
Python for Nonprofits

Kenneth Burchfiel

Mar 26, 2025

CONTENTS

I	Introduction	5
1	An Introduction to Python for Nonprofits	7
1.1	Overview of PFN's contents	7
1.2	My reasons for writing PFN	7
1.3	Development notes	8
1.4	Gratitude and acknowledgments	10
1.5	Dedication	11
2	The Case for Using Python at Nonprofits	13
2.1	Python is <i>flexible</i>	13
2.2	Python is <i>fast</i>	14
2.3	Python is <i>free</i>	14
2.4	Python is <i>famous</i>	15
2.5	Python is <i>fun</i> (to me, at least!)	15
2.6	Don't take my word for it—give it a try yourself!	15
3	Getting Started	17
3.1	Initial setup	17
3.2	Errata	18
3.3	Let's begin!	18
II	Importing and Prepping Data	19
4	Data Retrieval	21
4.1	A Pandas primer	21
4.2	Back to data retrieval	22
4.3	Importing .csv data	22
4.4	Importing .xlsx data	24
4.5	Importing SQL data	24
4.6	Importing HTML data	25
4.7	What about exports?	26
4.8	Conclusion	27
5	Data Prep	29
5.1	Reformatting and cleaning our winter results dataset	31
5.2	Removing duplicates	34
5.3	Combining winter survey results with our fall/spring dataset	36
5.4	Merging college and level data into our survey results table	37
5.5	Conclusion	38

III Analyzing Data	39
6 Descriptive Stats	41
6.1 Evaluating changes in average university-wide results during the school year	42
6.2 Calculating response rates	43
6.3 Using the <code>columns</code> argument within <code>pivot_table()</code> to show seasons side by side	44
6.4 Adding additional pivot index values	46
6.5 Comparing rows via <code>sort_values()</code> and <code>rank()</code>	50
6.6 Calculating percentiles and ranks	52
6.7 Conclusion	54
7 Descriptive Stats: Part 2	55
7.1 Calculating weighted average results by student (and dealing with missing values)	56
7.2 The danger of relying on column index positions when analyzing datasets	60
7.3 Why column-wise operations should be preferred over for loops	61
7.4 Microdata, pre-baked data, and the ‘average of averages’ problem	63
7.5 Handling missing data when creating pivot tables	67
7.6 The advantages of <code>map()</code> and <code>np.select()</code> over <code>np.where()</code> when missing data are present	71
7.7 Conclusion	75
8 Census Data Imports	77
8.1 An introduction to the American Community Survey	77
8.2 Deciding where to move to start a family	78
8.3 Examining connections between education levels and median incomes	78
8.4 Getting started with the Census API	78
8.5 Importing custom Census functions	80
8.6 Retrieving variable and group information	81
8.7 Creating aliases and specifying retrieval years	82
8.8 Calling <code>retrieve_census_data()</code>	83
8.9 Adding in comparison fields	87
8.10 Ranking counties by population growth	87
8.11 Repeating these steps in order to compare 25-to-29-year-old population growth rates	90
8.12 Retrieving data for our education/income regression analyses	96
8.13 Appendix	101
8.14 Conclusion	103
8.15 <code>census_import_scripts.py</code>	103
IV Visualizing Data	111
9 Creating Interactive and Static Charts with Plotly	113
9.1 Creating bar graphs	114
9.2 Creating line graphs	125
9.3 Scatter plots	129
9.4 Histograms	131
9.5 Treemaps	134
9.6 Comparing these treemaps to bar charts	140
10 Pivot and Graph Functions	143
10.1 The motivation behind this chapter	143
10.2 Creating a detailed survey results dataset	144
10.3 A basic autopivot function	145
10.4 A basic autobar function	147
10.5 Testing out <code>autopivot_simple()</code> and <code>autobar_simple()</code>	147
10.6 Importing autopivot and autobar functions	151

10.7 Conclusion	159
11 Mapping Census Data using Plotly (Work in progress)	161
11.1 A note on Plotly vs. Folium	161
11.2 Importing data to be mapped	162
11.3 Creating an initial choropleth map with a linear scale	166
11.4 Creating map with a percentile scale	168
11.5 Mapping 25-to-29-year-old population growth at the county level	175
11.6 Performing simliar steps to map state-level growth data	176
11.7 Rendering our state-level maps	178
11.8 Creating tiled choropleth maps	180
11.9 <code>plotly_choropleth_map_functions.py</code>	184
12 Creating Choropleth Maps with Folium	195
12.1 Gathering our net migration data	196
12.2 Importing county and state boundaries	203
12.3 Creating a choropleth map with interactive tooltips	207
12.4 Creating a more efficient version of this map	213
12.5 <code>folium_choropleth_map_functions.py</code>	218
V Regressions	229
13 Regression Analyses	231
13.1 An introduction to regressions	231
13.2 Important caveats	232
13.3 Part 1: Analyzing bookstore sales	233
13.4 Part 2: Evaluating the relationship between bachelor's degree prevalence and median incomes	247
13.5 Part 3: A word of caution regarding significance testing	281
13.6 Conclusion	284
VI Publishing Analyses Online	285
14 Creating a Simple Static Webpage	287
14.1 Creating Static Sites with GitHub Pages	287
14.2 More advanced sites	290
15 Updating Online Spreadsheets	291
15.1 Prerequisites	291
15.2 Importing weather data	293
15.3 Reading these datasets into DataFrames	294
15.4 Importing these DataFrames into a Google Sheets workbook	295
15.5 Appendix	296
15.6 <code>weather_import.py</code>	298
16 Simple Dash App Without Login	305
16.1 Introduction to the Online Visualizations section of Python for Nonprofits	305
16.2 The 'Simple App Without Login' project itself	305
16.3 Source code for Simple App Without Login	308
17 PFN Dash App Demo	313
17.1 Readme	313
17.2 Source code for PFN Dash App Demo	319

VII Appendix	347
18 NVCU Database Generator	349
18.1 Connecting to our NVCU database via the SQLAlchemy library	349
18.2 Creating a current enrollment table	350
18.3 Creating a survey results table	356
18.4 Creating a dining transactions table	360
18.5 Creating a table of fall and spring bookstore sales by student for a given year:	361
19 helper_funcs.py	363

Python for Nonprofits (PFN) is a guide to applying Python in nonprofit settings. Specifically, it teaches you how to use Python to:

- Import data, then clean and reformat it
- Analyze data using descriptive statistics and linear regressions
- Create both charts and maps
- Share spreadsheets and interactive visualizations online

This project is *not* meant to replace an introductory Python course or textbook. If you're new to Python, I suggest getting started with a resource like the 3rd edition of Think Python (<https://greenteapress.com/wp/think-python-3rd-edition/>). (I found the 2nd edition of this book to be very helpful in my own studies.) However, as long as you have an introductory background in the language, you should be in a great position to benefit from this project.

Many Python for Nonprofits sections incorporate *simulated* data from a *fictional* university (Northern Virginia Catholic University, or NVCU for short). Educational institutions have an immense amount of data and many reporting needs, so it made sense to center much of the text on data analysis work for a university. However, PFN also makes use of real-world data from the US Census Bureau.

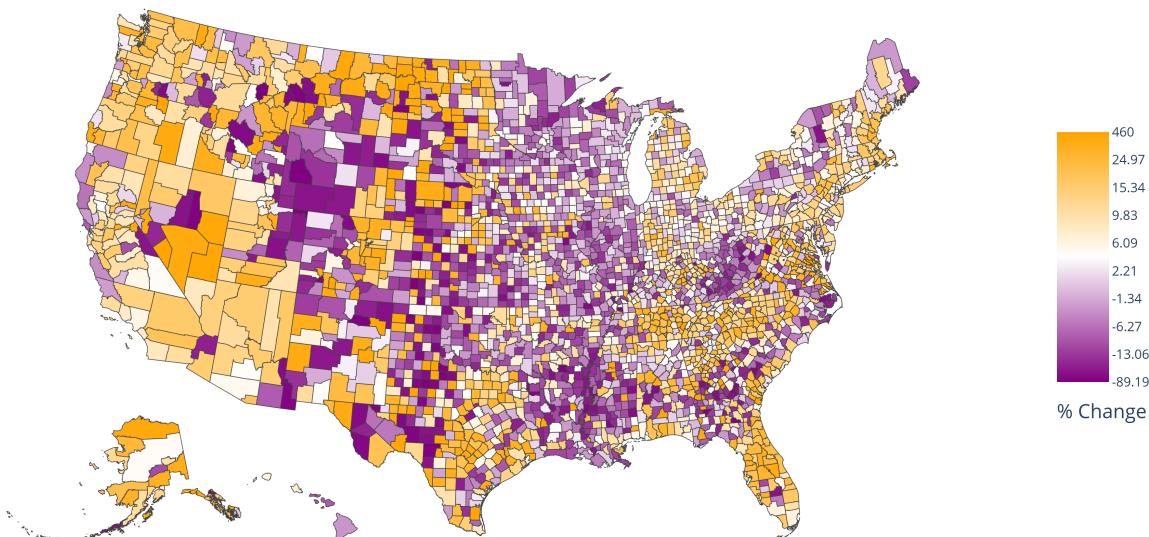
I've released PFN under the MIT license. This means that you can use this code in both commercial and personal applications *and* that you can choose whatever license you want for your own project. You would just need to give me credit if you use a substantial amount of the code. This choice of license makes PFN much more flexible than many other educational Python resources.

For a more comprehensive guide to Python for Nonprofits, reference the Introduction section.

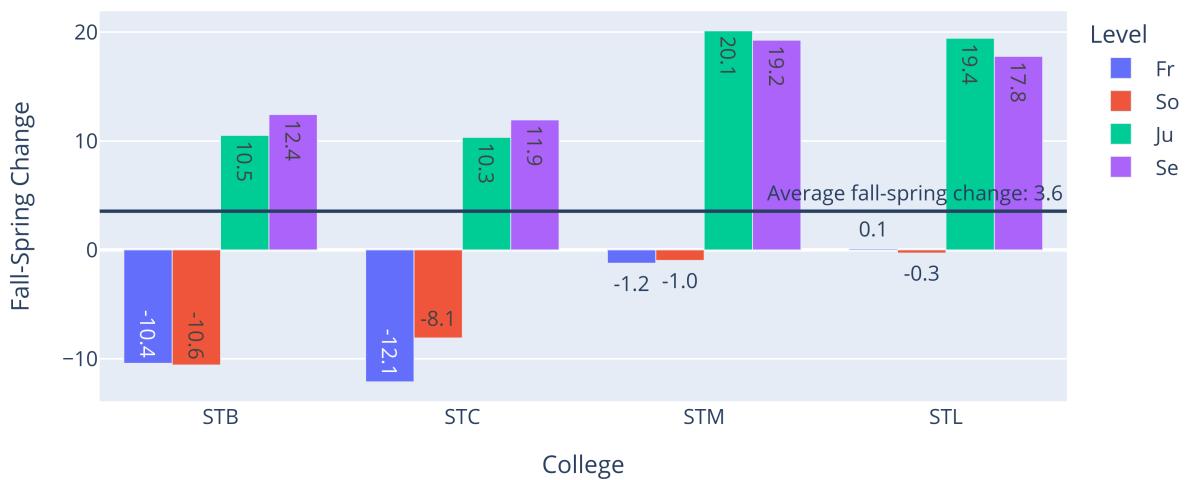
Sample output

Python for Nonprofits will teach you how to create visuals like the following:

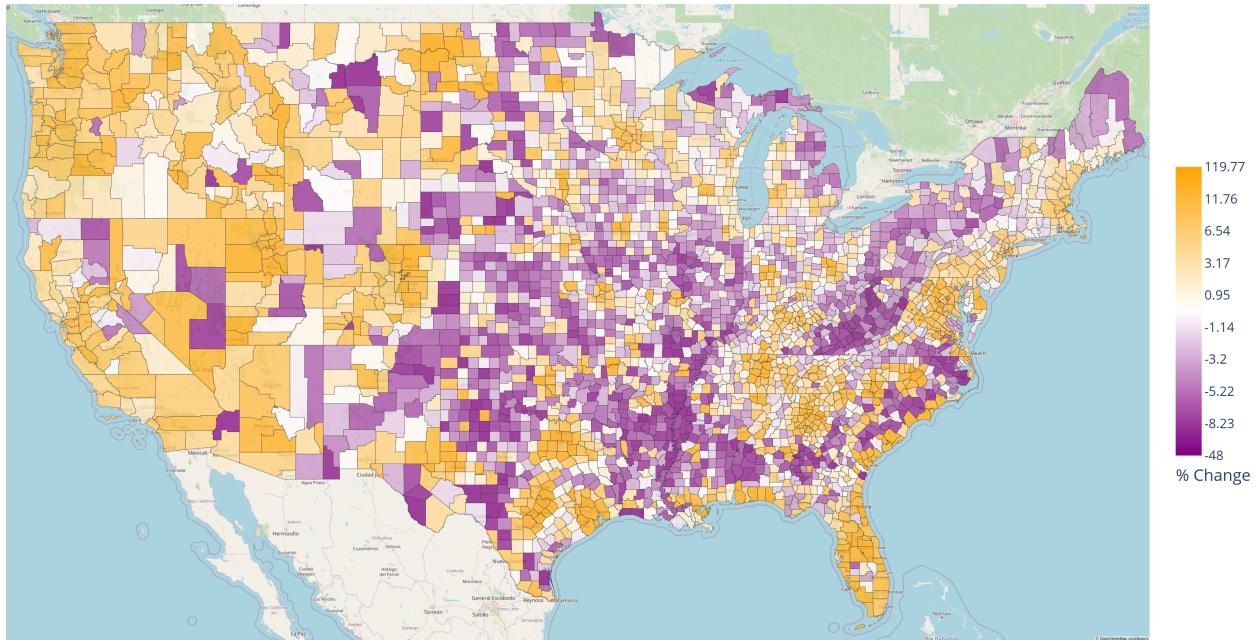
25-29 Year-Old Population % Change by County from 2011 to 2021



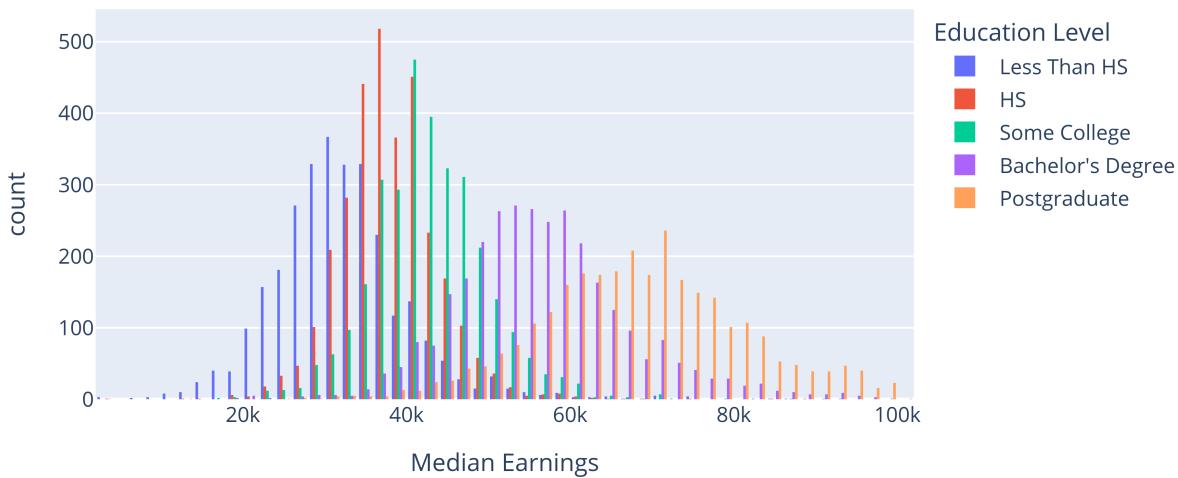
Average Fall-Spring Sales Growth by College and Level



Population % Change by County from 2011 to 2021



Distribution of County-Level Median Earnings by Education Level



(These are all static images, but you'll also learn how to create interactive HTML versions of these charts as well.)

Visualizations are just one component of Python for Nonprofits, but I did want to add some images to this section—and maps and charts make for more interesting images than do screenshots of tables!

I had a great time putting Python for Nonprofits together, and I hope that you find that this work helps advance your own nonprofit career. Feel free to contact me at kburghfiel3@gmail.com (or post a message at <https://github.com/kburghfiel/pfn/issues>) if you have any questions or notice any errors with the text.

Happy coding!

—Ken

Part I

Introduction

AN INTRODUCTION TO PYTHON FOR NONPROFITS

1.1 Overview of PFN's contents

Python for Nonprofits covers a wide range of nonprofit data analysis tasks. First, the **Importing and Prepping Data** section will teach you how to read data into your Python scripts, then clean and reformat them. Next, the **Analyzing Data** section will demonstrate how to calculate various descriptive statistics.

Once you've calculated such statistics, you'll often want to visualize them; therefore, this component is followed by the **Visualizing Data** section, which shows how to display your findings in both graph and map form.

PFN then provides a brief introduction to inferential statistics via its **Regressions** section. (I had initially planned to place this component within the Analyzing Data section, but because it relies so heavily on visualizations, I realized that I should have it follow my data visualization code.) Finally, in the **Publishing Analyses Online** section, you'll learn how to use Python to export data to Google Sheets and display it within online dashboards.

Later sections of Python for Nonprofits often reference content covered in earlier sections. For instance, data cleaning and reformatting will come up in many areas of PFN, not just the data prep chapter. In addition, the Regressions and Publishing Analyses Online sections will make extensive use of the graphing tools that the Visualizing Data section introduces. Therefore, I would recommend reading through PFN's sections in the order shown above.

1.2 My reasons for writing PFN

To explain why I wrote this text, it might help to provide some background about my own education and career path. I should first clarify that I do *not* have a formal background in mathematics or computer science. Therefore, if you (like many nonprofit staff) have focused more on qualitative than quantitative skills in your schooling and career, don't worry—I'm very confident that you can still learn and apply Python in your work!

I focused mostly on writing in high school, then majored in psychology at Middlebury College. After working as an AmeriCorps VISTA member in Houston, I obtained a Master of Science in Social Work at UT Austin and worked as a mental health counselor with Catholic Charities of Evansville for two years.

I first began studying Python seriously as a first-year MBA student at Columbia Business School, when I enrolled in Professor Mattan Griffel's Introduction to Programming in Python course. I was amazed with how easily Python facilitated various data analysis and visualization tasks, and I also loved how programming engaged both the creative and problem-solving portions of my brain. I then had the chance to take several other programming-related classes at CBS with Daniel Guetta, Hardeep Johar, and Jared Lander; these courses further opened my eyes to the capabilities of Python and other open-source tools.

I was fortunate to be able to apply Python during a summer internship at Seton Education Partners in The Bronx. My supervisor, David Morales, gave me lots of freedom to use Python for many analysis-related tasks. I greatly enjoyed my summer at Seton, so I returned there full time after completing my MBA. At Seton, I not only applied the Python knowledge that I had picked up during my MBA but also learned many new libraries and applications.

My wife, son, and I then moved to Virginia in 2024 to be closer to our families. As a Research Fellow with the Institute for Family Studies, I have been applying Python (along with R, a powerful open-source statistical programming language) to analyze survey data and create visualizations.

My professional experiences have taught me that Python is a very powerful tool for nonprofit work. It not only saves a great deal of time, but also allows for the creation of analyses and visualizations that would be difficult or impossible to perform in spreadsheet-based tools like Excel. (For a more detailed list of reasons to apply Python at your nonprofit, read ‘The Case for Python at Nonprofits’ within this same section.)

Therefore, I wanted to create a guide that would allow others to learn how to apply Python at their present (or future) nonprofit jobs. Much of the text focuses on applications and methods that I have used in my own work, so I’m confident that you’ll find this work to be relevant to many of your own nonprofit data analysis tasks as well. (I should also clarify, though, that I did not reference or copy any code from my work in the process of writing this text.)

I also hope to teach Python on a part-time basis one day; if that opportunity arises, I would plan to incorporate this guide into my curriculum.

