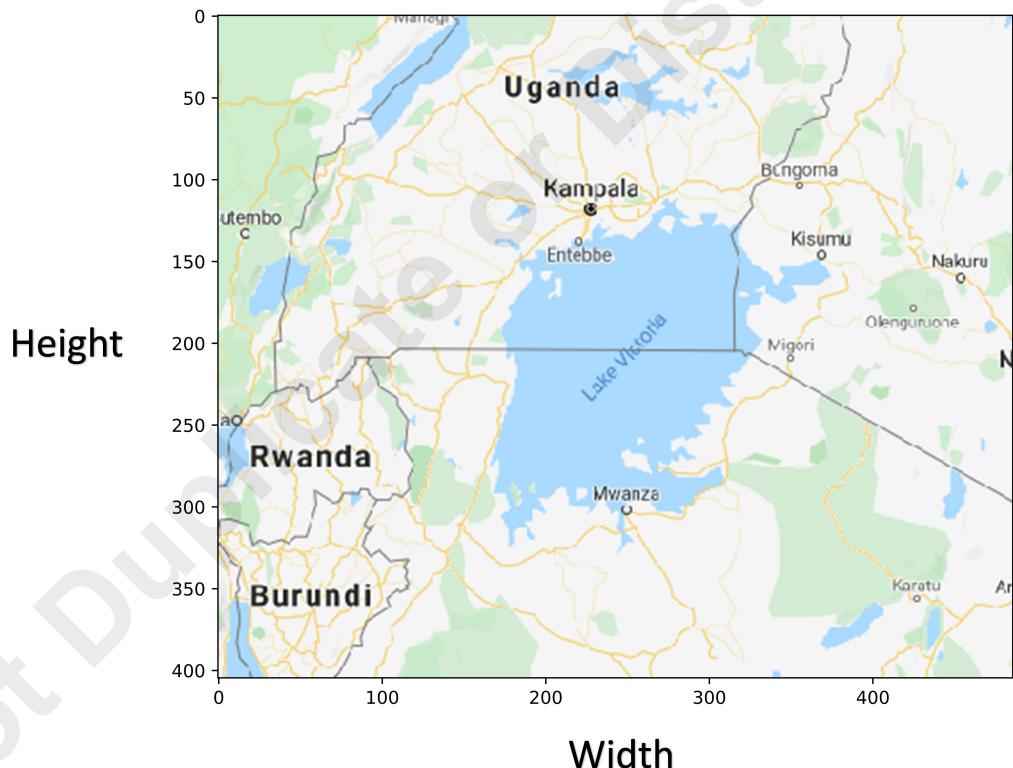


Figure 6-12: The generated image plot. Note that the axes correspond to the pixel values for both width and height.

Contextual Image Plot Example

As mentioned, it's common for an image plot to be used beneath another type of plot to add context to the data. Consider the quiver plot generated earlier. You can see the longitude and latitude values for the wind direction arrows, but it's difficult for a human observer to visualize the location this is all occurring in. It would be much more useful to plot the arrows on an actual map of the relevant area. In other words, you can combine the quiver plot with the image plot.

Before you simply call both plots, you need to address the fact that both the quiver plot and the image plot have different axes. The quiver plot has a range of longitude and latitude values, but the image plot is a range of pixel width and height values. You need to make these axes match. You can do this by constraining the image plot's axes to values that correspond to the quiver plot. When you call `imshow()`, pass in a list to the `extent` argument that follows this pattern:

```
[x start, x end, y start, y end]
```

In other words, you're specifying where the x-axis starts (left) and where it ends (right), as well as where the y-axis starts (bottom) and ends (top). In the case of plotting an image of Lake Victoria and its surrounding region on the proper longitude and latitude limits, you can set `extent` to the quiver plot's limits. To plot both image and quiver, use the familiar modular approach:

```
fig, ax = plt.subplots(figsize = (8, 8))
ax.imshow(img, extent = [numpy.amin(lon), numpy.amax(lon),
                        numpy.amin(lat), numpy.amax(lat)])
ax.quiver(lon, lat, wind_u, wind_v)
```

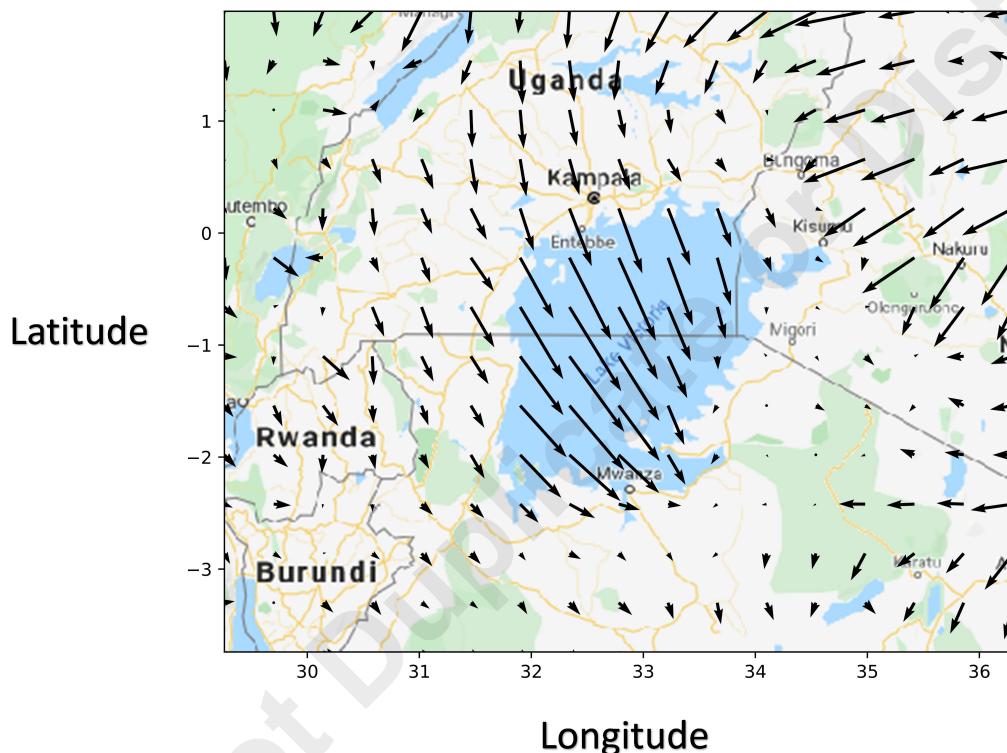


Figure 6–13: Now that the map is showing, you can see the wind sweep over the lake in a north-to-south and slightly west-to-east direction.

3-D Plots

A three-dimensional plot isn't a specific type, but is instead just a three-dimensional visualization of one or more of the types of plots you've already seen, and some that are unique to 3-D plotting. Translating a two-dimensional visual into three dimensions can make certain data easier to analyze, especially data from contour plots, violin plots, and other types that normally represent higher-dimensional data in lower dimensions.

The primary Matplotlib library doesn't provide 3-D plotting functionality, but the `mpl_toolkits` extension does—specifically, the `mpl_toolkits.mplot3d` module. Plotting in 3-D is essentially the

same process as plotting in 2-D; you can either use the stateful or object-oriented interfaces. There is one main difference: you create an `Axes3D` object, rather than a normal `Axes` object, by specifying `projection = '3d'` in `add_subplot()`. Note that `add_subplot()` is the preferred approach for 3-D plotting.

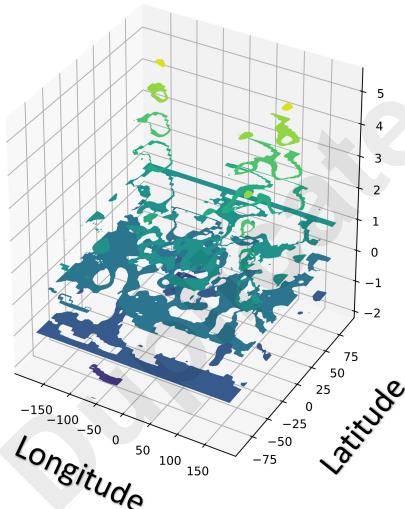
If you want to project a normally 2-D plot into 3-D, you just call the same function. For the most part, you need to provide an extra `z` value as an argument. In the following example, `contourf()` is called on the temperature anomaly data to cast that data into a 3-D filled-in contour plot. However, there are some plot types that are unique to three dimensions. The temperature data is also plotted on a surface plot, which projects the data in three dimensions with solid colors across the entire data surface. The function for this is `plot_surface()`.

```
fig = plt.figure()
ax1 = fig.add_subplot(1, 2, 1, projection = '3d')
ax2 = fig.add_subplot(1, 2, 2, projection = '3d')
ax1.contourf(lon_grid, lat_grid, temp_anomaly)
ax2.plot_surface(lon_grid, lat_grid, temp_anomaly, cmap = 'viridis')
```



Note: The `cmap` argument specifies the color map to use.

3D Contour



Surface Plot

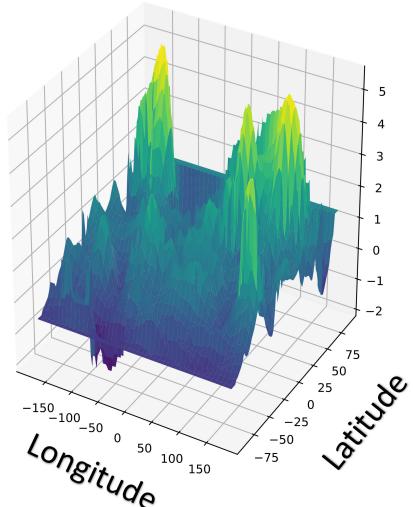


Figure 6-14: Two different 3-D plots of the same data. The temperature anomaly values are given height to distinguish their magnitude.

The `view_init()` Function

You can change the perspective of a 3-D plot by calling the `view_init()` function. It takes two arguments, both of which rotate the plot by the degrees you specify:

- `elev` —Change the elevation angle of the z-axis (i.e., tilt this axis forward or backward).
- `azim` —Change the azimuth angle of the x-axis and y-axis (i.e., rotate these axes clockwise or counter-clockwise).

Guidelines for Creating Common Types of Plots

Follow these guidelines when you are creating common types of plots in Matplotlib.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

Create Common Types of Plots

When creating common types of plots:

- Use the familiar plot types—scatter, line, bar, histogram, etc.—for the same purposes mentioned earlier.
- Use violin plots to demonstrate distribution using kernel density estimation (KDE), rather than through histogram bins.
 - Use to help alleviate the issue of choosing bin size in a histogram.
 - Interpret wider areas of the violin as having a greater probability of data being in that range, and vice versa.
- Use contour plots to demonstrate how a third dimension changes with respect to the first two dimensions.
 - Use to show topographical data.
 - Use to show density of data in large datasets.
 - Apply `numpy.meshgrid()` to all three dimension inputs if they aren't already in a grid format.
- Use quiver plots to demonstrate four-dimensional data in two dimensions.
 - Use to show processes with a magnitude and direction.
 - Use a one-dimensional input for a dimension or a two-dimensional grid.
- Use image plots to demonstrate bitmap images.
 - Use to change properties of an image, like color scale or orientation.
 - Use to add context to another plot in a modular fashion.
 - Use `image.imread()` to load PNG images, or a different library for other file types.
- Use 3-D plots to demonstrate data in three visible dimensions.
 - Use `Figure.add_subplot()` with `projection = '3d'` to construct a 3-D plot.
 - Use many of the same plotting functions to create a 3-D version of that plot.
 - Use plotting functions that are specific to three dimensions, like `plot_surface()`.

ACTIVITY 6–3

Creating Common Types of Plots

Data Files

/home/student/DSTIP/Matplotlib/Creating Common Plots.ipynb
 /home/student/DSTIP/Matplotlib/data/stores_data_full_clean.csv

Before You Begin

Jupyter Notebook is open.

Scenario

There are many ways to plot GCE's data, each of which can reveal more about the stores' sales performance. So, you'll generate different types of plots to see if you can extract actionable information from them. What you learn from these plots can help steer GCE's business into new and more profitable directions.

1. In Jupyter Notebook, open the **DSTIP/Matplotlib/Creating Common Plots.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 Lines 16 through 18 load the full `stores_df` DataFrame.
 - b) Run the code cell.
3. Generate stack plots to compare total items sold for each day in January and February.
 - a) Scroll down and view the cell titled **Generate stack plots to compare total items sold for each day in January and February**, and examine the code listing below it.

```

1 jan = stores_df[stores_df['Date'].dt.month == 1]
2 feb = stores_df[stores_df['Date'].dt.month == 2]
3
4 jan_quant = stores_df.groupby(jan['Date'].dt.day)[['Quantity']].sum()
5 feb_quant = stores_df.groupby(feb['Date'].dt.day)[['Quantity']].sum()
6 feb_quant = feb_quant.append(pd.Series([np.nan, np.nan, np.nan]),
7                               ignore_index = True)
8
9 jan_quant.head()

```

- Lines 1 and 2 filter `stores_df` by each month.
 - Lines 4 and 5 group each month by separate days, taking the total of items sold for each day.
 - Lines 6 and 7 append some null values to the February `Series`. In a stack plot, each dataset must have the same number of rows. Because February has three fewer days than January, you need to pad it out.
- b) Run the code cell.

- c) Examine the output.

```
Date
1.00    81
2.00    48
3.00    37
4.00    32
5.00    55
Name: Quantity, dtype: int64
```

The returned `Series` shows the total items sold for each day (in this case, the first five days of January).

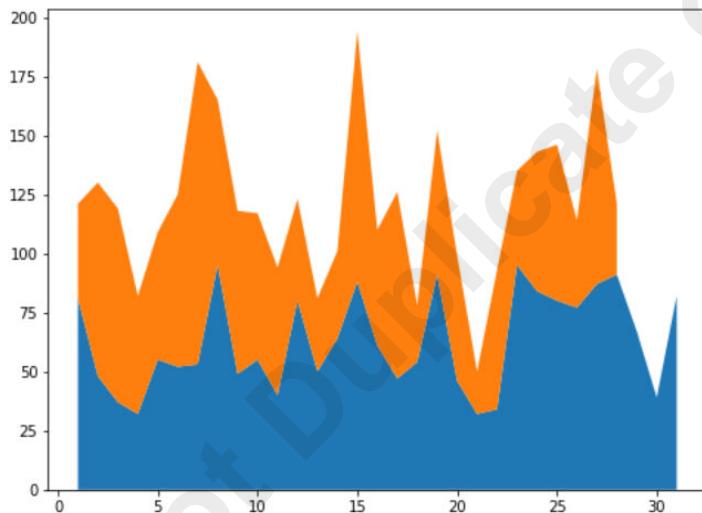
- d) Select the next code cell, and then type the following:

```
1 fig, ax = plt.subplots(figsize = (8, 6))
2 ax.stackplot(jan_quant.index, jan_quant, feb_quant)
```

Line 2 generates a stack plot, with one stack for January and the other for February.

- e) Run the code cell.
f) Examine the output.

```
[<matplotlib.collections.PolyCollection at 0x7fa347a1f310>,
 <matplotlib.collections.PolyCollection at 0x7fa346ea9650>]
```



- A stack plot was generated, where the bottom blue stack represents January and the top orange stack represents February.
- Stack plots are similar to area plots, except the area portions are stacked on top of each other. Since the areas do not overlap, this can make the y-axis somewhat misleading. For example, February 15th did not see just under 200 items sold; you would instead need to subtract the height of January's stack at that point from the height of February's stack at that point to get the actual value.
- So, a stacked plot like this is most useful when you compare the thickness of each filled-in portion. For example, the thickness of the January stack around day 21 is greater than its counterpart day in February (due to a sharp drop from the previous day).

4. Generate scatter plots to see how different variables correlate with gross income.

- Scroll down and view the cell titled **Generate scatter plots to see how different variables correlate with gross income**, then select the code cell below it.
- In the code cell, type the following:

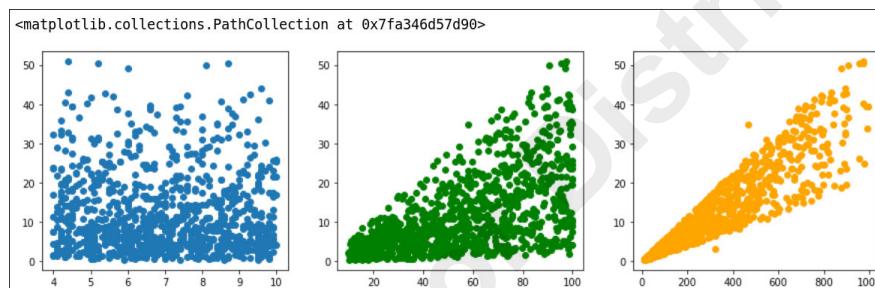
```

1 y = stores_df['GrossIncome']
2 fig, ax = plt.subplots(1, 3, figsize = (15, 4))
3 ax[0].scatter(stores_df['CustomerRating'], y)
4 ax[1].scatter(stores_df['UnitPrice'], y, c = 'green')
5 ax[2].scatter(stores_df['Revenue'], y, c = 'orange')

```

Lines 3 through 5 generate subplots where `x` is different accounting values, but `y` is always gross income.

- Run the code cell.
- Examine the output.



- The dependent variable in all three plots is gross income; the independent variables are customer rating, unit price, and revenue, respectively.
- These scatter plots demonstrate how changes in each independent variable impact the gross income.
- Customer rating doesn't appear to correlate that much with gross income because of how spread out the points are.
- Unit price seems to have some positive correlation with gross income due to the left-to-right spread of the points. However, because many points still cluster toward the bottom, the relationship between these variables is not entirely linear. In other words, there are several examples where low unit prices and high unit prices lead to the same income.
- Revenue seems to have the strongest linear correlation with gross income, which makes sense given how the former is calculated from the latter. The higher the revenue, the higher the gross income.

5. Generate bar charts comparing branch revenue per product line.

- Scroll down and view the cell titled **Generate bar charts comparing branch revenue per product line**, then select the code cell below it.
- In the code cell, type the following:

```

1 prod_rev = stores_df.groupby(['ProductLine', 'Branch'])['Revenue'].sum()
2 prod_rev

```

This code groups the dataset by both product line and store branch, and then finds the total revenue.

- Run the code cell.

- d) Examine the output.

```

ProductLine      Branch
Clothing         Carbon Creek   15,554.77
                  Greene City    15,631.73
                  Olinger        20,092.04
Electronics      Carbon Creek   17,444.87
                  Greene City    16,239.47
                  Olinger        18,065.69
Food and beverages Carbon Creek  16,345.81
                  Greene City    14,490.37
                  Olinger        22,016.52
Health and beauty  Carbon Creek  11,997.86
                  Greene City    18,567.26
                  Olinger        15,793.38
Home and lifestyle Carbon Creek  20,626.21
                  Greene City    16,713.49
                  Olinger        13,233.86
Sports and travel  Carbon Creek  18,450.19
                  Greene City    19,036.38
                  Olinger        15,011.36
Name: Revenue, dtype: float64

```

The returned multi-index Series contains the total revenue per store branch, per product line.

- e) Select the next code cell, and then type the following:

```

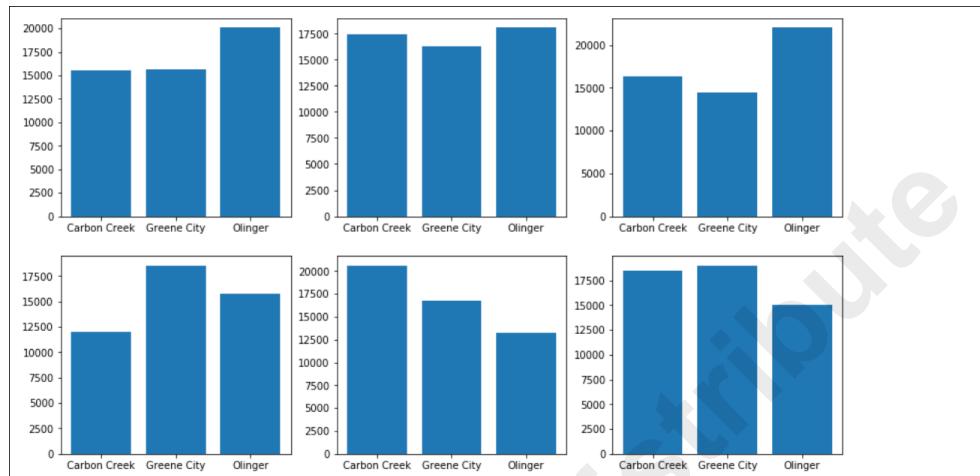
1 # Define lists of product lines and branches to iterate over.
2 prods = prod_rev.index.get_level_values('ProductLine').unique()
3 branch = prod_rev.index.get_level_values('Branch').unique()
4 i = 0
5
6 fig, ax = plt.subplots(2, 3, figsize = (14, 8))
7
8 for col in range(0, 3):
9     ax[0, col].bar(branch, prod_rev.loc[[prods[i]]])
10    i += 1
11
12 for col in range(0, 3):
13     ax[1, col].bar(branch, prod_rev.loc[[prods[i]]])
14    i += 1

```

- Lines 2 and 3 define lists of all unique product lines and branches, respectively. These will be used to make the plotting code a little cleaner.
- Lines 8 through 10 use a `for` loop to plot the first row of the subplot grid.
- Line 9 provides each branch location as the horizontal axis, with the current product line is plotted on the vertical axis.
- Lines 12 through 14 do the same for the second row of the subplot grid for the remaining product lines.

- f) Run the code cell.

g) Examine the output.



- Bar charts are plotted for each product type, where the revenue for each branch location for that product type is compared. The `Series` in the previous code cell's output shows the left-to-right order of each product bar chart.
- One example observation is that Olinger seems to have a clear edge in revenue for clothing, while the revenue for electronics is a little more even between the branches.

6. Generate histograms and box plots that show the distributions of revenue and income.

- Scroll down and view the cell titled **Generate histograms and box plots that show the distributions of revenue and income**, then select the code cell below it.
- In the code cell, type the following:

```
1 fig, ax = plt.subplots(1, 2, figsize = (15, 6))
2 ax[0].hist(stores_df['Revenue'], bins = 20)
3 ax[1].hist(stores_df['GrossIncome'], bins = 20);
```

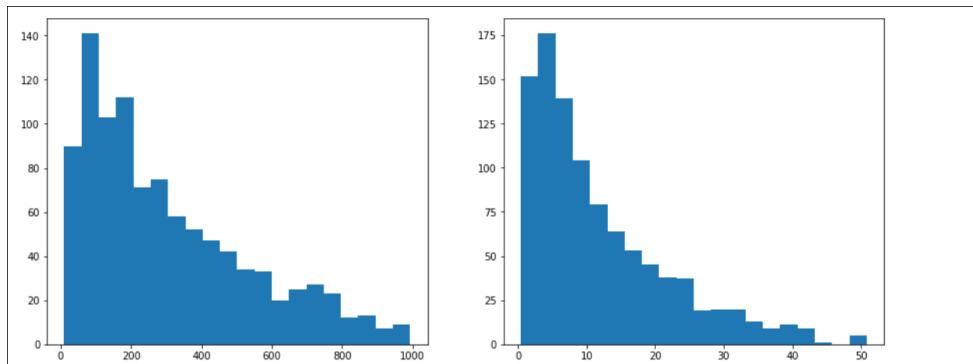
- Lines 2 and 3 generate histograms for revenue and gross income, using an arbitrary value for the number of bins.



Note: The semicolon (;) at the end of the last plot tells the kernel to suppress Matplotlib's text outputs for that cell, which can sometimes be verbose.

- Run the code cell.

d) Examine the output.



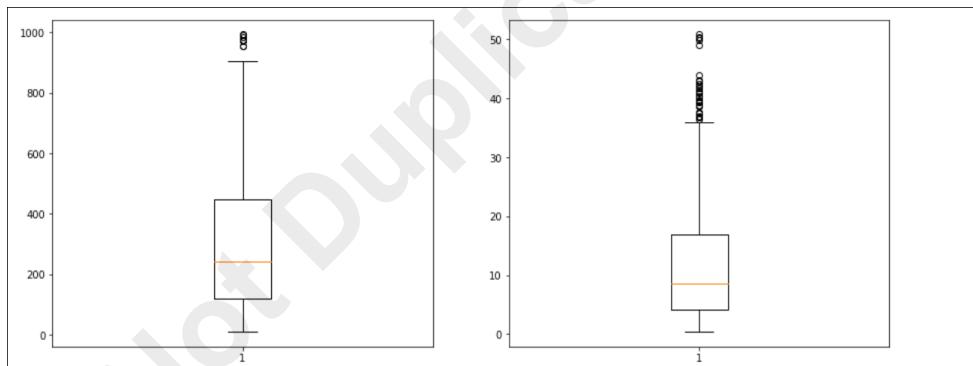
- Both variables seem very much right skewed; more transactions lead to lower sales values than higher ones. Revenue between \$0 and \$200 and income between \$0 and \$10 seem to be the most common.
- The number of bins can affect the shape of the histogram. There is not necessarily a "correct" number of bins.

e) Select the next code cell, and then type the following:

```
1 fig, ax = plt.subplots(1, 2, figsize = (15, 6))
2 ax[0].boxplot(stores_df['Revenue'])
3 ax[1].boxplot(stores_df['GrossIncome']);
```

Lines 2 and 3 generate box plots of the same values as the histograms.

- f) Run the code cell.
g) Examine the output.



- Box plots are an alternative method of analyzing variable distribution. They focus more on visualizing summary statistics, rather than just the spread of the data like a histogram does.
- While each variable is on a different scale, they appear to have a similar distribution.
- One example observation is that gross income has a lower relative maximum and more outliers than revenue has.

7. Generate violin plots that compare the distributions of unit price and tax price.

- a) Scroll down and view the cell titled **Generate violin plots that compare the distributions of unit price and tax price**, then select the code cell below it.

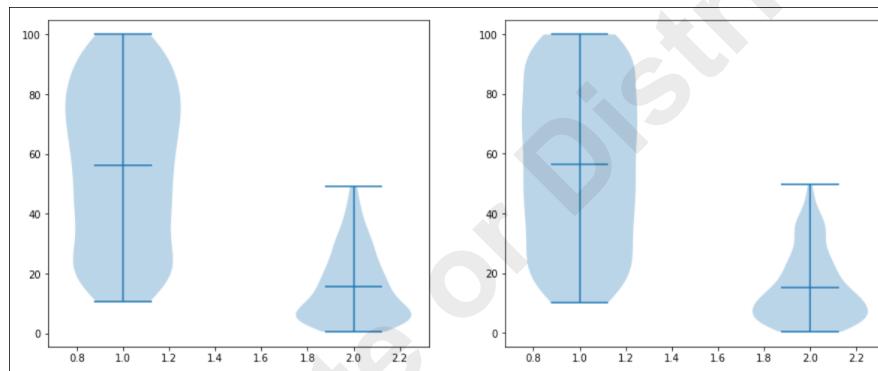
- b) In the code cell, type the following:

```

1 # Define lists of prices for each month.
2 jan_unit_tax = [jan['UnitPrice'].values, jan['TaxPrice'].values]
3 feb_unit_tax = [feb['UnitPrice'].values, feb['TaxPrice'].values]
4
5 fig, ax = plt.subplots(1, 2, figsize = (15, 6))
6 ax[0].violinplot(jan_unit_tax, showmeans = True)
7 ax[1].violinplot(feb_unit_tax, showmeans = True);

```

- Lines 2 and 3 define lists of each value to use in the violin plots for January and February.
 - Lines 6 and 7 generate each violin plot.
- c) Run the code cell.
d) Examine the output.



- Violin plots for unit price and tax price are displayed for January and February. For each plot, unit price is the violin on the left, and tax price is the violin on the right.
- Violin plots are another way to represent the distribution of variables. The thickness of the curve represents the probability of values falling within that portion of the distribution.
- The mean of each variable is shown as the middle line of the violin.
- One example observation is that tax price for both months has a much higher likelihood of being on the low end, while unit price seems to be more evenly distributed.

8. How does a violin plot differ from a histogram in terms of how it calculates the distribution of variables?

9. Generate a `GridSpec` of multiple plots showing the distributions of revenue and income.

- a) Scroll down and view the cell titled **Generate a GridSpec of multiple plots showing the distributions of revenue and income**, and examine the code listing below it.