Those are some of my ‘practical’ reasons for writing Python for Nonprofits. However, I also found this project to be intrinsically rewarding—and truly *fun* as well. I greatly enjoy learning new things, than teaching others about them; therefore, it has been immensely enjoyable to develop the code and documentation for this project. There were many evenings in which I had trouble shutting down my computer and going to bed because I was having such a good time with PFN!

I recognize that writing educational Python materials is a strange hobby, but it’s one that I hope to continue for years to come. I hope that you will have as much fun reading this text—and putting its examples into practice in your own personal and professional life—as I have had writing it.

1.3 Development notes

It took me quite a while to write PFN. I was working full time during most of the writing period, and the birth of our first child also kept me and my wife (but especially my wife!) quite busy. Therefore, I generally worked on the project later in the day—and often right before going to bed. The benefit of this arrangement, though, was that the time I spent on PFN felt more like a fun break from other responsibilities than a chore in itself. I found that, as long as I had 4 or 6 hours a week to spend on this project, I could write a decent amount over the course of a year.

Most PFN files were written in Jupyter Notebook (.ipynb) format, though quite a few—such as this chapter—were produced in Markdown format (.md). I also moved some Python code into standalone .py files to make them easier to incorporate into various projects. I used JupyterLab Desktop (<https://github.com/jupyterlab/jupyterlab-desktop>) for almost all of the writing process, though the code should also be viewable and executable within other code editors that support .ipynb files.

I switched over to Linux Mint in 2024 for most of my professional and personal work; however, the code should also run fine on Windows and Mac. (Please let me know if you have any trouble with certain parts of PFN on one of those operating systems.)

1.3.1 Why I released PFN under the MIT license

In my own personal and professional work, I strongly prefer permissively-licensed* code (such as the MIT and Apache licenses) over both proprietary code and ‘copyleft’ licenses like the GPL. That’s because these licenses allow me to incorporate such code into proprietary works without then needing to use that same copyleft license for those works. I’m also not a fan of ‘noncommercial’ licenses (such as the CC BY-NC license), as the prohibition on commercial use severely limits the utility of the text or code for certain use cases. Given the many ways that I have benefit from MIT-licensed code, it seemed fitting to release PFN under it as well.

I also believe that this choice will make PFN more accessible *and* practical to prospective readers. As a result of this licensing decision, you are free to copy large chunks of code from the book (such as the functions that under-

lie certain mapping and online visualization code) into your own commercial projects, then release those projects under a proprietary license. (You'd just need to provide attribution and reference the license; for more details, see <https://github.com/kburchfiel/pfn?tab=MIT-1-ov-file#readme>). I would love for as many people as possible to read and benefit from PFN, and a restrictive license would have made that goal harder to achieve.

*(For a useful overview of permissive licenses, visit https://en.wikipedia.org/wiki/Permissive_software_license).

1.3.2 The book version of PFN

Although I imagine that most people prefer to access resources like Python for Nonprofits online, it was also important to me that I make it available in print form. Although web-based resources offer incredible convenience and accessibility, to truly learn a subject, I find that it helps to read through a book from cover to cover.

Therefore, I made use of the fantastic Jupyter Book (<https://jupyterbook.org/en/stable/intro.html>) tool to convert the .ipynb and .md files that comprise Python for Nonprofits into a single PDF. Jupyter Book took care of much of the formatting for me, which thus allowed me to focus more on the actual content. It appeared that Jupyter Book couldn't process .py files on its own, though, so I copied those scripts into standalone Jupyter Notebooks, thus allowing them to get incorporated into the book.

I plan to make the PDF version of Python for Nonprofits available to order in printed form in the near future—though you are of course welcome to print it out yourself in the meantime. It is available as ‘pfn_book.pdf’ at https://github.com/kburchfiel/pfn/blob/main/Print_Book/pfn_book.pdf . You may also want to check the folder in which it's stored (https://github.com/kburchfiel/pfn/blob/main/Print_Book) to see whether a newer verison has become available.

If you're interested in using Jupyter Book for your own articles and books, check out the documentation at <https://jupyterbook.org/> . Note that the _config.yml, _toc.yml, create_html_book.sh, and create_pdflatex_book.sh files in PFN's root folder are part of my workflow for turning PFN into a Jupyter Book project. (These files were also based on Jupyter Book's documentation.)

Jupyter Book *also* allowed me to convert these files into an HTML-based book that you may find useful. (I haven't yet gotten it to display onine correctly, but if you download PFN's source code, you can access a local copy by going to docs/index.html . See the Getting Started section, found in the main PFN readme on GitHub or the ‘Getting Started’ chapter of the Print/PDF book, for more details.)

1.3.3 My reasons for eschewing generative AI

I chose not to use generative AI tools in the process of writing Python for Nonprofits. When I needed to learn how to perform a particular task, I preferred to consult the documentation for the library that I was using; StackOverflow answers; or similar human-created resources.

I had a few reasons for this approach. First, I believed that it was important for me to understand how my code was working and what it was doing. If I relied on generative AI to solve challenges for me, my own understanding of Python—and the libraries that PFN utilizes—would suffer. If Kristen Nygaard is correct that ‘*Programming is understanding*’ (https://www.stroustrup.com/3rd_pref.html), then copying and pasting Generative AI-produced code could lead to a *lack of understanding* on my part. And in the world of data analysis, this deficit in understanding could lead to all sorts of insidious errors.

I also have substantial concerns about potential copyright infringement caused by generative AI. If a generative API tool provided me with code that was actually protected by a ‘copyleft’ license like the GPL, I might then need to release my entire project under the GPL rather than the more ‘permissive’ MIT license that I selected. If it instead gave me a block of code that was actually covered under a proprietary license, I might then get sued for using someone else’s work without his or her permission.

In addition, I wanted my own style—both in writing and programming terms—to manifest itself within Python for Nonprofits. It would have been harder to maintain that voice if I interspersed generative AI-generated content within my own text.

Finally, as noted earlier, I find the process of writing my own code to be a great deal of fun—which is a relevant consideration for a hobby project like this. If I outsourced that process to an AI-based tool, I doubt that I would have enjoyed working on PFN as much.

None of this is meant to imply that individuals who use generative AI, whether for programming, writing, or other forms of content creation, are mistaken in their approach. I simply believe that avoiding the use of generative AI so far has been the best choice *for me*.

I also believe that, even for those who do utilize generative AI in their development work, it's very helpful—and, at times, crucial—to know how to write that code yourself. Although using AI might help speed up the process of writing code, there's still a need to refine, evaluate, and correct it—and knowing what the code *should* look like will help with that process. As Isaac Lyman noted, “It’s the AI’s job to be fast, but it’s your job to be good.”*

Aviation provides a useful analog here. Although commercial airline pilots heavily utilize autopilot**, it's still essential that they know how to fly the plane on their own if needed—and that they can recognize when that autopilot might be acting up. Similarly, even if you plan to make heavy use of generative AI in your work (or are already doing so), knowing how to write the same code on your own will remain a valuable skill, as you can then better evaluate your AI tool's suggestions. Therefore, this book—a flight manual of sorts for nonprofit data analysis—should still be a helpful resource for you.

1.4 Gratitude and acknowledgments

First, I am grateful to God for giving me the opportunity to write this text and share what I've learned with others. God's infinite love is a gift that cannot be put into words, and I hope that PFN can serve as a very small offering of love back to Him.

I am also grateful to Kenneth, my son, who has brought me and my wife joy, laughter, and a deeper sense of meaning and purpose for our lives. Although seeing Python for Nonprofits grow has been fun, *your* growth has been far more exciting—and, of course, far more important! This text and code will pass away, but your soul is eternal. I know you're a bit young to use a computer at the moment, but I hope you will find this project useful one day.

I'm grateful to my parents, Linda and Ken, as well. You provided me with immense love growing up, and I hope I can share that same love with Kenneth (and any other children with whom God might bless us). Dad, you showed me (*with Biotechnology and the Federal Circuit*) that it's possible to write a book while raising a young family and holding down a job. I'm honored to be able to follow in your footsteps 30 years later—and I hope that we will be able to reunite in Heaven one day.

I also want to thank my professors at Columbia Business School—especially Mattan Griffel, Daniel Guetta, Hardeep Johar, and Jared Lander—for introducing me to the valuable role that Python and R can play within organizations. In addition, I'm grateful to my supervisors at Seton Education Partners and the Institute for Family Studies: David Morales, Michael Carbone, Michael Toscano, and Wendy Wang. You gave me the opportunity to apply Python to real-world challenges, which has allowed me to extend my study of programming far beyond the classroom.

This book would not have been possible without the developers of Python and the many, many libraries that it makes use of (particularly Pandas, Plotly, Folium, and Dash). Thank you for making your work accessible to the world!

I also wish to thank the developers of the open-source software and tools (including, but not limited to, JupyterLab Desktop, LibreOffice, Jupyter Book, Linux, and Linux Mint) that I used to develop Python for Nonprofits.

I'm also grateful to Bjarne Stroustrup, the creator of C++. His *Programming: Principles and Practice Using C++* set a high bar for teaching programming that this book benefited from, even if my effort comes nowhere close to his work. I extend the same gratitude to Allen B. Downey, the author of *Think Python*.

Finally, I want to thank you for reading (or at least flipping through) this book. I hope that, regardless of where you are in your Python and nonprofit career journey, it will be a helpful tool for you.

1.5 Dedication

This work is dedicated to my wife, Allie Burchfiel. Allie generously allowed me to take the time I needed to put this project together, even though that meant that I was less available to help her with chores, childcare, and the many other tasks (big and small) that go into running a household. She also believed in my vision for this work—and in my dreams of sharing my (still growing) understanding of Python with others. Allie is an amazing wife and mother, and I am incredibly fortunate to spend my life with her.

Blessed (and soon to be Saint) Carlo Acutis, pray for us!

1.5.1 Footnotes

*(I came across this quote within the '*Developers with AI assistants need to follow the pair programming model*' StackOverflow blog post, which is worth reading in its entirety. You can find it at <https://stackoverflow.blog/2024/04/03/developers-with-ai-assistants-need-to-follow-the-pair-programming-model/>.)

**I had the impression that pilots usually land their planes automatically, but that they still do so manually every 10th time or so in order to keep their skills fresh. However, Josh noted within a flyingbynumbers piece (<https://flyingbynumbers.com/pilots-land-planes-manually/>) that, in fact, pilots tend to land planes *manually* even though the technology for automated landings is available. He notes that pilot-led landings are typically smoother and that doing them yourself “is just plain fun.” These rationales may apply to ‘manual’ programming also.

THE CASE FOR USING PYTHON AT NONPROFITS

By Kenneth Burchfiel

Python is a great option for managing, analyzing, and visualizing nonprofit data for four reasons: it's *flexible*, *fast*, *free*, and *famous*. And you might even find it to be *fun* as well.

This programming language won't be the right fit for every nonprofit, and it does require more study and practice than many commercial solutions. However, I would suggest that nonprofits at least consider incorporating Python into their data workflows.

2.1 Python is *flexible*

Python's versatility is one of its key strengths. You can use it to read in data from numerous different sources; reformat and analyze that data; create visualizations; and even share your analyses online via an interactive dashboard. Other tools, like spreadsheet programs or business intelligence software, may specialize in one or more of these components, but Python is a fantastic choice for all of them.

Python not only supports many different use cases but also allows you to create scripts that are tailored to your nonprofit's specific needs. If you want a closed-source commercial software program to perform an additional task, you'd probably need to submit a feature request and then wait for the developers to incorporate it into the next version of their program—if they choose to do so at all. Meanwhile, if you want a Python script to perform a certain task, you can add in code for that task yourself (provided, of course, that you have the time and capabilities to do so).

This flexibility is greatly enhanced by the wide variety of libraries offered for Python. For instance, *Pandas* is a powerful tool for reading, reformatting, and analyzing data. *Plotly* allows you to easily create interactive charts and maps. And *Dash*, together with *Plotly*, can power interactive online dashboards in order to make your analyses and visualizations easier to share with others. (Each of these libraries, among others, will get featured in *Python for Nonprofits*.) These packages aren't part of the core Python programming language, but they are easy to install, and detailed documentation on each of them is accessible online.

One drawback of this flexibility, however, is that users of a Python script that another user wrote will need to take additional time to understand how it works—especially if it was not documented well. As long as someone has learned how to use a proprietary program, he or she should have little trouble making updates to a file that someone else created within that program, given that such software offers less room for variation. However, a Python expert will probably still need to take time to understand a script that another expert created, especially if it incorporates libraries or methods with which the expert isn't yet familiar.

2.2 Python is *fast*

Once you've created a Python script that completes a certain task, you should then be able to finish that task much more quickly the next time around than you could have within a spreadsheet editor or other tool. Therefore, the more you incorporate Python into your workflow, the more productive you and your team can be.

For example, let's say that your nonprofit sends out a survey to its service recipients each month, then compiles that data into a report for its board. It's true that pivoting and charting this data within a spreadsheet editor would likely be quicker than writing a set of Python code to perform these same steps. *However*, once you've created that Python code, you could then have it create your tables and charts in a matter of minutes or even seconds. (Or, if you configure your script to read in the latest survey data automatically and then instruct your computer to run it on a regular basis, you could produce your analyses without any keypresses at all.)

Similarly, suppose your supervisor requests that you use a new color scheme for 20 different charts that you created as part of a report. If you created these charts in Python and defined the color scheme at the beginning of your script, you could easily update all 20 charts by changing that scheme's definition and rerunning the script.

This speed advantage does have a few caveats. For instance, if a new set of data that you receive uses a different layout, different value types (e.g. '5%' instead of '0.05'), or different column names, you'll likely need to adjust your code to accommodate that new setup. However, Python also helps speed up data cleaning tasks, so if your nonprofit's data is imperfect (which I imagine to be the case just about anywhere), Python might become an even more useful tool as a result.

I also recognize that individuals with experience in Visual Basic for Applications (https://en.wikipedia.org/wiki/Visual_Basic_for_Applications) could likely create Excel macros that would provide similar speed advantages. However, I would suggest that the time spent learning VBA might be better spent on Python given the latter's versatility and popularity (more on this soon).

2.3 Python is *free*

The more companies rely on a SAAS (Software As A Service) model, in which you must provide ongoing payments to continue using a program, the more appealing Python and other free and open-source tools will appear as a result. This is the case for all organizations, but Python's cost advantage may be particularly compelling to nonprofits.

Suppose that you use a proprietary data visualization program that costs 200 dollars per user per year. If you have 300 staff members who use this software, your annual payments for it will amount to 60,000 dollars.

However, because Python is free to use and open source, you could dramatically reduce these costs by building a web dashboard using Python instead. You would probably need to pay for server and hosting costs, but you'd be able to avoid per-user fees and thus save significant amounts of money as a result.

Again, there are some caveats here. First, unless you build the web dashboard yourself (which would present opportunity costs), you'd need to pay someone to recreate it in Python; if your dashboard is particularly complex, these upfront costs may be too large for your nonprofit to bear. However, if you're already paying someone to manage your proprietary visualization program, having them rebuild your dashboard in Python instead *might* be a financially sound decision.

Second, not all versions of Python are free. For instance, if you use Anaconda for an organization with 200 or more employees, you may need to pay 50 dollars a month. (See Anaconda's Terms of Service (<https://legal.anaconda.com/policies/en?name=terms-of-service#terms-of-service>) for more details; the exact monthly cost is subject to change, if it hasn't changed already.) My understanding is that using Miniforge (<https://github.com/conda-forge/miniforge>) *should* allow you to legally avoid this monthly payment, but I can't guarantee that this is the case. (I do use Miniforge for my own personal and professional use and would recommend that you consider doing so as well.)

2.4 Python is *famous*

Python, according to the TIOBE Index (<https://www.tiobe.com/tiobe-index/>), is the world's most popular language. The rating of 23.85% that it received in March 2025 was over 9 times the rating of VBA, the language behind Excel macros that I mentioned earlier, and 25 times that of R (another commonly-used language for data science applications). As just one example of its broad adoption, over 3.2 million users have enrolled in The University of Michigan's 'Programming for Everybody' Python course (<https://www.coursera.org/learn/python>). Due to the language's popularity, I would expect that nonprofits can find a broad pool of prospective talent for Python-related work.

Python's wide usage has another important benefit: if you're facing an error message or a bug, it's likely that you'll find someone on Stack Overflow or GitHub who encountered a similar issue—and someone else who was able to resolve it for them.

2.5 Python is *fun* (to me, at least!)

Although your experience may vary, I have found great enjoyment in learning and applying Python. The mix of creativity, problem-solving, and real-world application that Python (like other programming languages) offers means that writing code often feels more like play than work. Granted, there have been plenty of frustrating moments along the way, but that frustration dissolves into gratification once a solution to the problem is found.

Although the first four of the 5 'Fs' that I've presented here might be more relevant to a nonprofit's management team or board, I'd argue that the 'fun' factor is one of the best reasons to study and apply Python. If you find working in Python to be entertaining and enjoyable, that's a great reason to continue building up your skills! Alternatively, if you come to loathe the programming experience, consider using another method to analyze data (or find someone else to take care of the Python work).

2.6 Don't take my word for it—give it a try yourself!

I recognize that this essay glosses over the complexity of choosing what tools to use to meet a given nonprofit's data needs. There are undoubtedly many cases when proprietary software would be a better fit than Python for a given organization. However, I'm also confident that Python is an underutilized tool within the nonprofit sector. I hope that Python for Nonprofits can demonstrate the utility of this language for your organization *and* give you the confidence you need to begin applying it.

GETTING STARTED

If you already have experience with Python (which I *do* recommend as a prerequisite for reading *Python for Nonprofits*), you shouldn't need to perform too much additional setup in order to begin going through PFN's chapters.

I am a big fan of print-based programming guides, so I do encourage you to read this book or PDF front to back if you prefer to learn that way. However, I also recommend that you download PFN's GitHub repository, available at <https://github.com/kburchfiel/pfn>. That way, you can run and modify the code contained within this book—and, of course, incorporate it into your own projects. (The repository also contains certain source data files that you'll need to correctly run certain scripts.) As you read through each section of PFN, you may also find it helpful to keep that section's corresponding Jupyter Notebook open on your computer.

Note: When running these notebooks on your own computer, you may want to confirm that the render_for_pdf setting within the Appendix/helper_funcs.py file is set to False. That way, many visualizations will appear in interactive HTML rather than static PNG format. (If render_for_pdf was initially set to True, you'll need to rerun each section after changing it to False in order for that change to take effect.)

An HTML-based version of this book is also available; you can access it by downloading the entire GitHub repository, then opening docs/index.html within the downloaded folder. I do plan to make this HTML-based book available online as well, but I'll first need to work out some bugs with the [online version](#).

In reading through PFN, you can of course choose to skip over sections with which you are already familiar or don't (yet) have a need for; however, keep in mind that later sections assume that you have already read through earlier ones.

Finally, the best way to internalize this content is to use it as the starting point for your own projects. In the future, I may also add suggested 'homework' that will make this practice a bit more formal—but for now, you'll need to 'BYOA' (bring your own assignments). One option would be to find a dataset related to a subject that interests you, then apply the contents of each section to clean, analyze, and visualize it.

3.1 Initial setup

I created much of PFN within Jupyter Desktop (<https://github.com/jupyterlab/jupyterlab-desktop>), an open-source tool for opening and viewing Jupyter Notebooks. I recommend that you use this program for viewing PFN notebooks, though PFN should run well within other Jupyter Notebook viewers as well.

You'll of course also want to have Python set up on your computer. As noted within *The Case for Python at Nonprofits*, I recommend that you install and use Python (and the libraries that PFN applies) via Miniforge rather than Anaconda, as you may not be able to use the latter for free.

Although most of the libraries that *Python for Nonprofits* uses are available within conda-forge (<https://github.com/conda-forge>), some may need to be installed via pip (<https://pypi.org/>).

3.2 Errata

When I become aware of errors within printed versions of the book, I will plan to make note of those errors within https://github.com/kburchfiel/pfn/tree/main/Book_Specific_Materials/book_errata.md.