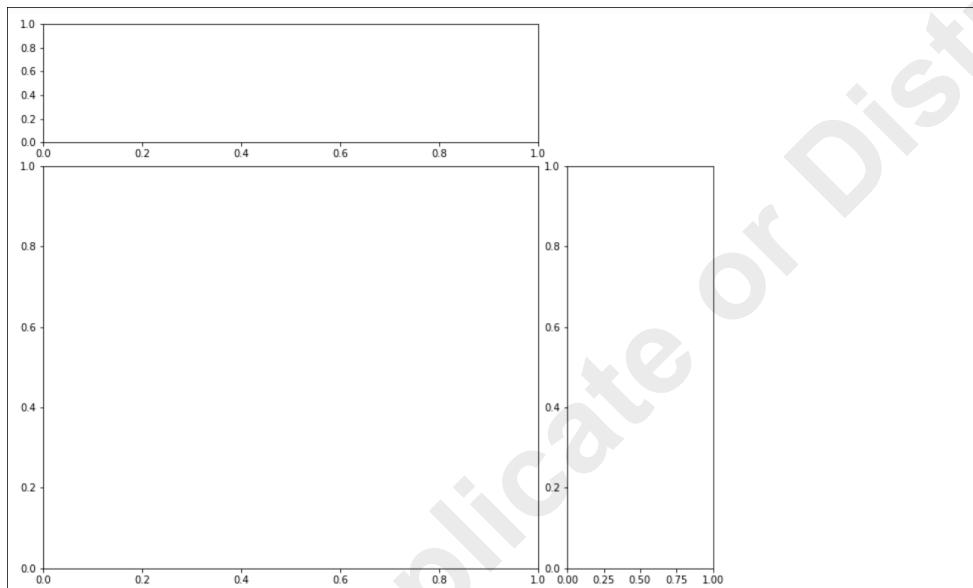
```

1 # Set up grid first.
2 fig = plt.figure(figsize = (12, 10))
3 grid = fig.add_gridspec(4, 4)
4
5 ax_top = fig.add_subplot(grid[0, :3])
6 ax_right = fig.add_subplot(grid[1:4, 3])
7 ax_mid = fig.add_subplot(grid[1:4, :3])

```

This code block sets up a `GridSpec` object to generate a custom subplot layout.

- b) Run the code cell.
c) Examine the output.



The custom grid layout has one large square-shaped subplot in the middle, with two smaller rectangular subplots that align with each axis.

- d) Select the next code cell, and then type the following:

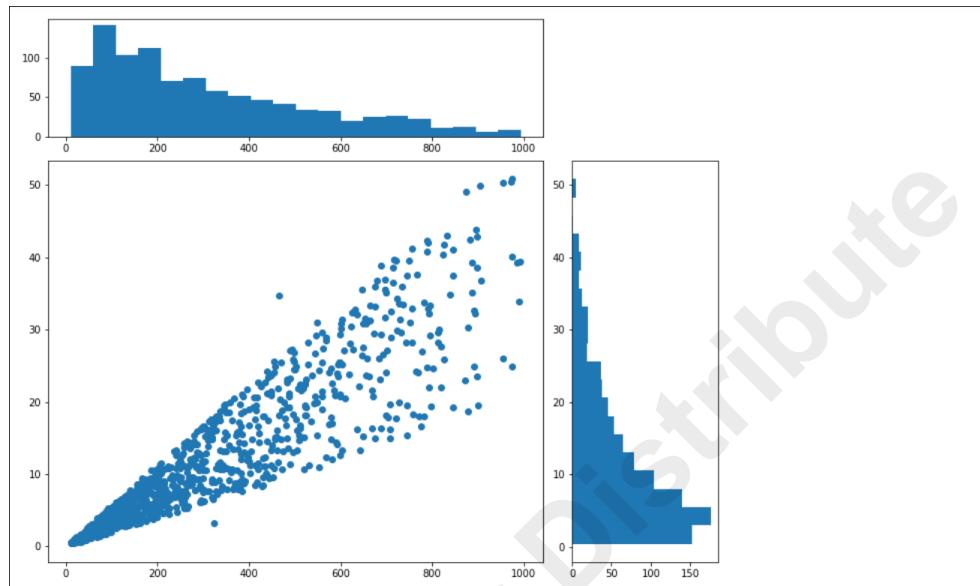
```

1 ax_top.hist(stores_df['Revenue'], bins = 20)
2 ax_right.hist(stores_df['GrossIncome'],
3               bins = 20, orientation = 'horizontal')
4 ax_mid.scatter(stores_df['Revenue'], stores_df['GrossIncome'])
5 fig

```

- Line 1 assigns the top subplot to a histogram of revenue values.
 - Line 2 assigns the right subplot to a histogram of gross income values.
 - Line 3 assigns the middle subplot to a scatter plot where revenue is on the x-axis and gross income is on the y-axis.
- e) Run the code cell.

f) Examine the output.



- This combination of plot types is often used to examine the distribution of two variables from multiple perspectives.
- The scatter plot shows the spread of revenue and income as they relate to one another.
- The revenue histogram on top aligns with the x-axis in the scatter plot, which also shows revenue.
- The gross income histogram on the right aligns with the y-axis in the scatter plot, which also shows gross income.

10. Generate a 3-D scatter plot comparing revenue, gross income, and customer rating.

- Scroll down and view the cell titled **Generate a 3-D scatter plot comparing revenue, gross income, and customer rating**, then select the code cell below it.
- In the code cell, type the following:

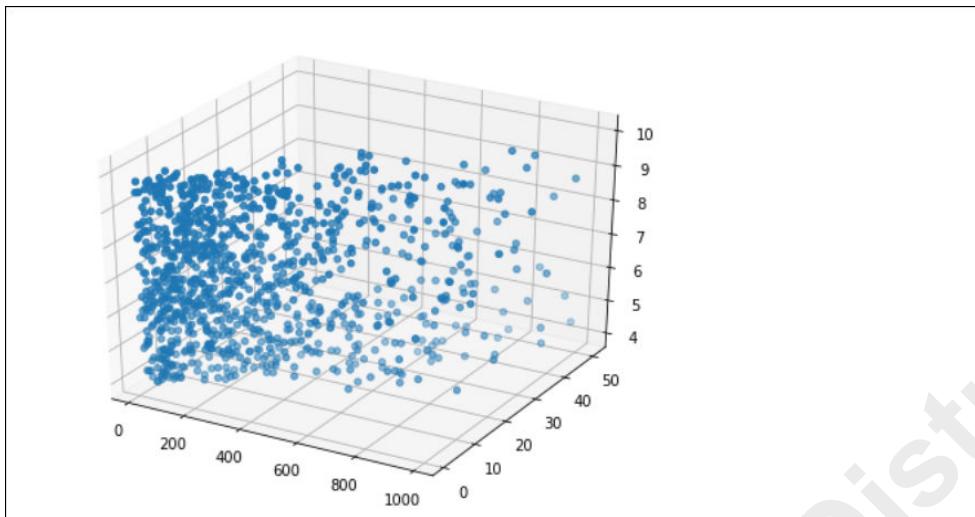
```

1 from mpl_toolkits.mplot3d import Axes3D
2
3 fig = plt.figure(figsize = (8, 6))
4 ax = fig.add_subplot(projection = '3d')
5 ax.scatter(stores_df['Revenue'],
6            stores_df['GrossIncome'],
7            stores_df['CustomerRating']);

```

- Line 1 imports the module and class that are required for creating three-dimensional plots in Matplotlib.
- Line 4 uses the typical `add_subplot()` function, but this time the `projection` argument specifies that it will create an `Axes3D` object instead of its normal two-dimensional equivalent.
- Lines 5 through 7 call `scatter()` similar to earlier, but there is now a third `z` argument that specifies what will be plotted along the third dimension.
- Run the code cell.

- d) Examine the output.



- The 3-D scatter plot shows revenue on the x-axis and gross income on the y-axis (the "bottom" axes in this graphic), while also showing customer rating on a third z-axis (the axis that appears vertical).
 - From this perspective, you can see that the markers are all along the z-axis, no matter where they are on the x- and y-axes. This reinforces the idea that customer rating seems not to have much correlation with revenue and income.
 - The beauty of a 3-D plot is that you can see multiple relationships on a single graph. However, you may need to re-orient a 3-D plot to get a better look at one type of relationship.
- e) Select the next code cell, and then type the following:

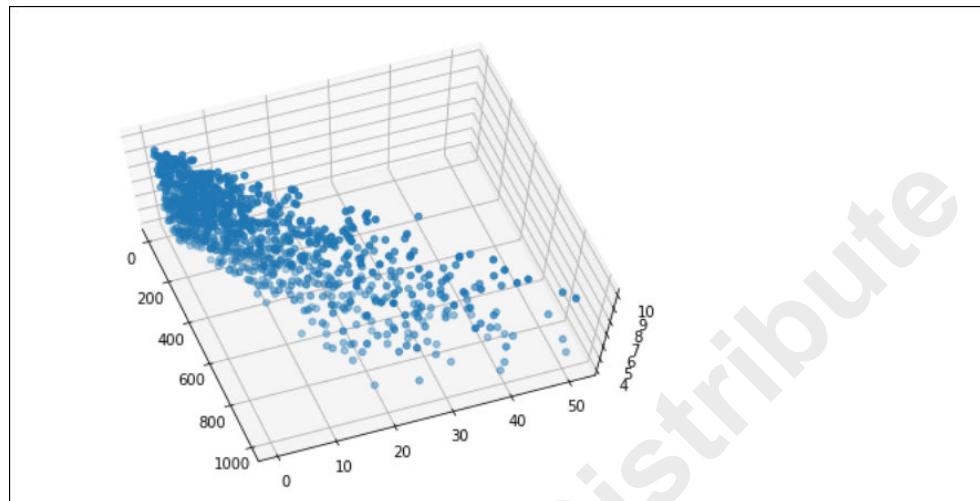
```
1 ax.view_init(elev = 70, azim = -20)
2 fig
```

The `view_init()` function re-orientates the 3-D plot according to two parameters:

- `elev` tilts the plot forward by 70 degrees.
- `azim` rotates the plot clockwise by 20 degrees.

- f) Run the code cell.

- g) Examine the output.



Now you have a better look at the linear relationship between the values along the x- and y-axes, while still seeing the "height" of the values along the z-axis.

11. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel→Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Creating Common Plots** tab in Firefox, but keep a tab open to **DSTIP/Matplotlib/** in the file hierarchy.

TOPIC D

Format Plots

So far, your plotting has come a long way. You've generated many different types of plots in many different configurations. Still, something has been missing thus far: a way to adjust their look and feel. In this topic, you'll apply formatting to your plots to ensure they provide the proper context for the data, as well as look more aesthetically pleasing.

The `style.use()` Function

Matplotlib's `style` module provides several pre-configured styles for you to apply to your plots. Each style has its own aesthetic properties—everything from the way it colors elements, to line widths and fills, to marker sizes, and more. Therefore, the purpose of a style is to set a consistent look and feel for the entirety of your plots; in other words, if you find a style you like, you won't need to tweak every individual property. Though, even when you apply a style, you can still override some of that style's properties if you so desire.

The `style.use()` function applies a style. You would typically call this function at the top of your code, before you actually do any plotting, so that the style is applied to every plot thereafter. The argument it takes is either the name of a built-in style or the path to an external style. A path can refer to the local system, or it can be a compatible URL. In the following example, a few different expenses from `income_df` are plotted on a stacked area chart. This chart is plotted in a built-in style called "Bayesian Methods for Hackers":

```
plt.style.use('bmh')
fig, ax = plt.subplots()
months = slice_2019.index.get_level_values(0)
expenses_2019 = [slice_2019['COGS'], slice_2019['Payroll'],
                 slice_2019['Taxes']]
ax.stackplot(months, expenses_2019)
```

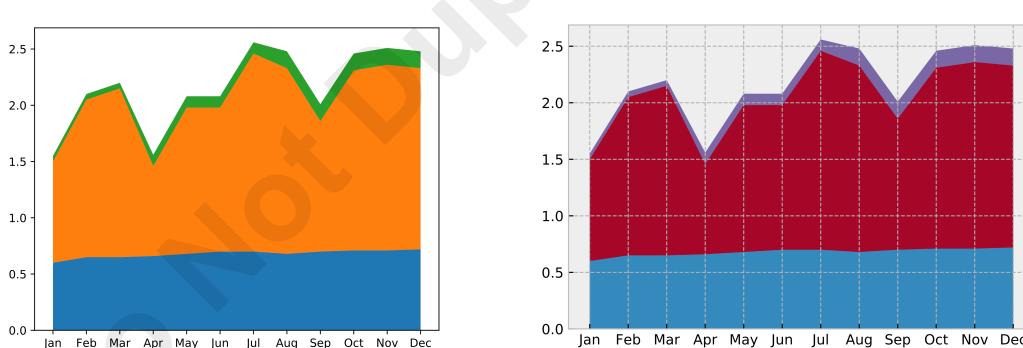


Figure 6-15: Comparing the `default` style with another built-in style.

Not only does the '`bmh`' style change the colors used, it also adds a grid and uses different spacing and font sizes.

Available Styles

Styles are typically saved as style sheet files. To get a list of all styles that are currently available on your system, including any custom styles you've defined, enter the following code:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

```
plt.style.available
```

The pyplot.rc() Function

The default style definitions are loaded as part of Matplotlib's runtime configuration (RC). In addition to using or creating your own style sheets, you can change the runtime configuration dynamically to modify the appearance of your plots. This is done through the `pyplot.rc()` function. The function takes two or more arguments: the group name of the component you're modifying, such as `lines` for modifying lines, and the actual parameter(s) you're changing, such as `linewidth` to change the width of lines. You can call this function multiple times, with each call referring to a different element you wish to change. Any elements not configured through `rc()` remain at their default values.

Line Colors and Styles

Although you're perfectly free to rely on the `styles` module for setting a visual theme, you can still adjust visual aspects on an individual level for a greater degree of plot customization. One of the primary ways to customize a plot is by specifying the colors it should use. When it comes to line plots, you can set the color of one line or multiple lines, depending on whether or not the plot is modular. This is done by specifying the color through the `color` argument, which is often shortened to `c`.

In addition to color, you may also wish to change the style of a line. Lines can be solid, dashed, dash-dotted, or dotted. Like with colors, you can choose which line to format when you go to plot it, this time by using the `linestyle` argument.

In the following example, each yearly revenue line is given its own color and line style:

```
fig, ax = plt.subplots()
ax.plot(months, slice_2019['Revenue'], c = 'grey', linestyle = 'dashed')
ax.plot(months, slice_2020['Revenue'], c = 'green')
```

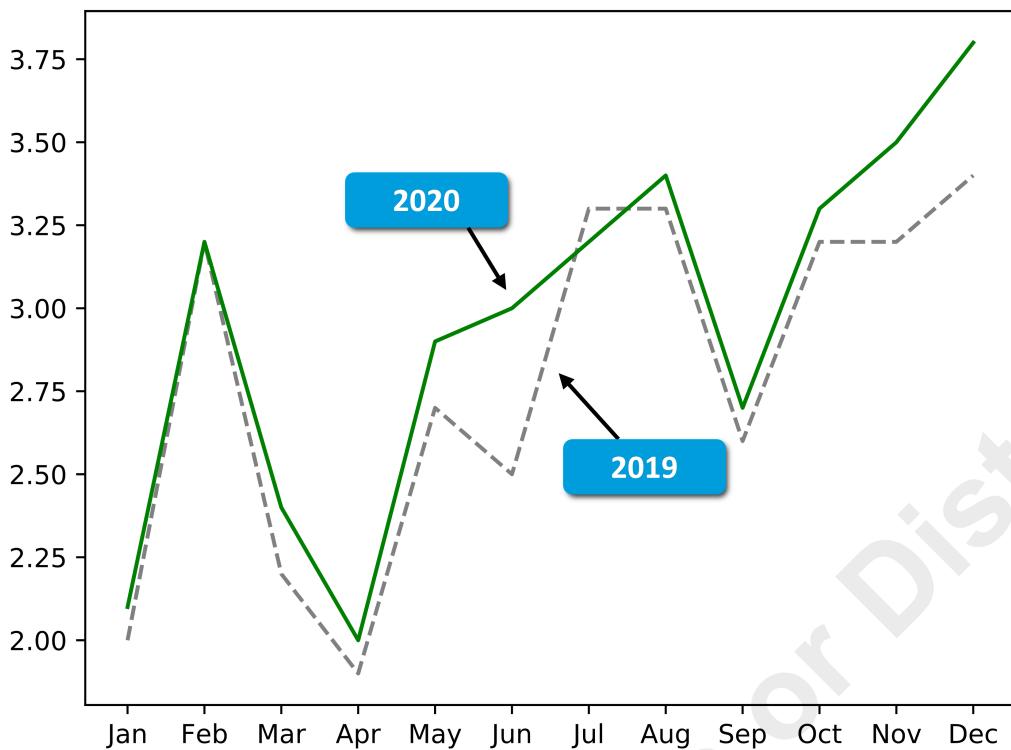


Figure 6-16: Plotting revenue per month where each year has a different color and style of line.

Color and Line Style Formats

Supplying the name of the color is just one of many methods for specifying color. The following are some other common methods you can use:

Method	Example
X11 color name	'dodgerblue'
xkcd color name	'xkcd:cerulean'
Short color code	'b'
RGB color code (decimal form)	(0, 0.616, 0.863)
RGB color code (hexadecimal form)	'#009ddc'
Grayscale value (decimal form)	'0.3'

Likewise, there are two ways of supplying the line style:

Name	Symbol
'solid'	' - '
'dashed'	' -- '
'dashdot'	'-..'
'dotted'	' : '

Color Settings in Other Plot Types

Just as you can set line colors in line plots, you can also set the color of elements in other types of plots. Many plotting functions accept a `color` or `c` argument that you can use in much the same way.

Color Maps

A **color map** is a collection of multiple colors that are assigned to plotted values in different ways. The purpose of a color map is to generate colors based on the values being plotted. A color map can reveal meaning and relationships between data points by using gradations of color, often by representing a third dimension like magnitude. Color maps are, therefore, more functional than just an aesthetic choice that you would get with a style or by applying colors individually yourself.

To apply a color map, specify the name of the map in the `cmap` argument of the plotting function you're using. For example, recall the anomalous temperature surface plot from earlier. It used the '`viridis`' color map, which starts at a dark purple at low values and goes from blue to green to yellow as values increase. This is somewhat appropriate for conveying changes in temperature, but a simpler color map might do a better job:

```
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')
ax.plot_surface(lon_grid, lat_grid, temp_anomaly, cmap = 'Reds')
```

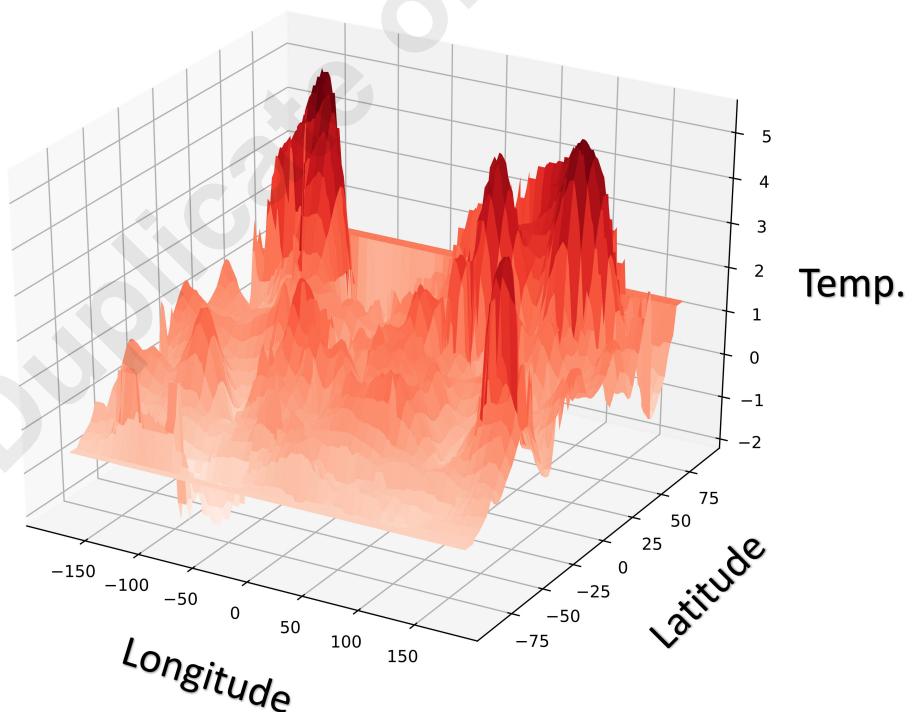


Figure 6–17: The surface plot with a simpler color map.

Now the surface plot, in addition to demonstrating relative temperature changes in three-dimensional peaks and valleys, also demonstrates those changes by the intensity of a single color.



Note: Many people have some degree of color blindness, which can make interpretation of color-mapped data difficult. If you plan on sharing your plots with colleagues or presenting plots to an audience, consider avoiding color maps that include both red and green, as red-green color blindness is the most common type.

Available Color Maps

To get a list of all color maps that are currently available on your system, enter the following code:

```
plt.colormaps()
```

Color Map Selection

Choosing the right color map for the problem you're trying to solve is challenging. Maps differ not just in the choice of the individual colors, but in the differences between those colors, and the amount of different colors. There are actually four conceptual categories of color map:

- **Sequential**—These maps typically use a single hue that changes in lightness or saturation over increments, though they can also use multiple hues. Sequential maps are best applied to data with a natural order, like increasing or decreasing numerical values (e.g., temperature changes).

Sequential



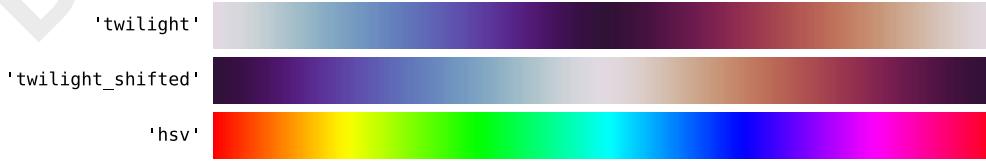
- **Diverging**—These maps typically use two different colors that change in lightness or saturation, where each color meets in an unsaturated middle. Diverging maps are best applied to data where the middle values are of special prominence (e.g., elevation as it relates to sea level).

Diverging

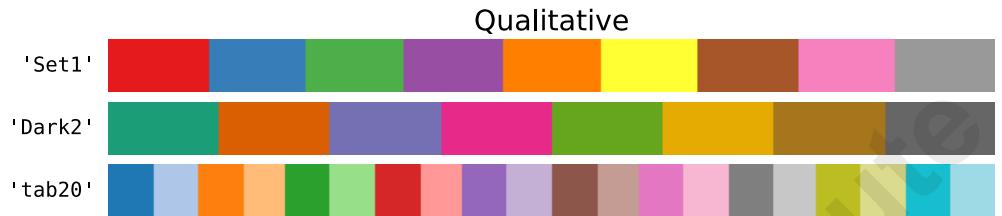


- **Cyclic**—These maps typically use two or more different colors that change in lightness, where each color meets in an unsaturated middle *and* an unsaturated beginning and end. Cyclic maps are best applied to data where the beginning and end values wrap around each other (e.g., time of day).

Cyclic



- **Qualitative**—These maps use numerous different colors. Unlike the other map types, qualitative maps do not show the relationship between values and are meant for categorizing non-ordinal data (e.g., classes of products in a catalog).



There are also some miscellaneous color maps that don't fit into any of these categories, like the 'prism' map.

Title Labels

Giving a plot a title is a good way to summarize the overall purpose of a plot or group of subplots. That way, someone with adequate knowledge will be able to get the gist of what the plot is showing without being told. Matplotlib won't be able to guess your intentions in plotting a set of data, so it's up to you to specify a title.

You can specify the title for a subplot by calling `set_title()` on the desired `Axes` object. The required argument is the label itself, a string. You can also specify the alignment of the title through `loc` (the default is center alignment), and its padding through `pad` (offset from the top of the axis). The following example gives titles to two subplots:

```
fig, ax = plt.subplots(1, 2)
ax[0].plot(months, slice_2019['Revenue'])
ax[1].plot(months, slice_2020['Revenue'])
ax[0].set_title('2019 Revenue', loc = 'left', pad = 15)
ax[1].set_title('2020 Revenue', loc = 'right', pad = 15)
```

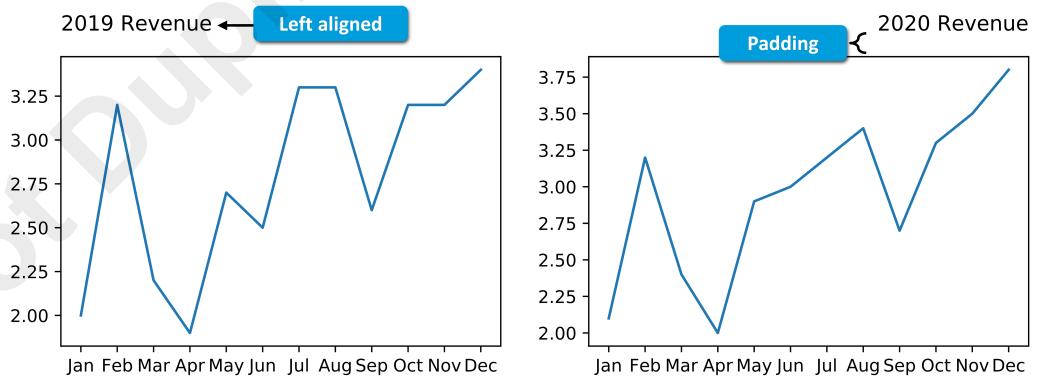


Figure 6-18: Two subplots with titles.

However, what if you want to title an entire subplot grid instead of, or in addition to, each individual subplot? The function for that is `pyplot.suptitle()`, and you can call it on your entire `Figure` object:

```
... # Statements that plot subplots and give them individual titles.
fig.suptitle('2019 Expenses')
```

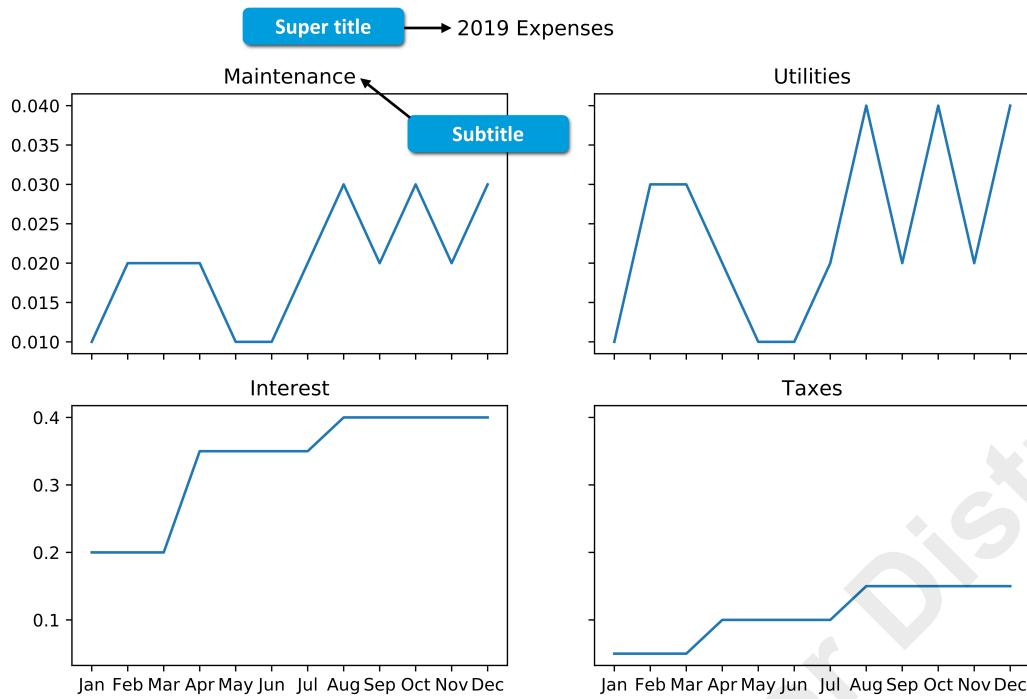


Figure 6-19: The entire subplot grid has a title, as does each individual subplot.



Note: Do not confuse the `suptitle()` function with the similarly spelled `subtitle()` function.

Axis Labels

If you don't properly label your axes, especially if you're working with numerical values, it will be difficult or even impossible for your audience to know what they represent. Matplotlib isn't always smart enough to automatically label your axes, even if you provide a `DataFrame` with a column label as an axis. So, in many cases, you'll need to manually populate labels for your plots' axes.