3.3 Let's begin!

Now that these introductory matters are out of the way, you can start learning how to use Python in a nonprofit setting. We'll begin, as many analysis tasks do, with data retrieval.

Part II

Importing and Prepping Data

DATA RETRIEVAL

By Kenneth Burchfiel

Released under the MIT License

In order to begin analyzing data, we'll first need to load it into our program. This script will demonstrate how to import a variety of data formats into Python using **pandas**, a fantastic library for loading, reformatting, and analyzing data. Before I dive into the code, I'd like to provide a brief overview of Pandas for those who aren't yet familiar with this library.

4.1 A Pandas primer

You can think of Pandas as a tool for performing spreadsheet-based tasks within Python programs. One benefit of performing such tasks in Python (rather than Excel, Google Sheets, or another spreadsheet program) is that, once you have them scripted, you can quickly rerun them whenever the original data gets updated.* You could even have your computer run your analysis script on a daily or hourly basis, thus freeing up time you'd need to spend on busywork for more interesting tasks.

A central feature of Pandas is the *DataFrame*, which is essentially a within-Python spreadsheet that you can filter, modify, and extend as needed. You can import data into DataFrames (as this section will demonstrate); perform calculations on specific DataFrame columns; graph DataFrame data using libraries like Plotly; and then export DataFrames to a variety of formats. DataFrames and Jupyter Notebooks work very well together: by executing Pandas code within Jupyter Notebook cells, you can easily see how particular operations will modify your DataFrame.

If you need more information about a given Pandas function, make sure to consult the library's documentation (https://pandas.pydata.org/docs/user_guide/index.html); this is a well-written resource that I've relied on heavily in my own work. (In general, I recommend consulting official documentation (including the official Python documentation at <https://docs.python.org/>) whenever possible, though some libraries will have better documentation than others.)

Pandas will play a key or leading role in many sections of Python for Nonprofits. Although PFN will introduce you to many essential Pandas operations, you'll likely come across many more in your own Python programming journey.

* There are certainly ways to automate Excel tasks as well (e.g. using Visual Basic). I don't have any experience with Visual Basic, so I'm not the best person to compare these two tools; however, I have no doubt that learning it would take some time, and given Python's versatility and power, I would recommend applying that time to learning Python instead. (You can get an estimate of the world's interest in Python versus Visual Basic by checking out the [TIOBE index](#).)

4.2 Back to data retrieval

One of the key strengths of Pandas is its compatibility with a wide variety of data formats. This script will demonstrate how to use pandas to import data from .csv files, .xlsx files, SQL tables, and HTML pages; however, *many* other data types are supported, either by Python itself or via additional libraries. Later sections of Python for Nonprofits will introduce additional data sources.

```
import pandas as pd
from sqlalchemy import create_engine
# The above code can be found at
# https://docs.sqlalchemy.org/en/20/core/engines.html .

# The following two lines temporarily update your sys.path value
# to include the Appendix folder; that way, our import statement can
# access that folder in order to import its helper_funcs.py file.
# (This code was derived from Cameron's StackOverflow answer at
# https://stackoverflow.com/questions/4383571/
# importing-files-from-different-folder).
import sys
sys.path.insert(1, '../Appendix')

from helper_funcs import config_notebook # This function improves the
# appearance of PDF-based copies of notebooks, as it limits the number
# of rows and columns that can be displayed. It also allows us to display
# charts and maps in PNG rather than HTML format, thus making these
# graphics easier to view in PDF format.
# For more details on this function, consult the documentation
# for config_notebook within helper_funcs.py.

display_type = config_notebook(display_max_columns=7)
```

4.3 Importing .csv data

Pandas' `read_csv()` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html) function simplifies the process of reading .csv data into your Python script. The following cell provides an example of this function.

Note that the path argument begins with `.. /`. This instructs the code to move up one directory within the file system (to the root Python for Nonprofits folder), which in turn allows it to enter into the Appendix folder and retrieve the `curr_enrollment.csv` file stored there.

```
df_curr_enrollment_csv = pd.read_csv(
    '../Appendix/curr_enrollment.csv')
df_curr_enrollment_csv.head() # head() prints the first 5 rows of
# a DataFrame by default, though you can specify a larger or smaller number
# within the parentheses if needed. The last 5 rows can be viewed via
# .tail().
```

	first_name	last_name	gender	...	class_of	level	level_for_sorting
0	Amanda	Murphy	F	...	2024	Se	3
1	Terri	Washington	F	...	2024	Se	3
2	Crystal	Davis	F	...	2024	Se	3
3	Theresa	Joseph	F	...	2024	Se	3
4	Chelsea	Martin	F	...	2024	Se	3

(continues on next page)

(continued from previous page)

```
[5 rows x 11 columns]
```

However, you'll sometimes need to add additional arguments to `read_csv()` in order to correctly import your data. For example, if a separator other than a comma was used, you'll want to specify that separator via the 'sep' argument.

The following example shows what you'll see if you try to use `read_csv()` to import *tab*-separated .csv data:

```
df_curr_enrollment_tab_csv = pd.read_csv(
    'curr_enrollment_tab_separated.csv')
df_curr_enrollment_tab_csv.head()
```

```
first_name\tlast_name\tgender\tmatriculation_year\tmatriculation_number\tstudent_
id\tdate_of_birth\tcollege\tclass_of\tlevel\tlevel_for_sorting
0 Amanda\tMurphy\tF\t2020\t1\t2020-1\t2002-12-16...
1 Terri\tWashington\tF\t2020\t2\t2020-2\t2002-09...
2 Crystal\tDavis\tF\t2020\t3\t2020-3\t2002-05-18...
3 Theresa\tJoseph\tF\t2020\t4\t2020-4\t2002-11-1...
4 Chelsea\tMartin\tF\t2020\t5\t2020-5\t2002-06-2...
```

The '\t' strings within the column and field values *and* the lack of column separators are dead giveaways that this file used tab separators rather than commas. To correctly import this information, you'll need to add `sep='\t'` within your `read_csv()` call, as shown below. ('\t' represents tabs, just as '\n' represents newlines.)

```
df_curr_enrollment_tab_csv = pd.read_csv(
    'curr_enrollment_tab_separated.csv', sep='\t')
df_curr_enrollment_tab_csv.head()
```

	first_name	last_name	gender	...	class_of	level	level_for_sorting
0	Amanda	Murphy	F	...	2024	Se	3
1	Terri	Washington	F	...	2024	Se	3
2	Crystal	Davis	F	...	2024	Se	3
3	Theresa	Joseph	F	...	2024	Se	3
4	Chelsea	Martin	F	...	2024	Se	3

```
[5 rows x 11 columns]
```

In addition, if your .csv file uses an encoding other than UTF-8 (the default), you may need to specify an alternative codec (<https://docs.python.org/3/library/codecs.html#standard-encodings>) using the 'encoding' argument.

`read_csv()` can also be used for .csv files hosted online; you'll simply substitute a URL in your function call for your local file path. For instance, the following cell reads in a .csv file from GitHub that will be processed by a later section of Python for Nonprofits:

```
df_web_csv_example = pd.read_csv(
    'https://raw.githubusercontent.com/kburchfiel/pfn/
refs/heads/main/Data_Prep/winter_results/STB_So_results.csv')
df_web_csv_example
```

	MATRIC#	SEASON	SURVEY_SCORE	STARTINGYR	MATRICYR
0	2875	W	74.0%	23	22
1	2018	W	52.0%	23	22
...

(continues on next page)

(continued from previous page)

```
959      2194      W      56.0%      23      22
960      3804      W      77.0%      23      22
[961 rows x 5 columns]
```

While this is a handy feature to have, if you're dealing with a large and unchanging dataset, you may want to consider downloading it to your computer beforehand—as reading in the file from your local hard drive will likely be faster than retrieving it via your internet connection.

4.4 Importing .xlsx data

Importing .xlsx files is also easy to do within Python, although I've found that this process can take longer to execute than does importing .csv files. (Therefore, I tend to prefer .csv files over .xlsx ones when importing data into Python.)

The following code imports an .xlsx version of the same current enrollment dataset that we imported earlier. You may need to install the openpyxl library in order for this code to run on your computer.

```
df_curr_enrollment_xlsx = pd.read_excel('curr_enrollment.xlsx')
df_curr_enrollment_xlsx.head()
```

```
first_name    last_name gender ... class_of level level_for_sorting
0     Amanda      Murphy     F ... 2024    Se          3
1      Terri  Washington     F ... 2024    Se          3
2   Crystal       Davis     F ... 2024    Se          3
3  Theresa      Joseph     F ... 2024    Se          3
4   Chelsea      Martin     F ... 2024    Se          3
[5 rows x 11 columns]
```

This cell took 1.38 seconds to execute on my computer, whereas the first .csv import cell ran in just 21 miliseconds (or 0.021 seconds). In other words, the .csv import was around 65 times faster than the .xlsx import.

4.5 Importing SQL data

Python's SQLAlchemy and Pandas libraries make it easy to import SQL tables into your script. Many different types of SQL (such as PostgreSQL) are supported, but this example will focus on a SQLite table created within PFN's appendix.

In order to import data from a database, we'll first need to connect to it via SQLAlchemy's `create_engine` function:

```
e = create_engine('sqlite:///..../Appendix/nvcu_db.db')
# Based on:
# https://docs.sqlalchemy.org/en/20/dialects/sqlite.html#pysqlite
```

Note that the first 3 forward slashes indicate that a relative path will be used. The actual relative path ('..../Appendix/nvcu_db.db') follows those forward slashes.

For guidance on creating engines for other database types (such as PostgreSQL, MySQL, and others), visit <https://docs.sqlalchemy.org/en/20/core/engine.html#database-urls>.

Once this SQLAlchemy engine has been created, we can use it to read in data from our database:

```
df_curr_enrollment_sql = pd.read_sql(
    'select * from curr_enrollment', con=e)
df_curr_enrollment_sql.head()
```

	first_name	last_name	gender	...	class_of	level	level_for_sorting
0	Amanda	Murphy	F	...	2024	Se	3
1	Terri	Washington	F	...	2024	Se	3
2	Crystal	Davis	F	...	2024	Se	3
3	Theresa	Joseph	F	...	2024	Se	3
4	Chelsea	Martin	F	...	2024	Se	3

[5 rows x 11 columns]

SQL is a whole language in itself, but you need not be a SQL expert to use Python to connect to database tables. In the above cell, `'select * from curr_enrollment'` is a line of SQL code that requests that all fields (and, thus, all data) be retrieved from the `curr_enrollment` SQLite table.

SQL makes it possible to select specific columns, choose only particular groups of rows, and perform other more advanced operations. However, many of these operations can also be performed within Python. If you're already a SQL whiz, feel free to pass more advanced code to `read_sql()`; if you're a SQL novice, `'select * from [table]'` will still get you pretty far!

Also note that a SQLAlchemy engine can be used as the 'con' argument within both `read_sql()` and `to_sql()`. This is mentioned explicitly within Pandas' `to_sql()` documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html) but can also be inferred from pandas' `read_sql()` page (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql.html). This page states that 'con' needs to be a 'SQLAlchemy connectable,' and the source code for `sql.py()` (<https://github.com/pandas-dev/pandas/blob/v2.2.2/pandas/io/sql.py#L570-L743>) specifies that 'SQLAlchemy connectable' can be either an engine or a connection. I mention this in part because using an engine as your argument for the 'con' parameter in `read_sql()` and `to_sql()` can save you a bit of code.

4.6 Importing HTML data

There are a number of ways to access data directly from the internet via Python. One means of doing so is `pd.read_html()`, which lets you read HTML tables from websites directly into DataFrames.

The following code imports a list of sample Census API connection strings into a DataFrame. The `[0]` at the end of the `read_html()` call instructs Pandas to convert the *first* table retrieved by `read_html()` into a DataFrame. (There's only one HTML table on this page, but it's still necessary to add `[0]`.)

Note that the import isn't perfect: for some reason, the data on the 'Number' field on the right didn't get downloaded successfully. As noted in the `read_html()` documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html), "Expect to do some cleanup after you call this function."

```
df_census_examples = pd.read_html(
    'https://api.census.gov/data/2022/acs/acs5/examples.html')[0]
df_census_examples.head()
```

	Geography	Hierarchy	Geography	Level	\
0		us		010	
1		us		010	
2		region		020	
3		region		020	

(continues on next page)

(continued from previous page)

	division	030	Example URL	Number
0	https://api.census.gov/data/2022/acs/acs5?get=...			NaN
1	https://api.census.gov/data/2022/acs/acs5?get=...			NaN
2	https://api.census.gov/data/2022/acs/acs5?get=...			NaN
3	https://api.census.gov/data/2022/acs/acs5?get=...			NaN
4	https://api.census.gov/data/2022/acs/acs5?get=...			NaN

4.7 What about exports?

The above examples focused on *importing* data; however, Pandas also makes it easy to *export* data to a variety of formats.

4.7.1 Exporting data to a .csv file:

```
df_curr_enrollment_csv.to_csv(  
    'curr_enrollment_export.csv', index=False)
```

`index=False` specifies that we do not want to include the DataFrame's index within our .csv file. This is the best option in this case, since the DataFrame's index doesn't contain any meaningful data—and it will get recreated anyway when we import the .csv file back into Python.

As the following two cells demonstrate, If we leave this component out, then attempt to import the DataFrame, we'll end up with an ugly Unnamed: 0 column:

```
df_curr_enrollment_csv.iloc[0:5].to_csv(  
    'curr_enrollment_export_with_index.csv') # Using .iloc to export  
# only the first 5 rows will reduce the size of this file within  
# our project folder.  
  
pd.read_csv('curr_enrollment_export_with_index.csv')
```

```
Unnamed: 0 first_name last_name ... class_of level level_for_sorting  
0 0 Amanda Murphy ... 2024 Se 3  
1 1 Terri Washington ... 2024 Se 3  
2 2 Crystal Davis ... 2024 Se 3  
3 3 Theresa Joseph ... 2024 Se 3  
4 4 Chelsea Martin ... 2024 Se 3  
  
[5 rows x 12 columns]
```

4.7.2 Exporting data to an .xlsx file:

```
df_curr_enrollment_xlsx.to_excel(  
    'curr_enrollment_export.xlsx', index=False)
```

On my computer, this .xlsx export took 52 times as long as the .csv export (1.36 seconds vs. 26 milliseconds). Therefore, as noted earlier, you may want to use .csv files in place of .xlsx files when working in Python.

For examples of exporting data to SQL tables via `to_sql()`, reference the `nvcu_db_gen.ipynb` file within the Appendix.

4.7.3 A caveat about data types

You'll often find that using `read_csv()`, `read_excel()`, and `read_sql()` will create identical DataFrames as long as the source data is the same. However, in some cases, the data types returned by these functions will differ—requiring you to add in some additional code to resolve this discrepancy.

For instance, if you retrieve a dataset from a SQL table, dates of birth might be formatted as datetime values. Meanwhile, those same values may be formatted as strings if they were imported from a .csv file. Therefore, if you're switching your import source from a SQL table to a .csv file or vice versa, it's not a bad idea to check the data types of your imported fields using `df.dtypes` (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dtypes.html>).

4.8 Conclusion

As this notebook demonstrates, Pandas tends to make data retrieval very simple. Cleaning and reformatting data, the subject of the following chapter, isn't always so simple—but Pandas will play a very valuable role with this process as well.

For a more comprehensive listing of the many data import (and export) tasks that Pandas can perform, visit <https://pandas.pydata.org/pandas-docs/stable/reference/io.html>.

DATA PREP

By Kenneth Burchfiel

Released under the MIT License

Before a dataset can be analyzed and visualized within Python, it often needs to be reformatted and cleaned. In order to demonstrate how Python can easily perform these reformatting and cleaning tasks, this script will clean and reformat a set of NVCU winter student survey results; combine those results with fall and spring survey data; and merge in information from our current enrollment table.

Our survey_results database file (which we'll import from PFN's Appendix folder) already contains student survey responses for the fall and spring. However, let's say that you've been asked to add a set of *winter* results to this dataset as well, then calculate a weighted average of fall, winter, and spring survey results for each student.

If these results were in the same format as the fall and spring ones and had no missing data, this process would be very simple. Unfortunately, that's not the case with the winter results that we'll be processing within this script. These results feature:

1. Column names that differ from those in the fall/spring results
2. Different data formats
3. A missing column
4. Duplicate values
5. Missing values for certain students

In addition, to make matters even more complex, these winter results are spread out over 16 different files (one for each level within each college).

It would be cumbersome and mind-numbing to modify each of these 16 datasets within Excel, Google Sheets, or a similar program so that they could be combined with our pre-existing fall and spring data. However, the Python code shown below will make this data cleaning process much easier. And once this script is in place, if you happened to get next year's winter results in the same format* as this year's, you'd be able to get them cleaned up and reformatted in no time.

**You may find in your work, however, that the results are in yet another format the following year, followed by a different format the year after that. Data-related tasks are always made easier when inputs stay the same, but in the real world, you'll often need to rework datasets in order to make them compatible with pre-existing processes.*

```
import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook
display_type = config_notebook(display_max_columns = 7)

import os
import pandas as pd
from sqlalchemy import create_engine
```

Our first step in preparing our winter survey results will be to import the 16 files that comprise them into a DataFrame. We'll start this process by calling `os.listdir()` to create a list of all files within our `winter_results` folder:

```
file_list = os.listdir('winter_results')
file_list[0:5]
```

```
['STM_Ju_results.csv',
 'STB_Se_results.csv',
 'STM_Se_results.csv',
 'STC_So_results.csv',
 'STB_Ju_results.csv']
```

Note that each file represents a single college/level pair.

(*Note: as an American university, NVCU has four 'levels': freshmen, sophomore, junior, and senior. These four terms (which will often be truncated to 'Fr', 'So', 'Ju', and 'Se' within our datasets and tables) correspond to the first, second, third, and fourth years of enrollment at the university.*

'Year' is a more common term than 'level' to refer to these four names, but it could easily get confused with school year values in our datasets. Therefore, I'll normally use 'level' to represent these terms within PFN.)