To set an axis label, call `set_xlabel()` and/or `set_ylabel()` on the desired `Axes` object for the x-axis and y-axis, respectively. Other than the label string itself, both functions take a `labelpad` argument with which you can specify the padding to use (offset from the axes' boundary, including any ticks). The following code gives labels to a scatter plot comparing income with age:

```
fig, ax = plt.subplots()
ax.set_title('Richland Income Based on Age')
ax.set_xlabel('Age', labelpad = 15)
ax.set_ylabel('Income (USD)', labelpad = 15)
ax.scatter(census_df['Age'], census_df['Income'])
```

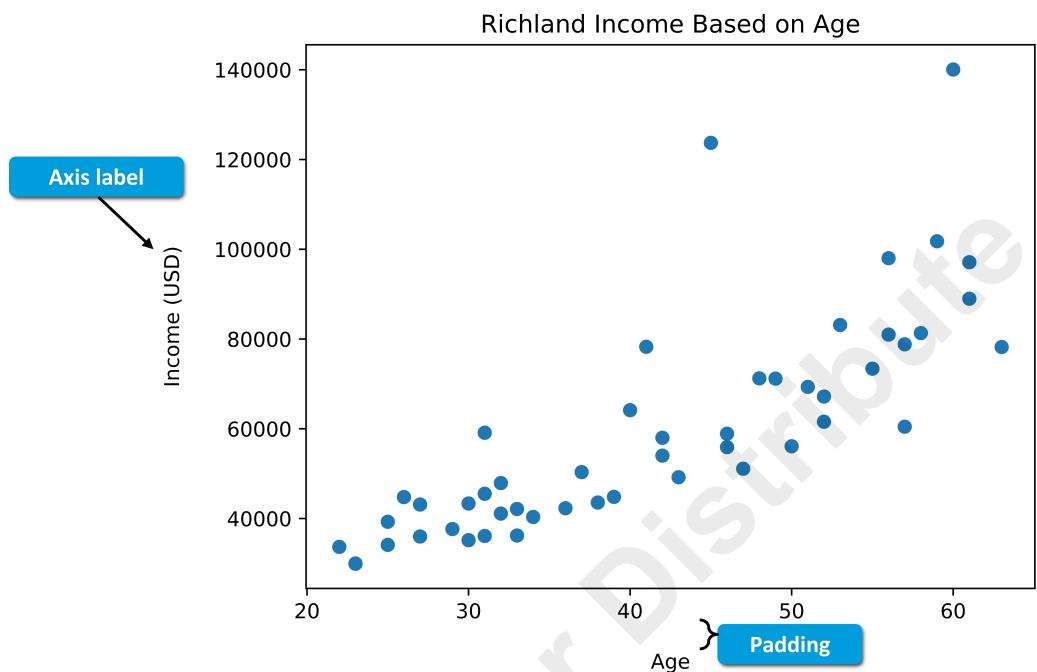


Figure 6–20: The generated plot with labels for both the x-axis and y-axis.

Turning Off Axis Information

Even if you don't provide a label, the axis text that automatically gets plotted might be more than you want to show. Simply call `ax.set_axis_off()` to hide axis information.

Axis Ticks

A **tick** is a value that aligns with an axis that is used to visually demonstrate where a data point falls on that axis. By default, each axis usually has multiple ticks that fill out the entire axis. In many cases, the default tick behavior will suffice, but there may be times when you want to change it. There are many ways to change axis ticks, and several methods for implementing those changes. One method is to use the `tick_params()` function on your `Axes` object. This function enables you to adjust various visual properties of the ticks. It has multiple arguments, including:

- `axis` —Specify whether the operation should apply to the x-axis, y-axis, or both.
- `which` —Specify which type of ticks the operation should apply to: major or minor. Major ticks are the most common and have larger gaps, and minor ticks are less common and have smaller gaps.
- `direction` —Specify whether the ticks should be inside the axis, outside the axis, or both.
- `length` —Specify the length of each tick.
- `width` —Specify the width of each tick.
- `color` —Specify the color of each tick.
- `pad` —Specify the distance between the ticks and the axis label.
- `labelsize` —Specify the font size of the tick labels.
- `labelrotation` —Specify the rotation of the tick labels.

The following code alters the ticks in the revenue line plot to make them a little more visually interesting:

```
... # Plotting and labeling code.  
ax.tick_params(axis = 'x', direction = 'out', length = 8, width = 2,
```

```
color = '#009ddc', labelsize = 12, labelrotation = -30)
ax.tick_params(axis = 'y', direction = 'in', length = 10, width = 2,
               color = '#009ddc')
```

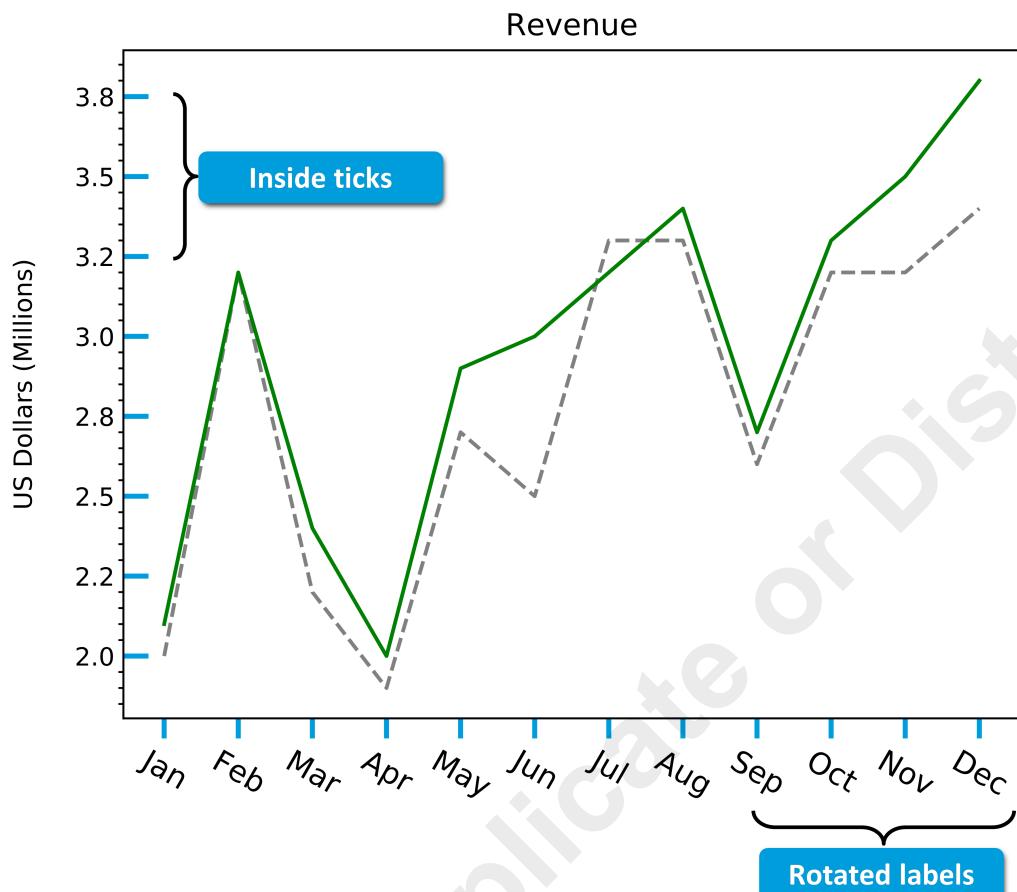


Figure 6-21: The generated plot with modified ticks.

The Locator and Formatter Classes

For more fine-grained control over your ticks, you can modify the properties of Locator and Formatter classes that are included with each axis. A Locator determines where to place ticks, and a Formatter determines how to format ticks. Each is controllable through the `xaxis` or `yaxis` properties of the `Axes` object, using the `set_major/minor_locator()` or `set_major/minor_formatter()` functions. When you call the function, you supply a class that pertains to what you're trying to do. For example, to set minor ticks on the y-axis:

```
from matplotlib.ticker import AutoMinorLocator
ax.yaxis.set_minor_locator(AutoMinorLocator())
```

The `AutoMinorLocator()` class automatically determines where to place the minor ticks. Note that you must import this class from the `ticker` module before using it.

That's an example of Locator. The following is an example of using Formatter with a string formatter to show only one decimal place for the tick labels:

```
from matplotlib.ticker import FormatStrFormatter
ax.yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
```



Note: For a list of tick Locator classes, visit: https://matplotlib.org/gallery/ticks_and_spines/tick-locators.html



Note: For a list of tick Formatter classes, visit: https://matplotlib.org/gallery/ticks_and_spines/tick-formatters.html

Legends

Legends can provide much-needed context to your plots. For example, it's not immediately obvious in the plot showing revenue what each line is referring to. Obviously, the values progress over a period of 12 months, so your audience might infer that each line is a separate year, but you shouldn't leave something like that up to their imaginations. What's more, they won't necessarily know *which* years are being represented. This is a good scenario for a legend.

The `pyplot.legend()` function controls the presence and appearance of legends. If you assign a `label` to your `plot()` call, simply calling `legend()` without any arguments generates the legend automatically. But, there are plenty of arguments you can use to configure how the legend looks, including:

- `loc` —Specify the location of the legend with respect to the rest of the figure.
- `frameon` —Specify whether or not the legend is drawn with a frame.
- `shadow` —Specify whether or not to draw a shadow behind the legend.
- `framealpha` —Specify the transparency of the frame.
- `borderpad` —Specify the whitespace padding inside the legend border.
- `borderaxespad` —Specify the padding between the legend border and the axes.

The following example adds a legend to the revenue chart:

```
... # Plotting and labeling code.
ax.legend(loc = 'lower right', frameon = False, borderaxespad = 1.5)
```

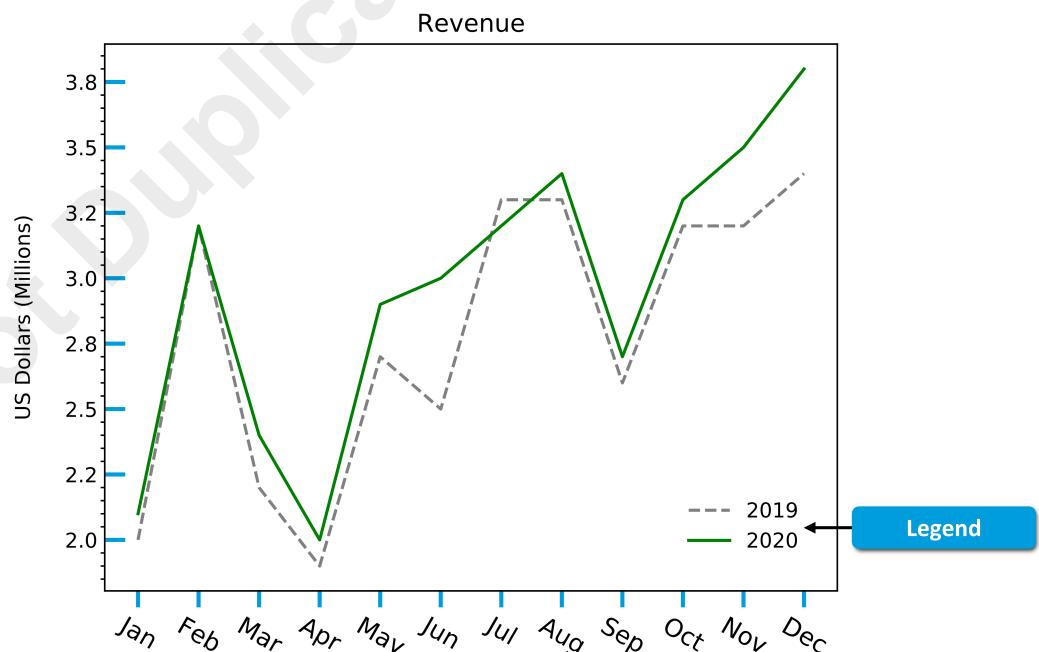


Figure 6–22: The generated plot with a legend.

Color Bars

Standard legends are most useful when plotting categorical data or data that has a small number of discrete data points. However, continuous data is not well represented by such a legend, as it would be too unwieldy. Color bars, on the other hand, are a great tool for communicating the meaning of data that follows a gradation. As the name suggests, a **color bar** includes the range of colors plotted on the main graph. The data values assigned to each color are placed on the color bar similar to how data values are plotted on a typical axis—labels, ticks, and all. The colors on the main plot help show values as they relate to each other, but the color bar can give you a better view into the quantitative measurements of the data.

In Matplotlib, a color bar is just another axis to add to the figure. You can use `pyplot.colorbar()` directly if you're using a stateful approach; otherwise, you can call `colorbar()` on a `Figure` object. In either case, you may need to specify the `mappable` argument, which determines the color mapping to use in the color bar itself. If you're using the object-oriented approach, just pass in the plot containing the appropriate color map as the argument:

```
... # Formatting code.
plot = ax.plot_surface(lon_grid, lat_grid, temp_anomaly, cmap = 'Reds')
fig.colorbar(plot)
```

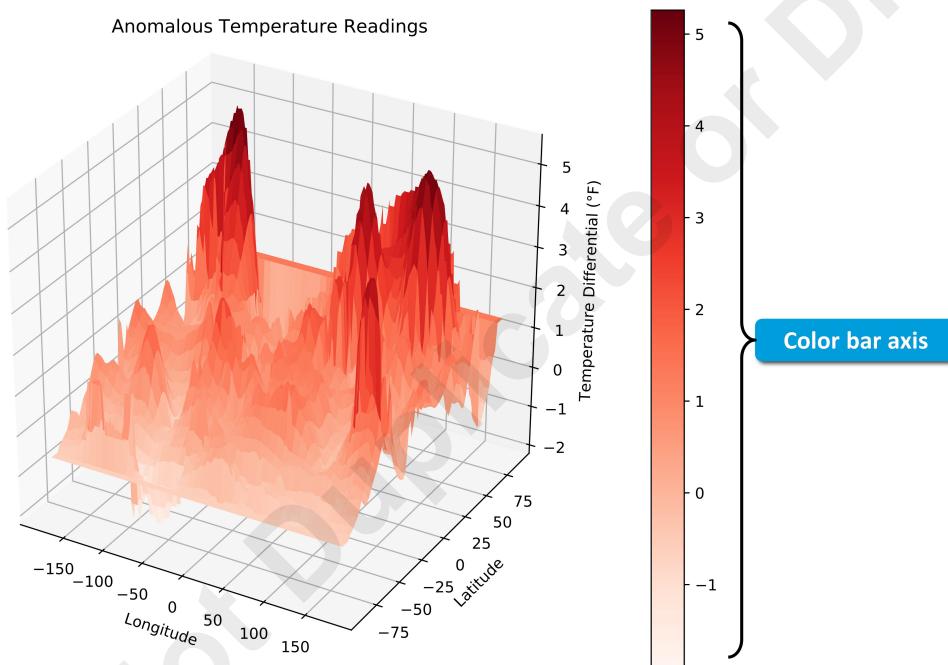


Figure 6–23: The generated plot with a color bar, where temperature differentials are plotted along a color gradient.

Color Bar Customization

Because a color bar is essentially just another axis, you can configure its appearance like any other axis, including its labels, ticks, etc. Some properties you can configure include:

- `orientation`—Specify whether the color bar is horizontal or vertical.
- `pad`—Specify the padding to use between the color bar and the main plot axes.
- `shrink`—Specify the fraction by which to multiply the size of the color bar.
- `aspect`—Specify the aspect ratio of the color bar.
- `extend`—Specify that the color bar should have pointed ends if a value is out of range.
- `ticks`—Specify the ticks to use or a `Locator` object.

- `format` —Specify a `Formatter` object to use for the ticks.

Axis Limits

By default, Matplotlib usually plots your data on axes that conform to the minimum and maximum values of that data. However, there may be situations where you want more control over the upper and lower bounds of the axes. Perhaps you have outliers that you would rather not show. Or, maybe you want to contract the axes to focus on what you think matters the most, while cutting out distractions; alternatively, you may wish to expand the axes beyond the default to ensure your audience has the full picture.

You can use the `set_xlim()` and `set_ylim()` functions to set limits on an `Axes` object. The former takes `left` and `right` arguments, and the latter takes `bottom` and `top`. These are essentially the lower and upper bounds you want to constrain (or expand) the axes to. You can specify both values, or you can specify just one.

In the following example, the bottom and top limits of the y-axis are being expanded for the revenue chart. The rationale for the bottom is that the audience may quickly glance at the chart and mistakenly assume that revenue started at 0 because the line begins at the bottom corner. For the top, you want to leave more space to reinforce the idea that there is still room for growth, whereas before one might assume that the growth has hit its limit.

```
... # Plotting and formatting code.  
ax.set_ylim(bottom = 0, top = 5) # Start at 0, end at 5.
```

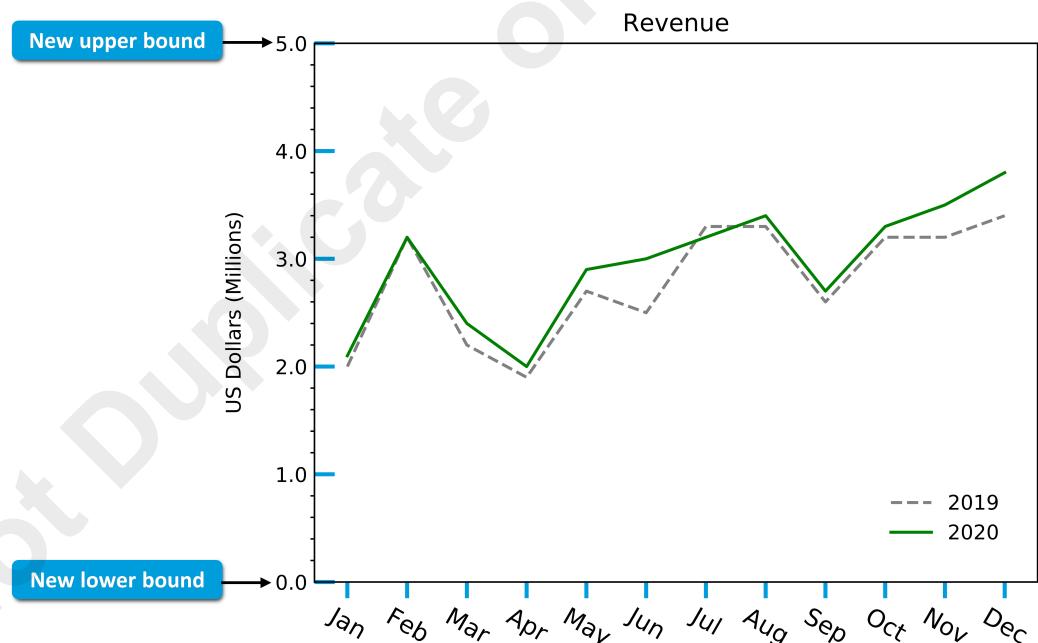


Figure 6–24: The generated plot with new axis limits.

Axis Scales

Setting limits is a way of scaling the appearance of data on a plot. If you want to scale the data itself before plotting, you can accomplish this by using the `set_xscale()` and/or `set_yscale()` functions. The main argument these functions take is `value`, through which you supply the type of scaling. The default scaling types that are supported are:

- `'linear'` —Linear scaling. This is the default.
- `'log'` —Logarithmic scaling (positive values).

- 'symlog' —Symmetrical logarithmic scaling (positive or negative values).
- 'logit' —Logit scaling (positive values less than 1).

Grids

Plots that involve a lot of precise numbers with a great deal of variability can be difficult to read, especially if you want to determine exactly where a data point lies on each axis in a line graph. Major and minor ticks are meant to help with this, but they can only do so much. In some cases, you might want to consider adding a grid to your plot so that picking out specific data values is easier. You can easily use your eyes to trace the grid lines from one axis to another.

The `grid()` function draws a grid according to your specifications. There are two main arguments: `which` specifies major or minor grid lines, much like major or minor ticks; and `axis` is which axis to apply the grid lines to (or both). There are many other formatting options you can leverage, including some you've seen before like `linestyle`, `linewidth`, `color`, etc. The following code adds grid lines to the revenue chart:

```
... # Plotting and formatting code.
ax.grid(which = 'both', axis = 'both', color = '#d8dcd6')
```

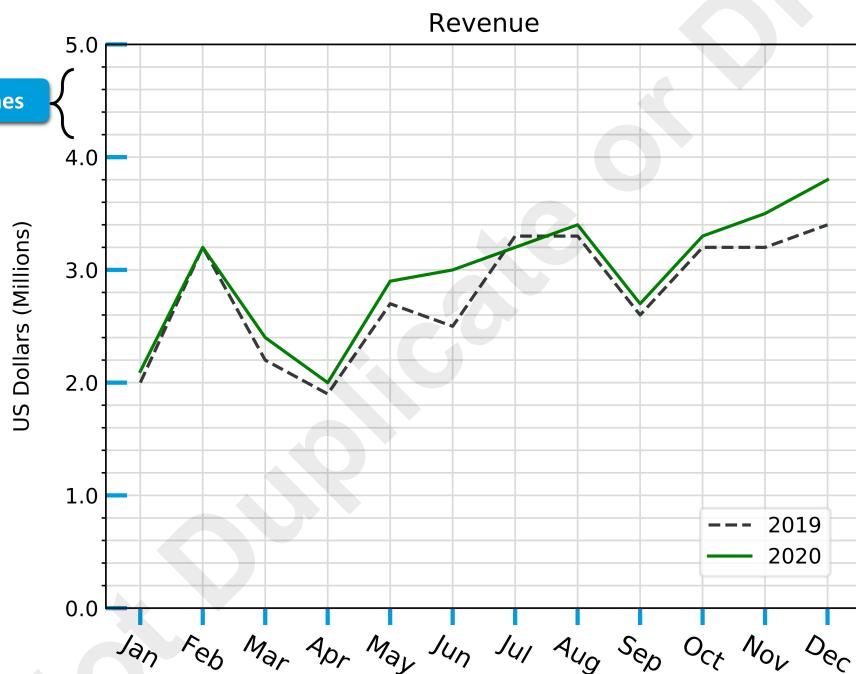


Figure 6–25: The generated plot with grid lines.



Note: Just like not having a grid can make it difficult to read a chart, including a grid can do likewise if it just unnecessarily adds to the noise of the chart.

Text Properties

Ensuring the text on your plots is readable is just as important as ensuring you have the proper axis limits, ticks, legends, and other informative elements. You want the text to be a good size according to its role (e.g., a subplot title should typically be larger than a tick label), a color that makes the text pop when you need it to, an alignment that makes sure the text doesn't interfere with other parts of the plot, and much more.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2021

Most Matplotlib functionality that incorporates text leverages a `Text` object. This object has various properties you can use to set the look and feel of the text. Functions like `set_title()` and `set_xlabel()` enable you to pass keyworded arguments that set these properties on the `Text` object. Some of these properties include:

- `alpha` — Set the transparency of the text.
- `backgroundcolor` — Set the text's background color.
- `color` — Set the text color.
- `size` — Set the font size.
- `weight` — Set the font weight.
- `family` — Set the font family.
- `name` — Set the specific font.
- `rotation` — Set the text's rotation.



Note: For a full list of text properties, visit https://matplotlib.org/3.1.1/tutorials/text/text_props.html.

Other functions, like `tick_params()`, have their own function-specific arguments for altering text.

The following example modifies the text properties of the plot title and the y-axis label:

```
... # Plotting and formatting code.
ax.set_title('Revenue', color = '#009ddc', size = 14,
             weight = 'bold', family = 'serif')
ax.set_ylabel('US Dollars\n(Millions)', labelpad = 50,
              rotation = 'horizontal', weight = 'bold')
```

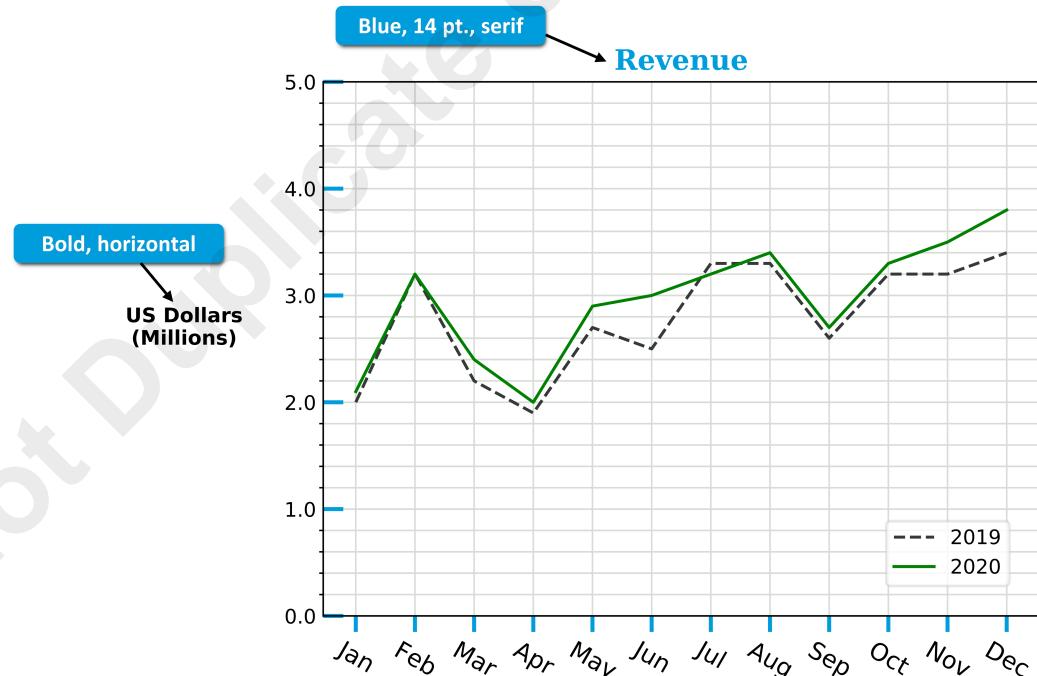


Figure 6–26: The generated plot with modification to text properties.

Annotation

Sometimes, the plot generated from your data won't be able to tell the whole story on its own. You may wish to call out specific data points, or specific trends in a graph, or anything else of interest to

your audience. Annotations enable you to add text and other elements to the plot to enhance their storytelling capabilities.

You can add a standard text annotation to a plot by calling the `text()` function on an `Axes` object. The `s` argument takes the text string itself. The `x` and `y` arguments take the `x` and `y` coordinates, respectively, of where you want the text to appear. You can also supply any of the typical `Text` object properties that have been mentioned. The following example adds a possible explanation for a lull in revenue by using a text annotation:

```
ax.text(x = 'Jun', y = 2.2, s = 'Product Recall',
        horizontalalignment = 'center', color = 'darkred')
```

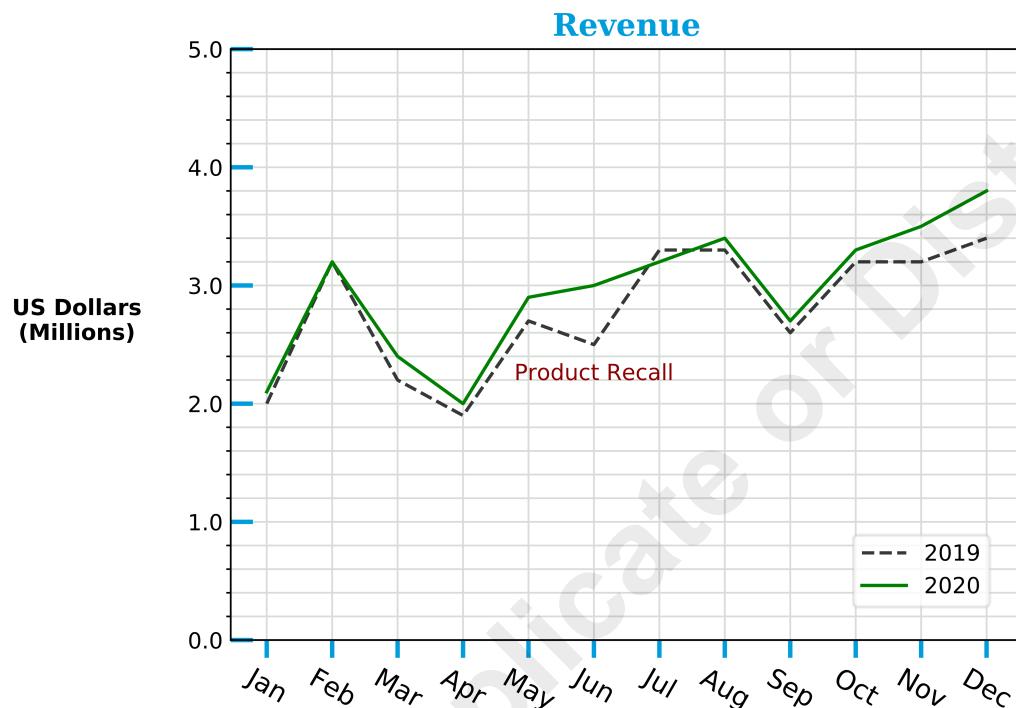


Figure 6-27: The generated plot with a simple text annotation.