Next, we'll use a for loop to (1) read each file within this list into a DataFrame, then (2) add those files to a list of DataFrames. We'll then apply `pd.concat()` to combine these file-specific DataFrames into a single DataFrame.

```
df_list = []
for file in file_list:
    df = pd.read_csv(f'winter_results/{file}')
    df_list.append(df)

# Combining these individual DataFrames into a single file:
df_winter_results = pd.concat(
    [df for df in df_list]) # df for df in df_list is a list comprehension
# that contains all DataFrames in df_list.
# Replacing the original indices with a new, unified index:
df_winter_results.reset_index(drop=True, inplace=True)
df_winter_results.head()
```

	MATRIC#	SEASON	SURVEY_SCORE	STARTINGYR	MATRICYR
0	2689	W	58.0%	23	21
1	72	W	54.0%	23	21
2	1612	W	82.0%	23	21
3	2404	W	76.0%	23	21
4	1552	W	56.0%	23	21

The following cell shows a more concise means of creating the same DataFrame. Although this approach requires fewer lines of code, it's also less flexible (as the former method allows you to make individual updates to each DataFrame if needed).

```
df_winter_results = pd.concat(
    [pd.read_csv(f'winter_results/{file}')
     for file in os.listdir('winter_results')]).reset_index(drop=True)
df_winter_results.head()
```

	MATRIC#	SEASON	SURVEY_SCORE	STARTINGYR	MATRICYR
0	2689	W	58.0%	23	21

(continues on next page)

(continued from previous page)

1	72	W	54.0%	23	21
2	1612	W	82.0%	23	21
3	2404	W	76.0%	23	21
4	1552	W	56.0%	23	21

5.1 Reformatting and cleaning our winter results dataset

Our next step is to combine these winter survey results with the fall and spring results in our NVCU database. Here's what those results look like:

```
# Connecting to our database:
e = create_engine('sqlite:///../Appendix/nvcu_db.db')
df_fall_spring_results = pd.read_sql(
    "Select * from survey_results", con=e)
df_fall_spring_results.head()
```

	student_id	starting_year	season	score
0	2020-1	2023	Fall	88
1	2020-2	2023	Fall	37
2	2020-3	2023	Fall	54
3	2020-4	2023	Fall	56
4	2020-5	2023	Fall	77

If we naively tried to add our winter results to our fall/spring results, we'd end up with a very messy DataFrame with numerous blank (i.e. NaN) cells:

```
df_messy_combo = pd.concat([df_fall_spring_results,
                            df_winter_results])
```

Compare the beginning of this DataFrame (which has fall/spring results) with the end (which has the new winter results we just imported):

```
df_messy_combo.head()
```

	student_id	starting_year	season	...	SURVEY_SCORE	STARTINGYR	MATRICYR
0	2020-1	2023.0	Fall	...	NaN	NaN	NaN
1	2020-2	2023.0	Fall	...	NaN	NaN	NaN
2	2020-3	2023.0	Fall	...	NaN	NaN	NaN
3	2020-4	2023.0	Fall	...	NaN	NaN	NaN
4	2020-5	2023.0	Fall	...	NaN	NaN	NaN

[5 rows x 9 columns]

```
df_messy_combo.tail()
```

	student_id	starting_year	season	...	SURVEY_SCORE	STARTINGYR	MATRICYR
14617	NaN	NaN	NaN	NaN	65.0%	23.0	23.0
14618	NaN	NaN	NaN	NaN	79.0%	23.0	23.0
14619	NaN	NaN	NaN	NaN	51.0%	23.0	23.0

(continues on next page)

(continued from previous page)

14620	NaN	NaN	NaN	...	62.0%	23.0	23.0
14621	NaN	NaN	NaN	...	57.0%	23.0	23.0
[5 rows x 9 columns]							

This unkempt output is caused by discrepancies in column names between the two tables. To rectify this issue, we'll need to rename our winter results fields to match their corresponding fields within the fall/spring table. Thankfully, Pandas makes this process very straightforward:

```
df_winter_results.rename(columns = {
    'SEASON':'season', 'STARTINGYR':'starting_year',
    'SURVEY_SCORE':'score'}, inplace=True)
df_winter_results.head()
```

	MATRIC#	season	score	starting_year	MATRICYR
0	2689	W	58.0%	23	21
1	72	W	54.0%	23	21
2	1612	W	82.0%	23	21
3	2404	W	76.0%	23	21
4	1552	W	56.0%	23	21

We'll also need to convert our MATRIC# and MATRICYR fields into a single student_id field. (This student_id field simply combines students' matriculation years with their matriculation numbers; see nvcu_db_gen.ipynb within the Appendix for more details.) This can be done as follows:

```
df_winter_results['MATRICYR'] += 2000 # Converts our MATRICYR
# values from YY to YYYY format so that they'll match the format of the
# matriculation year component of the student_id values within
# df_fall_spring_results

# Converting students' MATRICYR and MATRIC# values into student IDs (which
# are based on these two values):
# (Note that both columns must be converted to strings in order for
# this code to work. If we just added the two integers together, we'd
# get their sum--which isn't what we want in this case.)
df_winter_results['student_id'] = (
    df_winter_results['MATRICYR'].astype('str')
    + '-'
    + df_winter_results['MATRIC#'].astype('str'))

# Now that we've used our MATRICYR and MATRIC# columns to create
# our student IDs, we no longer need to retain them:
df_winter_results.drop(
    ['MATRICYR', 'MATRIC#'],
    axis=1, inplace=True)
df_winter_results.head()
```

	season	score	starting_year	student_id
0	W	58.0%	23	2021-2689
1	W	54.0%	23	2021-72
2	W	82.0%	23	2021-1612
3	W	76.0%	23	2021-2404
4	W	56.0%	23	2021-1552

The columns in `df_winter_results` now match those within `df_fall_spring_results`. That's great! Let's try combining the two datasets to see if we're ready to calculate average survey scores for each season:

```
df_results = pd.concat([df_fall_spring_results,
                       df_winter_results])
df_results.head()
```

	student_id	starting_year	season	score
0	2020-1	2023	Fall	88
1	2020-2	2023	Fall	37
2	2020-3	2023	Fall	54
3	2020-4	2023	Fall	56
4	2020-5	2023	Fall	77

Note the differences between the previous and following outputs (which show fall and winter data, respectively):

```
df_winter_results.tail()
```

	season	score	starting_year	student_id
14617	W	65.0%	23	2023-376
14618	W	79.0%	23	2023-1699
14619	W	51.0%	23	2023-3091
14620	W	62.0%	23	2023-999
14621	W	57.0%	23	2023-2736

These excerpts show that, unfortunately, we're not quite ready to analyze this data just yet: there are several formatting differences that we'll need to address.

For instance, the 'score' column within `df_fall_spring_results` uses an integer format, whereas these same numbers are formatted as strings within `df_winter_results`. This will produce errors when we attempt to perform numerical calculations on this field:

```
# df_results['score'].mean()
```

This line was commented out because it would produce the following `TypeError`:

```
unsupported operand type(s) for +: 'int' and 'str'
```

The following cell resolves this issue by converting our string-formatted score values to integers:

```
df_winter_results['score'] = df_winter_results[
    'score'].str.replace('.0%', '').astype('int') # Chaining these
# column-specific operations together helps reduce the number of
# times we need to write the column name--which can help prevent
# errors related to incorrect field names.
df_winter_results.head()
```

	season	score	starting_year	student_id
0	W	58	23	2021-2689
1	W	54	23	2021-72
2	W	82	23	2021-1612
3	W	76	23	2021-2404
4	W	56	23	2021-1552

We'll also need to reformat our winter results' `season` and `starting_year` values so that they match the formats found in the fall/spring table.

The following cell replaces the ‘W’ values within the ‘season’ column with ‘Winter’ so that they’ll better match the ‘Fall’ and ‘Spring’ values in `df_fall_spring_results`:

```
df_winter_results['season'] = (
    df_winter_results['season'].replace({'W':'Winter'}))
df_winter_results.head()
```

	season	score	starting_year	student_id
0	Winter	58	23	2021-2689
1	Winter	54	23	2021-72
2	Winter	82	23	2021-1612
3	Winter	76	23	2021-2404
4	Winter	56	23	2021-1552

The following code would also have worked; however, it assumes that every row within the DataFrame is indeed a winter result. This is the case in our simulated data, but in the real world, some data from other seasons might have leaked in, causing this code to incorrectly reclassify certain results.

```
# df_winter_results['season'] = 'Winter'
```

Finally, we’ll add 2000 to every `starting_year` value so that our years will show up within YYYY format—just as they do within our fall/spring dataset.

```
df_winter_results['starting_year'] += 2000
df_winter_results.head()
```

	season	score	starting_year	student_id
0	Winter	58	2023	2021-2689
1	Winter	54	2023	2021-72
2	Winter	82	2023	2021-1612
3	Winter	76	2023	2021-2404
4	Winter	56	2023	2021-1552

5.2 Removing duplicates

We’ve now successfully made our winter dataset’s field names and values compatible with those in our fall/spring dataset. However, before we can combine the two together, we’ll need to remove some duplicate results. (The NVCU administration has specified that they want only *one* survey score per student/season pair to get retained; otherwise, students who fill out a given survey multiple times will get weighted more often than those who complete it only once.)

Let’s first identify these duplicate rows. The following code filters `df_winter_results` to include any rows whose `season`, `starting_year`, and `student_id` columns match. (The inclusion of `keep=False` instructs Pandas to return all copies of a duplicated row, not just the first one that it encounters.)

```
df_winter_results[df_winter_results.duplicated(
    subset = ['season', 'starting_year', 'student_id'],
    keep=False)]
```

	season	score	starting_year	student_id
13	Winter	54	2023	2021-3597
22	Winter	73	2023	2021-2260

(continues on next page)

(continued from previous page)

```

...
...
...
14620  Winter      62          2023  2023-999
14621  Winter      57          2023  2023-2736

[1392 rows x 4 columns]

```

These duplicate values can easily be removed using Pandas' `drop_duplicates()` function. However, before removing duplicate rows, it's a good idea to first consider which of these duplicated entries to retain. (For instance, you might choose to keep the earliest copy of a set of duplicated surveys; the latest copy; or the one with the highest/lowest score.)

In our case, we'll keep the duplicated row with the *highest* score and remove all others. We can do this by (1) sorting our DataFrame to show higher scores before lower ones, then (2) keeping the first row (i.e. the one with the highest score) when removing our duplicates.

```

df_winter_results.sort_values(
    'score', ascending=False, inplace=True)
df_winter_results.head()

```

	season	score	starting_year	student_id
2358	Winter	99	2023	2020-616
1545	Winter	98	2023	2020-35
757	Winter	97	2023	2021-468
7718	Winter	97	2023	2020-395
8457	Winter	97	2023	2020-816

Removing duplicate values:

Note: when removing duplicates, think carefully about which columns to include in your `subset` argument. For instance, if we had multiple years' worth of data in our table, using `['season', 'student_id']` as your subset would cause *only one* result for each student/season pair to get retained, thus removing valid data for other years from your table.

```

df_winter_results.drop_duplicates(
    subset = ['season', 'starting_year', 'student_id'],
    keep = 'first', inplace = True)
df_winter_results.head()

```

	season	score	starting_year	student_id
2358	Winter	99	2023	2020-616
1545	Winter	98	2023	2020-35
757	Winter	97	2023	2021-468
7718	Winter	97	2023	2020-395
8457	Winter	97	2023	2020-816

Rerunning our duplicate check code confirms that no duplicate entries remain within our dataset:

```

df_duplicated_winter_results = (
    df_winter_results[df_winter_results.duplicated(
        subset = ['season', 'starting_year', 'student_id'],
        keep = False)])
len(df_duplicated_winter_results)

```

0

Since we may need to run this code again in the future with new sets of winter data, we should include a block of code at this point that will check whether we have any duplicates, then raise an error if so. (This will prevent the script from continuing on and possibly generating incorrect findings.)

```
if len(df_duplicated_winter_results) > 0:  
    raise ValueError("Duplicate winter survey results are still present!")  
else:  
    print("No duplicate entries are present within the winter dataset.")
```

```
No duplicate entries are present within the winter dataset.
```

5.3 Combining winter survey results with our fall/spring dataset

We're now finally ready to combine df_winter_results with df_fall_spring results. However, one final issue remains with this table: winter survey results are missing for a number of students. This won't cause any issues with the following code, but we'll need to take these missing entries into account when analyzing our survey data within the upcoming Descriptive Stats chapter.

```
df_results = pd.concat(  
    [df_fall_spring_results,  
     df_winter_results]).sort_values(  
    ['starting_year', 'season']).reset_index(drop=True)  
df_results.head()
```

	student_id	starting_year	season	score
0	2020-1	2023	Fall	88
1	2020-2	2023	Fall	37
2	2020-3	2023	Fall	54
3	2020-4	2023	Fall	56
4	2020-5	2023	Fall	77

The following output confirms that the format of the winter results within this table matches that of the fall/spring rows:

```
df_results.query("season == 'Winter'").head()
```

	student_id	starting_year	season	score
32768	2020-616	2023	Winter	99
32769	2020-35	2023	Winter	98
32770	2021-468	2023	Winter	97
32771	2020-395	2023	Winter	97
32772	2020-816	2023	Winter	97

Note that, when our seasons are sorted alphabetically, Fall will come first, followed by Spring and then Winter. In order to allow for a chronological sort (which will prove useful when creating charts and pivot tables), we'll add in a 'season_order' column that maps these seasons to integers.

```
df_results['season_order'] = df_results['season'].map(  
    {'Fall':0,'Winter':1,'Spring':2})  
df_results.head()
```

	student_id	starting_year	season	score	season_order
0	2020-1	2023	Fall	88	0
1	2020-2	2023	Fall	37	0
2	2020-3	2023	Fall	54	0
3	2020-4	2023	Fall	56	0
4	2020-5	2023	Fall	77	0

5.4 Merging college and level data into our survey results table

In its current form, `df_results` allows us to calculate survey results at the university-wide level. However, in order to determine whether these results differ by college and by level, we'll also need to merge college and level data into our dataset.

We can accomplish this by reading data from our `curr_enrollment` SQL table into a DataFrame, then merging that data with `df_results`. (The `student_id` field present in both DataFrames can serve as a merge key.)

Note: if we had survey data for multiple years, our current enrollment table wouldn't be a good candidate for our merge, since it would show only the current college and levels for each student. We'd instead want to source our enrollment data from a historical enrollment table so that students' yearly survey results could be matched with their levels and colleges during the years that they took the survey.

```
# The following code reads in only the fields from df_curr_enrollment
# that we'll need for our analyses. (If we wanted to read in all fields,
# we could replace the field names in our SQL query with *.)

df_curr_enrollment = pd.read_sql(
    "Select student_id, college, level, \
    level_for_sorting from curr_enrollment", con=e)
df_curr_enrollment.head()
```

	student_id	college	level	level_for_sorting
0	2020-1	STC	Se	3
1	2020-2	STM	Se	3
2	2020-3	STC	Se	3
3	2020-4	STC	Se	3
4	2020-5	STM	Se	3

The next cell uses a `left` merge to add enrollment data into `df_results`. The `left` argument for the `how` parameter instructs the code to retain all rows in `df_results` even if no corresponding enrollment data were found. (If we had instead used `right` as our argument, all rows in `df_curr_enrollment` would have been retained whether or not we had corresponding `survey` data for them.)

Other options for the `how` parameter include `outer`, which preserves all rows in *both* datasets regardless of whether or not a given key was found in both of them, and `inner`, which would only keep rows whose student IDs were found in both datasets.

For more on the `df.merge()` function used below, visit <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html>.

```
df_results = df_results.merge(
    df_curr_enrollment, on='student_id', how='left')
df_results.head()
```

```
student_id    starting_year   season   ...   college   level   level_for_sorting
0      2020-1           2023     Fall   ...     STC     Se          3
1      2020-2           2023     Fall   ...     STM     Se          3
2      2020-3           2023     Fall   ...     STC     Se          3
3      2020-4           2023     Fall   ...     STC     Se          3
4      2020-5           2023     Fall   ...     STM     Se          3

[5 rows x 8 columns]
```

Now that we've merged in college and level data, we'll save this dataset to a .csv file so that it can be processed by the code in our descriptive statistics chapters:

```
df_results.to_csv('2023_survey_results.csv', index = False)
```

5.5 Conclusion

This script has provided a brief introduction to data cleaning and reformatting. Other PFN sections will provide further examples of data reformatting, as this process is often a necessary prerequisite for analysis and visualization tasks.

Next up is the Descriptive Stats chapter, in which we'll perform various statistical tests on the new .csv file that we created within this section.

Part III

Analyzing Data

DESCRIPTIVE STATS

Now that we've learned how to retrieve, reformat, and clean data, we can finally begin analyzing it! This notebook demonstrates how to calculate descriptive statistics in Python using Pandas.

Suppose leaders at NVCU would like to know, on a *daily* basis, how incoming spring survey results differ from fall and winter ones. (These metrics could change each day as new spring survey data gets released, and the administration does not wish to wait until all results are in before they begin reviewing the numbers.)

One way to accomplish this task would be to manually import survey data from your database each day; paste it into Excel or Google Sheets; pivot the data, and then share the output. However, you could also accomplish these same steps in Python. While this would likely take you longer the first time around, you could then create updated analyses of your data in mere seconds. This notebook will show you how!

```
import time
start_time = time.time()
import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook
display_type = config_notebook(display_max_columns = 8,
                                 display_max_rows = 16)

import pandas as pd
import numpy as np
from sqlalchemy import create_engine
```

We'll first import our combined set of fall, spring, and winter student survey results; these results were created within `data_prep.ipynb`. (Note that these results also include college and level data that we merged in from NVCU's 'curr_enrollment' SQL table; that way, we can evaluate average results by level and college.)

```
df_survey_results = pd.read_csv('../Data_Prep/2023_survey_results.csv')
# Removing a few non-essential columns from the display so that the table
# will show up better in PDF format:
df_survey_results.drop(['season_order', 'level_for_sorting'],
                      axis = 1).head(5)
```

	student_id	starting_year	season	score	college	level
0	2020-1	2023	Fall	88	STC	Se
1	2020-2	2023	Fall	37	STM	Se
2	2020-3	2023	Fall	54	STC	Se
3	2020-4	2023	Fall	56	STC	Se
4	2020-5	2023	Fall	77	STM	Se

6.1 Evaluating changes in average university-wide results during the school year

Our dataset contains fall, winter, and spring survey results. In order to determine how the mean survey score has changed over the course of the year, we can use Pandas' `pivot_table()` function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot_table.html). I consider `pivot_table()` to be one of the most useful functions in the Pandas library, and you'll see it used again and again with PFN.

The `pivot_table()` call below introduces three key arguments:

`index`: the list of values by which to group results. Although our dataset only contains data for one year, we'll still include `starting_year` in our results in order to (1) allow the function to accommodate other school years and (2) demonstrate to the viewer that all of this data comes from 2023. We'll also add both `season_order` and `season` to our list (in that order) so as to display results by season in chronological order. (Without the `season_order` argument, our seasons would be sorted alphabetically, with spring results preceding winter ones.)

`values`: the metric(s) to assess. We're interested in analyzing changes in average score by year, so we'll pass `score` as our argument. Multiple values can be passed to this argument if needed.

`aggfunc`: the aggregate function to apply to our list of values. We'll use `mean` here, but we could also have chosen `median` as a measure of the average.

I generally like to add `reset_index()` to the result of `pivot_table()` in order to remove any blank index values.

```
df_results_by_season = df_survey_results.pivot_table(
    index=['starting_year', 'season_order', 'season'],
    values='score', aggfunc='mean').reset_index()
df_results_by_season
```

	starting_year	season_order	season	score
0	2023	0	Fall	69.682251
1	2023	1	Winter	64.216214
2	2023	2	Spring	72.049622

These results show that the average score fell around 5 points from the fall to the winter, then increased nearly 8 points from the winter to the spring.

Here's what the output looks like without the trailing `reset_index()` call:

```
df_survey_results.pivot_table(
    index=['starting_year', 'season_order', 'season'],
    values='score', aggfunc='mean')
```

	starting_year	season_order	season	score
2023	0	Fall	69.682251	
	1	Winter	64.216214	
	2	Spring	72.049622	

We can also find the average score across seasons by setting `margins` to True. The `margins_name` argument lets us assign a name to this row; if we leave it blank, the row will be titled 'All.'

```
df_survey_results.pivot_table(
    index = ['starting_year', 'season_order', 'season'],
```

(continues on next page)

(continued from previous page)

```
values = 'score', aggfunc = 'mean', margins=True,
margins_name='2023 Average').reset_index()
```

	starting_year	season_order	season	score
0	2023	0	Fall	69.682251
1	2023	1	Winter	64.216214
2	2023	2	Spring	72.049622
3	2023 Average			68.882726

6.2 Calculating response rates

The `pivot_table()` function will also let us determine survey response rates as a percentage of our current enrollment. We can import this current enrollment value from our NVCU database:

```
e = create_engine('sqlite:///../Appendix/nvcu_db.db')
```

Calculating our current enrollment by counting the number of rows in our `curr_enrollment` table:

```
enrollment_count = len(pd.read_sql(
    "Select * from curr_enrollment", con=e))
enrollment_count
```

```
16384
```

A faster way of computing this number is to request it within the original SQL query via `COUNT(*)`. The following line, which demonstrates this approach, took only 6 milliseconds to run on my computer—one tenth the duration of the previous line (which took 62 milliseconds). If we were dealing with millions of rows instead of thousands, this performance difference would probably be even greater.

```
enrollment_count = pd.read_sql(
    "Select COUNT(*) from curr_enrollment", con=e).iloc[0]['COUNT(*)']
enrollment_count
```

```
np.int64(16384)
```

Now that we have the denominator* for our response rate calculations, let's go ahead and calculate the numerator (e.g. the number of responses per season). For teaching purposes, the following cell will also show how to use a dictionary to specify separate `aggfunc` values for different value entries. In this case, we'll pass `mean` as our aggregate function for the `score` column, but `count` as our aggregate function for the `responses` field.

Note: When calculating row counts, make sure that the column you pass to the `values` argument doesn't contain null values; otherwise, your row counts will be incorrect (as null values will get excluded from your counts.) To prevent this issue, I often like to create a column that stores a value of 1 for every row. Using this column (titled `responses` in the following cell) ensures that my pivot table will show the correct row counts for each group.

*A more robust approach here would have been to calculate separate enrollment figures for the fall, winter, and spring periods in order to take new matriculations and withdrawals into account.

```
df_survey_results['responses'] = 1
df_response_rates = df_survey_results.pivot_table(
    index=['starting_year', 'season_order', 'season'],
    values=['score', 'responses'],
    aggfunc={'score':'mean', 'responses':'count'}).reset_index()
# Because all 'responses' values are 1, we could also have made 'sum'
# the aggfunc for 'responses' rather than 'count.'
df_response_rates
```

	starting_year	season_order	season	responses	score
0	2023	0	Fall	16384	69.682251
1	2023	1	Winter	13926	64.216214
2	2023	2	Spring	16384	72.049622

Calculating response rates as the quotient of survey counts and NVCU's current enrollment:

```
df_response_rates['response_rate'] = 100 * (
    df_response_rates['responses'] / enrollment_count)
df_response_rates
```

	starting_year	season_order	season	responses	score	response_rate
0	2023	0	Fall	16384	69.682251	100.000000
1	2023	1	Winter	13926	64.216214	84.997559
2	2023	2	Spring	16384	72.049622	100.000000

This table shows that our survey response rates were 100% during the fall and spring and around 85% during the winter.

6.3 Using the columns argument within pivot_table() to show seasons side by side

Currently, the DataFrame is in 'long' format: each row shows data for one specific season. However, in order to more easily calculate the change in results from one season to another, we can also use the `columns` argument when creating a pivot table in order to show scores for each season side by side. (This will prove especially useful when we add additional index variables to our `pivot_table()` call.)

The following function is similar to our earlier `pivot_table` calls except that the `season_order` and `season` values have been moved from the `index` argument to the argument for `columns`. This change makes the seasons appear horizontally rather than vertically.

```
df_results_by_season_wide = df_survey_results.pivot_table(
    index='starting_year', columns=['season_order', 'season'],
    values='score', aggfunc='mean').reset_index()
df_results_by_season_wide
```

season_order	starting_year	0	1	2
season		Fall	Winter	Spring
0	2023	69.682251	64.216214	72.049622

Note that, because we passed two values to the `columns` parameter, two levels of headers are now visible. However, I'd like to show just one level of columns that will comprise the `starting_year` value in the top row and the season names in the bottom row. We can accomplish this by first calling `to_flat_index()` to 'flatten' the columns into tuples:

```
df_results_by_season_wide.columns = (
    df_results_by_season_wide.columns.to_flat_index())
df_results_by_season_wide
```

	(starting_year,)	(0, Fall)	(1, Winter)	(2, Spring)
0	2023	69.682251	64.216214	72.049622

Next, I'll use a list comprehension to replace our tuple-based columns with string-based ones. Note that I want to keep the first entry ('starting_year') in the first tuple and the second entries (Fall, Winter, and Spring) in the others; this can be accomplished by adding an if/else statement to our list comprehension.

```
df_results_by_season_wide.columns = [
    column_tuple[0] if column_tuple[1] not in ['Fall', 'Winter', 'Spring']
    else column_tuple[1] for column_tuple in
    df_results_by_season_wide.columns]
df_results_by_season_wide
```

	starting_year	Fall	Winter	Spring
0	2023	69.682251	64.216214	72.049622

Now that we have our seasons next to one another, we can easily calculate changes in average scores between them:

```
df_results_by_season_wide['Fall-Spring Change'] = (
    df_results_by_season_wide['Spring']
    - df_results_by_season_wide['Fall'])

df_results_by_season_wide['Fall-Winter Change'] = (
    df_results_by_season_wide['Winter']
    - df_results_by_season_wide['Fall'])

df_results_by_season_wide['Winter-Spring Change'] = (
    df_results_by_season_wide['Spring']
    - df_results_by_season_wide['Winter'])

df_results_by_season_wide
```

	starting_year	Fall	Winter	Spring	Fall-Spring Change	Fall-Winter Change	Winter-Spring Change
0	2023	69.682251	64.216214	72.049622	2.367371	-5.466037	7.833407

We can simplify this code by creating a list of the seasons between which we want to calculate changes in survey scores, then looping through that list. Note that this code produces the following output as the previous cell.

```
for sp in [('Fall', 'Spring'), ('Fall', 'Winter'),
           ('Winter', 'Spring')]: # 'sp' here is short for 'season pair.'
    df_results_by_season_wide[f'{sp[0]}-{sp[1]} Change'] = (
        df_results_by_season_wide[sp[1]]
        - df_results_by_season_wide[sp[0]])

df_results_by_season_wide
```

	starting_year	Fall	Winter	Spring	Fall-Spring	Change	\
0	2023	69.682251	64.216214	72.049622		2.367371	
0		Fall-Winter Change	Winter-Spring Change				
0		-5.466037	7.833407				

6.4 Adding additional pivot index values

We now know that our average NVCU student survey scores declined from the fall to the winter and then rose from the winter to the spring. Was this trend the same across colleges and levels? We can answer this question by adding our college and level fields to the `index` argument of our pivot table function.

In order to make this section more efficient, we can create a function that performs the pivot table, column renaming, and growth calculations shown above for `df_results_by_season_wide`. This will greatly reduce the amount of code that we need to write to perform these additional analyses.

```
def create_wide_table(index_values):
    '''This function creates a wide pivot table of df_survey_results, then
    performs additional column renaming steps and growth calculations.

    index_values: a list of values to pass to the index argument of
    pivot_table().'''

    df_wide = df_survey_results.pivot_table(
        index = index_values, columns = ['season_order', 'season'],
        values = 'score', aggfunc = 'mean').reset_index()
    df_wide.columns = (df_wide.columns.to_flat_index())
    df_wide.columns = [column_tuple[0] if column_tuple[1] not in
                       ['Fall', 'Winter', 'Spring'] else column_tuple[1]
                       for column_tuple in df_wide.columns]
    for sp in [('Fall', 'Spring'), ('Fall', 'Winter'),
               ('Winter', 'Spring')]:
        df_wide[f'{sp[0]}-{sp[1]} Change'] = (
            df_wide[sp[1]] - df_wide[sp[0]])
    return df_wide
```

6.4.1 Evaluating changes in survey scores by season and college:

```
df_results_by_season_and_college_wide = create_wide_table(
    index_values = ['starting_year', 'college'])
# Saving these results to a .csv file so that they can be used within
# other parts of Python for Nonprofits:
df_results_by_season_and_college_wide.to_csv(
    'survey_results_by_college_wide.csv', index = False)
df_results_by_season_and_college_wide
```

	starting_year	college	Fall	Winter	Spring	Fall-Spring	Change	\
0	2023	STB	69.797119	64.406522	67.077551		-2.719568	
1	2023	STC	69.568665	63.934845	66.911444		-2.657221	
2	2023	STL	69.596675	64.146248	76.727809		7.131134	
3	2023	STM	69.735685	64.318689	76.639004		6.903320	

(continues on next page)

(continued from previous page)

	Fall-Winter Change	Winter-Spring Change
0	-5.390596	2.671029
1	-5.633820	2.976599
2	-5.450427	12.581561
3	-5.416995	12.320315

Although we found earlier that university-wide survey results grew from the fall to the spring, this table shows that results for two colleges (STB and STC) actually *dropped* over that time period. (Their average spring survey scores were also markedly lower than STL's and STM's.) It also demonstrates that every college saw both a drop in scores from the fall to the winter *and* an increase in scores from the winter to the spring.

The following cell creates a ‘long’ version of this table that features only one season per row and doesn’t display changes in scores between seasons. We’ll utilize a .csv copy of this table within PFN’s graphing section. (We could also use the wide-formatted table shown above within our graphing script, but I wanted to make sure to demonstrate how to graph long-formatted data.)

Note that `season_order` is added before `season` within the `index` list in order to get Winter results to precede Spring ones; however, it’s then dropped in order to help streamline the table.

```
df_results_by_season_and_college_long = df_survey_results.pivot_table(
    index=['starting_year', 'college', 'season_order', 'season'],
    values='score', aggfunc='mean').reset_index().drop(
    'season_order', axis=1)
df_results_by_season_and_college_long.to_csv(
    'survey_results_by_college_long.csv', index=False)
df_results_by_season_and_college_long
```

	starting_year	college	season	score
0	2023	STB	Fall	69.797119
1	2023	STB	Winter	64.406522
2	2023	STB	Spring	67.077551
3	2023	STC	Fall	69.568665
4	2023	STC	Winter	63.934845
5	2023	STC	Spring	66.911444
6	2023	STL	Fall	69.596675
7	2023	STL	Winter	64.146248
8	2023	STL	Spring	76.727809
9	2023	STM	Fall	69.735685
10	2023	STM	Winter	64.318689
11	2023	STM	Spring	76.639004

6.4.2 Evaluating changes in survey scores by season and level:

For this task, we’ll pivot the data by `level_for_sorting` so as to order the rows from youngest to oldest.

```
df_results_by_season_and_college_wide = create_wide_table(
    index_values=['starting_year', 'level_for_sorting', 'level'])
df_results_by_season_and_college_wide.drop([
    'starting_year', 'level_for_sorting'], axis=1) # I chose to drop
# these columns so that none of the more important ones would get
# cut off by my notebook's 8-column limit
```

	level	Fall	Winter	Spring	Fall-Spring	Change	\
0	Fr	69.609774	64.193667	74.833180		5.223406	
1	So	69.688672	64.231086	69.495124		-0.193548	
2	Ju	69.768957	64.221396	69.350671		-0.418286	
3	Se	69.698085	64.229462	73.546671		3.848586	
		Fall-Winter Change	Winter-Spring Change				
0		-5.416107	10.639513				
1		-5.457586	5.264038				
2		-5.547561	5.129274				
3		-5.468623	9.317209				

This table shows that, whereas scores for sophomores and juniors did not change much from the fall to the spring, scores for freshmen and seniors increased quite a bit during that period. All levels showed a fall-to-winter drop followed by a winter-to-spring rise.

6.4.3 Evaluating changes in survey scores by season, level, and college:

(I originally named the following DataFrame `df_results_by_season_level_and_college_wide`, but since that's a rather long name for a DataFrame that we'll use quite frequently, I chose the abbreviated name `df_results_slc` instead.)

```
df_results_slc = create_wide_table(
    index_values=['starting_year', 'college',
                  'level_for_sorting', 'level'])
df_results_slc.to_csv('survey_results_slc_wide.csv', index = False)
df_results_slc.head()
```

	starting_year	college	level_for_sorting	level	...	Spring	\
0	2023	STB		0	Fr	...	69.177235
1	2023	STB		1	So	...	65.377306
2	2023	STB		2	Ju	...	64.950769
3	2023	STB		3	Se	...	68.543287
4	2023	STC		0	Fr	...	69.079348
		Fall-Spring Change	Fall-Winter Change	Winter-Spring Change			
0		-0.416348	-5.403776	4.987428			
1		-5.000000	-5.237847	0.237847			
2		-5.000000	-5.507824	0.507824			
3		-0.611041	-5.426788	4.815747			
4		-0.519565	-5.674532	5.154967			

[5 rows x 10 columns]

The following cell creates a 'long' version of this table that can be incorporated into PFN's graphing section.

```
df_survey_results_slc_long = df_survey_results.pivot_table(
    index=['starting_year', 'college',
           'level_for_sorting', 'level', 'season'],
    values='score', aggfunc = 'mean').reset_index()
df_survey_results_slc_long.to_csv(
    'survey_results_slc_long.csv', index = False)
df_survey_results_slc_long.head()
```

	starting_year	college	level_for_sorting	level	season	score
0	2023	STB		0	Fr	Fall 69.593583
1	2023	STB		0	Fr	Spring 69.177235
2	2023	STB		0	Fr	Winter 64.189807
3	2023	STB		1	So	Fall 70.377306
4	2023	STB		1	So	Spring 65.377306

6.4.4 Creating a wide student-level version of our survey results dataset

We'll also create a table that shows fall, winter, and spring survey results for each individual student within the same row. (This table will get used within Part 2 of the Descriptive Stats section.)

Since we're simply condensing and reshaping `df_survey_results` here rather than calculating averages, we can use `pivot` rather than `pivot_table`. Note that `pivot` doesn't take `aggfunc` as an argument (as we're not creating any aggregate statistics here).

```
df_student_results_wide = df_survey_results.copy().pivot(
    index = ['starting_year', 'student_id'],
    columns = 'season', values = 'score').reset_index()
df_student_results_wide.head()
```

	season	starting_year	student_id	Fall	Spring	Winter
0		2023	2020-1	88.0	86.0	81.0
1		2023	2020-10	69.0	73.0	63.0
2		2023	2020-100	68.0	88.0	60.0
3		2023	2020-1000	58.0	65.0	55.0
4		2023	2020-1001	88.0	100.0	84.0

Note that 'Spring' is located to the left of 'Winter'. To move it to the right, I'll call `df.insert()`, which allows us to add a new column into the DataFrame. This 'new' column will actually be the current Winter column, which I'll first remove from the DataFrame via `df.pop()`.

The first argument within `df.insert()` is the location of the new column. The code calculates this location by subtracting 1 from the number of columns within `df_student_results_wide`; this will represent the position to the right of the 'Winter' column once the 'Spring' column is removed. (I could have hardcoded this value, but this approach will better accommodate changes to `df_student_results_wide`'s structure).

`df.insert()`'s second argument is the name (which we'll keep as 'Spring'), and its third argument is the data to add to this new column (which, in this case, will be the existing 'Spring' column entries).

For more details on these functions, visit <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.insert.html> and <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pop.html>. The idea of chaining these two functions together isn't an original one; I probably got it from a StackOverflow answer like this one from Marc Maxmeister : <https://stackoverflow.com/a/77463008/13097194>)

```
df_student_results_wide.insert(
    len(df_student_results_wide.columns)-1, 'Spring',
    df_student_results_wide.pop('Spring'))
df_student_results_wide.head()
```

	season	starting_year	student_id	Fall	Winter	Spring
0		2023	2020-1	88.0	81.0	86.0
1		2023	2020-10	69.0	63.0	73.0

(continues on next page)

(continued from previous page)

2	2023	2020-100	68.0	60.0	88.0
3	2023	2020-1000	58.0	55.0	65.0
4	2023	2020-1001	88.0	84.0	100.0

I could also have reordered the columns via the following code; however, this code would then need to be updated to account for any new columns that we chose to add to the dataset, whereas the method shown above is a bit more flexible.

```
df_student_results_wide = df_student_results_wide[['starting_year', 'student_id', 'Fall', 'Winter', 'Spring']].copy()
df_student_results_wide.head()
```

season	starting_year	student_id	Fall	Winter	Spring
0	2023	2020-1	88.0	81.0	86.0
1	2023	2020-10	69.0	63.0	73.0
2	2023	2020-100	68.0	60.0	88.0
3	2023	2020-1000	58.0	55.0	65.0
4	2023	2020-1001	88.0	84.0	100.0

Saving the table to a .csv file so that it can get imported into Part 2 of the Descriptive Stats section:

```
df_student_results_wide.to_csv(  
    'survey_results_by_student_wide.csv', index = False)
```

6.5 Comparing rows via `sort_values()` and `rank()`

Which college/level pairs had the highest and lowest spring survey results? We could examine `df_results_slc` line by line to answer this question; however, two Pandas functions—`sort_values()` and `rank()`—can help us answer this question more efficiently.

First, here are the five college/level pairs with the highest average spring results:

```
df_results_slc.sort_values('Spring', ascending=False).drop(['starting_year', 'level_for_sorting'], axis=1).head()
```

college	level	Fall	Winter	Spring	Fall-Spring	Change
15	STM	Se	70.058590	64.707852	79.309831	9.251241
8	STL	Fr	69.585030	64.255102	79.111622	9.526592
12	STM	Fr	69.650503	64.286313	78.488468	8.837966
11	STL	Se	69.026639	63.520482	77.915984	8.889344
10	STL	Ju	69.878706	64.372549	74.634771	4.756065
Fall-Winter Change		Winter-Spring Change				
15		-5.350738		14.601979		
8		-5.329928		14.856520		
12		-5.364190		14.202156		
11		-5.506157		14.395502		
10		-5.506157		10.262222		

And here are the five pairs with the *lowest* spring results:

```
df_results_slc.sort_values('Spring').drop(['starting_year', 'level_for_sorting'], axis=1).head()
```

	college	level	Fall	Winter	Spring	Fall-Spring	Change	\
6	STC	Ju	69.180932	63.502506	64.180932		-5.000000	
5	STC	So	69.331325	63.760684	64.331325		-5.000000	
2	STB	Ju	69.950769	64.442945	64.950769		-5.000000	
1	STB	So	70.377306	65.139459	65.377306		-5.000000	
3	STB	Se	69.154329	63.727541	68.543287		-0.611041	
			Fall-Winter Change	Winter-Spring Change				
6			-5.678425	0.678425				
5			-5.570642	0.570642				
2			-5.507824	0.507824				
1			-5.237847	0.237847				
3			-5.426788	4.815747				

Note that the use of `sort_values()` here did not actually change the underlying order of the DataFrame. Although the output appeared in sorted order immediately after `sort_values()` got called, the DataFrame will then revert to its original sort order during subsequent lines of code. The following cell demonstrates this:

```
df_results_slc.head() # Note that the DataFrame  
# is once again sorted by college and level
```

	starting_year	college	level_for_sorting	level	...	Spring	\	
0	2023	STB		0	Fr	...	69.177235	
1	2023	STB		1	So	...	65.377306	
2	2023	STB		2	Ju	...	64.950769	
3	2023	STB		3	Se	...	68.543287	
4	2023	STC		0	Fr	...	69.079348	
			Fall-Spring Change	Fall-Winter Change	Winter-Spring Change			
0			-0.416348	-5.403776	4.987428			
1			-5.000000	-5.237847	0.237847			
2			-5.000000	-5.507824	0.507824			
3			-0.611041	-5.426788	4.815747			
4			-0.519565	-5.674532	5.154967			

[5 rows x 10 columns]

This behavior, which is seen in many other Pandas functions, is actually quite helpful: it allows you to test out changes and modifications without making them permanent (which, if you make a mistake, could force you to restart your script).

To make a sort persistent, you can use one of the following two lines:

```
# First option (my preference because it often requires fewer characters):  
df_results_slc.sort_values('Spring', inplace=True)  
  
# An alternative option (which can come in handy when making multiple  
# changes to a dataset at once):  
df_results_slc = (df_results_slc.sort_values('Spring')).copy()  
  
# Make sure NOT to add 'inplace = True' as an argument when using the  
# second method, as your DataFrame will then get replaced with  
# a 'None' object!
```

(continues on next page)

(continued from previous page)

```
# For an explanation of None, see:
# https://docs.python.org/3/library/constants.html#None
```

Now that we've made our sort persistent, we will continue to see it reflected within subsequent sets of output:

```
df_results_slc.head()
```

	starting_year	college	level_for_sorting	level	...	Spring	\
6	2023	STC		2	Ju	...	64.180932
5	2023	STC		1	So	...	64.331325
2	2023	STB		2	Ju	...	64.950769
1	2023	STB		1	So	...	65.377306
3	2023	STB		3	Se	...	68.543287

	Fall-Spring Change	Fall-Winter Change	Winter-Spring Change
6	-5.000000	-5.678425	0.678425
5	-5.000000	-5.570642	0.570642
2	-5.000000	-5.507824	0.507824
1	-5.000000	-5.237847	0.237847
3	-0.611041	-5.426788	4.815747

[5 rows x 10 columns]

It's also worth mentioning that none of these changes, even the persistent sort that we just effected, are affecting the underlying .csv file from which we retrieved our data. That file will only get modified if we use `to_csv()` to save our table to that same filename. (You'll usually want to save tables to a different filename anyway so that your output doesn't overwrite your input.)