Another option is to use the `annotate()` function, which can add both text and an arrow, and has some advanced functionality. Instead of individual `x` and `y` arguments, it takes a tuple `xy` of both coordinates. If you go with text only, you just need to supply a string to the `text` argument. However, if you want an arrow pointing to a place on the graph, with text next to the arrow, you should use `xy` to specify where the arrow points to and `xytext` to specify where to place the text. The arrow itself is defined through `arrowprops`, which can take a dictionary of various properties to format the appearance of the arrow. The following example makes it a little more clear what is being pointed out in the revenue graph by using an arrow:

```
arrow_format = {'color': 'black', 'width': 2, 'headwidth': 6}
ax.annotate('Product Recall', xy = ('Jun', 2.4),
            xytext = ('Jul', 1.7), arrowprops = arrow_format,
            color = 'darkred')
```

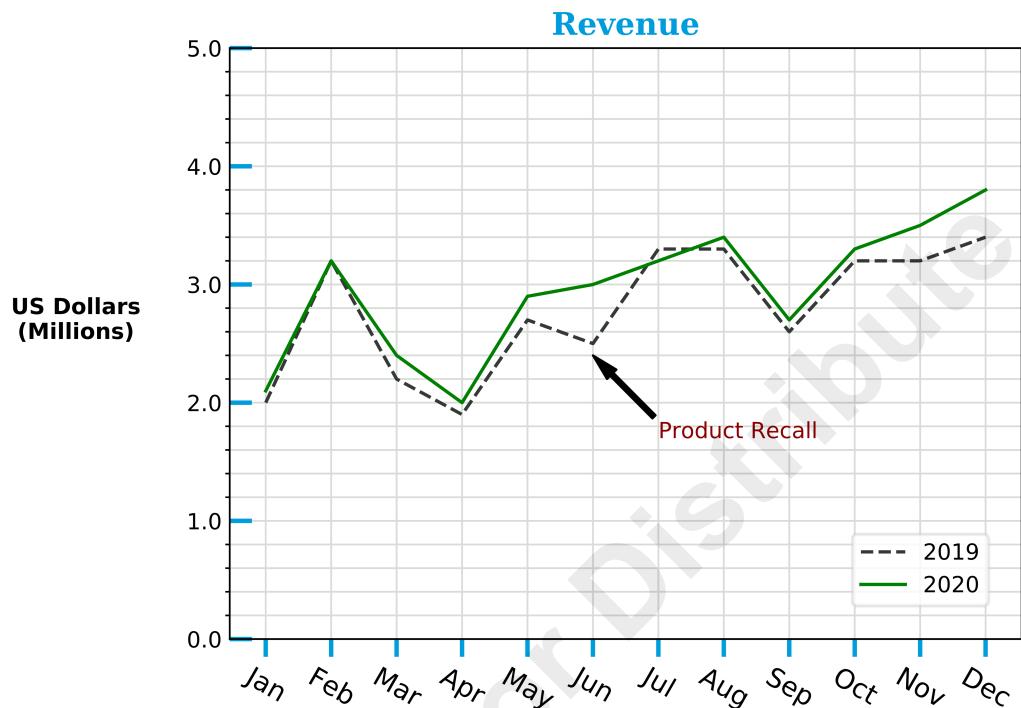


Figure 6-28: The generated plot with an arrow annotation.

Stateful vs. Object-Oriented Function Differences

Several of the functions for formatting plots discussed thus far are specific to the object-oriented interface. Their stateful equivalents have slightly different names. The following table maps the functions that differ between both interfaces. Functions not listed in the following table have the same name, regardless of which interface you use.

Stateful Function	Object-Oriented Function
<code>pyplot.title()</code>	<code>Axes.set_title()</code>
<code>pyplot.xlabel()</code>	<code>Axes.set_xlabel()</code>
<code>pyplot.ylabel()</code>	<code>Axes.set_ylabel()</code>
<code>pyplot.xlim()</code>	<code>Axes.set_xlim()</code>
<code>pyplot.ylim()</code>	<code>Axes.set_ylim()</code>

The `Axes.set()` Function

Setting a bunch of different titles, labels, and limits on a plot can quickly get tedious if you're calling each function line by line. Thankfully, Matplotlib provides a shortcut for setting these elements. The `Axes.set()` function takes one or more keyword arguments that map to a related "set" function—namely, the object-oriented functions mentioned in the previous table. So, rather than call each function individually, like so:

```
ax.set_title('Revenue')
ax.set_ylabel('US Dollars\n(Millions)')
ax.set_ylim((0, 5))
```

You can instead call `set()` once:

```
ax.set(title = 'Revenue', ylabel = 'US Dollars\n(Millions)', ylim = (0, 5))
```

However, if you need to include multiple arguments for an element you set, it's best just to call that element's function directly.

Guidelines for Formatting Plots

Follow these guidelines when you are formatting plots in Matplotlib.

Format Plots

When formatting plots:

- Use the `style.use()` function to set an overall style to your plots.
- Change the styles and colors of lines and other plot elements to emphasize or de-emphasize certain portions of a plot.
- Consider using colors that are easy for people with color blindness to read.
- Use color maps to demonstrate a gradation of changing values, especially with 3-D plots.
- Choose a color map type that best supports the data you're plotting:
 - Sequential maps for data with a natural order.
 - Diverging maps for data with prominent middle values.
 - Cyclic maps for data with wrapping start and end values.
 - Qualitative for categorical data.
- Apply title labels to subplots and entire figures to briefly explain the overall purpose of a subplot or figure.
- Apply a label to each axis of a plot that indicates which variable the axis is representing.
 - Consider omitting axis labels if the tick labels make it obvious what the variable is (e.g., months of the year).
- Consider customizing how axis ticks are plotted and labeled to make it easier to identify specific points.
- Add legends and color bars to plots to explain what different styles and/or colors refer to.
- Consider expanding or constraining axis limits to adjust the scale in which data appears.
- Consider adding grids to plots to make it easier to trace individual data points.
- Adjust the formatting of text to emphasize or de-emphasize certain labels.
- Consider adding an annotation to a plot where it's necessary to point out something important, like a sharp deviation.
- Avoiding cluttering your plots with too many formatting elements, like grids, ticks, annotations, etc.
- Consider applying formatting using the `Axes.set()` function to keep simple formatting to a single statement.

ACTIVITY 6–4

Formatting Plots

Data Files

/home/student/DSTIP/Matplotlib/Formatting Plots.ipynb
 /home/student/DSTIP/Matplotlib/data/stores_data_full_clean.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've been generating more and more visualizations that have helped you analyze the Greene City Emporium store data, but you also need to think about how you're going to present your findings to management. Since you're the one who's been creating these visualizations, it's easy for you to interpret them—but the same cannot be said for someone else, especially someone without the same technical knowledge as you. So, you need to format your plots to not only make them look nice, but also to provide context that is necessary for others to understand the stories you're trying to tell through these visuals.

To begin with, you have two visualizations you want to eventually present to business leaders at GCE: a line graph showing how revenue fluctuated between January and February on a day-by-day basis, and a group of scatter plots showing how unit price and other key attributes have an effect on gross income. You'll dress up these visualizations to make them presentable.

1. In Jupyter Notebook, open the **DSTIP/Matplotlib/Formatting Plots.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
 Lines 15 through 18 load the full `stores_df` DataFrame.
 - b) Run the code cell.
3. Generate revenue line plots using a built-in style.
 - a) Scroll down and view the cell titled **Generate revenue line plots using a built-in style**, and examine the code listing below it.

```

1 jan = stores_df[stores_df['Date'].dt.month == 1]
2 feb = stores_df[stores_df['Date'].dt.month == 2]
3
4 jan_rev = stores_df.groupby(jan['Date'].dt.day)[['Revenue']].sum()
5 feb_rev = stores_df.groupby(feb['Date'].dt.day)[['Revenue']].sum()
6
7 jan_rev.head()
```

This code gets the sum of each day's revenue for January and February.

- b) Run the code cell.

- c) Examine the output.

```
Date
1.00    4,519.22
2.00    1,852.86
3.00    1,979.17
4.00    1,546.37
5.00    3,368.27
Name: Revenue, dtype: float64
```

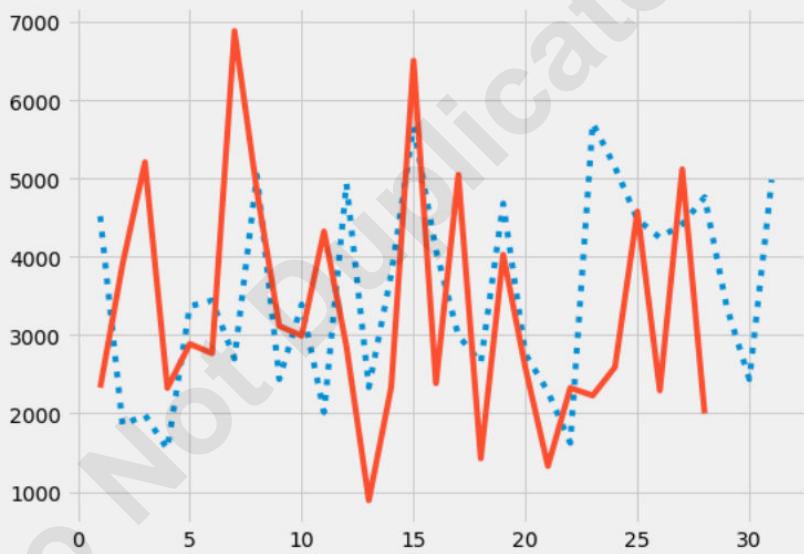
- d) Select the next code cell, and then type the following:

```
1 plt.style.use('fivethirtyeight')
2 fig, ax = plt.subplots(figsize = (8, 6))
3 ax.plot(jan_rev.index, jan_rev, linestyle = ':', label = 'January')
4 ax.plot(feb_rev.index, feb_rev, label = 'February')
```

- Line 1 sets the style that Matplotlib will use on the plots that follow. This particular style emulates the style used by opinion poll analysis website *FiveThirtyEight*.
- Lines 3 and 4 plot the revenue for each month on the same `Axes` object. Each line includes a `label` that will be used for identification in a legend.
- Line 3 sets the line style of the January plot to dotted to distinguish it from February's default line style (solid).

- e) Run the code cell.
f) Examine the output.

```
[<matplotlib.lines.Line2D at 0x7f4148956a10>]
```



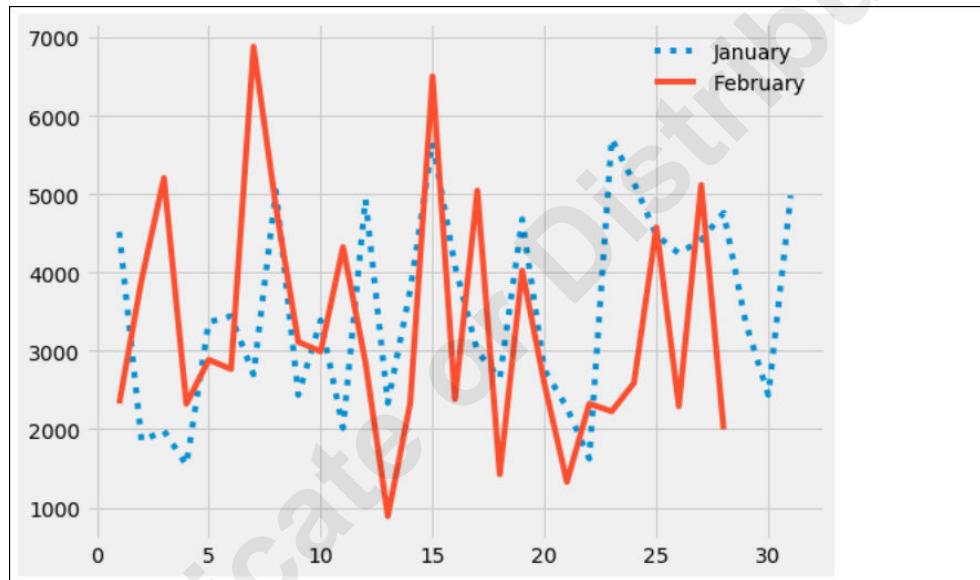
- The '`fivethirtyeight`' style has more visual "pop" than the default Matplotlib style. It includes a grey background, a grid, and thick data lines. It also uses different default colors for the data lines.
- The default January line style was overridden to use a dotted format. This makes it less prominent than the solid February line. You might do this if February has just concluded and you want to show how it performed as compared to the previous month's. February is the more immediate concern in this scenario, so it should catch the eye more than January.
- Although this style is a good start, the figure is still missing some key components.

4. Add a legend, title, and axis labels to the plot.

- Scroll down and view the cell titled **Add a legend, title, and axis labels to the plot**, then select the code cell below it.
- In the code cell, type the following:

```
1 ax.legend(frameon = False)
2 fig
```

- Run the code cell.
- Examine the output.



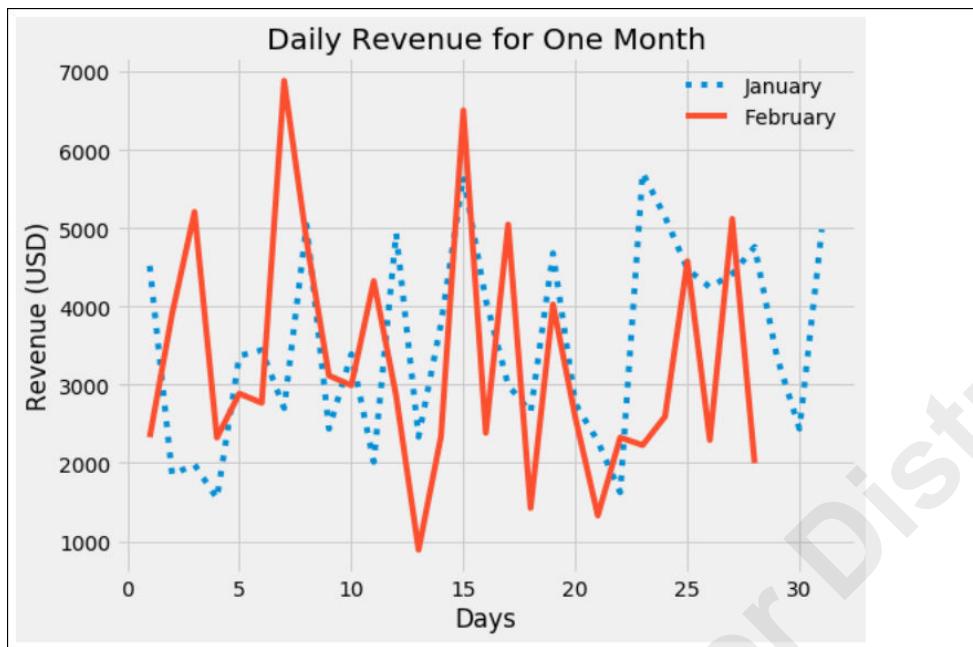
- A legend is added to the top-right corner of the figure.
 - The legend is automatically generated based on the `label` values you provided during plotting.
- Select the next code cell, and then type the following:

```
1 ax.set(title = 'Daily Revenue for One Month',
2        xlabel = 'Days', ylabel = 'Revenue (USD)')
3 fig
```

The `set()` function sets multiple parameters in one statement. In this case, it is setting:

- The title of the figure.
 - The label of the x-axis.
 - The label of the y-axis.
- Run the code cell.

- g) Examine the output.



The addition of a title and axis labels provides much-needed context to the intended audience. There are still a few more tweaks to make, however.

5. Change axis limits and modify axis ticks.

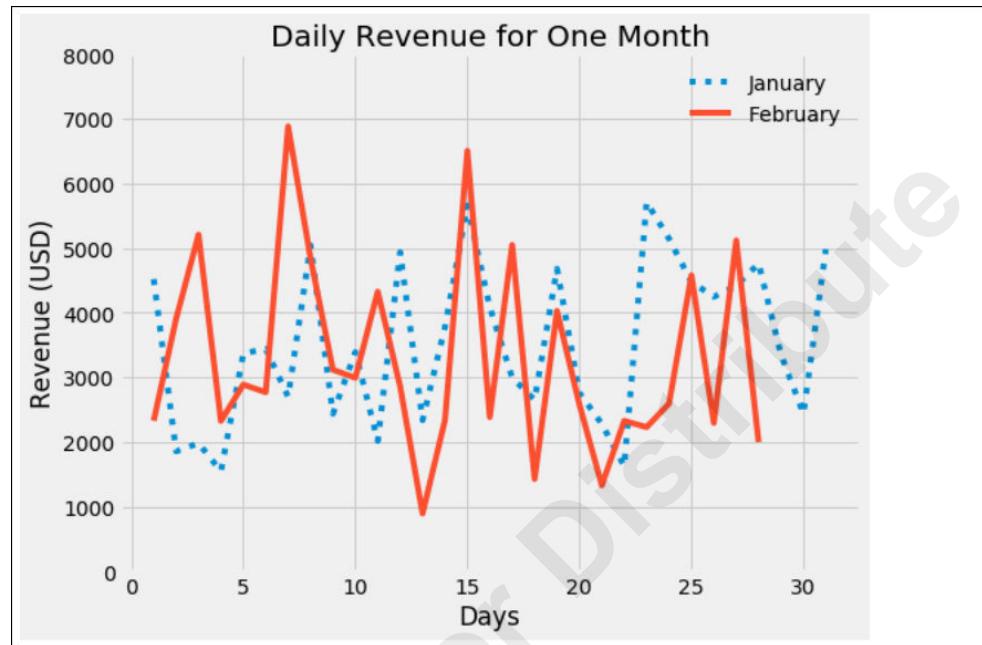
- a) Scroll down and view the cell titled **Change axis limits and modify axis ticks**, then select the code cell below it.
- b) In the code cell, type the following:

```
1 ax.set_ylim(bottom = 0, top = 8000)
2 fig
```

This expands the limits of the y-axis so that it starts at 0 and ends at 8,000. Previously, the y-axis was bounded by the lowest and highest values of the data, as you can see in the previous output.

- c) Run the code cell.

- d) Examine the output.



Now that the y-axis limits have expanded, the graph has a little more vertical breathing room.

- e) Select the next code cell, and then type the following:

```

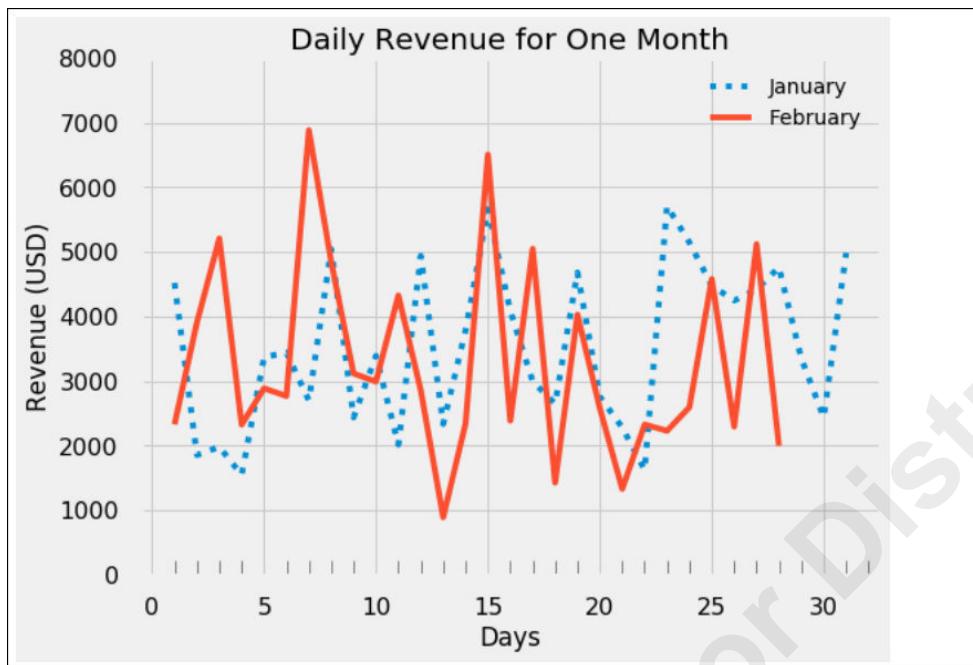
1 from matplotlib.ticker import AutoMinorLocator
2
3 ax.tick_params(axis = 'both', pad = 15, labelsize = 16)
4 ax.xaxis.set_minor_locator(AutoMinorLocator())
5 ax.tick_params(axis = 'x', which = 'minor', direction = 'in',
6                 length = 10, width = 1, color = 'grey')
7 fig

```

- Line 1 imports the `AutoMinorLocator` class from the `ticker` module. This is a `Locator` class that is necessary in order to add minor ticks to the figure.
- Line 3 changes the tick parameters so that the ticks on both axes are offset from their labels, and the text size of the labels is increased.
- Line 4 uses the `AutoMinorLocator` class to add minor ticks to the x-axis.
- Lines 5 and 6 adjust the ticks on the x-axis, specifically the minor ticks, to make them actually appear with the specified formatting.

- f) Run the code cell.

- g) Examine the output.



- The tick labels on both axes are somewhat larger and are given more room as compared to the previous output.
- The minor ticks on the x-axis show up on the inside of the figure. These might make it easier for someone to determine the specific day that a revenue point is mapping to, especially if it's between the five-day range of the major ticks. For example, the February day with the lowest revenue clearly occurred on the 13th.
- There's one last change you'll make to this figure.

6. Add an annotation explaining that a particularly sharp drop in revenue was caused by a snowstorm in the region.

- Scroll down and view the cell titled **Add an annotation explaining that a particularly sharp drop in revenue was caused by a snowstorm in the region**, then select the code cell below it.
- In the code cell, type the following:

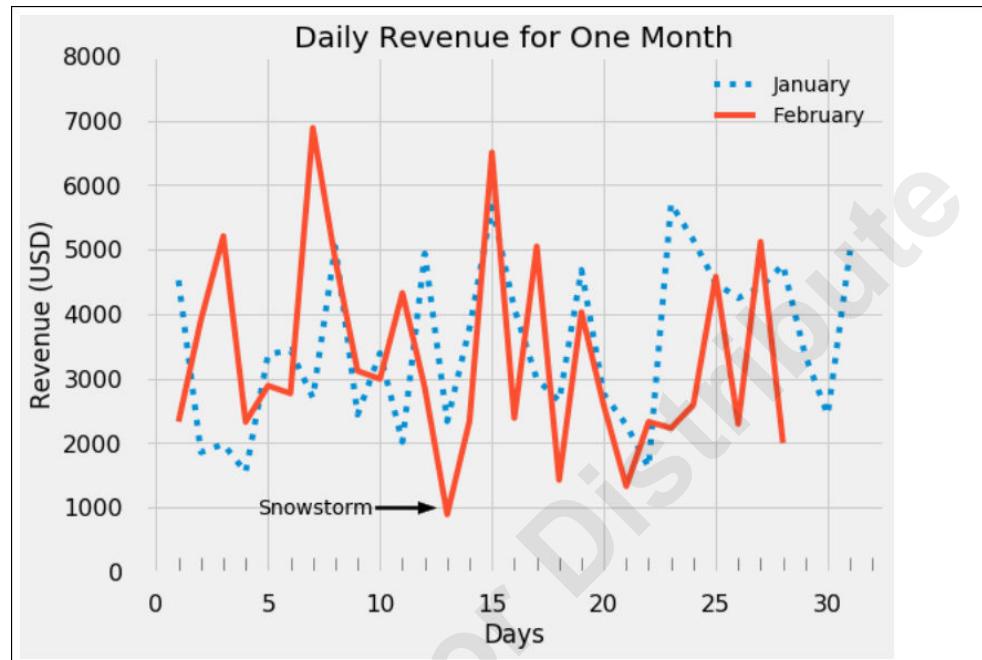
```

1 arrow_format = {'color': 'black', 'width': 2, 'headwidth': 8}
2 ax.annotate('Snowstorm', xy = (12.5, 990), xytext = (4.5, 900),
3             arrowprops = arrow_format, color = 'black')
4 fig

```

- Line 1 sets up a dictionary of arrow formatting styles.
- Lines 2 and 3 generate an annotation.
 - The `xy` argument specifies where on the figure the arrow is pointing, which is just a tuple of `x` and `y` values, respectively.
 - In this case, the arrow will be pointing slightly left of day 13 and slightly below the \$1,000 revenue threshold.
 - The `xytext` argument is the same, but positions the label text.
 - The `arrowprops` argument takes the formatting dictionary defined on line 1.
- Run the code cell.

- d) Examine the output.



- The annotation is pointing to the lowest dip in revenue to attempt to explain why it occurred.
 - Annotations like this can be helpful for explaining parts of a visual that may warrant it, but they should be used sparingly. Too many annotations can make a figure too noisy.
 - Also, adding annotations is a somewhat tedious process, as it takes some trial and error to create a good-looking and well-positioned arrow and label.
 - In all, this graph should be sufficient for presenting to a business audience.
7. Generate scatter plots of unit price and gross income, in which each plot is color mapped to COGS, quantity, or customer rating.
- a) Scroll down and view the cell titled **Generate scatter plots of unit price and gross income, in which each plot is color mapped to COGS, quantity, or customer rating**, then select the code cell below it.

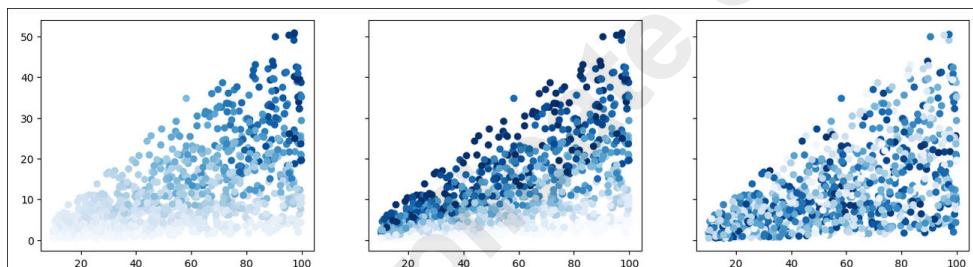
- b) In the code cell, type the following:

```

1 x = stores_df['UnitPrice']
2 y = stores_df['GrossIncome']
3 cmap = 'Blues'
4
5 plt.style.use('default')
6 fig, ax = plt.subplots(1, 3, sharey = 'row', figsize = (16, 4))
7 plot1 = ax[0].scatter(x, y, c = stores_df['COGS'], cmap = cmap)
8 plot2 = ax[1].scatter(x, y, c = stores_df['Quantity'], cmap = cmap)
9 plot3 = ax[2].scatter(x, y, c = stores_df['CustomerRating'], cmap = cmap)

```

- Lines 1 through 3 set up some variables that will be used in the upcoming function calls:
 - The x-axis for all three subplots will be unit price.
 - The y-axis for all three subplots will be gross income.
 - The color map is a sequential type that uses blue gradients. All three subplots will use this color map, so it'll be easier to compare the third attribute across the subplots.
 - Line 5 resets the style to use Matplotlib's default.
 - Line 6 sets a 1×3 figure with the y-axis shared along the entire row.
 - Lines 7 through 9 plot unit price against gross income. The `c` argument defines the set of values to map the color gradients to. In other words, the data markers for a transaction will change in color intensity depending on that transaction's COGS, quantity, and customer rating—one for each subplot.
- c) Run the code cell.
d) Examine the output.



- Each subplot shows the same somewhat linear relationship between unit price and gross income, but with different color intensities for the markers.
- Darker blue markers indicate higher values, whereas lighter blue markers indicate lower values. Note that some very low values are near white and may blend in with the background.
- These color maps show how a third dimension of the data may play a role in the relationship between the dependent and independent variables. For example, the middle subplot shows that higher quantity values tend to lead to comparatively higher income values, regardless of the unit price. Likewise, the left subplot shows that high COGS values tend to lead to high income values only if the unit price is also high. The right subplot seems to indicate that customer rating has no real bearing on the correlation between unit price and gross income.
- It's not immediately clear what value a color gradient refers to; you need color bars for that.

8. What type of color map is best used for data where the middle values are of special importance?

- Diverging
- Sequential
- Qualitative
- Cyclic

9. Assign color bars to each axis in the figure.

- Scroll down and view the cell titled **Assign color bars to each axis in the figure**, then select the code cell below it.
- In the code cell, type the following:

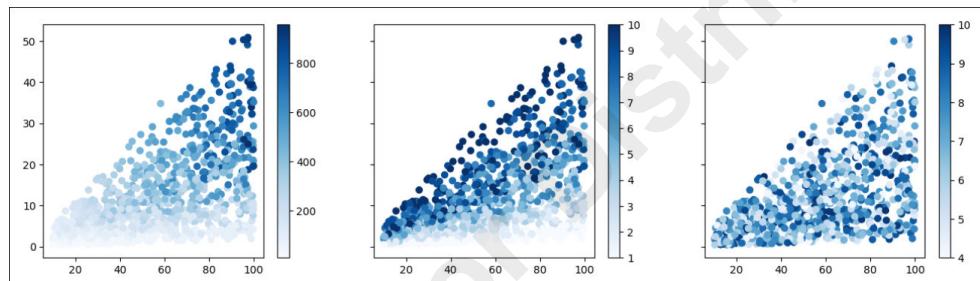
```

1 fig.colorbar(plot1, ax = ax[0])
2 fig.colorbar(plot2, ax = ax[1])
3 fig.colorbar(plot3, ax = ax[2])
4 fig

```

Each line generates a color bar for a different `Axes` object.

- Run the code cell.
- Examine the output.



- The color bars show what values the blue gradients represent.
- The figure could still use some formatting to make it easier to interpret.

10. Add titles and axis labels, and format text.

- Scroll down and view the cell titled **Add titles and axis labels, and format text**, then select the code cell below it.
- In the code cell, type the following:

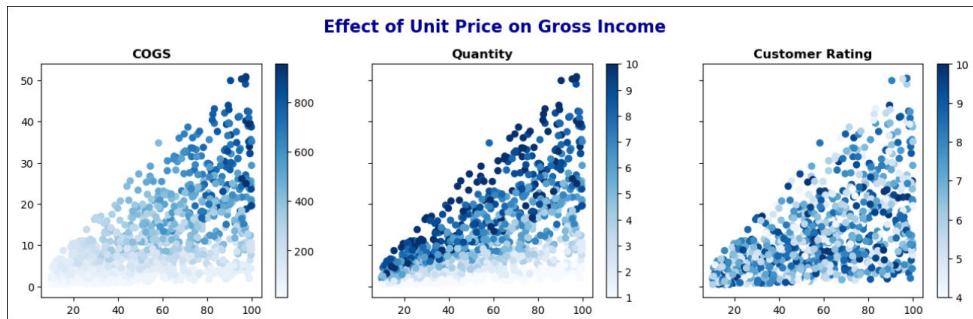
```

1 ax[0].set_title('COGS', weight = 'bold')
2 ax[1].set_title('Quantity', weight = 'bold')
3 ax[2].set_title('Customer Rating', weight = 'bold')
4 fig.suptitle('Effect of Unit Price on Gross Income', size = 16,
5               weight = 'bold', c = 'darkblue', y = 1.03)
6 fig

```

- Lines 1 through 3 set titles for each individual subplot.
- Lines 4 and 5 set the title for the entire figure, along with some additional text formatting. The `y` argument specifies where the super title should be located.
- Run the code cell.