6.6 Calculating percentiles and ranks

Ranks and percentiles are alternative ways to evaluate values relative to their peers. Let's say that the NVCU administration would like you to calculate both the rank *and* the percentile of each college/level pair's average spring score. However, they'd also like you to round the spring results to integers before making these calculations so that pairs with similar scores will get treated equally.

First, we'll create a new condensed DataFrame that can store these integer-based results, ranks, and percentiles. We'll then assign ranks to each integer.

```
# Creating a condensed DataFrame:
df_spring_ranks = df_results_slc.copy()[['starting_year', 'college',
                                         'level_for_sorting', 'level', 'Spring']].sort_values(
    'Spring', ascending=False)
# Converting average spring results to integers:
df_spring_ranks['Spring'] = df_spring_ranks['Spring'].astype('int')

# Calculating our ranks:
# Note: the inclusion of "method = 'min'" ensures that, in the case of
# ties, each tied row will show the lowest (i.e. best) possible rank.
# This is the ranking convention that I'm more familiar with, but Pandas
# allows for other methods also. Meanwhile, "ascending = False" assigns the
# best ranks to the highest results.
# See the df.rank() documentation for more details:
```

(continues on next page)

(continued from previous page)

```
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.
# DataFrame.rank.html

df_spring_ranks['Spring_Rank'] = df_spring_ranks[
    'Spring'].rank(ascending = False, method = 'min')
df_spring_ranks
```

	starting_year	college	level_for_sorting	level	Spring	Spring_Rank
15	2023	STM		3	Se	79
8	2023	STL		0	Fr	79
12	2023	STM		0	Fr	78
11	2023	STL		3	Se	77
10	2023	STL		2	Ju	74
14	2023	STM		2	Ju	74
9	2023	STL		1	So	74
13	2023	STM		1	So	73
7	2023	STC		3	Se	69
0	2023	STB		0	Fr	69
4	2023	STC		0	Fr	69
3	2023	STB		3	Se	68
1	2023	STB		1	So	65
2	2023	STB		2	Ju	64
5	2023	STC		1	So	64
6	2023	STC		2	Ju	64

Our code for calculating percentiles will also use `df.rank()`; we can instruct that function to display its output as percentiles by adding the argument `pct=True`. We'll also add (1) `ascending=True` so that the highest scores will get the highest percentiles and (2) `method='max'` so that, in the case of ties, the highest possible percentile will get displayed.

Note that, while the highest percentile in the following output is 100, the lowest percentile is not 0. I'm quite sure that this is because Pandas calculates percentiles as the percentage of results *equal to or lower than* the current result. Therefore, even the lowest row won't get a percentile of 0 during percentile calculations, as it will at least be equal to itself. (The larger your dataset, the closer the smallest percentile will be to 0.)

```
df_spring_ranks['Spring_Percentile'] = (
    100 * df_spring_ranks['Spring'].rank(
        ascending=True, pct=True, method='max'))
df_spring_ranks.head()
```

	starting_year	college	level_for_sorting	level	Spring	Spring_Rank	\
15	2023	STM		3	Se	79	1.0
8	2023	STL		0	Fr	79	1.0
12	2023	STM		0	Fr	78	3.0
11	2023	STL		3	Se	77	4.0
10	2023	STL		2	Ju	74	5.0
					Spring_Percentile		
15					100.00		
8					100.00		
12					87.50		
11					81.25		
10					75.00		

Although our DataFrame was sorted by Spring results, `df.rank()` would still have successfully calculated ranks and

percentiles regardless of how the DataFrame happened to be sorted. We can demonstrate this via the following code (which sorts the DataFrame by a different field before calling `rank()`):

```
df_spring_ranks['Spring_Percentile'] = (
    100 * df_spring_ranks.sort_values('college')['Spring'].rank(
        ascending=True, pct=True, method='max'))
df_spring_ranks.head()
```

	starting_year	college	level_for_sorting	level	Spring	Spring_Rank	\
15	2023	STM		3	Se	79	1.0
8	2023	STL		0	Fr	79	1.0
12	2023	STM		0	Fr	78	3.0
11	2023	STL		3	Se	77	4.0
10	2023	STL		2	Ju	74	5.0

	Spring_Percentile
15	100.00
8	100.00
12	87.50
11	81.25
10	75.00

```
end_time = time.time()
run_time = end_time - start_time
print(f"Finished running this script in {round(run_time, 3)} seconds.")
```

```
Finished running this script in 0.602 seconds.
```

The above line shows how long it took this notebook to run all of its Python code. As you can see, my claim at the start of this notebook that you could use Python to perform these tasks in ‘mere seconds’ wasn’t hyperbole!

6.7 Conclusion

This notebook has provided a brief introduction to descriptive statistics calculations within Python. PFN’s graphing section will teach you how to convert some of the pivot tables created here into line and bar charts, thus making these data easier to interpret. However, before we can dive into the fun of Python-based visualizations, we should first cover a few more important descriptive stats topics.

DESCRIPTIVE STATS: PART 2

By Kenneth Burchfiel

This second part of Python for Nonprofits' descriptive stats section covers several potential data analysis pitfalls. Specifically, it will explore:

1. How to adjust for missing values when calculating weighted averages
2. The risk of relying on column index positions
3. Why column-wise operations should be preferred over for loops
4. Challenges with finding average values for fields whose rows are themselves averages
5. Methods of accounting for missing data when creating pivot tables
6. Issues with using `np.where()` to create derivatives of columns with missing data (and why `map()` and `np.select()` are better fits)

I have to admit that, even by programming textbook standards, you may not find this to be the most exciting chapter. You may well be anxious to get ahead to the graphing, mapping, and online dashboard sections of PFN—i.e. the ‘fun’ stuff.

However, in order to be confident that your graphs, maps, and dashboards will provide an accurate view of your underlying data, it’s crucial to exercise caution when pivoting and transforming tables. This chapter is meant to help you be more cautious—and thus more successful—in your data analysis adventures.

```
import pandas as pd
import numpy as np
from sqlalchemy import create_engine
e = create_engine('sqlite:///../Appendix/nvcu_db.db')

import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook
display_type = config_notebook(display_max_columns=6,
                                display_max_rows=8)
```

7.1 Calculating weighted average results by student (and dealing with missing values)

Suppose that the NVCU administration also wishes to see what percentage of students had a weighted average annual survey score below 60. Because they are more interested in students' most recent survey results, they would like you to assign a weight of 0.2 to the fall results; 0.3 to the winter results; and 0.5 to the spring results. (Thus, students' weighted survey averages will equal $0.2*F + 0.3*W + 0.5*S$, with F, W, and S referring to students' fall, winter, and spring results, respectively.)

We'll begin this analysis by loading in 'survey_results_by_student_wide.csv', which shows fall, winter, and spring scores side by side for each student.

```
df_student_results_wide = pd.read_csv(  
    'survey_results_by_student_wide.csv')  
# Creating a copy of this DataFrame that includes only students with  
# valid winter survey results:  
# (We'll make use of this copy later within this notebook.)  
df_valid_survey_results = (  
    df_student_results_wide.query("Winter.notna()").copy().drop(  
        'starting_year', axis=1))  
df_student_results_wide.head()
```

	starting_year	student_id	Fall	Winter	Spring
0	2023	2020-1	88.0	81.0	86.0
1	2023	2020-10	69.0	63.0	73.0
2	2023	2020-100	68.0	60.0	88.0
3	2023	2020-1000	58.0	55.0	65.0
4	2023	2020-1001	88.0	84.0	100.0

Because not all students took the winter survey, some winter results have NaN (not a number) values. We can count the number of NaN results for each column by (1) calling the `isna()` function, which displays whether or not each cell is NaN, then (2) following that call with `.sum()` in order to add up all NaN entries.

```
df_student_results_wide.isna().sum()
```

	0
starting_year	0
student_id	0
Fall	0
Winter	2458
Spring	0
dtype: int64	

These missing results will make our weighted average calculations a bit more complicated. For instance, suppose we tried to create our weighted averages using the following code:

```
avg_score_cols_to_display = [  
    'student_id', 'Fall', 'Winter', 'Spring', 'weighted_avg_score']  
# Displaying only these columns in the following cells will help  
# condense the output
```

```
df_student_results_wide['weighted_avg_score'] = (  
    df_student_results_wide['Fall'] * 0.2  
    + df_student_results_wide['Winter'] * 0.3
```

(continues on next page)

(continued from previous page)

```
+ df_student_results_wide['Spring'] * 0.5)
df_student_results_wide[avg_score_cols_to_display].head()
```

	student_id	Fall	Winter	Spring	weighted_avg_score
0	2020-1	88.0	81.0	86.0	84.9
1	2020-10	69.0	63.0	73.0	69.2
2	2020-100	68.0	60.0	88.0	75.6
3	2020-1000	58.0	55.0	65.0	60.6
4	2020-1001	88.0	84.0	100.0	92.8

This code works fine for students with valid scores for all 3 seasons, but those with a NaN winter value will also end up with a NaN average score:

```
df_student_results_wide.query("Winter.isna()") [
    avg_score_cols_to_display].head()
```

	student_id	Fall	Winter	Spring	weighted_avg_score
10	2020-1007	93.0	NaN	96.0	NaN
18	2020-1014	66.0	NaN	66.0	NaN
21	2020-1017	70.0	NaN	66.0	NaN
28	2020-1023	72.0	NaN	67.0	NaN
31	2020-1026	74.0	NaN	75.0	NaN

A naive approach to compensate for these NaN results would be to call `fillna()` to replace all NaN values with 0, as shown in the following block of code. However, **this approach will result in inaccurately low average score values for students with missing winter results.** This is because our valid score weights for these students (0.2 for fall and 0.5 for spring) add up to only 0.7.

In the following output, note how the weighted averages for students with missing winter scores are lower than both their fall *and* spring scores, which doesn't make sense.

```
df_student_results_wide['weighted_avg_score'] = (
    df_student_results_wide['Fall'].fillna(0) * 0.2
    + df_student_results_wide['Winter'].fillna(0) * 0.3
    + df_student_results_wide['Spring'].fillna(0) * 0.5)
df_student_results_wide.query("Winter.isna()") [
    avg_score_cols_to_display].head()
```

	student_id	Fall	Winter	Spring	weighted_avg_score
10	2020-1007	93.0	NaN	96.0	66.6
18	2020-1014	66.0	NaN	66.0	46.2
21	2020-1017	70.0	NaN	66.0	47.0
28	2020-1023	72.0	NaN	67.0	47.9
31	2020-1026	74.0	NaN	75.0	52.3

Here's a better approach that, while a bit more complex, successfully adjusts for missing values. First, we'll create a weight column for each season that displays either our predetermined weight (if a student has a valid score for that season) or 0 (if the student does not). We'll also create a column that adds all of these weights together.

The following code applies `np.where()` to determine whether or not to assign a given student/season pair a weight of 0. If a score is missing for a given season, that season will receive a weight entry of 0; otherwise, it will be assigned the usual weight. (See <https://numpy.org/doc/stable/reference/generated/numpy.where.html> for more information about `np.where()`.)

`np.where()` is a great option for filling in DataFrame fields based on two specific conditions (e.g. is survey data for a season *missing* or *present*?). However, if you ever need to handle three or more conditions, consider using `np.select()` instead. (We'll cover this function later within PFN.)

```
season_weight_dict = {'Fall':0.2,'Winter':0.3,'Spring':0.5}
# Using a for loop to create these columns makes our code a bit more
# concise.
for season in ['Fall', 'Winter', 'Spring']:
    df_student_results_wide[season+'_weight'] = np.where(
        df_student_results_wide[season].isna(), 0,
        season_weight_dict[season])

# adding axis=1 as an argument to df.sum() ensures that the calculations
# will be made row-wise rather than column-wise.
df_student_results_wide['weight_sum'] = df_student_results_wide[['
    'Fall_weight', 'Winter_weight', 'Spring_weight']].sum(axis=1)

df_student_results_wide.head()
```

	starting_year	student_id	Fall	...	Winter_weight	Spring_weight	\
0	2023	2020-1	88.0	...	0.3	0.5	
1	2023	2020-10	69.0	...	0.3	0.5	
2	2023	2020-100	68.0	...	0.3	0.5	
3	2023	2020-1000	58.0	...	0.3	0.5	
4	2023	2020-1001	88.0	...	0.3	0.5	

	weight_sum
0	1.0
1	1.0
2	1.0
3	1.0
4	1.0

[5 rows x 10 columns]

We can now accurately calculate average scores for all students by (1) multiplying each score by its corresponding weight value (which will be 0 in the case of missing scores); (2) adding these products together; and then (3) dividing the sum by the `weight_sum` column. If a student has missing values for a given season, his or her `weight_sum` value will also be lower, thus compensating for his/her lower sum of scores.

(Note that we'll still fill in missing scores with 0 in order to prevent final NaN outputs.)

```
df_student_results_wide['weighted_avg_score'] = (
    df_student_results_wide['Fall'].fillna(0)
    * df_student_results_wide['Fall_weight']
    + df_student_results_wide['Winter'].fillna(0)
    * df_student_results_wide['Winter_weight']
    + df_student_results_wide['Spring'].fillna(0)
    * df_student_results_wide['Spring_weight']) / (
        df_student_results_wide['weight_sum'])

df_student_results_wide.query("Winter.isna()").head()
```

	starting_year	student_id	Fall	...	Winter_weight	Spring_weight	\
10	2023	2020-1007	93.0	...	0.0	0.5	
18	2023	2020-1014	66.0	...	0.0	0.5	

(continues on next page)

(continued from previous page)

```

21      2023  2020-1017  70.0  ...          0.0      0.5
28      2023  2020-1023  72.0  ...          0.0      0.5
31      2023  2020-1026  74.0  ...          0.0      0.5

    weight_sum
10      0.7
18      0.7
21      0.7
28      0.7
31      0.7

[5 rows x 10 columns]

```

The following output confirms that students' weighted average scores are no longer being dragged down by missing winter results:

```
df_student_results_wide.query("Winter.isna()") [
    avg_score_cols_to_display].head()
```

	student_id	Fall	Winter	Spring	weighted_avg_score
10	2020-1007	93.0	NaN	96.0	95.142857
18	2020-1014	66.0	NaN	66.0	66.000000
21	2020-1017	70.0	NaN	66.0	67.142857
28	2020-1023	72.0	NaN	67.0	68.428571
31	2020-1026	74.0	NaN	75.0	74.714286

A few additional notes:

1. This approach would also successfully compensate for students with missing fall or spring scores. It would only fail to work for students who had no survey results at all—but such students should be excluded from these calculations to begin with.
2. Note that, for students with missing winter weights, this code uses a fall score weight of 0.2/0.7 (28.6%) and a spring score weight of 0.5/0.7 (71.4%). Thus, fall and spring results counted more for these students than they did for students with valid winter results.

Now that we've calculated weighted average results for all students, we can finally answer the administrators' original question: what percentage of students had a weighted average survey score below 60?

```
df_student_results_wide['weighted_avg_below_60'] = np.where(
    df_student_results_wide['weighted_avg_score'] < 60, 1, 0)

# Calling value_counts(normalize=True), then multiplying the results
# by 100, allows us to calculate the percentage of students with weighted
# averages below 60.
100*df_student_results_wide['weighted_avg_below_60'].value_counts(
    normalize=True)
```

```

weighted_avg_below_60
0     81.292725
1     18.707275
Name: proportion, dtype: float64

```

It turns out that around 18.7% of students had a weighted average score below 60.

7.2 The danger of relying on column index positions when analyzing datasets

There are several different ways to specify the DataFrame columns on which you would like to perform operations. So far, I have been selecting columns using their names (e.g. `df_student_results_wide['Spring']`). However, it's also possible to select them via their index positions. In this section, I'll explain why this is often *not* a good idea.

`df_valid_survey_results`, a copy of our student-level results table that excludes missing winter results, has four columns: 'student_id', 'Fall', 'Winter', and 'Spring'. These columns' index positions are 0, 1, 2, and 3, respectively.

```
df_valid_survey_results.head()
```

	student_id	Fall	Winter	Spring
0	2020-1	88.0	81.0	86.0
1	2020-10	69.0	63.0	73.0
2	2020-100	68.0	60.0	88.0
3	2020-1000	58.0	55.0	65.0
4	2020-1001	88.0	84.0	100.0

If I wanted to create a non-weighted average of students' winter and spring scores, I *could* use `.iloc[]` (which allows columns to be selected via these index positions) to retrieve the scores for these two seasons.

In the following code, `.iloc[:, 2:4]` selects all rows for the columns between index positions 2 (inclusive) and 4 (exclusive)—or, in other words, the columns with index positions 2 and 3.

```
df_valid_survey_results[
    'Winter/Spring Avg.'] = df_valid_survey_results.iloc[
        :, 2:4].mean(axis=1)
df_valid_survey_results
```

	student_id	Fall	Winter	Spring	Winter/Spring Avg.
0	2020-1	88.0	81.0	86.0	83.5
1	2020-10	69.0	63.0	73.0	68.0
2	2020-100	68.0	60.0	88.0	74.0
3	2020-1000	58.0	55.0	65.0	60.0
...
16380	2023-996	62.0	58.0	69.0	63.5
16381	2023-997	47.0	42.0	42.0	42.0
16382	2023-998	77.0	67.0	74.0	70.5
16383	2023-999	64.0	62.0	65.0	63.5

[13926 rows x 5 columns]

This code accurately calculates the average of each student's winter and spring survey scores. What makes it dangerous, though, is that a minor change to the table could end up filling this field with incorrect data.

For instance, suppose that a new column gets added to the left of our survey scores in the future:

```
df_valid_survey_results.insert(1, 'University', 'NVCU')
df_valid_survey_results.head()
```

	student_id	University	Fall	Winter	Spring	Winter/Spring Avg.
0	2020-1	NVCU	88.0	81.0	86.0	83.5
1	2020-10	NVCU	69.0	63.0	73.0	68.0

(continues on next page)

(continued from previous page)

2	2020-100	NVCU	68.0	60.0	88.0	74.0
3	2020-1000	NVCU	58.0	55.0	65.0	60.0
4	2020-1001	NVCU	88.0	84.0	100.0	92.0

If we call the same code on this modified column, our ‘Winter/Spring Avg.’ column will now show the average of *fall* and winter scores, since the ‘University’ column has changed which columns correspond to the index positions of 2 and 3.

```
df_valid_survey_results[
    'Winter/Spring Avg.']
    = df_valid_survey_results.iloc[:, 2:4].mean(axis=1)
df_valid_survey_results.head()
```

	student_id	University	Fall	Winter	Spring	Winter/Spring Avg.
0	2020-1	NVCU	88.0	81.0	86.0	84.5
1	2020-10	NVCU	69.0	63.0	73.0	66.0
2	2020-100	NVCU	68.0	60.0	88.0	64.0
3	2020-1000	NVCU	58.0	55.0	65.0	56.5
4	2020-1001	NVCU	88.0	84.0	100.0	86.0

Thus, a much safer approach is to explicitly name the columns that you wish to incorporate into a calculation. The following code will work fine regardless of the index positions of the ‘Winter’ and ‘Spring’ columns.

```
df_valid_survey_results[
    'Winter/Spring Avg.']
    = df_valid_survey_results[
        ['Winter', 'Spring']].mean(axis=1)
df_valid_survey_results.head()
```

	student_id	University	Fall	Winter	Spring	Winter/Spring Avg.
0	2020-1	NVCU	88.0	81.0	86.0	83.5
1	2020-10	NVCU	69.0	63.0	73.0	68.0
2	2020-100	NVCU	68.0	60.0	88.0	74.0
3	2020-1000	NVCU	58.0	55.0	65.0	60.0
4	2020-1001	NVCU	88.0	84.0	100.0	92.0

Note that the code that explicitly states the name of each column is also more intuitive, as it eliminates the need to cross-reference which column corresponds to each index position. And the more intuitive you can keep your code, particularly within complex projects, the better.

7.3 Why column-wise operations should be preferred over for loops

If you’re a newcomer to Pandas, you may initially want to try using for loops to update values within DataFrames (especially if you’ve been used to using such loops within languages like C or C++). However, I recommend that you use *column-wise operations* (e.g. code that will apply to all rows within a column) rather than loops whenever possible, simply because the former are much, much faster than the latter.

Let’s say that we wanted to create an unweighted average of our fall, winter, and spring scores within `df_valid_survey_results`. One option would be to initialize an empty ‘Average’ column, then fill it in by looping through all rows in our dataset.

Creating our ‘Average’ column:

```
# Removing columns that we no longer need:  
df_valid_survey_results.drop(  
    ['University', 'Winter/Spring Avg.'], axis=1, inplace=True)  
df_valid_survey_results['Average'] = np.nan  
df_valid_survey_results.head()
```

	student_id	Fall	Winter	Spring	Average
0	2020-1	88.0	81.0	86.0	NaN
1	2020-10	69.0	63.0	73.0	NaN
2	2020-100	68.0	60.0	88.0	NaN
3	2020-1000	58.0	55.0	65.0	NaN
4	2020-1001	88.0	84.0	100.0	NaN

Determining the index position of our ‘Average’ column:

(If we tried to use this column’s name within our for loop rather than its index position, the averages wouldn’t actually get added in, and we’d end up with a `SettingWithCopyWarning`.)

```
avg_index = df_valid_survey_results.columns.get_loc('Average')  
avg_index
```

4

Calculating this average for each row:

```
for i in range(len(df_valid_survey_results)):  
    df_valid_survey_results.iloc[i, avg_index] = (  
        # Note: the following alternative to the previous line would not work,  
        # as noted in the documentation above:  
        # df_valid_survey_results.iloc[i]['Average'] = ( # Incorrect!  
            df_valid_survey_results.iloc[i]['Fall'] +  
            df_valid_survey_results.iloc[i]['Winter'] +  
            df_valid_survey_results.iloc[i]['Spring']) / 3  
df_valid_survey_results.head()
```

	student_id	Fall	Winter	Spring	Average
0	2020-1	88.0	81.0	86.0	85.000000
1	2020-10	69.0	63.0	73.0	68.333333
2	2020-100	68.0	60.0	88.0	72.000000
3	2020-1000	58.0	55.0	65.0	59.333333
4	2020-1001	88.0	84.0	100.0	90.666667

I ran the previous cell 5 times on my computer and found that it took an average of 1.72 seconds to execute. There are 13,926 rows within `df_valid_survey_results`, so even on a fast laptop, it will take Python a decent while to process each one.

The following cell, in contrast, uses a column-wise operation to compute this average:

```
df_valid_survey_results['Average'] = (  
    df_valid_survey_results['Fall'] +  
    df_valid_survey_results['Winter'] +  
    df_valid_survey_results['Spring']) / 3  
df_valid_survey_results.head()
```

	student_id	Fall	Winter	Spring	Average
0	2020-1	88.0	81.0	86.0	85.000000
1	2020-10	69.0	63.0	73.0	68.333333
2	2020-100	68.0	60.0	88.0	72.000000
3	2020-1000	58.0	55.0	65.0	59.333333
4	2020-1001	88.0	84.0	100.0	90.666667

I ran this cell 5 times as well and found that it needed only 5.4 milliseconds (on average) to execute. In other words, it was over *300 times* faster than the for loop-based approach!

The difference between 1.72 seconds and 0.006 seconds may not seem like much in nominal terms. However, when you're dealing with a dataset that contains 13 *million* rows rather than 13 *thousand*, or a script that performs dozens of these sorts of calculations, the performance advantage of column-wise calculations will become much clearer.

There are many ways to execute even complex calculations in column-wise form; tools like `np.where()`, `np.select()`, `Series.map()`, and `Series.apply()` can help you do so.

7.4 Microdata, pre-baked data, and the ‘average of averages’ problem

`df_transactions`, a table of NVCU dining hall transactions that the following cell loads in, is an excellent candidate for creating a diverse set of pivot tables and charts because each transaction has its own row. (In other words, the table contains *microdata*; for more on this subject, reference [https://en.wikipedia.org/wiki/Microdata_\(statistics\)](https://en.wikipedia.org/wiki/Microdata_(statistics)).) Microdata-based tables allow you to easily calculate statistics for your choice of comparison variables by (1) grouping rows into unique combinations of these variables, then (2) applying one or more statistical functions (`mean`, `median`, etc.) to each group. (As you've already seen, pivot tables make this process very simple.)

```
df_transactions = pd.read_sql(
    'select * from dining_transactions', con=e)
df_transactions['transactions'] = 1
df_transactions.head(5)
```

	starting_year	weekday	level	amount	transactions
0	2023	We	Fr	13.36	1
1	2023	Tu	Se	12.22	1
2	2023	We	Se	23.10	1
3	2023	Su	Fr	18.78	1
4	2023	Tu	Fr	11.36	1

If you wanted to calculate the average dining hall transaction amount, you could simply find the mean of all items within this table:

```
df_transactions['amount'].mean()
```

```
np.float64(12.65563577250256)
```

If you instead wanted to find the average amount spent by *weekday and level*, you could easily do so via Pandas' `pivot_table()` function:

```
df_transactions.pivot_table(
    index=['weekday', 'level'], values='amount',
    aggfunc='mean').reset_index().sort_values(
        by='amount', ascending=False).head()
```

	weekday	level	amount
14	Su	Se	22.426887
26	We	Se	21.550155
6	Mo	Se	21.346364
22	Tu	Se	21.268938
18	Th	Se	21.072613

However, you may not always have access to this type of data. Instead, you might receive an *aggregated* dataset in which averages by different groups are *pre-baked*—e.g. already present in the output. As you’ll soon see, this makes the table much less flexible.

‘pre-baked’ is not, to my knowledge, a common statistical term, but I find that it works pretty well for describing this type of data. For instance, once you’ve *baked* a pie using apples, wheat, and sugar (or whatever goes into a pie—I’m not a baker!), it’s pretty hard to convert that dish into a caramelized apple. Similarly, as the following examples will show, once you’ve ‘baked’ a list of transactions into separate sets of averages by level and by weekday, it will be impossible to use that data to calculate total spending amounts by level *and* weekday—as we no longer know how each level value relates to each weekday value.

To illustrate this issues caused by pre-baked datasets, let’s generate two DataFrames, the first of which will show average transaction amounts by level, and the second of which will show average amounts by weekday.

```
df_average_spending_by_level = df_transactions.pivot_table(  
    index='level', values=[  
        'amount', 'transactions'], aggfunc={  
            'amount':'mean', 'transactions':'count'}).reset_index()  
df_average_spending_by_level
```

	level	amount	transactions
0	Fr	9.911620	12410
1	Ju	17.133287	3289
2	Se	21.294252	2046
3	So	13.006451	4708

```
df_average_spending_by_weekday = df_transactions.pivot_table(  
    index='weekday', values=[  
        'amount', 'transactions'], aggfunc={  
            'amount':'mean', 'transactions':'count'}).reset_index()  
df_average_spending_by_weekday
```

	weekday	amount	transactions
0	Fr	12.721265	2466
1	Mo	12.632190	3608
2	Sa	12.253912	708
3	Su	12.775699	1123
4	Th	12.592931	3964
5	Tu	12.747500	5267
6	We	12.624990	5317

We can see that lower levels (e.g. freshmen and sophomores) tend to spend less per dining hall visit than upper levels; in addition, we can see that spending doesn’t vary too much by day of the week. However, because these tables are pre-baked, we don’t have any way of calculating average spending by level *and* weekday like we could with df_transactions.

In addition, let’s say that we wanted to calculate average dining hall spending for all students using df_average_spending_by_level. A naive approach would be to simply calculate the average of each row:

```
# The following approach is incorrect:

df_average_spending_by_level['amount'].mean()

np.float64(15.336402324109617)
```

This average is way off the actual average, which we calculated earlier in this code using `df_transactions`. Why is this the case? As `df_average_spending_by_level` shows, seniors and juniors spend much more than freshmen and sophomores, yet they also use the dining hall less (as shown by their smaller transaction counts). As a result, if we simply average the mean transaction amounts for each level, we'll overrepresent upperclassmen and thus skew our average transaction amount upward.

In order to avoid this ‘average of averages’ issue, which arises whenever differences in group sizes skew an average that’s in turn based on averages for those groups, we’ll need to create a *weighted* average. We can do so by multiplying each row’s ‘amount’ column by its ‘transaction’ column; adding these products together; and then dividing them by the ‘transaction’ column.

```
df_average_spending_by_level['amount_x_transactions'] = (
    df_average_spending_by_level['amount']
    * df_average_spending_by_level['transactions'])
df_average_spending_by_level
```

	level	amount	transactions	amount_x_transactions
0	Fr	9.911620	12410	123003.20
1	Ju	17.133287	3289	56351.38
2	Se	21.294252	2046	43568.04
3	So	13.006451	4708	61234.37

Here’s our weighted average, which matches the average calculated earlier:

```
(df_average_spending_by_level['amount_x_transactions'].sum()
 / df_average_spending_by_level['transactions'].sum())

np.float64(12.65563577250256)
```

The following function can be used to calculate weighted means for other datasets:

```
def weighted_mean(original_df, metric_col, weight_col):
    '''This function calculates, then returns, a weighted mean for
    the DataFrame passed to original_df.
    metric_col: the column storing the variable for which to calculate
    a mean.
    weight_col: the column storing weight values that will be incorporated
    into this weighted mean.
    '''
    df = original_df.copy() # Prevents the function from modifying
    # the original DataFrame
    df['metric_x_weight'] = df[metric_col] * df[weight_col]
    weighted_mean = (df['metric_x_weight'].sum() /
    df[weight_col].sum())
    return weighted_mean
```

Let’s try putting this function into action by calculating the average transaction amount using `df_average_spending_by_weekday`:

```
weighted_mean(df_average_spending_by_weekday,  
               metric_col='amount',  
               weight_col='transactions')
```

```
np.float64(12.65563577250256)
```

This average matches the weighted average that we calculated within `df_average_spending_by_level`.

Incidentally, because average transaction amounts are relatively constant across weekdays, simply averaging all ‘amount’ values within `df_average_spending_by_weekday` will get us very close to the actual average (despite the considerable variation in transaction counts by weekday). These kinds of ‘believable’, yet incorrect values are particularly insidious: they may go unnoticed for quite a while, whereas an obviously incorrect value (e.g. an average transaction amount of -5 or 83,000) would get caught right away.

```
df_average_spending_by_weekday['amount'].mean()
```

```
np.float64(12.621212399856491)
```

We were able to calculate correct averages within these pre-baked tables because we also knew the number of transactions within each row. In the real world, though, such sample size information may not be available.

From a data analysis perspective, it would be ideal to have all of your source data in microdata (rather than pre-baked) form. However, the microdata approach has its own drawbacks.

First, data privacy needs may preclude the issuance of microdata. Imagine, for instance, that NVCU released a dataset that contained individual course grades for each student along with those students’ majors. If you happened to be the only music major who took a C++ course, anyone with that knowledge could find out how you performed in the class.

A more private approach, in this case, would be to release separate ‘pre-baked’ tables that showed averages for courses and by major (but not averages for each course/major pair). (Even with this strategy, if a given pre-baked average was based on only a few students, it might be best to remove that row’s data so as to protect those students’ privacy.)

Second, microdata can take up much more storage size than pre-baked data. To illustrate this, let’s compare the amount of memory, in kilobytes, used by `df_transactions` (a microdata-based table) with that used by our two pre-baked tables:

```
microdata_kb = df_transactions.memory_usage(  
    index = True, deep = True).sum() / 1000  
# For more on df.memory_usage(), see  
# https://pandas.pydata.org/pandas-docs/stable/reference/api/  
# pandas.DataFrame.memory_usage.html  
microdata_kb
```

```
np.float64(2829.21)
```

```
pre_baked_kb = (df_average_spending_by_level.memory_usage(  
    index = True, deep = True).sum() +  
    df_average_spending_by_weekday.memory_usage(  
        index = True, deep = True).sum()) / 1000  
pre_baked_kb
```

```
np.float64(1.033)
```

```
microdata_kb / pre_baked_kb
```

```
np.float64(2738.828654404647)
```

The microdata table takes up over 2,700 times more memory than our two pre-baked tables combined! Therefore, it's understandable that data providers may prefer to share pre-calculated averages rather than original datasets.

7.5 Handling missing data when creating pivot tables

If a field passed to the `index` or `columns` argument of a `pivot_table()` call has a missing value, that missing value won't get incorporated into the final table. This can cause calculation errors if you end up using the pivot table for further analyses. However, mitigating this issue isn't too difficult.

To demonstrate this issue, I'll create a 'faulty' version of `df_transactions` in the following cell that has `np.nan` (i.e.. missing) entries for all 'Wednesday' weekday values and all 'So' level values. As you'll see, these missing values will cause issues when we (1) create a pivot table of this data, then (2) attempt to use that pivot table to determine the sum of all transactions in our dataset.

```
# Passing the 'level' and 'weekday' fields as the final arguments
# within these function calls allows these fields' original
# values to get retained should the conditions in the first
# arguments not be met. (For instance, in the first np.where()
# call, Wednesday values will get replaced with missing entries,
# but non-Wednesday values will get replaced with themselves
# (thus leaving them unchanged).

df_transactions_faulty = df_transactions.copy()
df_transactions_faulty['weekday'] = np.where(
    df_transactions_faulty['weekday'] == 'We',
    np.nan, df_transactions_faulty['weekday'])
df_transactions_faulty['level'] = np.where(
    df_transactions_faulty['level'] == 'So',
    np.nan, df_transactions_faulty['level'])

df_transactions_faulty.head()
```

	starting_year	weekday	level	amount	transactions
0	2023	NaN	Fr	13.36	1
1	2023	Tu	Se	12.22	1
2	2023	NaN	Se	23.10	1
3	2023	Su	Fr	18.78	1
4	2023	Tu	Fr	11.36	1

First, let's try converting this dataset into a pivot table that shows total transaction amounts by week.

```
df_transactions_faulty_pivot = df_transactions_faulty.pivot_table(
    index='weekday', values='amount',
    aggfunc='sum').reset_index()
df_transactions_faulty_pivot
```

	weekday	amount
0	Fr	31370.64
1	Mo	45576.94
2	Sa	8675.77
3	Su	14347.11
4	Th	49918.38
5	Tu	67141.08

There's no sign whatsoever of the 'Wednesday' rows. This shouldn't be too surprising (since we removed those values), but it's worth highlighting that no 'Missing' or 'N/A' row gets returned by the pivot table function.

The complete absence of the 'Wednesday' row makes this missing data issue easier to identify. However, many cases of missing data are far more insidious. For instance, suppose only a handful of 'Wednesday' entries were missing. We'd see a 'Wednesday' row within the pivot table, but we might not realize that its 'amount' sum was incorrect. (For this reason, it's often a good idea to check for missing entries within any fields that you pass to the `index` or `column` arguments of a `pivot_table()` call.)

If we try to use this pivot table to calculate the sum of all our transactions, we'll end up with an incorrect result, since rows with missing 'weekday' values won't factor into the calculation.

```
df_transactions_faulty_pivot['amount'].sum()
```

```
np.float64(217029.9199999998)
```

For reference, here's the correct sum (that also includes Wednesday transactions):

```
df_transactions['amount'].sum()
```

```
np.float64(284156.99)
```

Thankfully, it's not too hard to prevent this issue. We simply need to fill in missing values within any fields passed to the `index` or `columns` arguments of the `pivot_table()` call.

I'll demonstrate two ways to replace these values with a 'Missing' string. First, if you don't wish to permanently modify your original DataFrame, you can call `.fillna()` on the DataFrame being passed to the `pivot_table()` function. This ensures that rows with missing data get incorporated into the pivot table but leaves the original DataFrame untouched.

The following code implements this approach via a *dict comprehension*. (For more on this valuable tool, see <https://docs.python.org/3/tutorial/datastructures.html#dictionaries>.) This approach allows me to use the same list of `index` values that I'll pass to `pivot_table()` to specify which columns' missing entries should get filled in. This results in cleaner and less error-prone code, as if I needed to update my index values twice, once for `pivot_table()` and once for `fillna()`, the two lists could easily get out of sync.

The pivot table we'll create in the following cell will show total transactions by both level and weekday, but a similar method would work for tables that show sums for only one of these two groups.

(Another option, by the way, would be to fill missing entries within *all* columns with a 'Missing' string; this could be accomplished via `df_transactions_faulty.fillna('Missing')`. However, if some columns with missing data use float or integer types, this approach could either raise a warning or an error—as you'd be attempting to pass a string into a list of missing values. Thus, I recommend using the more precise solution shown below.)

```
index = ['level', 'weekday'] # These values will get passed to the
# index argument of our pivot_table() call.
# The following dict comprehension will produce the dictionary
# {'level': 'Missing', 'weekday': 'Missing'}--which, when passed to
```

(continues on next page)

(continued from previous page)

```
# fillna(), will instruct Pandas to fill in missing data with 'Missing'
# only within those two fields.
# (For more on this argument, see
# https://pandas.pydata.org/pandas-docs/stable/reference/
# api/pandas.DataFrame.fillna.html )
df_transactions_corrected_pivot = df_transactions_faulty.fillna(
    {field:'Missing' for field in index}).pivot_table(
    index=index, values='amount',
    aggfunc='sum').reset_index()
df_transactions_corrected_pivot.head()
```

	level	weekday	amount
0	Fr	Fr	13183.14
1	Fr	Missing	28856.95
2	Fr	Mo	20214.29
3	Fr	Sa	3739.85
4	Fr	Su	6230.97

Filling in missing ‘level’ and ‘weekday’ rows with ‘Missing’ allowed them to appear within our updated pivot table. As a result, we can now produce an accurate sum of all transactions by adding up the ‘amount’ rows within this table:

```
df_transactions_corrected_pivot['amount'].sum()
```

```
np.float64(284156.99)
```

The following approach, unlike the one shown above, permanently replaces missing weekday and level values with ‘Missing’, thus saving you the trouble of repeating this step later in your code. It also iterates through each field in the `index` list via a `for` loop in order to inform you which columns, in particular, have missing data.

In some cases, it may be preferable for your code to halt upon finding missing data; this halt serves as an alert that missing data exists, thus prompting you to fill in those missing entries with their actual values. The commented-out code above the first print statement in the following cell would stop this script’s operation (by raising a `ValueError`) if it came across any missing data.

```
for val in index:
    if df_transactions_faulty[val].isna().sum() > 0:
        # raise ValueError(f"{val} has NaN values; address these before \
        # running the following pivot table operation.")
        print(f"{val} has NaN values; these will be filled with 'Missing' \
so that their rows' data can still get incorporated into the following \
pivot table.\n")
        df_transactions_faulty[val] = df_transactions_faulty[val].fillna(
            'Missing').copy()
        # If a column with missing data was float-based rather than
        # string-based, it would be best to enter a float-based missing
        # data indicator rather than this string.

df_transactions_corrected_pivot = df_transactions_faulty.pivot_table(
    index=index, values='amount',
    aggfunc='sum').reset_index()
df_transactions_corrected_pivot
```

```
level has NaN values; these will be filled with 'Missing' so that their rows' data  
can still get incorporated into the following pivot table.
```

```
weekday has NaN values; these will be filled with 'Missing' so that their rows'  
data can still get incorporated into the following pivot table.
```

```
level  weekday      amount  
0      Fr        Fr  13183.14  
1      Fr    Missing  28856.95  
2      Fr        Mo  20214.29  
3      Fr        Sa  3739.85  
...    ...      ...  
24     Se        Sa   1260.53  
25     Se        Su   2377.25  
26     Se        Th   7017.18  
27     Se        Tu  10613.20  
  
[28 rows x 3 columns]
```

This new version of `df_transactions_corrected_pivot` also lets us calculate an accurate sum of all transactions within the original dataset:

```
df_transactions_corrected_pivot['amount'].sum()
```

```
np.float64(284156.99)
```

I'll now undo my corrections to `df_transactions_faulty` in order to prepare the dataset for the following section.

```
df_transactions_faulty.replace(  
    'Missing', np.nan, inplace=True)  
df_transactions_faulty
```

```
starting_year  weekday  level      amount  transactions  
0            2023    NaN    Fr    13.36          1  
1            2023    Tu     Se   12.22          1  
2            2023    NaN    Se   23.10          1  
3            2023    Su     Fr   18.78          1  
...          ...      ...      ...      ...  
22449        2023    Tu     Fr    5.98          1  
22450        2023    Fr     NaN   7.39          1  
22451        2023    Sa     Fr   13.25          1  
22452        2023    Mo     Fr   1.95          1  
  
[22453 rows x 5 columns]
```

7.6 The advantages of `map()` and `np.select()` over `np.where()` when missing data are present

Suppose we'd like to find the total number of dining hall transactions that took place on the weekend versus those that didn't. To make this calculation easier, we can add a 'weekday' column that will store a value of 1 if a transaction fell on Saturday or Sunday and 0 otherwise.