- d) Examine the output.



The titles help make the figure easier to understand, but it still needs axis labels.

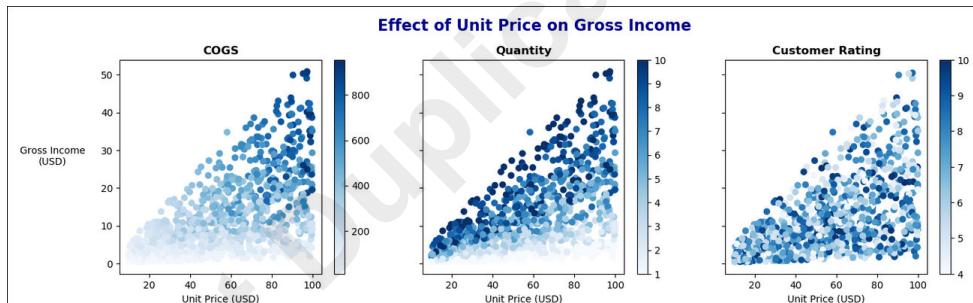
- e) Select the next code cell, and then type the following:

```

1 for i in range(0, 3):
2     ax[i].set_xlabel('Unit Price (USD)')
3
4 ax[0].set_ylabel('Gross Income\n(USD)', labelpad = 50,
5                  rotation = 'horizontal')
6 fig

```

- Lines 1 and 2 iterate through all three subplots to set the same x-axis label for each one.
 - Lines 4 and 5 set a y-axis label for just the left-most subplot. All of the subplots share the same y-axis, so this should be sufficient.
- f) Run the code cell.
g) Examine the output.



The figure is now in presentable state.

11. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel→Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Formatting Plots** tab in Firefox, but keep a tab open to **DSTIP/Matplotlib/** in the file hierarchy.

TOPIC E

Streamline Plotting with Seaborn

You've effectively generated some useful and attractive visualizations using Matplotlib, but as you may have noticed, the experience can be somewhat tedious at times. Thankfully, Seaborn helps to streamline the plotting process, and you can use it to create even more effective and good-looking plots with minimal lines of code.

Seaborn vs. Matplotlib

Matplotlib is a very powerful library for visualizing data in different forms. However, it does have some shortcomings that can make plotting a chore. Configuring individual details about a plot can lead to repetitive, verbose code. This is due to the low-level nature of Matplotlib's interfaces, which in some ways is one of its strengths. Still, there are times when you'll want to streamline the process to save yourself some hassle.

Another downside to Matplotlib is that it doesn't provide full support for pandas data structures. You can certainly use `Series` and `DataFrame` objects with Matplotlib, but the library isn't particularly smart about how it interprets these structures. For example, it would make sense to plot a column and use the column's name as the axis label, but Matplotlib doesn't do that automatically. Therefore, its integration with pandas is limited and requires more manual effort on your part.

The last major downside is somewhat subjective, but still important to a lot of data scientists—Matplotlib plots are not always the most aesthetically pleasing. You can certainly improve the appearance of a plot through styles and parameters, but it would be nice to have good-looking plots right out of the box.

All of these downsides were addressed in the development of Seaborn. Seaborn provides a more high-level interface to plotting than Matplotlib does, so much of the tedium of plot configuration is minimized. In addition, Seaborn provides full support for pandas data structures and can intelligently use `DataFrame` metadata in creating plots. And, lastly, default Seaborn plots just tend to look nicer than the Matplotlib equivalents.

Integration with Matplotlib

Seaborn is not a library created in a vacuum; like most everything else in the Python data science world, it builds on a precursor library. In this case, Seaborn is essentially just a cleaner, more user-friendly interface into Matplotlib. It uses Matplotlib to actually draw its plots. So, everything you do in Seaborn can be done in Matplotlib. The idea is that using Seaborn requires less effort on your part.

Because Seaborn builds on Matplotlib, you can plot data in much the same way. For example, you can initialize Seaborn and then use Matplotlib's plotting functions, and your resulting plot will take on the styles provided by Seaborn. You can also use either the stateful or object-oriented interface. The following example plots the data using Matplotlib:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(months, slice_2019['Revenue'])
```

Then, the same data is plotted using Seaborn:

```
import seaborn
seaborn.set()
fig, ax = plt.subplots()
ax.plot(months, slice_2019['Revenue'])
```

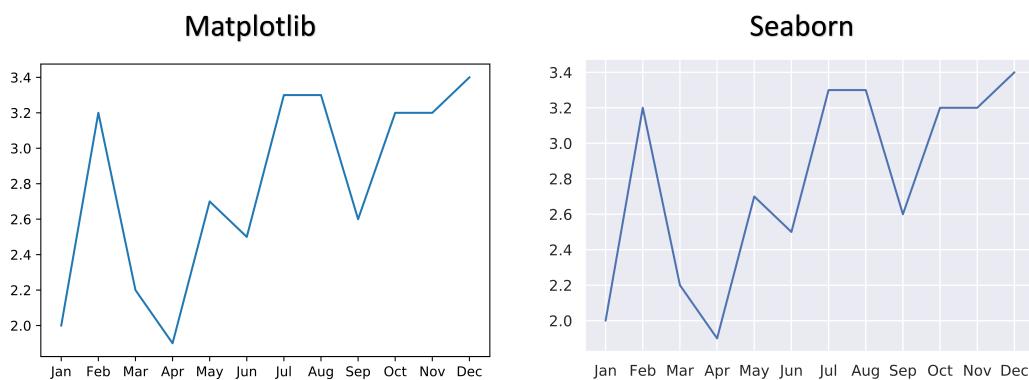


Figure 6-29: Both plots compared. Notice the addition of a grid, the lack of tick marks, and a slightly different line color with Seaborn.

However, this is not the only way to generate a Seaborn plot. Seaborn has many of its own functions that you can call, rather than calling a function directly from the `pyplot` module. You'll see examples of this soon.

The `seaborn.set()` Function

The `seaborn.set()` function initializes the look and feel of Seaborn plots in a single call. Calling this function overrides Matplotlib's styles and other aesthetic parameters. You can see this in the previous example—simply calling the function without any arguments changed the look of the plot, despite the fact that it was plotted in the exact same way. While you can use Seaborn's default aesthetics, the `seaborn.set()` function enables you to customize the look of plots further. This is done through various arguments, including:

- `style` —Specify the style for the axes to use.
- `palette` —Specify the color palette to use.
- `font` —Specify the font family to use.
- `rc` —Supply runtime configuration parameters that override any that are defined in the previous arguments.

However, it's more common to call the functions that set formatting options directly. You would typically use `seaborn.set()` with no arguments to reset the visual parameters to their defaults.



Note: If you prefer to use Matplotlib's styles, you don't need to call `seaborn.set()`. You can still use Seaborn's plotting functions, but they'll adopt the Matplotlib styles.

The `seaborn.set_style()` Function

Use the `seaborn.set_style()` function to set the overall style of the plot. There are five default styles:

- '`darkgrid`' — Grid with a gray background. This is the default.
- '`whitegrid`' — Grid with a white background.
- '`dark`' — No grid with a gray background.
- '`white`' — No grid with a white background.
- '`ticks`' — Adds tick marks and has a white background.

This is a simple way to set the look and feel of your plots. It's also "permanent" in the sense that, whenever you set a style this way, all of the plots after it will adopt the style unless you reset it with `seaborn.set()`.

The `seaborn.axes_style()` Function

The `seaborn.axes_style()` function is a more flexible version of `seaborn.set_style()`. It takes the same five style types, but you can override the specific parameters in each style if you want to customize things further. You do this by supplying a dictionary of the parameters you want to override to the `rc` argument, like so:

```
seaborn.axes_style('dark', {'xtick.bottom': True, 'ytick.left': True})
```

To get a list of all available parameters, just call `seaborn.axes_style()` with no arguments. You can also override parameters with `seaborn.set_style()`, but the only way to list the parameters is by using `axes_style()`.

The flexibility of `axes_style()` comes in its ability to set styles temporarily, rather than permanently. When you use `axes_style()`, place whatever plots you want to adopt the style inside of a `with` statement, like so:

```
fig = plt.figure(figsize = (13, 10))

with seaborn.axes_style('darkgrid'):
    ax = fig.add_subplot(2, 2, 1)
    ax.plot(months, slice_2019['Revenue'])

with seaborn.axes_style('whitegrid'):
    ax = fig.add_subplot(2, 2, 2)
    ax.plot(months, slice_2019['Revenue'])

... # More plots.
```

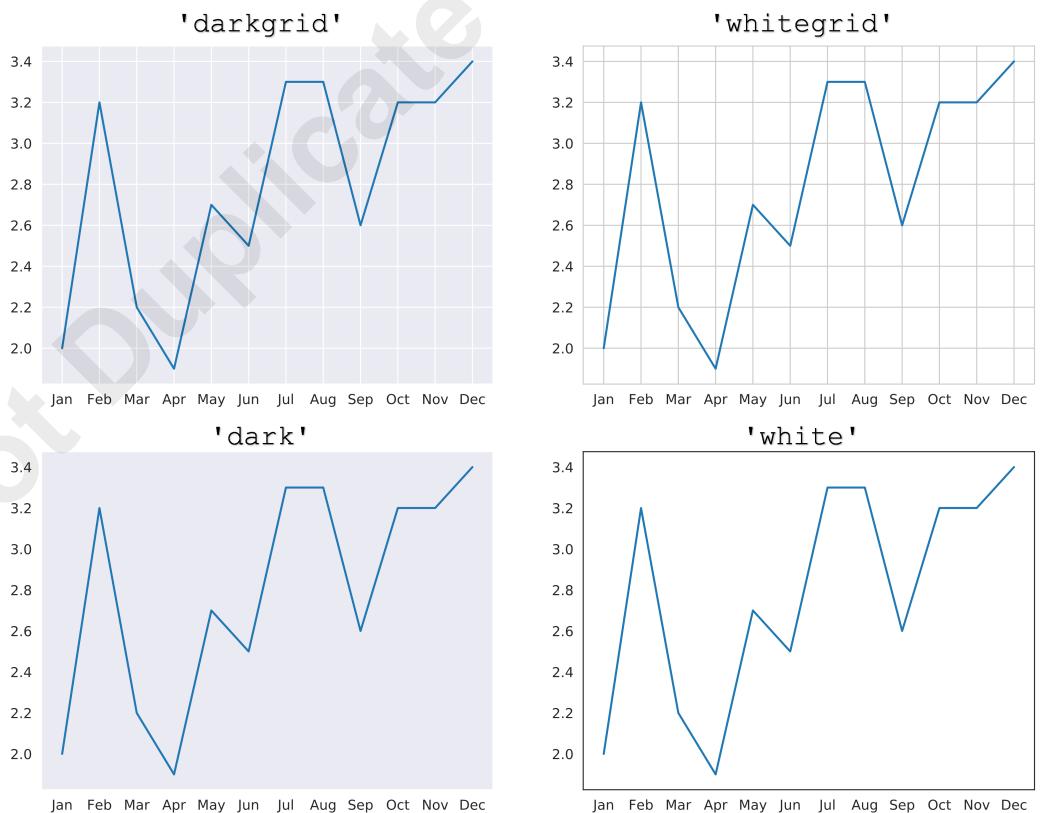


Figure 6-30: A grid of subplots, each of which adopts a different style.



Note: There is no "correct" way to style Seaborn plots. You can use whatever approach works best for you.

The seaborn.color_palette() Function

The `seaborn.color_palette()` function enables you to retrieve and configure the colors that make up one of Seaborn's color palettes. The function takes three arguments: the palette name itself, `n_colors` to specify the number of colors, and `desat` to desaturate the palette by a specified amount. Seaborn's built-in palettes are:

- 'deep'
- 'muted'
- 'bright'
- 'pastel'
- 'dark'
- 'colorblind'

These palettes use different tones of the same colors. You can also use any of Matplotlib's color maps as the `palette` argument if you want a different set of colors.

You use `color_palette()` in much the same way as `axes_style()` (i.e., on a temporary, per-plot basis):

```
with seaborn.color_palette('dark'):
    ax = fig.add_subplot()
    ax.stackplot(months, slice_2019['Revenue'])

... # More plots.
```

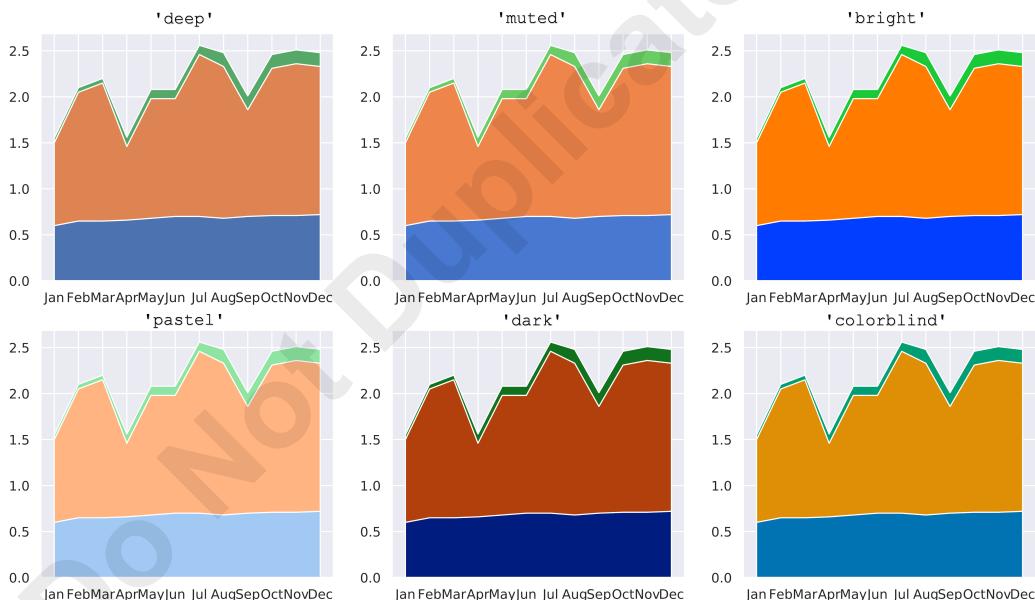


Figure 6–31: A grid of subplots, each of which adopts a different color palette.

The seaborn.despine() Function

The `seaborn.despine()` function removes the top and right "spines" from a plot. A *spine* is the axis line between the plotting area and the ticks. Removing these spines is often more aesthetically

pleasing with certain types of plots, especially line plots. If you would rather just move the spines, supply a value to the `offset` argument, where positive values move the spines outward and negative values move them inward.

The following example calls the `despine()` function without any arguments to remove the spines from a line plot:

```
seaborn.set_style('white')
fig, ax = plt.subplots()
ax.plot(months, slice_2019['Revenue'])
seaborn.despine()
```

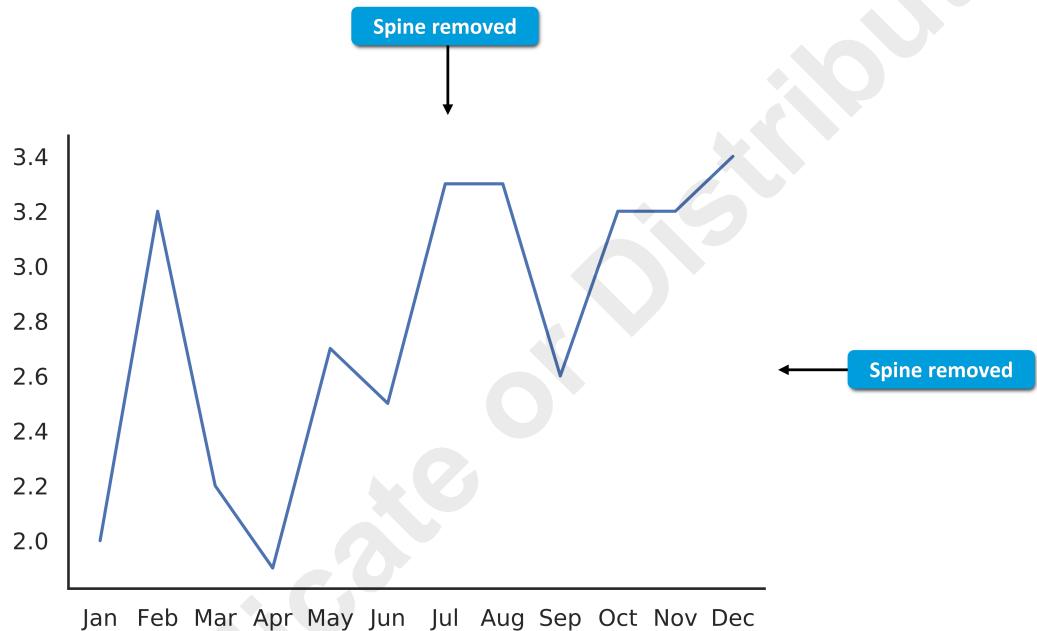


Figure 6–32: The generated plot with top and right spines removed.

Axes-Level Functions

Thus far, the examples shown have used standard Matplotlib functions to plot Seaborn-style plots. By doing this, you're really leveraging only the aesthetic advantages of Seaborn, rather than the more functional advantages. Seaborn has its own plotting functions, but before diving into them, you should know that they are divided into two broad categories: axes-level and figure-level.

Axes-level functions return a Matplotlib `Axes` object. They are drawn independent of the entire `Figure` object. Most functions that output a single type of plot, like a line plot, a scatter plot, a pie chart, etc., are considered axes-level functions. When you call one of these functions, you provide the data as input in one of several ways. This is where Seaborn's integration with pandas really shines. You can:

- Supply a list, NumPy array, or other compatible series of data.
- Supply the names of columns in a `DataFrame`.
- Supply the entire `DataFrame` to plot all columns at once. This only works with some types of plots.

Other than the input data, the other major argument that's important to axes-level functions is the `Axes` object you want to plot the data on. So, you can continue to use the object-oriented approach you learned in Matplotlib to draw your Seaborn plots.

Line Plot Example

The following axes-level example creates two subplots by calling Seaborn's equivalent of a line plot:

```
fig, ax = plt.subplots(1, 2, figsize = (12, 5))
seaborn.lineplot(x = months, y = 'COGS', data = slice_2019,
                  sort = False, ax = ax[0])
seaborn.lineplot(x = months, y = 'Payroll', data = slice_2019,
                  sort = False, ax = ax[1])
```

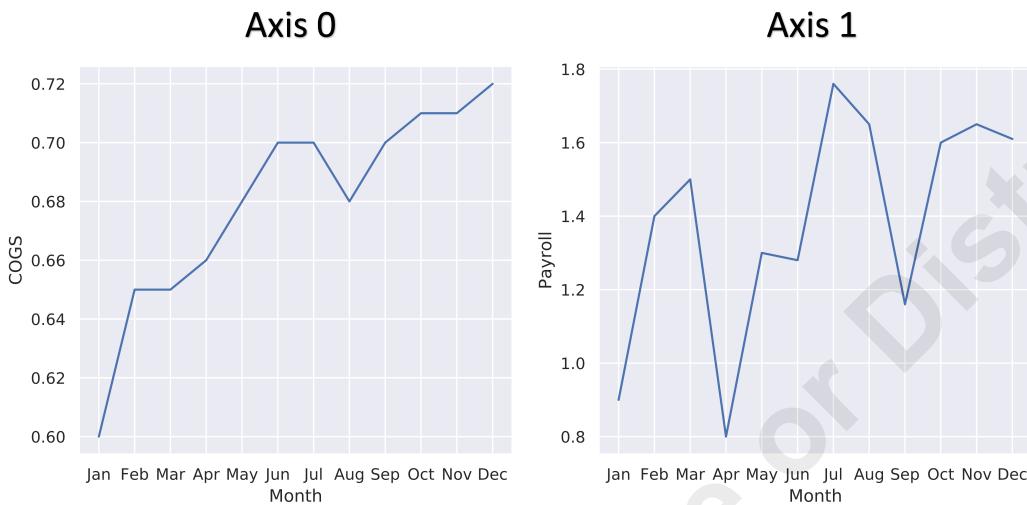


Figure 6-33: The generated axes-level subplots.

As you can see, all you need to do is call a standard Matplotlib subplotting function like you've been doing, and then just pass in the appropriate `Axes` object to the Seaborn plotting function. The `data` argument is the `DataFrame` itself, and `x` and `y` are either names of columns in that `DataFrame` or some other pandas structure (in this case, an `Index` object is supplied to `x`). Sorting is being turned off so that the month names are not automatically sorted in alphabetical order.

Also, notice how Seaborn automatically plotted the axis labels. It retrieved this information from the `DataFrame` itself, so, unlike Matplotlib, you don't need to label the axes manually. If you want to configure other aspects of the plot, you can also call formatting functions like `set_title()` on your `Axes` objects, much like you can do with Matplotlib.



Note: If you don't provide an `Axes` object, Seaborn plots on the current (default) axis, or, if none exists, it creates one.

Figure-Level Functions

The other type of Seaborn plotting function is a figure-level function. Unlike axes-level functions, figure-level functions control an entire `Figure` object. They generate new figures each time they are called, where each figure typically has multiple `Axes` objects. The purpose of a figure-level function is to generate multiple plots in a way that shows a relationship between those plots. This can be done manually through normal Matplotlib plotting, but figure-level functions alleviate the need to define axes individually; this is all done for you.

There are three general types of figure-level functions:

- `FacetGrid`—This shows the relationship between `x` and `y` using different subsets of data. Each subset is its own subplot on the overall grid. The subsets are all plotted against the same overall `x`

and y variables, but each subset has one or more special "conditions." So, the point is to compare how the variables change based on these different conditions.

- `PairGrid` —This takes multiple variables as input, and each variable is plotted against every other variable. The result is a grid of every possible combination. This type of figure is often used to get a quick overview of how (or if) variables correlate with one another.
- `JointGrid` —This takes two variables and generates three plots: one bi-variate distribution plot (i.e., a scatter plot) and two marginal univariate distribution plots (i.e., histograms of each variable). The histogram for x is usually placed above the scatter plot, and the histogram for y is usually placed to the right of the scatter plot. You would typically use this type of figure to help you analyze the distribution of two variables from multiple perspectives.

Ultimately, using any one of these figures can save you a great deal of time and minimize the plotting code that you type.

FacetGrid Example

Let's say you have that census `DataFrame`, only this time, it has an additional column with a categorical variable: whether or not there are two married adults in the household (`Married`). The `DataFrame` looks something like this:

	City	Size	Income	Age	Married
0	Olinger	2	43112.03	27	Yes
1	Greene City	5	78267.56	41	No
2	Olinger	2	37651.32	29	Yes
3	Carbon Creek	2	123714.98	45	No
4	Agerstown	4	83120.54	53	Yes
...					

Your goal is to compare the average age of income generators to the income they generate. *However*, you want to see if this relationship changes based on two conditions: what city the household is located in, and if the household members are married. This is where a `FacetGrid` can shine.

First, you define the `FacetGrid` object. The `row` argument specifies which column of the `DataFrame` will be plotted along the grid rows, and `col` does likewise for grid columns. The `data` argument is the `DataFrame`. So, to use the city and the married status as conditions:

```
grid = seaborn.FacetGrid(row = 'City', col = 'Married',
                         data = census_df, aspect = 1.5)
```

Then, you call the `FacetGrid.map()` function on your grid and pass in what type of plot you want to generate. You also provide the x and y variables each plot in the grid will be mapped to:

```
grid.map(seaborn.scatterplot, 'Age', 'Income')
```

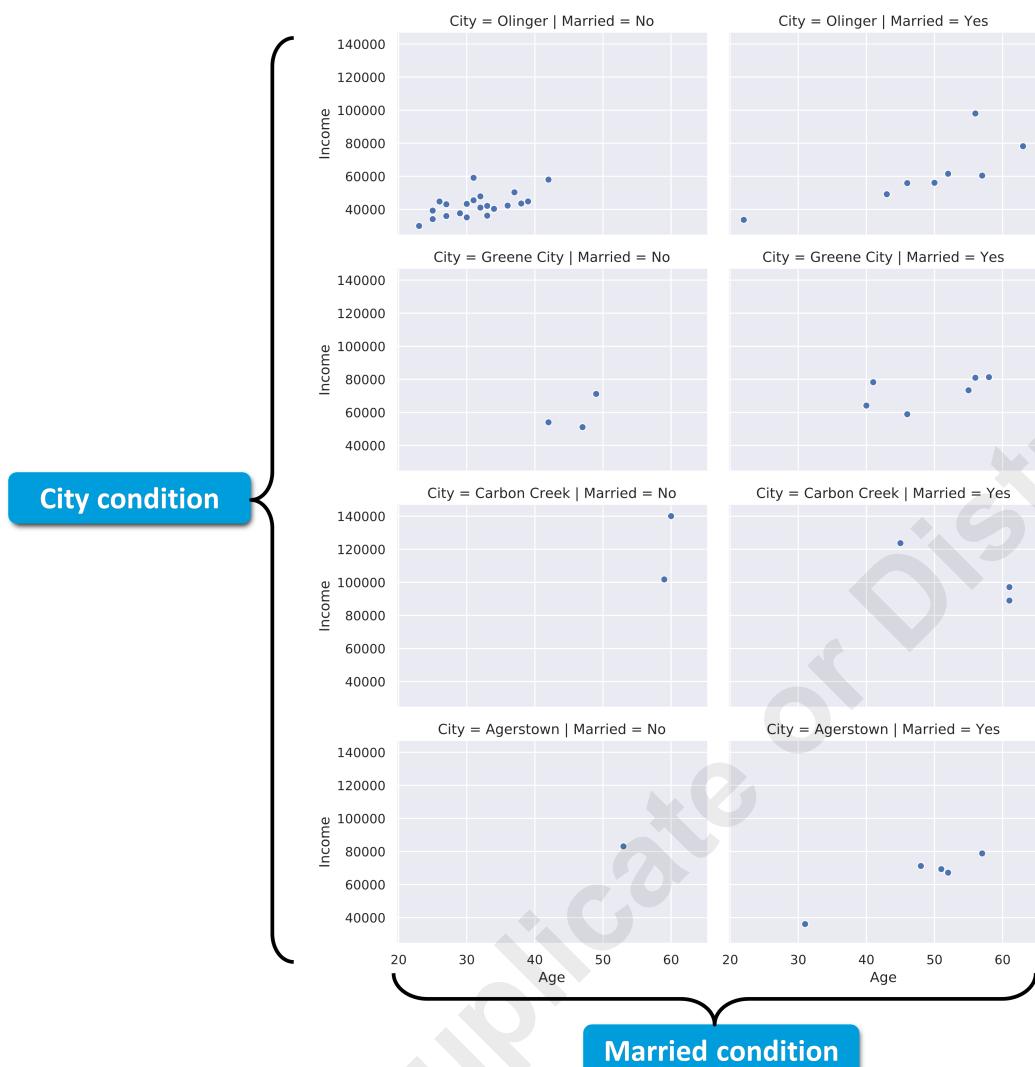


Figure 6–34: The generated FacetGrid plot.

It's that simple. One conclusion you can draw from this grid is that, in Olinger, non-married residents tend to be younger and make less money. Note that the preceding example used two conditions. You can use one, two, or three. The optional third condition is defined through the `hue` argument. When the grid is plotted, each data point will take on a different hue to indicate which category it belongs to. You might do this for the family size, for example.

Figure-Level Function Interface

Figure-level functions are like shortcuts to their verbose Matplotlib equivalents. Even still, Seaborn provides shortcuts to the shortcuts. These come in the form of high-level function interfaces into each grid type. For example, you can generate the previous example by using a single function call:

```
seaborn.relplot(x = 'Age', y = 'Income', row = 'City', col = 'Married',
                 data = census_df, kind = 'scatter')
```

The following is a list of all high-level interface functions for `FacetGrid` plots:

- `relplot()` —Relational plots (line or scatter).
- `catplot()` —Categorical plots (box, bar, violin, etc.).

- `lmplot()` —Linear regression model plot.

There is one interface function for `PairGrid` and one for `JointGrid`: `pairplot()` and `jointplot()`, respectively.

Plot Type Categories

Seaborn places each of its axes-level plotting functions into one of five categories:

- Relational plots
- Distribution plots
- Categorical plots
- Matrix plots (heatmaps)
- Regression plots

Relational Plots

A relational plot refers to either a line plot or a scatter plot—in other words, a type of plot that shows the relationship between variables `x` and `y`. You've seen examples of both of these plots so far, but to recap, the functions are:

- `seaborn.lineplot()`
- `seaborn.scatterplot()`

Both functions take arguments `x` and `y`, which can be `DataFrame` labels like column names, and `data`, which is the `DataFrame` itself. There's also a `hue` argument for plotting data on a third dimension. Both functions also take numerous other optional arguments for customizing the plot, such as `palette` to specify the color palette to use; `size` to specify line width or marker size; `style` to specify line type or marker type; `sort` to sort variable data on the applicable axis; and many more.

The following is an example of creating line plots in Seaborn:

```
seaborn.lineplot(x = months, y = 'Revenue', data = slice_2019,
                  sort = False, ax = ax)
seaborn.lineplot(x = months, y = 'Revenue', data = slice_2020,
                  sort = False, ax = ax)
... # Formatting code.
```

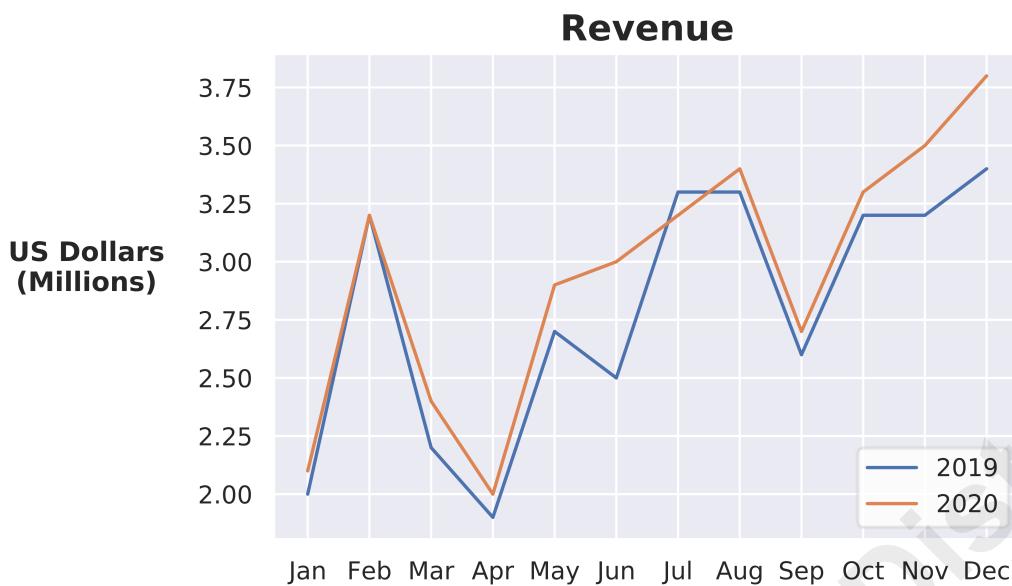


Figure 6–35: The generated line plots.

Distribution Plots

There are three types of plots in Seaborn that show the distribution of one or more variables:

- `seaborn.kdeplot()` —Generates a univariate or bivariate distribution using kernel density estimation (KDE).
- `seaborn.rugplot()` —Plots each data point as a small vertical line on the relevant axis.
- `seaborn.distplot()` —Combines Matplotlib's `hist()` and the previous two Seaborn distribution plots to generate a robust distribution visual.

Let's focus on `distplot()` since it combines everything into one function. Some of the more important arguments include:

- `a` —Specify the input data.
- `bins` —Specify the number of bins to use in the histogram portion of the plot.
- `hist` —Turn the histogram portion on or off. This is on by default.
- `kde` —Turn the KDE portion on or off. This is on by default.
- `rug` —Turn the rug portion on or off. This is off by default.
- `vertical` —Orient the plot vertically.

The following is an example of plotting the distribution of income values from `census_df`:

```
seaborn.distplot(a = census_df['Income'], bins = 10, rug = True)
... # Formatting code.
```

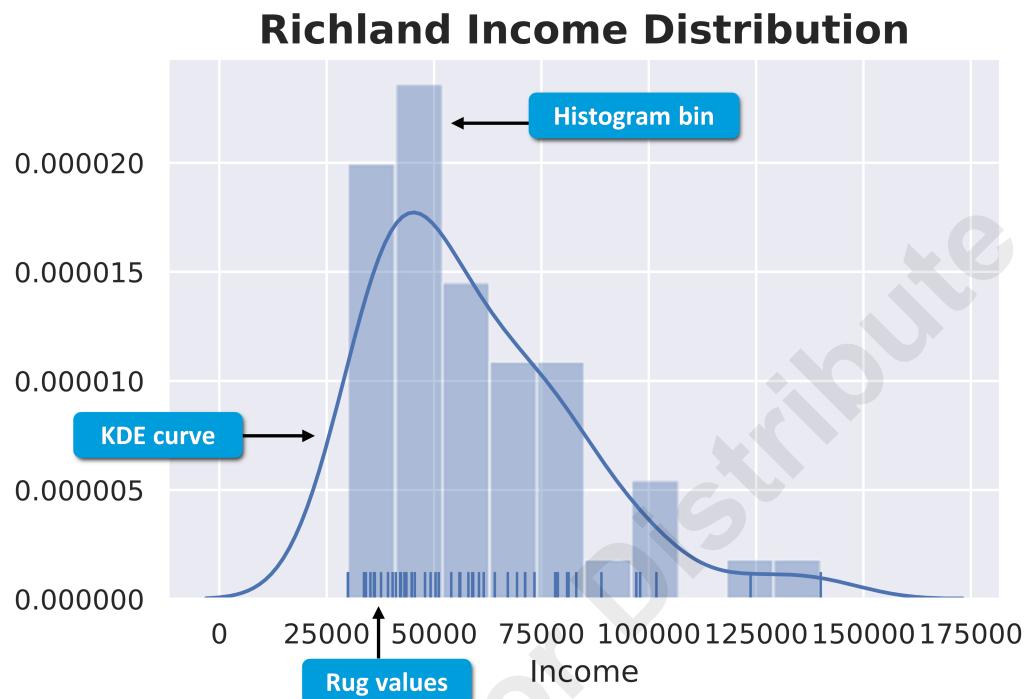


Figure 6–36: The generated distribution plot.

Categorical Plots

In Seaborn, a categorical plot is any plot that works with one or more categorical variables. Several of these categorical plots incorporate other plot types, like relational or distribution plots. The axes-level categorical plotting functions can be further divided into three families:

- Categorical distribution plots:
 - `boxplot()` —Generates a box plot.
 - `boxenplot()` —Generates an "enhanced" box plot that shows more *quantiles* than the standard four. This is typically used with large datasets, as it does a better job than a standard box plot of showing the shape of tails in the distribution.
 - `violinplot()` —Generates a violin plot with KDE.
- Categorical estimate plots:
 - `countplot()` —Generates a simple quantitative bar chart—i.e., the bars indicate the number of observations for each category.
 - `barplot()` —Generates a more complex bar chart that plots a categorical variable against some numeric variable using an estimate like the mean. This plot also adds error bars to each value indicating the amount of uncertainty with the estimate.
 - `pointplot()` —Shows categorical value estimates similar to a bar plot, but represents each estimate as a point. There is also a line connecting points, which makes it easier to compare categorical variable estimates in terms of a slope. Error bars are also added to each point.
- Categorical scatter plots:
 - `stripplot()` —Generates a scatter plot where one variable is categorical, and the other is numeric. In other words, it shows the spread of a category's numeric values by plotting each one as a point on the graph. Since there is really only one axis that has changing values, the plot introduces "jitter" to spread the points out on the graph, minimizing clutter.

- `swarmplot()` —Generates the same type of scatter plot, but with a different algorithm for spreading the points out. This algorithm does a more thorough job of ensuring that no points overlap, which results in a spread that resembles a swarm of bees. This is only really feasible in smaller datasets.

Most of these functions take `x`, `y`, `hue`, and `data` arguments as input. Each individual plot also has arguments for customizing the plot itself. You can use the `estimator` argument to specify which measurement to use in plots that perform estimates. The following is an example of a bar chart that shows the mean of each city's income:

```
seaborn.barplot(x = 'City', y = 'Income', hue = 'Married', data = census_df,
                 estimator = numpy.median)
... # Formatting code.
```

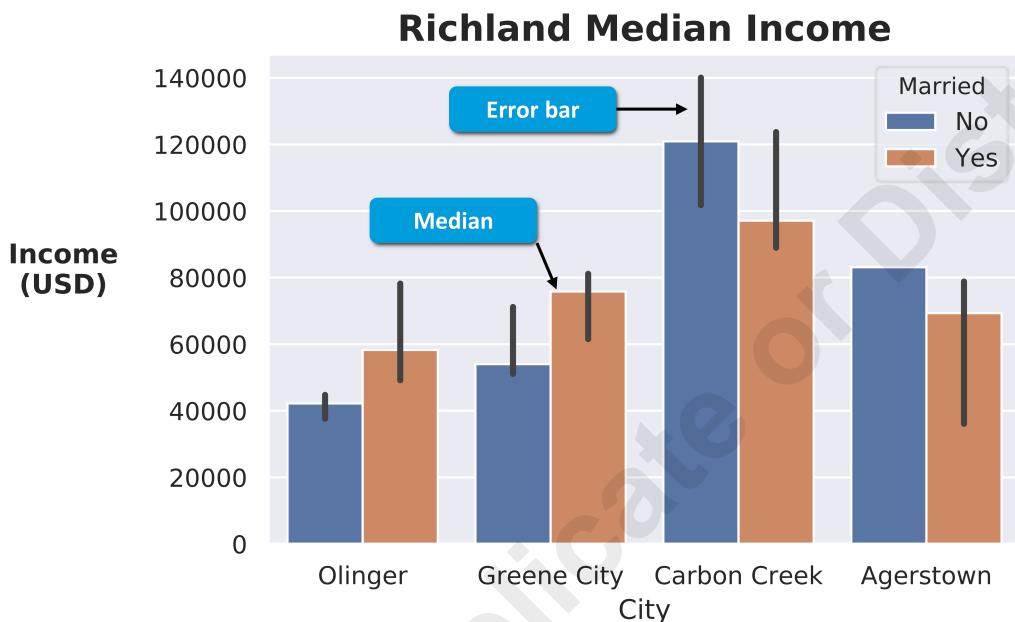


Figure 6–37: The generated bar chart.

Heatmaps

A [heatmap](#) plots different shades or intensities of color on a matrix based on data values in that location of the matrix. Each cell in the matrix is a certain color according to its value, much like how a surface plot or contour plot can use a color map to show the relationship between values.

Likewise, in a heatmap, color coding makes it easier to identify patterns of values by their color and location. This makes heatmaps very useful for revealing numerical disparities between location-based values (like temperature changes on a map). They're also commonly used to show correlations between variables in a matrix, similar to how a `PairGrid` works.

The primary heatmap function in Seaborn is `seaborn.heatmap()`. If you provide a `DataFrame` to the `data` argument, the heatmap automatically adopts its rows and columns. There are many optional arguments, including `cmap` to specify the color map; `annot` to actually write the data value in each cell; `linewidths` to specify the width of lines dividing each cell; and more. The following example generates a correlation matrix using a heatmap, where darker colors indicate a weak positive correlation between two variables, and lighter colors indicate a strong positive correlation.

```
seaborn.heatmap(slice_2019.corr(), annot = True)
... # Formatting code.
```

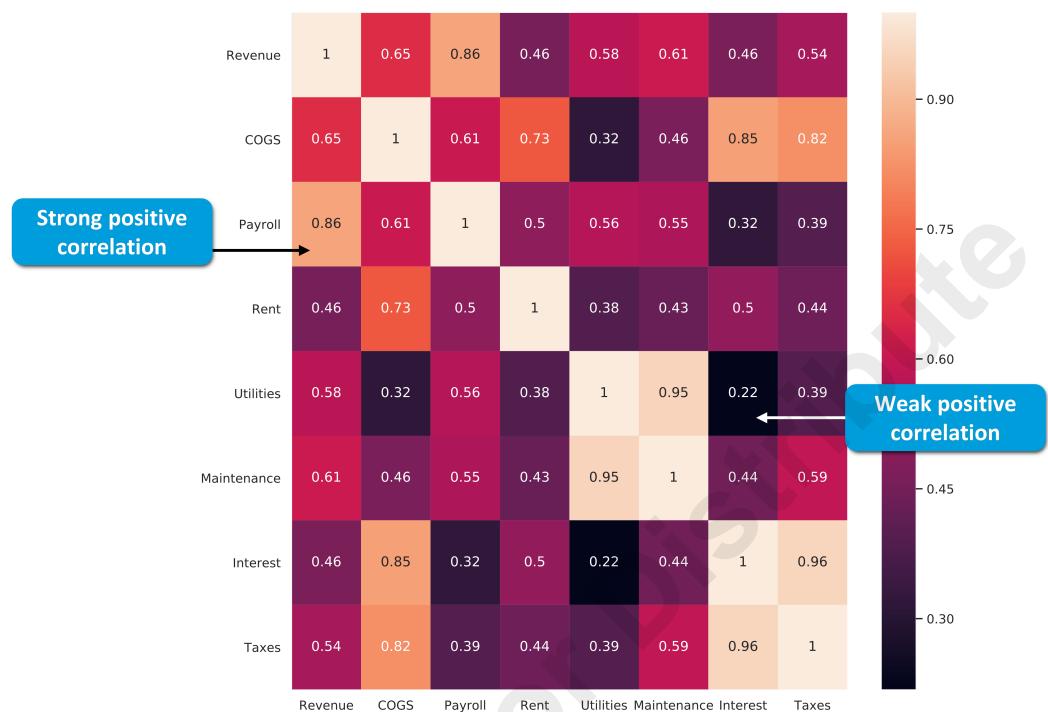


Figure 6–38: The generated heatmap. Notice that a color bar is automatically added.

The `seaborn.clustermap()` Function

The `seaborn.clustermap()` function generates a more advanced heatmap that incorporates a hierarchical clustering algorithm. Hierarchical clustering is commonly used in unsupervised machine learning to determine how best to group similar data points together.

Linear Regression Plots

A *linear regression plot* shows the relationship between an independent and a dependent variable on an *x*-*y* graph. Instead of simply plotting the data like in a scatter plot, a linear regression plot also shows the linear relationship between the values as a line of best fit. The line of best fit can be used to make predictions about new data. For example, if you plot average age (independent) against household income (dependent), you can determine how both are linearly related. Let's say there's a gap in the dataset—not all ages are represented. You can apply the missing ages to the line to see what that age's estimated income is. Because of its predictive power, linear regression is one of the most fundamental algorithms used in supervised machine learning.

The main linear regression function provided by Seaborn is `seaborn.regplot()`. Like standard relational plots, it takes an *x*, *y*, and *data* argument. In addition to plotting the line of best fit, it also plots a confidence interval on either side of the line by using translucent bands. The following code plots the age vs. income example to generate a linear regression:

```
seaborn.regplot(x = 'Age', y = 'Income', data = census_df)
... # Formatting code.
```

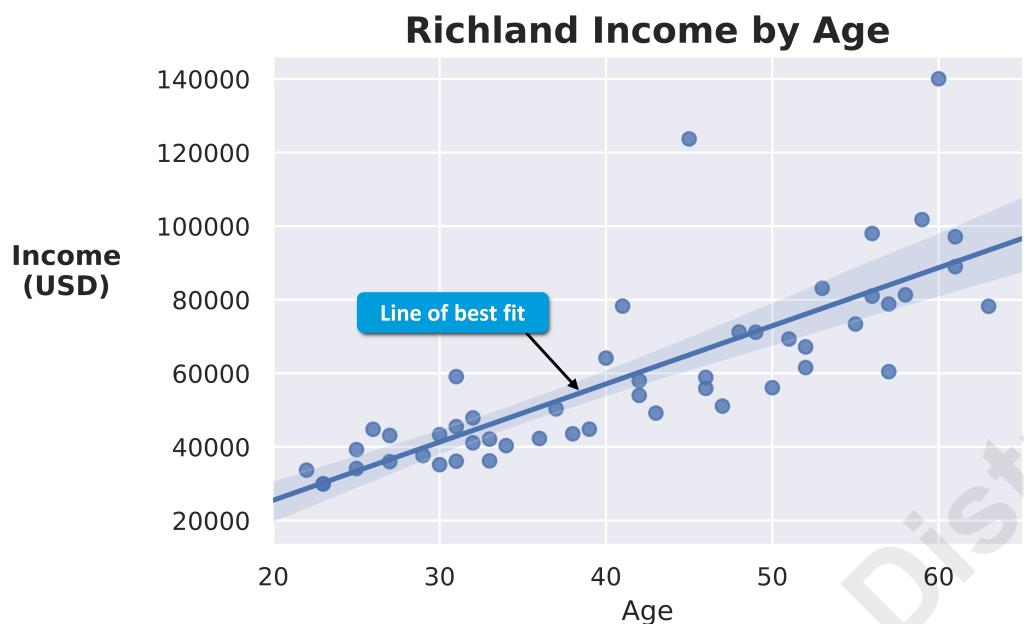


Figure 6-39: The generated linear regression plot.

Guidelines for Plotting with Seaborn

Follow these guidelines when you are plotting with Seaborn.

Plot with Seaborn

When plotting with Seaborn:

- Consider using Seaborn's plotting functions over their Matplotlib equivalents, especially when you are plotting DataFrame data.
- Remember that Seaborn integrates with Matplotlib.
- Supplement Seaborn plotting functions with Matplotlib formatting functions.
- Use `seaborn.set()` to reset the look and feel of plots.
- Use `seaborn.set_style()` to set a "permanent" style for plots.
- Use a `with` statement with `seaborn.axes_style()` to set a temporary style for plots.
- Use a `with` statement with `seaborn.color_palette()` to set a temporary color palette for plots.
- Consider using `seaborn.despine()` on certain plots to make them more aesthetically pleasing.
- Use Seaborn's axes-level functions to plot individual plots.
 - Use line and scatter plots to show relationships between numeric variables.
 - Use various distribution plots to show the spread of continuous variables.
 - Use various categorical plots to demonstrate categorical variables in multiple ways.
 - Use heatmaps to show variable correlations and numerical changes based on location.
 - Use linear regression plots to show a line of best fit for predicting unseen data.
- Use Seaborn's figure-level functions to show the relationships between multiple plots.
 - Use a `FacetGrid` to show the relationship between variables using different subsets of data.
 - Use a `PairGrid` to plot each variable against every other variable.
 - Use a `JointGrid` to plot a scatter plot of two variables, and one histogram for each variable.

ACTIVITY 6–5

Streamlining Plotting with Seaborn

Data Files

/home/student/DSTIP/Matplotlib/Streamlining Plotting.ipynb

/home/student/DSTIP/Matplotlib/data/stores_data_full_clean.csv

Before You Begin

Jupyter Notebook is open.

Scenario

You've had success presenting your visualizations to GCE's business leaders. They've asked you to present reports like these on a regular schedule so that they can stay up to date on the company's performance. Now that you know you'll be generating more and more plots in the foreseeable future, you want to streamline your processes to make things easier. You decide to try Seaborn, which can help automate plot generation and modification, reducing the time spent on coding. In addition, Seaborn can help you generate advanced plots that aren't readily available through Matplotlib. Once you have some experience with Seaborn, you'll be able to incorporate streamlined plot generation processes into the team's overall data science pipeline.

1. In Jupyter Notebook, open the **DSTIP/Matplotlib/Streamlining Plotting.ipynb** file.
2. Import the relevant software libraries, and load the dataset.
 - a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
Lines 16 through 18 load the full `stores_df` DataFrame.
 - b) Run the code cell.
 - c) Verify that the version of Python is displayed, as are the versions of NumPy, pandas, Matplotlib, and Seaborn that were imported.

```
Libraries used in this project:
- Python 3.7.6 (default, Jan  8 2020, 19:59:22)
[GCC 7.3.0]
- NumPy 1.18.1
- pandas 1.0.1
- Matplotlib 3.1.3
- Seaborn 0.10.0
```

Loaded dataset.

3. Generate scatter plots using different Seaborn styles.
 - a) Scroll down and view the cell titled **Generate scatter plots using different Seaborn styles**, then select the code cell below it.

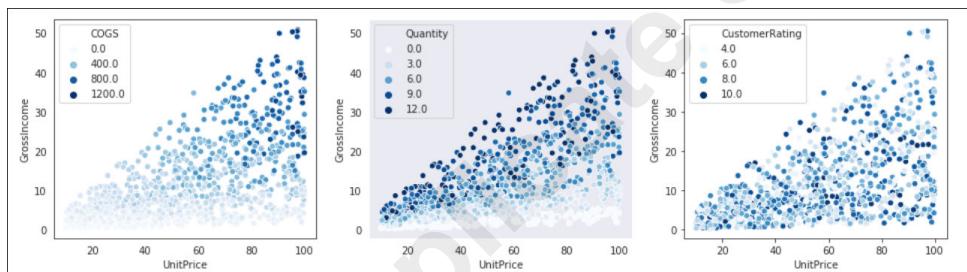
- b) In the code cell, type the following:

```

1  kwargs = {'x': 'UnitPrice', 'y': 'GrossIncome',
2             'data': stores_df, 'palette': 'Blues'}
3
4  fig = plt.figure(figsize = (16, 4))
5
6  with sb.axes_style('white'):
7      ax = fig.add_subplot(1, 3, 1)
8      sb.scatterplot(hue = 'COGS', ax = ax, **kwargs)
9
10 with sb.axes_style('dark'):
11     ax = fig.add_subplot(1, 3, 2)
12     sb.scatterplot(hue = 'Quantity', ax = ax, **kwargs)
13
14 with sb.axes_style('ticks'):
15     ax = fig.add_subplot(1, 3, 3)
16     sb.scatterplot(hue = 'CustomerRating', ax = ax, **kwargs)

```

- Line 1 sets up a dictionary of parameters you'll pass in to the plotting functions.
 - Line 6 begins a `with` statement that sets a temporary style for the plot inside it to use.
 - Line 8 uses Seaborn's `scatterplot()` function for the first subplot.
 - Lines 10 through 16 repeat this process for the other two subplots, where each one uses a different style.
- c) Run the code cell.
d) Examine the output.



- This is similar to the scatter plot grid you generated before—unit price is being compared with gross income, with COGS, quantity, and customer rating represented through a color map for each subplot.
- As expected, each subplot's style is different.
- Seaborn automatically added axis labels to the plots without you needing to specify them.
- Seaborn also added legends for the color maps. These might not be as useful as color bars, but they can still help you identify the general value of a marker based on its color intensity.

4. Generate a compound distribution plot with a customized style.

- a) Scroll down and view the cell titled **Generate a compound distribution plot with a customized style**, then select the code cell below it.

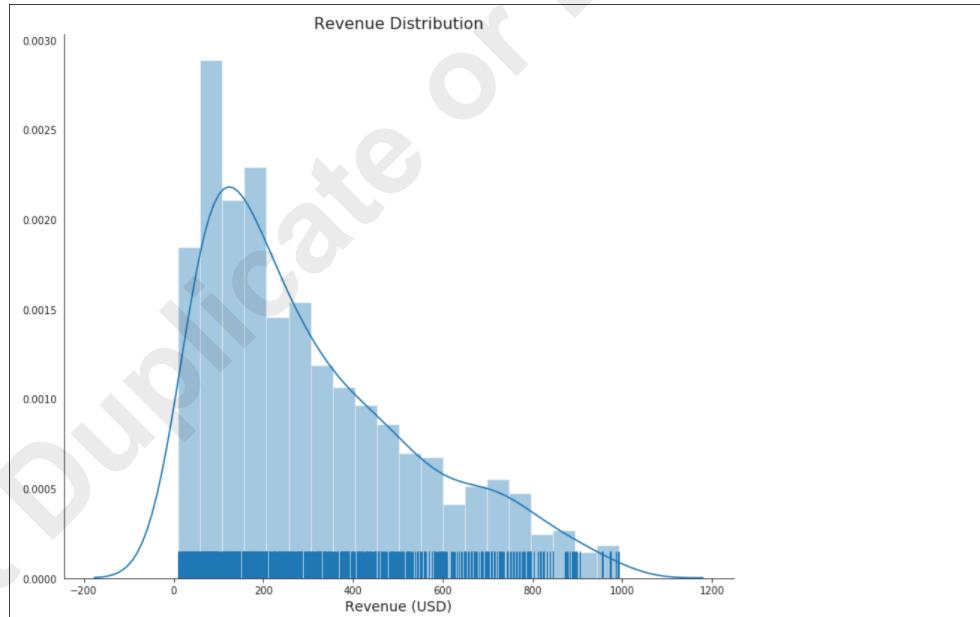
- b) In the code cell, type the following:

```

1 sb.set_style('white', {'xtick.bottom': True})
2
3 fig, ax = plt.subplots(figsize = (12, 10))
4 sb.distplot(a = stores_df['Revenue'], bins = 20, rug = True)
5 ax.set_title('Revenue Distribution', size = 16)
6 ax.set_xlabel('Revenue (USD)', size = 14)
7
8 sb.despine()

```

- Line 1 sets a "permanent" style. Any plots following this statement will adopt this style, unless the styles are reset. A dictionary is also being passed in to add ticks to the x-axis.
 - Line 4 calls `distplot()`, which combines a histogram with KDE and rug plots.
 - Line 5 uses a typical Matplotlib formatting function to add a title. Seaborn does not add titles automatically.
 - Line 6 uses a typical Matplotlib formatting function to change the x-axis label. While Seaborn adds axis labels automatically, you may wish to change them to make them more readable.
 - Line 8 removes the spine from the figure.
- c) Run the code cell.
d) Examine the output.



- All three elements of the compound distribution plot are shown—histogram bins, KDE curve, and rug marks.
- Notice how the top and right edge lines of the graph are gone.
- One example observation is that the density curve levels off when compared to the first four bins of the histogram, despite the second bin containing more values than its surrounding bins.
- You could generate this same graph by using Matplotlib directly, but the code would be more verbose.

5. Generate a bar chart using a Seaborn color palette.

- a) Scroll down and view the cell titled **Generate a bar chart using a Seaborn color palette**, then select the code cell below it.

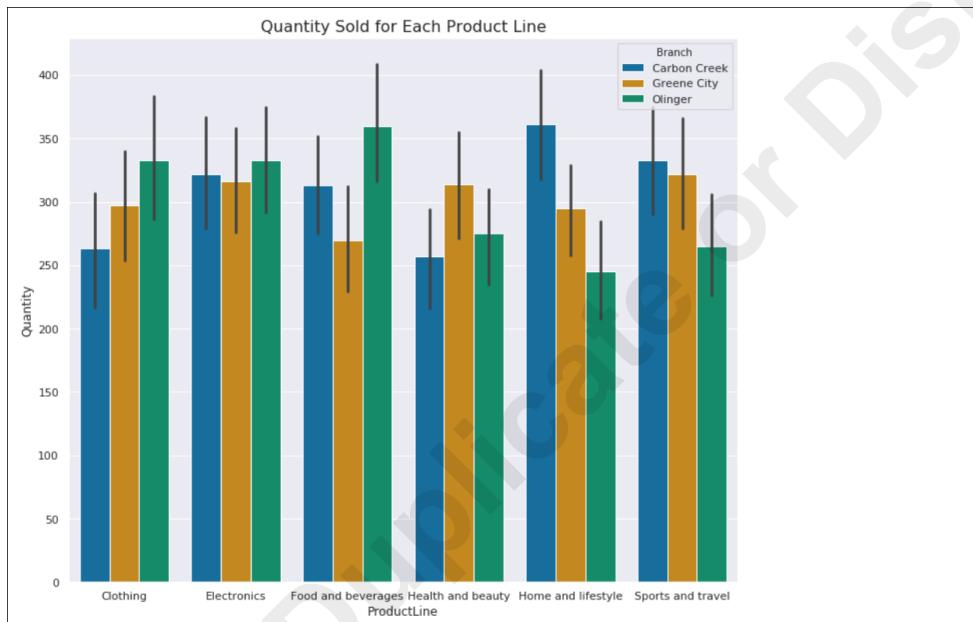
- b) In the code cell, type the following:

```

1 sb.set() # Reset style to default.
2
3 with sb.color_palette('colorblind'):
4     fig, ax = plt.subplots(figsize = (12, 10))
5     sb.barplot(x = 'ProductLine', y = 'Quantity', hue = 'Branch',
6                 data = stores_df, estimator = np.sum)
7     ax.set_title('Quantity Sold for Each Product Line', size = 16)

```

- Line 3 begins a `with` statement that sets a temporary color palette for the plot inside it to use.
 - Lines 5 and 6 call `barplot()`, a categorical estimation plot. Each product line will be compared according to total quantity sold. In addition, `hue` specifies that each product line will be further divided by store branch.
- c) Run the code cell.
d) Examine the output.



- The bar colors are generated using a color palette that supports people with color blindness.
- Each color represents a different store branch for a product line, with a legend showing which branch is which color.
- The vertical black lines over the top of each bar show the margins of error in the estimation.
- One example observation is that Olinger seems to sell the most clothing, electronics, and food and beverage products, but lags behind when it comes to health and beauty, home and lifestyle, and sports and travel products.

6. Generate a swarm plot using a style and color palette.

- a) Scroll down and view the cell titled **Generate a swarm plot using a style and color palette**, then select the code cell below it.

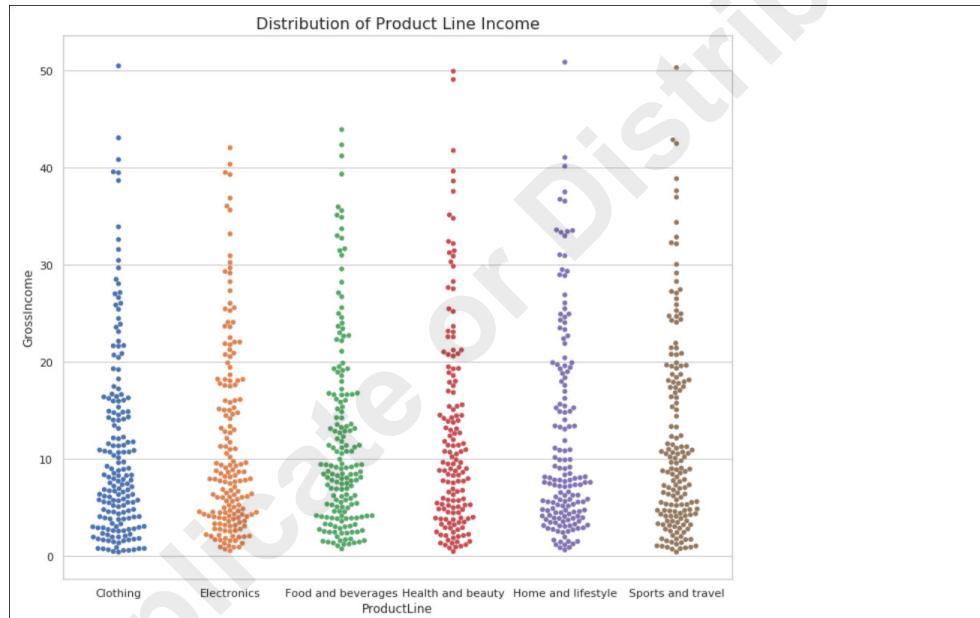
- b) In the code cell, type the following:

```

1 with sb.axes_style('whitegrid'), sb.color_palette('deep'):
2     fig, ax = plt.subplots(figsize = (12, 10))
3     sb.swarmplot(x = 'ProductLine', y = 'GrossIncome', data = stores_df)
4     ax.set_title('Distribution of Product Line Income', size = 16)

```

- Line 1 sets both a temporary style and color palette for the plot within.
 - Line 3 calls `swarmplot()`. The plot will show the spread of each transaction's gross income values, categorized by product line.
- c) Run the code cell.
d) Examine the output.



- The 'whitegrid' is self-explanatory. The 'deep' color palette has relatively low saturation and low luminance (brightness).
- Each marker represents a transaction for each product line. Its placement on the y-axis is indicated by its gross income. This particular plot type uses an algorithm that spreads the markers out to minimize overlap. This is only really effective in smaller datasets, otherwise you should consider using `stripplot()`.
- One example observation is that the gross income for each product line seems to have a similar spread. Several product lines have outliers around the \$50 threshold, excluding electronics and food and beverages.

7. Generate a heat map showing correlations between numeric variables.

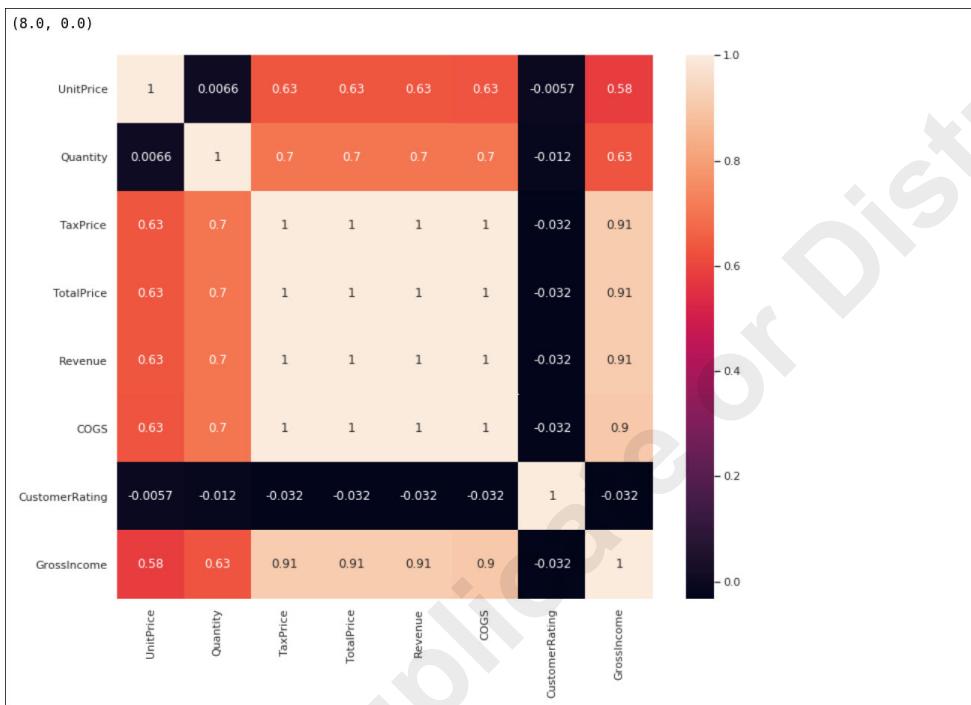
- a) Scroll down and view the cell titled **Generate a heat map showing correlations between numeric variables**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 fig, ax = plt.subplots(figsize = (12, 10))
2 sb.heatmap(stores_df.corr(), annot = True)
```

Line 2 creates a heat map from the correlation matrix generated from pandas' `DataFrame.corr()` function. If you recall, this correlation matrix showed how strongly each numeric variable in `stores_df` correlated with every other numeric variable.

- c) Run the code cell.
d) Examine the output.



- This is essentially the same as the correlation matrix you saw earlier, but made more visual.
- As the color bar shows, darker colors indicate little to no correlation, whereas brighter colors indicate strong positive correlations.
- In addition to the color map, the correlation value between -1 and 1 is shown for each matchup.
- One example observation is that unit price seems to have little to no correlation with quantity or customer rating, but has a positive correlation with the other pricing values.

8. Generate a linear regression plot for revenue and gross income.

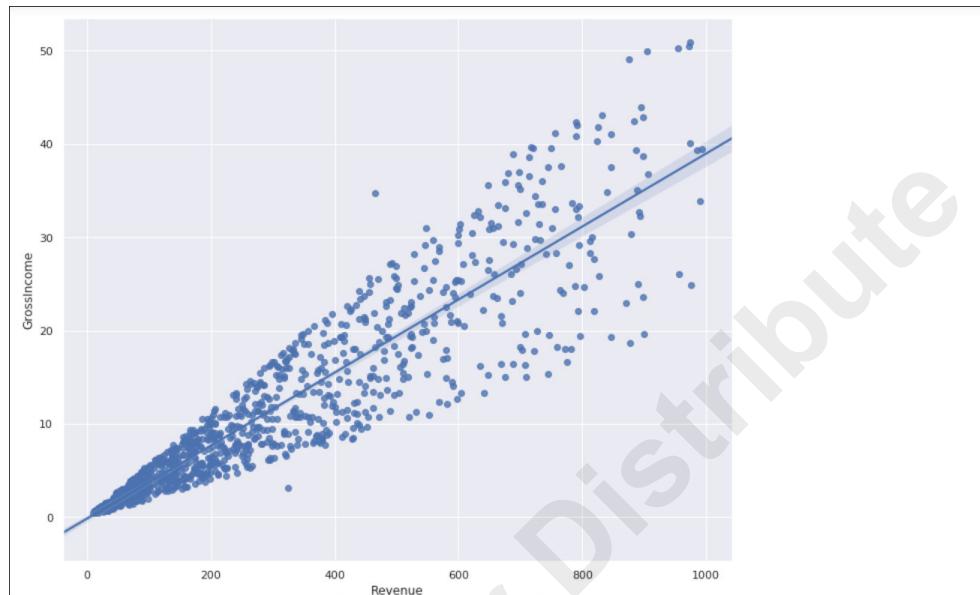
- a) Scroll down and view the cell titled **Generate a linear regression plot for revenue and gross income**, then select the code cell below it.
b) In the code cell, type the following:

```
1 fig, ax = plt.subplots(figsize = (12, 10))
2 sb.regplot(x = 'Revenue', y = 'GrossIncome', data = stores_df);
```

Line 2 calls `regplot()` to generate a linear regression model for revenue and gross income.

- c) Run the code cell.

- d) Examine the output.



- A scatter plot, much like ones you've seen before, was generated.
- The main difference is that a line of best fit was plotted. The line of best fit attempts to explain the linear relationship between the two variables, which can be used to make predictions for unseen data. This is one of the more simple algorithms involved in machine learning.
- The translucent bands on either side of the line represent the confidence interval.
- If, for example, you added a new store transaction with a revenue of around \$700, and you wanted to estimate what its gross income will be, you would identify where that revenue meets the line of best fit, which would give you the estimated gross income (around \$25 in this case). Note that linear regression models can provide you with the estimation directly—you don't need to rely on a visual approximation.

9. How do axes-level functions in Seaborn differ from figure-level functions?

10. Generate a FacetGrid of quantity sold per product line, categorized by branch and customer type.

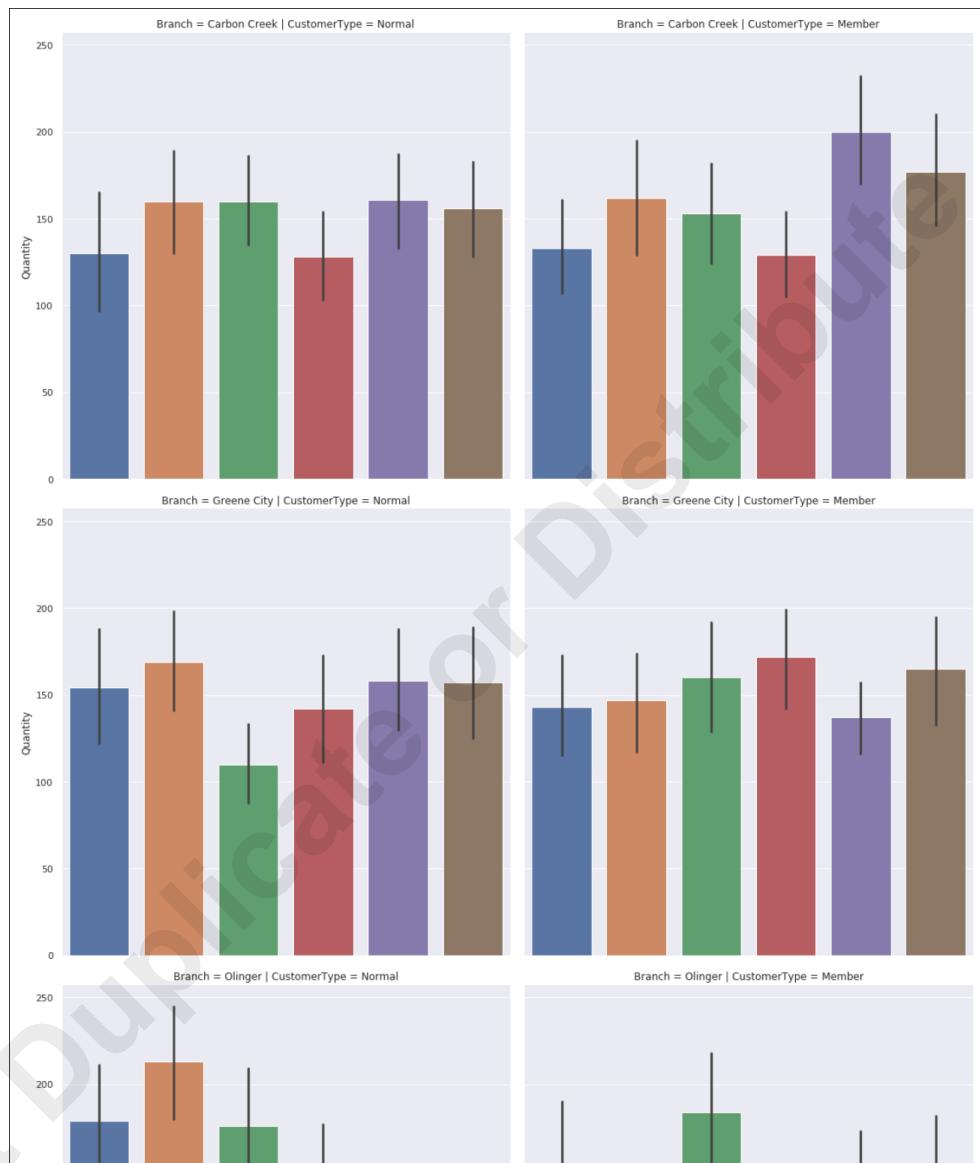
- a) Scroll down and view the cell titled **Generate a FacetGrid of quantity sold per product line, categorized by branch and customer type**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 plot = sb.catplot(x = 'ProductLine', y = 'Quantity',
2                     row = 'Branch', col = 'CustomerType', data = stores_df,
3                     kind = 'bar', estimator = np.sum, height = 8)
4 plot.set_xticklabels(stores_df['ProductLine'].unique(), rotation = -30);
```

- Lines 1 through 3 use the `catplot()` function interface to generate a `FacetGrid` object:
 - In this case, it's a categorical plot, specifically a group of bar charts (as specified by `kind`).
 - The `x` and `y` arguments are the `DataFrame` columns that will be plotted along the overall figure's x-axis and y-axis, respectively.
 - The `row` and `col` arguments plot `DataFrame` columns for each row of subplots and each column of subplots, respectively.
 - The `height` argument is used to increase the grid's size while maintaining its aspect ratio.
 - Line 4 adjusts the x-axis tick labels so that they'll be placed at an angle, preventing them from overlapping.
- c) Run the code cell.

d) Examine the output.



- In just a single plotting statement, an entire grid of bar charts was created.
- All six charts share the same x-axis (product line) and y-axis (quantity).
- Each row shows a different store branch, and each column shows a different customer type (member or not).
- In other words, not only can you compare each product line's total quantity sold, you can compare each product line's total quantity sold as per two different conditions.
- One example observation is that home and lifestyle products seem to be the most popular amongst members who shop at Carbon Creek, but non-members who shop at Olinger don't seem to buy many items from this department.

11. Generate a `JointGrid` showing the distributions of revenue and income.

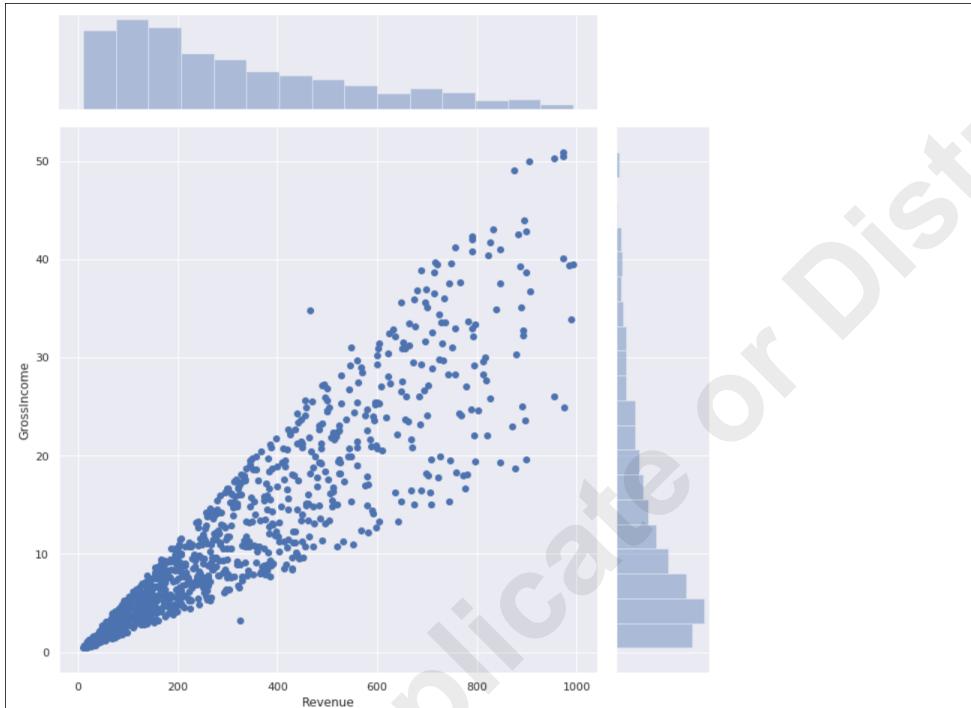
- Scroll down and view the cell titled **Generate a `JointGrid` showing the distributions of revenue and income**, then select the code cell below it.

- b) In the code cell, type the following:

```
1 sb.jointplot(x = 'Revenue', y = 'GrossIncome', data = stores_df,
2                 kind = 'scatter', height = 10);
```

This code uses the `jointplot()` function interface to generate a `JointGrid` object. It will plot revenue vs. gross income in multiple ways.

- c) Run the code cell.
d) Examine the output.



- The result is similar to the `GridSpec` you created earlier, but accomplished in much less code.
- A large scatter plot of revenue and gross income is in the middle, with smaller histograms on the top and right side.
- The top histogram shows the distribution of the `x` variable (revenue), and the right histogram shows the distribution of the `y` variable.
- A `JointGrid` like this one reinforces the distribution of the variables from multiple perspectives. The bi-variate scatter plot shows the distribution of both variables as they relate to one another, and the univariate histograms show you how each variable is distributed independent of the other.

12. Shut down this Jupyter Notebook kernel.

- From the menu, select **Kernel->Shutdown**.
- In the **Shutdown kernel?** dialog box, select **Shutdown**.
- Close the **Streamlining Plotting** tab in Firefox, but keep a tab open to the file hierarchy in Jupyter Notebook.

Summary

In this lesson, you transformed your raw data into illustrative visuals using the Matplotlib and Seaborn libraries. These visuals will aid your own analysis efforts and make it easier to present your findings to a wider audience. Not only did you generate these visuals, you tweaked them in various ways so that they communicate your ideas as effectively as possible.

What kinds of plots do you think you'll create most often on the job?

Do you think you'll use Seaborn over Matplotlib, or would you prefer just using Matplotlib directly? Or, do you think you'll use both? Why?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Course Follow-Up

Congratulations! You have completed the *Using Data Science Tools in Python®* course. You have successfully used Python to analyze, manipulate, and visualize business data in order to extract knowledge that may be used to benefit the organization. There are many more Python data science libraries out there, but having skill in NumPy, pandas, and Matplotlib will serve as the foundation of your work.

What's Next?

You may want to pursue additional courses in the Logical Operations Data Science learning path. Details and course information can be found at <https://logicaloperations.com/data-science-certification-path/>. For example, to learn how to use Visual Basic for Applications (VBA) to perform data wrangling in a Microsoft® Office environment, consider taking the *Programming and Data Wrangling with VBA and Excel®* course. For more advanced applications of data science in the world of artificial intelligence (AI), consider taking the CertNexus *Certified Artificial Intelligence (AI) Practitioner (Exam AIP-110)* course and its associated certification exam.

You are encouraged to explore data science further by actively participating in any of the social media forums set up by your instructor or training administrator through the **Social Media** tile on the CHOICE Course screen.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

A

Scraping Web Data Using Beautiful Soup

Appendix Introduction

The following appendix provides an overview of scraping web data using Beautiful Soup.

TOPIC A

Scrape Web Pages

Websites and other web-based documents are common sources of data. BeautifulSoup can help you pull data from these sources, enabling you to analyze, transform, and visualize this data using the Python® libraries you're familiar with.

Web Scraping and Parsing

Web scraping is the process of extracting data from web-based content formats, particularly those hosted on websites accessible through the Hypertext Transfer Protocol (HTTP). **Web parsing** is the act of analyzing and selecting specific portions of the content based on patterns in the text. Both activities are related, and they typically go hand-in-hand during data science tasks. The ability to extract, analyze, and manipulate web-based data can be useful for organizations or individuals who want to compile or convert data that is found on websites and similar formats. For example, say you work for an organization that has customer information documented on a private intranet website. The information includes the customers' names, addresses, email, etc. To make this information easier to analyze, you want to place it in a DataFrame.

Web scrapers typically work with Hypertext Markup Language (HTML) and other markup languages like Extensible Markup Language (XML). HTML is the standard markup format that most websites are written in and is used to display information. XML, on the other hand, is typically used to define, transport, and store data, rather than to display it. While HTML and XML have significant differences, both are similar in terms of their use of tags. Tags tell a parsing engine how to parse specific components of a document and what context to place them in. In HTML, the `<h1>` tag tells a parsing engine (e.g., a web browser) to format the text between the tags as a heading. For example:

```
<h1>Customer001<h2>
```

So, you might tell a web scraper to extract only heading text from the document. It will, therefore, extract `Customer001`, which you can use however you wish. That's just a simple example—using web scrapers, you can extract much more types and combinations of information from web documents. This appendix assumes you understand the basics of HTML and XML structure, namely the nesting of element tags and their attributes in a tree-like hierarchy.



Note: While pandas has an HTML loading function, it is limited to reading from tables, which the data in question may not be formatted in.

Beautiful Soup

Beautiful Soup is one of the most popular Python-based web scraping and web parsing libraries. It is used for many different purposes, including as a data extraction and munging tool by data scientists. It can scrape and parse both HTML and XML documents, regardless of how well they are formatted. Its name is taken from the concept of *tag soup*, which refers to HTML tagging that is poorly structured (e.g., unclosed tags) yet still renderable due to HTML's permissiveness. BeautifulSoup therefore takes the document and transforms it into a Python object that better represents the document tree, free of tagging errors.

To generate a Python object from a web document—"creating the soup"—you call the `BeautifulSoup` class constructor on the source file you want to work with. The following example assumes you've downloaded an HTML file to the current working directory:

```
from bs4 import BeautifulSoup
```

```
with open('customers.html') as file:
    soup = BeautifulSoup(file)
```

This simply uses the Python `with` statement to open a file using a relative path, then creates an object from that file using `BeautifulSoup`. However, downloading HTML files to your local system may not be feasible. To access the content of an HTML file directly from the live website it's hosted on, you'll need to use a library that can make web requests. The `Requests` library is, as you might expect, just such a library. It comes with most modern versions of Python®, including through Anaconda®. You can use it to send a GET request to an HTTP server, which can return an HTML document to you (assuming the request succeeded). For example:

```
import requests

page = requests.get('https://website.example/customers.html')

soup = BeautifulSoup(page.text)
```

The URL is passed in as an argument to `requests.get()`, which is then assigned to `page`. This constructs an object from that URL. Rather than supply `page` directly to the `BeautifulSoup` constructor, you need to supply `page.text` in order to actually retrieve the HTML markup text. The `soup` is then created, just as it was with a local file.



Note: Beautiful Soup is also a reference to a song in Lewis Carroll's *Alice's Adventures in Wonderland*.

Supported Parsers

Beautiful Soup calls one of several Python parsing libraries to actually parse an HTML or XML document. Each parsing library has its advantages and disadvantages, as well as some differences in terms of how it parses documents. You configure which parser Beautiful Soup should use by adding it as the second argument to the `BeautifulSoup` constructor. If you don't specify a parsing library to use, Beautiful Soup will use the HTML parser included with Python's standard library. Note that if you want to use a third-party parsing library, you must install it first. All of these libraries are included with Anaconda, however.

The following table describes the various parsers supported by Beautiful Soup.

Parser	Description
<code>BeautifulSoup(markup, 'html.parser')</code>	This is the standard Python HTML parser. It has decent speed compared to the other parsers and is somewhat lenient when it comes to errors. This is a good choice if you don't want to install external libraries and speed isn't crucial.
<code>BeautifulSoup(markup, 'lxml')</code>	This is the HTML version of a third-party parser called <code>lxml</code> . It is faster than the standard Python parser and has a similar degree of leniency. This is the best choice if you don't mind installing an external library and speed is crucial.
<code>BeautifulSoup(markup, 'lxml-xml')</code> <code>BeautifulSoup(markup, 'xml')</code>	Both of these call the XML version of the <code>lxml</code> parser. This is the only XML parser that is currently supported. This parser has equivalent speed to the HTML version of <code>lxml</code> .

Parser	Description
<code>BeautifulSoup(markup, 'html5lib')</code>	This parser is designed to parse an HTML document in a similar way to a web browser. It has the highest leniency among the HTML parsers and can create HTML5-compliant markup. However, it is slower than the other parsers. You might want to use this parser if HTML5 compliance is of greater importance than speed.

Differences Between Parsers

As mentioned, the parsers in the previous table parse documents differently. The differences occur wherever the document has tag soup. While this may not be a concern in most cases, if you plan on sharing your Python parsing code with others, you'll want to ensure everyone is using the same parser for the same document. Otherwise, you could introduce errors in the code's logic.

The difference between all of the HTML parsers and the one XML parser is that the former will correct an empty tag, but the latter will not. For example:

```
>>> BeautifulSoup('<h1><b></h1>')
<html><body><h1><b></b></h1></body></html>

>>> BeautifulSoup('<h1><b></h1>', 'xml')
<?xml version="1.0" encoding="utf-8"?>
<h1><b/></h1>
```

In the HTML version, `` became ``, but in the XML version, it was made into a closing tag. Both also differ in that standard HTML structural tags were added by the HTML parser, and an XML declaration was added by the XML parser.

When it comes to the three HTML parsers, they can differ based on how they handle certain unopened tags:

```
>>> BeautifulSoup('<body></p></body>')
<html><body></body></html>

>>> BeautifulSoup('<body></p></body>', 'lxml')
<html><body></body></html>

>>> BeautifulSoup('<body></p></body>', 'html5lib')
<html><head></head><body><p></p></body></html>
```

The standard HTML parser and the lxml HTML parser both stripped the unopened tag from the document. However, the html5lib parser kept the closing `</p>` tag and added an opening `<p>` tag. It also added `<head></head>` tags. Not all element tags are parsed in this manner, however.

The Tag Object

In Beautiful Soup, there are four primary object types. One of the most fundamental object types is the Tag. As the name suggests, a Tag object corresponds to an HTML or XML tag and its contents. Using a Tag, you can:

- Retrieve the name of the tag.
- Retrieve the tag's attributes.
- Navigate the tag's parent and child elements.
- Search for specific tags.

A Tag object is automatically created every time you construct a BeautifulSoup document. Each tag in the document is assigned to its own object. You can reference the tag like so:

```
>>> soup = BeautifulSoup('<a href="https://www.website.example">Link</a>')
>>> soup
<html><body><a href="https://www.website.example">Link</a></body></html>
>>> soup.a
<a href="https://www.website.example">Link</a>
```

The `soup.a` statement retrieved the `<a>` element and its entire contents (including the closing tag). Likewise, if you call `soup.body`, you get:

```
<body><a href="https://www.website.example">Link</a></body>
```

So, BeautifulSoup retrieved the `<body>` element and everything inside of it.

You can also get the name of a tag as follows:

```
>>> soup.body.name
'body'
```

This might be helpful if you've assigned a Tag to a variable and need to verify the HTML/XML tag it's referencing.

You can also retrieve a tag's attributes like so:

```
>>> soup.a.attrs
{'href': 'https://www.website.example'}
```

A dictionary of attributes and their values is provided. To retrieve a specific attribute value, access the Tag object like you would a dictionary:

```
>>> soup.a['href']
'https://www.website.example'
```



Note: This retrieves only the first instance of a tag; if you have multiple `<a>` tags, for instance, you will need a different approach to select a specific one.

The BeautifulSoup Object

The BeautifulSoup object represents the entire parsed HTML/XML document. Unlike a Tag object, the BeautifulSoup object has no specific name or attributes. However, you can use it to navigate and search an entire document tree in much the same way as you would with an individual tag.

The NavigableString Object

Retrieving tags and their attributes is useful for parsing the structure of an HTML/XML document tree, but it's not the only way to extract the relevant data. You can use the NavigableString object to select the string text that is *inside* tags, rather than the entire tag and its contents. You can get a NavigableString object by calling `.string` on a Tag object, like so:

```
>>> soup = BeautifulSoup('<p>This text is in <b>BOLD</b></p>')
>>> b_tag = soup.b
>>> b_tag.string
'BOLD'
```

Note that, although the output looks like a simple string, a NavigableString object contains a reference to the entire parsed document tree. If you want to use the string text outside of BeautifulSoup, convert it to a standard Python Unicode string to avoid wasting memory. For example:

```
>>> type(b_tag.string)
bs4.element.NavigableString
>>> b_str = str(b_tag.string)
>>> b_str
'BOLD'
```

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 2018

```
>>> type(b_str)
str
```

The Comment Object

The last of the main BeautifulSoup objects is the `Comment`. A `Comment` stores an HTML/XML-style comment that is found in the parsed document. It's really just a type of `NavigableString`, and can, therefore, be accessed in the same way:

```
>>> soup = BeautifulSoup('<p><!--This is a comment--></p>')
>>> soup.p.string
'<!--This is a comment--&gt;'
&gt;&gt;&gt; type(soup.p.string)
bs4.element.Comment</pre>

```

Each parser may handle `Comment` objects differently than `NavigableString` objects when it comes to formatting their appearance.

Tree Navigation

HTML and XML documents include markup in a tree-like hierarchy, where certain tags are placed within other tags, which are within yet more tags, and so on. This forms a parent–child relationship between tags in a document. So, you might need to access a tag that is within a specific tag. Or, you might need to access groups of tags and strings that are within a certain hierarchical context. You can navigate a document tree in one of four ways:

- Down the tree.
- Up the tree.
- Sideways in the tree.
- Back and forth based on parsing order.

The following HTML document will be used to demonstrate each type of navigation and search operation.

```
<html><head><title>Customer Information</title></head>
<body>
<p id="desc">Information for <b>Customer001</b> is listed below.</p>
<ul>
    <li><b>Name:</b> Aaron</li>
    <li><b>Email:</b> aaron@website.example</li>
    <li><b>Purchase total:</b> 125.45</li>
</ul>
<p id="upd">This information was last updated on 01-01-2020.</p>
<a href='https://www.website.example'>Home</a>
</body>
</html>
```

When rendered by a browser, this markup will look similar to the following:

Information for **Customer001** is listed below.

- **Name:** Aaron
- **Email:** aaron@website.example
- **Purchase total:** 125.45

This information was last updated on 01-01-2020.

[Home](#)

The BeautifulSoup object will be constructed as follows:

```
with open('customers.html') as file:
    soup = BeautifulSoup(file)
```

The parsed form of the document is quite similar to its original form; the only difference is that the `` elements are not indented.

Downward Tree Navigation

Navigating down the document tree involves stepping through tags from parents to their children. You can do this by continuing the chain of references on a Tag or BeautifulSoup object. For example, to get the first bold tag inside the first list item:

```
>>> soup.li.b
<b>Name:</b>
```

Instead of retrieving one specific tag, you may want to return all of the child tags of the parent element. You can do this by calling `.contents` on the parent. For example, the following code returns the children of the first list item:

```
>>> soup.li.contents
[<b>Name:</b>, ' Aaron']
```

Notice how a list was returned, where each child is an item in that list. In this case, the children of the first `` tag are the `` tag and its string text (' Aaron').

In addition to returning a list through `.contents`, you can iteratively return each child by using the `.children` generator:

```
>>> for child in soup.li.children:
...     print(child)
<b>Name:</b>
Aaron
```

However, what if you want to return each child tag *and* the children of the children? For that, you can use the `.descendants` generator:

```
>>> for child in soup.li.descendants:
...     print(child)
<b>Name:</b>
Name:
Aaron
```

Notice how this returned a third component where it didn't before—the `Name: string`. This is because `Name:` is a child of the `` tag, which is itself a child of the list item. By using `.descendants`, you will return all children and sub-children of the parent, while with `.children`, you will get only the direct children of the parent.

String Retrieval

Recall that you can retrieve a `NavigableString` object by appending `.string` to a `Tag` object. But, what if you want to retrieve all child strings of a parent tag instead of just one? You can use the `.strings` generator to do this. For example, to get all strings under the `` (unordered list) tag:

```
>>> for string in soup.ul.strings:
...     print(repr(string))
'\n'
'Name:'
'Aaron'
'\n'
'Email:'
'aaron@website.example'
'\n'
'Purchase total:'
' 125.45'
'\n'
```

As you can see, retrieving strings in this manner also retrieves newline characters (`\n`). Every time a tag is placed on a new line when it is parsed, a newline character is added. Referring to the original markup, you can see where the strings are on different lines.



Note: The `repr()` function demonstrates newline characters, rather than showing a blank space, as printing normally would.

To remove these newline characters, as well as any leading or trailing whitespace, use `.stripped_strings` instead:

```
>>> for string in soup.ul.stripped_strings:
...     print(string)
Name:
Aaron
Email:
aaron@website.example
Purchase total:
125.45
```

The newline characters are gone, as are the leading spaces in the field values.

Upward Tree Navigation

Just as you can navigate from parent to child, you can also navigate from child to parent. You can do this by calling `.parent` on a child tag. For example, to get the parent of the first `` tag:

```
>>> soup.b.parent
<p id="desc">Information for <b>Customer001</b> is listed below.</p>
```

The `<p id="desc">` tag is the parent of the `` tag, so it is returned.

You can also use the `.parents` generator to iteratively return all of a tag's parents. This can help you traverse the tree upward from child to parent. In the following example, the names of each parent tag for the `` tag are printed:

```
>>> for parent in soup.li.parents:
...     print(parent.name)
```

```
ul
body
html
[document]
```

In this case, the parent of `` is ``, the parent of `` is `<body>`, and so on until the top level of the document.

Sideways Tree Navigation

Another way to navigate the document tree is by going from one tag to another tag that is on the same level. In other words, the tags have the same direct parent. These are called *sibling* tags. To navigate from sibling to sibling, you can append `.sibling` to a tag. The following example retrieves the next sibling from the first `` tag:

```
>>> soup.li.next_sibling
'\n'
```

The result is a newline character, rather than the next list item. This is a common occurrence when you are navigating sideways through a document tree, as tags with the same parent are often placed on a new line. This is the case in the document example—each list item is on its own line. However, you can chain sibling calls to get the tags you actually want:

```
>>> soup.li.next_sibling.next_sibling
<li><b>Email:</b> aaron@website.example</li>
```

This is the second list item in the unordered list.

You can also go the opposite way by using `.previous_sibling`:

```
>>> soup.ul.previous_sibling.previous_sibling
<p id="desc">Information for <b>Customer001</b> is listed below.</p>
```

The `<p id="desc">` tag has the same parent as ``, and comes before ``.

Also, just like with downward and upward navigation, you can use the `.next_siblings` and `.previous_siblings` generators to return all siblings iteratively:

```
>>> for sibling in soup.li.next_siblings:
...     print(repr(sibling))
'\n'
<li><b>Email:</b> aaron@website.example</li>
'\n'
<li><b>Purchase total:</b> 125.45</li>
'\n'
```

Back and Forth Navigation

The last method of navigation involves going from element to element based on how a document is parsed. For example, consider the first `` tag in the document example:

```
<li><b>Name:</b> Aaron</li>
```

This markup is parsed as an event sequence:

1. The `` tag is opened.
2. The `` tag is opened.
3. The `Name:` string is added.
4. The `` tag is closed.
5. The `Aaron` string is added.
6. The `` tag is closed.

Using `.next_element`, you can go from step 1 to step 2:

```
>>> soup.li.next_element
<b>Name:</b>
To go from step 1 to step 3:
```

```
>>> soup.li.next_element.next_element
'Name:'
```

The equivalent statement for going in the opposite direction is `.previous_element`. Likewise, the `.next_elements` and `.previous_elements` generators return all elements in the order of parsing. For example, you can start at one tag and see how the rest of the document after that tag is parsed:

```
>>> for element in soup.a.next_elements:
...     print(repr(element))
'Home'
'\n'
'\n'
'\n'
```

Tree Searching

When you select a tag using notation like `soup.li`, only the first tag of this type is retrieved. But you may want to retrieve *all* tags matching a certain type. You can do this by searching the document from top to bottom by using the `find_all()` function. For example, to return all `` tags in the document:

```
>>> soup.find_all('li')
[<li><b>Name:</b> Aaron</li>,
 <li><b>Email:</b> aaron@website.example</li>,
 <li><b>Purchase total:</b> 125.45</li>]
```

The result is a list, where each item is a matching tag in the document. Because using `find_all()` is one of the most common tasks in Beautiful Soup, there is a shortcut:

```
>>> soup('li')
[<li><b>Name:</b> Aaron</li>,
 <li><b>Email:</b> aaron@website.example</li>,
 <li><b>Purchase total:</b> 125.45</li>]
```

You can also supply a list of strings to retrieve all occurrences of multiple tags:

```
>>> soup(['li', 'b'])
[<b>Customer001</b>,
 <li><b>Name:</b> Aaron</li>,
 <b>Name:</b>,
 <li><b>Email:</b> aaron@website.example</li>,
 <b>Email:</b>,
 <li><b>Purchase total:</b> 125.45</li>,
 <b>Purchase total:</b>]
```

Tree Searching Arguments

In addition to the name of the tag, you can also supply some arguments to the `find_all()` function to narrow your search. For example, you can set a limit on the number of items that are returned. The first two list items are returned in the following example:

```
>>> soup('li', limit = 2)
[<li><b>Name:</b> Aaron</li>, <li><b>Email:</b> aaron@website.example</li>]
```

You can also supply keyword arguments that correspond to attributes. For example, if you want to return all tags that have an `id` attribute equal to `'upd'`:

```
>>> soup(id = 'upd')
[<p id="upd">This information was last updated on 01-01-2020.</p>]
```

Another argument you can use is `string`. This enables you to search for `NavigableString` objects within a document that match a search term. The search term can be a simple string, but it can also be a regular expression, a list, and more. The following example searches for all instances of the string 'Home' in the document:

```
>>> soup(string = 'Home')
['Home']
```

You can also get the context of the tag that contains a string by supplying the tag name as an argument:

```
>>> soup('a', string = 'Home')
[<a href="https://www.website.example">Home</a>]
```

The `find()` Function

The `find()` function is similar to `find_all()` with `limit = 1`. The difference is that `find()` returns the result directly, and `find_all()` returns a list:

```
>>> soup.find_all('li', limit = 1)
[<li><b>Name:</b> Aaron</li>]
>>> soup.find('li')
<li><b>Name:</b> Aaron</li>
```

Additional Search Functions

There are additional search functions that essentially apply `find_all()` and `find()` to specific contexts within the tree. Those functions are:

- `find_parents()` and `find_parent()` —Searches the tree upward, from child to parent.
- `find_next_siblings()` and `find_next_sibling()` —Searches the tree sideways, from one sibling to the next sibling.
- `find_previous_siblings()` and `find_previous_sibling()` —Searches the tree sideways, from one sibling to previous sibling.
- `find_all_next()` and `find_next()` —Searches the tree back and forth based on parsing order, from one tag or string to the next.
- `find_all_previous()` and `find_previous()` —Searches the tree back and forth based on parsing order, from one tag or string to the previous.

In each pair of functions, the former returns all occurrences, and the latter returns only the first occurrence.

Pulling Scrapped Data into a DataFrame

The following code parses and scrapes the example HTML document and pulls that data into a `DataFrame`. More customer records have been added to the HTML document, and each one is formatted with the same markup structure as before.

```
records_df = pandas.DataFrame() # Set up empty DataFrame.

with open('customers.html') as file:
    soup = BeautifulSoup(file)

desc_tags = soup(id = 'desc') # Get all description paragraphs.
ul_tags = soup('ul') # Get all unordered lists.

# Add all customer IDs to a list.
cust_ids = []
```

```

for desc_tag in desc_tags:
    cust_ids.append(desc_tag.b.string)

# Construct DataFrame from list data for each customer.
for cust_id, ul_tag in zip(cust_ids, ul_tags):
    cols = []
    data = []
    for field in ul_tag('b'): # Start with bold tags.
        # Bold strings are column names.
        col_name = str(field.string).strip(':')
        cols.append(col_name)

        # Next string is data value.
        val = str(field.next_sibling.string).lstrip()
        data.append(val)

    # Set up temporary DataFrame for each record, then append to main.
    temp_df = pandas.DataFrame([data], index = [cust_id], columns = cols)
    records_df = records_df.append(temp_df)

```

The resulting DataFrame is as follows:

```

>>> records_df
      Name          Email Purchase total
Customer001   Aaron  aaron@website.example     125.45
Customer002  Bradley  bradley@website.example    83.12
Customer003  Cynthia  cynthia@website.example   213.05

```



Note: There are many ways to scrape data from this document. The preceding code was just one example.

Solutions

ACTIVITY 1-1: Selecting Python Data Science Tools

1. Why is Python such a popular programming language for data science? What are its advantages?

A: Answers may vary, as there are many reasons for Python's success in data science. One reason is the language itself—Python is a relatively simple, straightforward language to learn and implement. Another reason is that there is a major community-driven effort to create robust data science libraries for Python, which other languages may not have. The open source nature of these libraries means that developers can acquire and use them free of charge, as well as modify them if they so desire.

2. Which Python library is the foundation on which the other libraries are built?

- pandas
- NumPy
- Matplotlib
- SciPy

3. How does Seaborn differ from Matplotlib?

A: Seaborn is actually built on Matplotlib. However, Seaborn tends to be more user friendly and visually attractive.

4. Which tool suits this need?

- Anaconda
- Jupyter Notebook
- SciPy
- BeautifulSoup

5. If the company were to run these data science projects on local hardware, what are some of the things they need to consider?

A: Answers will vary. The hardware specifications, particularly the processing power and storage capacity of computers running the projects, will need to be adequate for what the team is trying to do. For example, if the team needs to perform complex mathematical operations on millions of points of data, the organization will need to invest in server-grade CPUs and high-capacity SSDs. Time constraints can also influence these decisions, as even underpowered hardware can accomplish these data science tasks—it'll just take much longer. And, the organization needs to consider that its requirements will be in a state of flux; the scale of its operations may start small, but as the organization acquires more and more data, the hardware will need to scale with it.

6. If the company were to run these data science projects on cloud services, what are some of the things they need to consider?

A: Answers will vary. Cloud services are a viable option for their elastic "pay only for what you use" business model. If the organization can't meet the hardware demands of its data science projects, it may want to offload those resources to the cloud. Of course, choosing the right cloud service is a challenge. The organization needs to consider factors like cost, ease of use, and available features. And, it must be aware of the pitfalls of cloud services, namely that it cannot claim total ownership of data that is out of its control.

ACTIVITY 1–3: Setting Up a Jupyter Notebook Environment

7. Why do you think this is?

A: The Jupyter Notebook environment maintains its execution state according to the cells that are run. Even though the cell that defines `my_var` comes before the `print` statement, the definition cell was not re-run, so the kernel still has the original value for `my_var`.

ACTIVITY 2–1: Creating NumPy Arrays

7. When might you use functions like `linspace()`, `arange()`, and `random.randint()`?

A: Answers may vary. Being able to create an array that fills a range of values is useful for testing certain NumPy functions or operations. They're also used in fields like regression. You can also use range arrays to transform your existing datasets. Creating arrays full of random data can be useful for adding a degree of unpredictability to your datasets or operations on those datasets.

ACTIVITY 2–2: Loading and Saving NumPy Data

9. Why were you unable to open this file in a text editor?

A: You saved this file in the Python pickle format, which is a binary format. Binary files are not text files, and their contents cannot be easily represented by a text editor.

10. What are some advantages and disadvantages of saving this type of file as compared to a CSV file?

A: Answers may vary, but a binary file can "clone" an array so its contents are an exact copy, while a text file may not be as true to the original array when it's loaded back in. However, text files are easier for humans and some programs to parse, and they are less of a security risk.

ACTIVITY 2-3: Analyzing Data in NumPy Arrays

9. What other libraries can you use to perform more specialized or advanced statistical analyses?

A: Answers may vary, but SciPy and statsmodels are both popular extensions to NumPy's statistical analysis functions. Both libraries are easy to apply to NumPy arrays.

ACTIVITY 3-1: Manipulating Data in NumPy Arrays

6. How many arrays has `finances` been split into, given the code cell you just ran?

- 2
- 3
- 4
- 5

9. What is the primary advantage of using a view over using a copy?

A: Using a view avoids having to create a separate object in memory when manipulating and modifying an existing array, which can be highly beneficial to performance when you are working with large datasets.

ACTIVITY 3-2: Modifying Data in NumPy Arrays

7. If you tried to compare these two arrays as is, what would happen?

- The arrays will be broadcast successfully since both arrays have the same dimensions.
- The arrays will be broadcast successfully since the `cogs` array will be padded with 1 to match the size of `total_and_rev`.
- Broadcasting will fail since both arrays have different sizes in the same axis, and neither of those sizes is 1.
- Broadcasting will fail since, even after being padded with 1, `cogs` will not have the same dimensions as `total_and_rev`.

ACTIVITY 4–3: Analyzing Data in DataFrames

6. If you wanted to index the DataFrame by numerical indices instead of labels, how would you retrieve the value in the fifth row, third column?
- stores_df[4, 2]
 - stores_df.iloc[4, 2]
 - stores_df[4, 2]
 - stores_df[[4, 2]]
11. What data type will a column adopt when any of its values are `NaN`?
- Boolean
 - String object
 - Integer
 - Float

ACTIVITY 5–1: Manipulating Data in DataFrames

5. Why would indexing a new column for `stores_df` and setting it equal to the `CustomerRating` values in `ratings_df` not be ideal?
- A: The customer ratings values are not in an order that is compatible with the order of rows in `stores_df`.
6. Why would appending `ratings_df` to `stores_df` not be ideal?
- A: Doing so would create duplicate rows in the main DataFrame, rather than merge the new column values where they belong.
10. Why will using the standard `pivot()` function not work on this DataFrame?
- A: The DataFrame has many duplicate values across its columns, which `pivot()` does not support. For example, there are multiple rows where `City` and `ProductLine` are the same, so you can't use a normal pivot.

ACTIVITY 6–2: Creating Subplots

4. Why would you want to avoid adding March to this modular plot?
- A: At some point, plotting too much data on one graph can lead to a noisy visual that is difficult to read and interpret.

ACTIVITY 6-3: Creating Common Types of Plots

8. How does a violin plot differ from a histogram in terms of how it calculates the distribution of variables?

A: Histograms fit the spread of values within a pre-defined number of bins, but choosing the right number of bins can be difficult. Violin plots use kernel density estimation (KDE) to automatically derive probabilities for the spread of values without needing to specify bins.

ACTIVITY 6-4: Formatting Plots

8. What type of color map is best used for data where the middle values are of special importance?

- Diverging
- Sequential
- Qualitative
- Cyclic

ACTIVITY 6-5: Streamlining Plotting with Seaborn

9. How do axes-level functions in Seaborn differ from figure-level functions?

A: An axes-level function returns a Matplotlib `Axes` object, and is typically used to draw an individual plot. A figure-level function returns a Matplotlib `Figure` object with multiple `Axes` objects, and is typically used to generate multiple plots at once in order to show relationships between these plots.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

Glossary

Anaconda

A cross-platform, open source distribution that includes many Python data science libraries.

Anaconda Navigator

A graphical user interface (GUI) for managing software that is available through the Anaconda distribution.

area plot

A type of line plot in which the space below the line is filled in with some color or texture.

bar chart

A type of plot that represents the proportion measurement of categorical variables by using either horizontal or vertical bars.

Beautiful Soup

A Python library for scraping and parsing HTML and XML documents.

box plot

A type of plot that represents the distribution of a numeric variable using five different summary statistics.

broadcasting

The process of making one NumPy array compatible with another when performing arithmetic operations on arrays of different shapes.

color bar

An element of a plot that shows a range of values through changes in color.

color map

A collection of multiple colors that are assigned to plotted data, typically to show the relationships between values.

Conda

The package manager that comes with the Anaconda distribution.

contour plot

A type of plot that represents three-dimensional data in a two-dimensional format by plotting contour lines.

data science

The discipline that involves accumulating data, analyzing the data, extracting value from the data, and presenting the value of the data in a meaningful way.

fancy indexing

A type of array indexing in which the indices are themselves arrays or lists of values.

grouping

In pandas, the process of facilitating the analysis and manipulation of data within related portions of the overall dataset.

heatmap

A type of plot that shows different shades or intensities of color on a matrix based on data values in that location of the matrix.

histogram

A type of plot that represents the distribution of a continuous variable.

image plot

A type of plot that shows a bitmap image on a two-dimensional plotting surface.

IPython

An interactive command shell for the Python programming language.

IQR

(interquartile range) A summary statistic that measures the difference between upper and lower quartiles in a distribution.

Jupyter Notebook

A web application that enables users to create, view, and share files that include live, executable code, as well as explanatory markup text.

kernel

In the context of Project Jupyter, a code execution environment that is usually specific to one programming language implementation.

line plot

A variant of a scatter plot in which a series of lines connects data points in order.

linear regression plot

A type of plot that shows the linear relationship between an independent and dependent variable by plotting a line of best fit.

magic command

A type of command unique to IPython that provides additional functionality within Python programming.

masking

The bitwise process of selecting a subset of a NumPy array to keep, while discarding the rest.

Matplotlib

A Python library that includes various methods for plotting data on graphs.

matrix

An array of numbers arranged in the form of rows and columns, commonly used in linear algebra.

multi-indexing

A method of giving a `DataFrame` multiple row indices to represent data hierarchically.

NumPy

A Python library for creating and performing mathematical calculations on large, multi-dimensional arrays

NumPy array

A multi-dimensional, single-type array object provided by the NumPy library for efficient processing of data.

open source

A development methodology in which users are permitted to view, copy, and modify computer code for any reason, as well as distribute it to anyone—all in support of open collaboration.

package manager

A program that streamlines the process of querying, installing, updating, and uninstalling packaged software.

pandas

A Python library that provides the `DataFrame` structure for data analysis and manipulation.

pickle

The main data serialization format in the Python programming language.

pie chart

A type of plot that represents the numerical proportion of some categorical variable in the form of a circle.

Project Jupyter

An open source project that provides interactive computing environments for many languages, including Python.

quantile

One of several possible ranges of values that divide a distribution. If divided into four ranges, each range is called a quartile.

quartile

See quantile.

quiver plot

A type of plot that represents four-dimensional data in a two-dimensional format by plotting arrows.

reindexing

The process of changing the order and/or labels of DataFrame indices.

scatter plot

A type of plot that represents the relationship between two variables through the use of points on a graph.

SciPy

A Python library that builds on NumPy by offering more powerful mathematical functions, particularly those used in the field of scientific computing.

Seaborn

A Python library that extends the functionality of Matplotlib by incorporating more types of plots and by streamlining the plotting process.

spine

The axis line between the plotting area and the ticks.

statsmodels

A Python library that provides functions for statistical modeling and testing.

subplotting

The process of creating multiple smaller plots within a larger overall plot.

tick

A value that aligns with an axis that is used to visually demonstrate where a data point falls on that axis.

universal function

A vectorization function that performs an element-wise mathematical operation on a NumPy array.

vectorization

The process of leveraging pre-existing NumPy functions to operate over an entire array.

view

The result of slicing a NumPy array, in which the slice object shares data with the original array, rather than copying that data.

violin plot

A type of plot that shows the distribution of a numerical value through probability density.

web parsing

The process of analyzing and selecting specific portions of web content based on patterns in the text.

web scraping

The process of extracting data from web-based content formats, particularly those hosted on websites accessible through the Hypertext Transfer Protocol (HTTP).

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Jan 13 20:

Index

3-D plots 251

A

Anaconda 3, 8
Anaconda Navigator 8
area plots 202

B

bar charts 203
Beautiful Soup
back and forth tree navigation 327
BeautifulSoup object 323
Comment object 324
defined 3
downward tree navigation 325
NavigableString object 323
overview 320
sideways tree navigation 327
supported parsers 321, 322
Tag object 322
tree navigation 324
tree searching 328
upward tree navigation 326
box plots 205

C

cloud services 4
code cells 18
color maps 268, 269
Conda 9
conda command
syntax 10
usage 10

contour plot 247

D

DataFrame.any() function 142
DataFrame.append() function 166, 167
DataFrame.apply() function 140
DataFrame.copy() function 166
DataFrame.corr() function 137
DataFrame.describe() function 138
DataFrame.drop() function 190
DataFrame.fillna() function 185
DataFrame.groupby() function 172
DataFrame.head() function 116
DataFrame.idxmin() and
DataFrame.idxmax() functions 136
DataFrame.iloc() function 135
DataFrame.isna() and DataFrame.notna()
functions 141
DataFrame.join() function 167
DataFrame.loc() function 135
DataFrame.pivot() function 168
DataFrame.plot() function 200
DataFrame.rename() function 185
DataFrame.sort_index() function 170
DataFrame.sort_values() function 169
DataFrame.tail() function 117
DataFrame.to_csv() function 127
DataFrame.to_html() function 127
DataFrame.transpose() function 169
DataFrame.where() function 186
DataFrame object
area plots 202
arithmetic functions 186, 189
attributes 115
bar charts 203

box plots 205
comparison functions 153
creation 115
filtering with masks 154
histograms 204
indexing 134
line plots 201
loading data 124, 126
logical operators 154
multi-indexing 173, 174
overview 114
pie charts 206
pulling scraped data 329
reindexing 137
saving data 126, 128
scatter plots 200
slicing 152
stacking functions 175
statistical summary functions 139
data science
cloud considerations 4
defined 2
hardware considerations 4
reference documentation 3
tools overview 2
dynamic typing 30

F

fancy indexing 51
Figure.add_axes() function 227
Figure.add_subplot() function 228
Figure.subplots() function 229

G

grouping
in pandas 171

H

heatmaps 303
histograms 204

I

image plots 249
interquartile range, *See* IQR
IPython 14, 16
IQR 246

J

Jupyter Notebook
dashboard 17
defined 3
document cells 18
document interface 15
document menu 19
overview 15
plot() 214, 226

K

kernels 16

L

linear regression plots 304
line plots 201

M

machine learning 2
machine learning libraries 3
magic commands 14
markup cells 18
masking 97
Matplotlib
annotation 278
Axes.set() function 280
axis limits 276
axis ticks 272
color bars 275
defined 3
grids 277
GridSpec object 231
legends 274
modular plotting 224
object-oriented interface 215
plot axis labels 271
plot formatting 266
plot title labels 270
plot types 242, 246, 247, 249, 251
pyplot.plot() function 214
pyplot.savefig() 214
pyplot.show() 214
shared axes using subplots() 230
stateful interface 215
stateful vs. object-oriented functions 280
style.use() function 265
subplotting 226
text properties 277

vs. Seaborn 292
 matplotlib.pyplot module 214
 mixed data types 187

N

ndarray object 31
 NumPy
 arithmetic functions 91
 broadcasting 92
 comparison functions 96
 defined 2
 exponential and logarithmic functions 94
 limitations 112
 logical functions 97
 mathematical functions 95
 missing data 141
 searching functions 79
 stacking functions 76
 string modification functions 98
 universal functions 91
 numpy.append() function 76
 numpy.argsort() function 78
 numpy.array_split() function 77
 numpy.array() function 32
 numpy.char module 98
 numpy.concatenate() function 75
 numpy.copy() function 72
 numpy.delete() function 90
 numpy.flip() function 74
 numpy.insert() function 90
 numpy.load() function 43
 numpy.loadtxt() function 42
 numpy.nditer object 53
 numpy.partition() function 79
 numpy.random module 35
 numpy.ravel() function 74
 numpy.reshape() function 73
 numpy.resize() function 73
 numpy.savetxt() function 43
 numpy.sort() function 77
 numpy.split() function 77
 numpy.stack() function 76
 numpy.transpose() function 75
 numpy.unique() function 55
 numpy.where() function 98
 NumPy arrays
 attributes 32
 creation functions 34
 data types 33
 defined 31

homogeneous data 31
 indexing 51
 iteration 53
 loading data 42
 saving data 43
 SciPy integration 56
 slicing 52
 statistical summary functions 54
 statsmodels integration 58
 type conversion 31

O

open source 2

P

package managers 9
 pandas
 DataFrame object 114, 116, 124, 126
 data structures 112
 defined 3
 grouping 171
 missing data 140
 multi-indexing 173
 Series object 112
 pandas.read_csv() function 124
 pandas.read_html() function 125
 pandas.to_datetime() function 188
 pandas.to_numeric() function 188
 pickle 43
 pie charts 206
 pip 9
 Project Jupyter 15
 pyplot.axes() function 228
 pyplot.contour() function 247
 pyplot.imshow() function 249
 pyplot.legend() function 274
 pyplot.quiver() function 249
 pyplot.rc() function 266
 pyplot.subplots() function 230
 pyplot.violinplot() function 246
 Python lists
 traditional 30

Q

quantiles 302
 quartile 205
 quiver plots 249

R

random number generation [35](#)

S

scatter plot [200](#)

SciPy

defined [2](#)

integration [56](#)

subpackage [56](#)

Seaborn

axes-level functions [296](#)

categorical plots [302](#)

defined [3](#)

distribution plots [301](#)

FacetGrid example [298](#)

figure-level functions [297](#)

heatmaps [303](#)

integration with Matplotlib [292](#)

linear regression plots [304](#)

line plot [297](#)

plot type categories [300](#)

relational plots [300](#)

seaborn.axes_style() function [294](#)

seaborn.clustermap() function [304](#)

seaborn.color_palette() function [295](#)

seaborn.despine() function [295](#)

seaborn.set_style() function [293](#)

seaborn.set() function [293](#)

vs. Matplotlib [292](#)

Series object

attributes [115](#)

creation [114](#)

overview [112](#)

statsmodels

defined [3](#)

integration [58](#)

statistics submodules [59](#)

T

ticks [272](#)

tree navigation

back and forth [327](#)

downward [325](#)

overview [324](#)

sideways [327](#)

upward [326](#)

tree searching [328](#)

V

vectorization [53](#)

view [72](#)

view_init() function [252](#)

violin plots [246](#)

W

web parsing [320](#)

web scraping [320](#)

Do Not Distribute or Duplicate

Do Not Duplicate or Distribute

094001S rev 1.0
ISBN-13 978-1-4246-4053-9
ISBN-10 1-4246-4053-9



9 781424 640539

Licensed For Use Only By: Abdulwahab Alweban dev at 12345@gmail.com Jan 13 2015