We could try initializing this column via `np.where()`. The following cell uses this function to assign a 'weekend' value of 0 to all transactions whose 'weekday' value corresponds to Monday, Tuesday, Wednesday, Thursday, or Friday. Transactions that don't have one of these weekday values are classified as weekend sales.

```
df_transactions_faulty['weekend'] = np.where(
    df_transactions_faulty['weekday'].isin(
        ['Mo', 'Tu', 'We', 'Th', 'Fr']), 0, 1)
df_transactions_faulty
```

	starting_year	weekday	level	amount	transactions	weekend
0	2023	NaN	Fr	13.36	1	1
1	2023	Tu	Se	12.22	1	0
2	2023	NaN	Se	23.10	1	1
3	2023	Su	Fr	18.78	1	1
...
22449	2023	Tu	Fr	5.98	1	0
22450	2023	Fr	NaN	7.39	1	0
22451	2023	Sa	Fr	13.25	1	1
22452	2023	Mo	Fr	1.95	1	0

[22453 rows x 6 columns]

The issue with this approach, as you might have recognized already, is that it classifies all transactions with missing 'weekday' values as taking place during the weekend. We know that this is inaccurate: the transactions with missing weekday entries actually took place on Wednesday. (In real life, of course, we wouldn't know this to be the case—but we still wouldn't want to assume that they fell on the weekend.)

Here is the total number of Saturday and Sunday dining hall transactions according to our 'weekend' column:

```
len(df_transactions_faulty.query("weekend == 1"))
```

7148

As it turns out, this sum much higher than the correct value (which we can calculate using our original transactions table):

```
len(df_transactions.query("weekday in ['Sa', 'Su']"))
```

1831

Let's now overwrite this faulty 'weekend' column by using `map()` instead. This function will allow us to map each individual weekday value to either 1 (for weekend) entries or 0 (for non-weekend values). Importantly, it also lets us map NaN entries to a third number, -1, which will represent missing values.

```
df_transactions_faulty['weekend'] = (
    df_transactions_faulty['weekday'].map(
```

(continues on next page)

(continued from previous page)

```
{'Su':0, 'Mo':0, 'Tu': 0, 'We':0, 'Th':0, 'Fr':0, 'Sa':1, 'Su':1,
 np.nan:-1}})
df_transactions_faulty
```

	starting_year	weekday	level	amount	transactions	weekend
0	2023	NaN	Fr	13.36	1	-1
1	2023	Tu	Se	12.22	1	0
2	2023	NaN	Se	23.10	1	-1
3	2023	Su	Fr	18.78	1	1
...
22449	2023	Tu	Fr	5.98	1	0
22450	2023	Fr	NaN	7.39	1	0
22451	2023	Sa	Fr	13.25	1	1
22452	2023	Mo	Fr	1.95	1	0

[22453 rows x 6 columns]

If we apply value_counts() to this column, we'll find that our weekend transactions count matches that shown above. The sum for the -1 entry shows the number of transactions that are missing

```
df_transactions_faulty['weekend'].value_counts()
```

```
weekend
0    15305
-1    5317
1    1831
Name: count, dtype: int64
```

I could have also chosen to use 'Missing' as my marker for invalid data; however, since the other values output by map () were integers, I wanted to make the missing data code an integer also.

As an aside, the use of 1 for weekend transactions and 0 for work-week ones makes it easy to determine the percentage of transactions that took place over the weekend:

```
print(f'{round(100*df_transactions_faulty.query(
    "weekend != -1")["weekend"].mean(), 2)}% of transactions with valid \
weekday entries took place on a Saturday or Sunday.')
```

10.69% of transactions with valid weekday entries took place on a Saturday or
Sunday.

Note the 'with valid weekday entries' caveat in the above print statement. Since we wouldn't know for sure on which weekdays the missing transactions took place, we wouldn't want our statement to make any assertions about them.

It was also crucial to exclude rows with weekend values of -1 from our above calculation. If we had kept them in, we would have ended up with a nonsensical percentage of weekend transactions:

```
print(f'{round(100 * df_transactions_faulty["weekend"].mean(), 2)}% \
of transactions took place on a Saturday or Sunday . . . wait, really? \
I think I need to double-check these numbers . . . '}')
```

-15.53% of transactions took place on a Saturday or Sunday . . . wait, really? I ↴think I need to double-check these numbers . . .

If you're categorizing continuous rather than categorical data, the `map()` approach above will be hard to implement—as you might end up having to code a ton of values. Therefore, you should consider using `np.select()` instead. This function generally involves a bit more code than `map()` for categorical data, but it easily accommodates continuous values as well.

Let's say that we want to add a 'large_purchase' flag to each row that will equal 1 if the amount was at or above the 90th percentile and 0 otherwise. We could then use this flag to determine each level's likelihood of making such a large purchase.

The 90th percentile can be calculated as follows:

```
large_purchase_threshold = df_transactions['amount'].quantile(0.9)
# See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.quantile.html
large_purchase_threshold
```

`np.float64(24.067999999999994)`

The following code will simplify `df_transactions_faulty` by removing some columns that won't be needed for the following section. It will also introduce missing numerical data by replacing all transactions whose final digit is a 6 with NaN.

```
df_transactions_faulty = df_transactions.copy().drop(
    ['weekday', 'transactions'],
    axis=1)

df_transactions_faulty['amount'] = np.where(
    df_transactions_faulty['amount'].astype('str').str[-1] == '6',
    np.nan, df_transactions_faulty['amount'])

df_transactions_faulty.head()
```

	starting_year	level	amount
0	2023	Fr	NaN
1	2023	Se	12.22
2	2023	Se	23.10
3	2023	Fr	18.78
4	2023	Fr	NaN

We wouldn't want to use `np.where()` to initialize our 'large_purchase' column, as we would inadvertently group missing transactions as large or non-large. In addition, `map()` would be a poor solution, as we now have thousands of unique values to code rather than just a few:

```
len(df_transactions_faulty['amount'].unique())
```

3131

Therefore, we'll instead use `np.select()`. This function applies a `condlist` (a list of possible conditions) and a `choicelist` (a list of values to apply for each of those conditions) in order to initialize or update a given column. (You don't have to name these items `condlist` and `choicelist`, but those are their corresponding parameter names; see

7.6. The advantages of `map()` and `np.select()` over `np.where()` when missing data are present

<https://numpy.org/doc/stable/reference/generated/numpy.select.html>). Importantly, this function also has a `default` argument that lets you determine what value to enter if none of the conditions were met.

The following code applies `np.select()` by defining two possible conditions (e.g. a transaction is above the threshold or it isn't) and two corresponding values to enter within our 'large_purchase' column (1 or 0). It also adds a 'default' value of -1; this value will get added to the large_purchase column when a given transaction amount is missing. (Note that, if no default value is specified, numpy will add values of 0—which would easily get mistaken here for the 'not a large purchase' condition.)

```
condlist = [
    df_transactions_faulty['amount'] >= large_purchase_threshold,
    df_transactions_faulty['amount'] <= large_purchase_threshold
]

choicelist = [1,
              0] # Make sure that the order of these entries matches
# that of your condlist entries!

df_transactions_faulty['large_purchase'] = np.select(
    condlist, choicelist, default=-1)
df_transactions_faulty
```

	starting_year	level	amount	large_purchase
0	2023	Fr	NaN	-1
1	2023	Se	12.22	0
2	2023	Se	23.10	0
3	2023	Fr	18.78	0
...
22449	2023	Fr	5.98	0
22450	2023	So	7.39	0
22451	2023	Fr	13.25	0
22452	2023	Fr	1.95	0

[22453 rows x 4 columns]

Now that we've added in this column, we can create a pivot table that shows the likelihood, for each level, that a given transaction was at or above the 90th percentile:

```
df_transactions_faulty.query("large_purchase != -1").pivot_table(
    index = 'level',
    values = 'large_purchase', aggfunc = 'mean').reset_index()
```

level	large_purchase
Fr	0.000000
Ju	0.297705
Se	0.434444
So	0.081307

This likelihood is evidently much higher for upperclassmen (juniors and seniors) than for underclassmen (freshmen and sophomores). Of course, this analysis isn't perfect, as it's limited to the rows for which we have valid transaction amounts.

7.7 Conclusion

These two chapters have offered a brief introduction—and certainly not a comprehensive reference—to the use of Pandas to calculate descriptive statistics within Python. My hope is that, now that you've seen how easily Pandas can perform various statistical calculations, you'll be inspired to test this library out for other tasks as well.

I have found in my own work that, while descriptive stats are less ‘trendy’ than more recent developments in the field of statistics (e.g. machine learning, neural networks, etc.), they’re still incredibly useful as a way to share, both internally and externally, how your nonprofit is performing and how those whom it serves are doing. These kinds of descriptive analyses will also serve as the foundation for many different visualizations, as we’ll see later in Python for Nonprofits.

Now that we’ve learned how to retrieve, reformat, and analyze data, we can move on to a task (importing Census data) that applies each of these skills.

CENSUS DATA IMPORTS

The US Census Bureau is a fantastic source of free demographic data. As this notebook will demonstrate, Python allows us to easily access large amounts of this data at once. In this script, we'll apply the Census Bureau's API to (1) calculate state- and county-level population growth data and (2) gather data on education levels and median incomes that will prove useful for future regression analyses.

(Note: I had initially planned to feature this script right after PFN's Data Retrieval chapter. However, this notebook not only imports Census data, but also reformats it and generates some descriptive statistics. Therefore, I decided it would be best to wait to cover this subject until now.)

8.1 An introduction to the American Community Survey

Many Americans probably associate the US Census Bureau with its decennial Census. However, the Census Bureau also conducts the American Community Survey (<https://www.census.gov/data/developers/data-sets/acs-5year.html>) each year, making it an ideal resource for recent demographic data.

This notebook will source data from the American Community Survey's 5-year estimates, which show an average of results for the past 5 years. (For example, the 2021 ACS5 dataset shows results between 2017 and 2021). The 1-year ACS estimates (<https://www.census.gov/data/developers/data-sets/acs-1year.html>) offer results for a more recent timeframe; however, because the 5-year estimates are sourced from a larger pool of data, they may be more reliable (especially for smaller regions). In addition, 1-year estimates aren't available for certain regions, such as counties with smaller populations and zip codes.

For the sake of brevity, I'll often refer to the American Community Survey's 5-year estimates as the 'ACS5' survey.

Another great source of Census data, which allows for a much wider variety of analyses, is American Community Survey *microdata*. These microdata tables show individual respondent rows rather than pre-calculated estimates, thus allowing for all sorts of custom filters, comparisons, and statistical tests. You can download 1- and 5-year ACS microdata from the Census website (<https://www.census.gov/data/developers/data-sets/census-microdata-api.html>); however, I would recommend using IPUMS (<https://usa.ipums.org/usa/>) for microdata retrieval, as it makes comparisons across different time periods much easier.

This script will not feature microdata imports, but if you plan to use Census data within your own work, you'll definitely want to learn more about the benefits (and disadvantages) of microdata. The US Census guide at https://www.census.gov/content/dam/Census/library/publications/2021/acs/acs_pums_handbook_2021.pdf is a good place to start. One notable disadvantage of microdata files is that, for confidentiality reasons, they do not include county- or zip-level information (though IPUMS has been able to map certain microdata records to certain US counties).

Finally, I'll provide a brief introduction to the two main data-gathering tasks that we'll tackle within this script:

8.2 Deciding where to move to start a family

Let's say that some NVCU seniors are interested in settling down and raising a family a few years after they graduate. Because they'd prefer to live in a growing region rather than a declining one, they want to know which areas have seen the highest growth rates in recent years. They'd like to see this data both for all residents within each county *and* those aged 25-29.

In order to answer these questions, we'll use the Census API to retrieve US county population growth data from the ACS5 for a selected set of years. We'll then use this data to calculate population growth rates across multiple periods.

8.3 Examining connections between education levels and median incomes

The NVCU admissions department would like to make the case to high schoolers on the fence about pursuing higher education that going to college is (generally) worth it from a financial perspective. Therefore, they would like you to determine how the median income for a given county and state increases as the percentage of residents with at least a bachelor's degree increases.* They figure that a strong relationship between the two could help convince teenagers to pursue a higher education.**

We can shed some light on these questions by performing a series of linear regressions—but first, we'll need to obtain Census data that can get incorporated into those regressions.

*The monetary value of a bachelor's degree can be better estimated by simply comparing the median incomes of state/county residents by different education levels. Therefore, this notebook will retrieve these medians as well. However, the continuous nature of the '% with bachelor's' metric will make it a great fit for our linear regression example.

**There are of course many non-monetary benefits to a college education—but given the size of student loans these days, it doesn't hurt to look into the financial rewards of a bachelor's degree.

8.4 Getting started with the Census API

We'll first import a few relevant libraries and set several configuration variables:

```
import time
program_start_time = time.time()
import pandas as pd
pd.set_option('display.max_columns', 1000) # This max column setting
# will prevent columns in the notebook from being hidden; however,
# if render_for_pdf (defined in the Appendix's helper_funcs.py file)
# is set to True, this setting will get overwritten.
import numpy as np
from iteration_utilities import duplicates

import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, render_for_pdf
display_type = config_notebook(display_max_columns=5,
                                display_max_rows=6)

acs5_year = 2021 # By updating this variable when future American
# Community Surveys get released, you should be able to retrieve the most
# recent data possible. (If changes to the survey's format are made,
```

(continues on next page)

(continued from previous page)

```
# however, updates to the scripts may be necessary.)

# Note: I had originally set acs5_year to 2022, the latest year for which
# ACS5 data were available at the time. However, due to a recent change
# in Connecticut's county-equivalent boundaries (see
# https://www.federalregister.gov/documents/2022/06/06/2022-12063/
# change-to-county-equivalents-in-the-state-of-connecticut for more
# information), ACS5 population growth data between previous
# years and 2022 appeared to be unavailable for that state. Therefore,
# I chose to retrieve data for 2021 instead.

acs5_latest_year = 2023 # This variable will be used to determine
# the ACS5 year for which a dataset containing only one year of data
# will be retrieved. Since this dataset won't show historical trends
# in data, we don't need to worry about changes in county definitions;
# thus, we can use the most recent year for which data are available.

download_new_variable_list = False # If set to True, a new list of
# variables will be downloaded from the Census API website. If False,
# this list of variables will instead be read in from a local .csv copy
# (thus saving processing time).
```

8.4.1 Importing a Census API Key

You can obtain a free Census API key at https://api.census.gov/data/key_signup.html. The following cell imports my own personal key, so you'll need to replace this code with one that loads in your own API key.

```
with open ('census_api_key_path.txt') as file:
    key_path = file.read()
with open(key_path) as file:
    key = file.read()
```

In order to get better acquainted with the Census API, you may want to review its documentation. For instance, you'll probably find the Census Data API User Guide (<https://www.census.gov/content/dam/Census/data/developers/api-user-guide/api-guide.pdf>) to be helpful in applying the Census API.

The list of ACS5 API call examples at (<https://api.census.gov/data/2021/acs/acs5/examples.html>) is another great resource. One of the sample URLs shown on this page for retrieving county-level data appears as follows:

https://api.census.gov/data/2021/acs/acs5?get=NAME,B01001_001E&for=county:*&key=YOUR_KEY_Goes_Here

If you replace the 'YOUR_KEY_Goes_Here' component of the URL with your actual key, then enter this link into your web browser, you'll receive a very long list of counties, population values, and state and county codes. The top of the list for the 2021 ACS5 looks like this:

```
[["NAME", "B01001_001E", "state", "county"],
["Autauga County, Alabama", "58239", "01", "001"],
["Baldwin County, Alabama", "227131", "01", "003"],
["Barbour County, Alabama", "25259", "01", "005"],
["Bibb County, Alabama", "22412", "01", "007"],
["Blount County, Alabama", "58884", "01", "009"],
["Bullock County, Alabama", "10386", "01", "011"],
```

'B01001_001E' refers to the total population estimates for a given county. We can find this out by going to the 2021 ACS5's Detailed Tables page (<https://api.census.gov/data/2021/acs/acs5/variables.html>) and navigating to the row with

a ‘Name’ value of ‘B01001_001E’. This link, which may take a little while to fully load, is available on the ACS5 API Documentation Page (<https://www.census.gov/data/developers/data-sets/acs-5year.html>).

We can use `pd.read_json()` to easily read this same data into a DataFrame:

```
df_results = pd.read_json(  
    f'https://api.census.gov/data/{acs5_year}/\\  
    acs/acs5?get=NAME,B01001_001E&for=county:*&key={key}')  
# read_json documentation:  
# https://pandas.pydata.org/pandas-docs/stable/reference/api/  
# pandas.read_json.html  
df_results.head()
```

	0	1	2	3
0	NAME	B01001_001E	state	county
1	Autauga County, Alabama	58239	01	001
2	Baldwin County, Alabama	227131	01	003
3	Barbour County, Alabama	25259	01	005
4	Bibb County, Alabama	22412	01	007

At this point, the DataFrame’s columns are [0, 1, 2, 3], whereas the columns we want to use are stored within the first row. The following code sets these row values as our column values, then deletes this row:

```
df_results.columns = df_results.iloc[0]  
df_results.drop(0, inplace=True)  
  
df_results.head()
```

	NAME	B01001_001E	state	county
1	Autauga County, Alabama	58239	01	001
2	Baldwin County, Alabama	227131	01	003
3	Barbour County, Alabama	25259	01	005
4	Bibb County, Alabama	22412	01	007
5	Blount County, Alabama	58884	01	009

8.5 Importing custom Census functions

Now that we’ve seen some basic applications of the Census API, we’ll import four Census data import and analysis functions that will help simplify some more complex procedures. I originally defined these functions within this script; however, I chose to move them to a separate Python file (‘census_import_scripts.py’) so that they can be incorporated more easily into other data projects. I highly recommend that you read the documentation for these files as well so that you can better understand what the functions are doing (and how to modify them for your own applications if needed).

```
from census_import_scripts import download_variable_list,  
create_variable_aliases, retrieve_census_data, create_comparison_fields
```

8.6 Retrieving variable and group information

In order to determine which variable codes to enter into our script, we'll first need to review a list of all American Community Survey variables and the overall groups into which they fit. The variable lists for each ACS5 are available on the Census website; for instance, the copy for 2021 is available at <https://api.census.gov/data/2021/acs/acs5/variables.html>. However, we can also use the `download_variable_list()` function we just imported to save this list (and its corresponding list of variable groups) to a local .csv file, as shown below.

```
if download_new_variable_list == True: # This process can take a little
    # while, so if you already have a copy of the variable list you need,
    # consider setting download_variable_list to False.
    download_variable_list(acss5_year, 'acs5')

# Reading the group and variable datasets into our script:
df_variables = pd.read_csv(
    f'Datasets/acs5_{acss5_year}_variables.csv')

df_groups = pd.read_csv(
    f'Datasets/acs5_{acss5_year}_groups.csv')

# The following code will show a condensed set of columns if the script
# is being run for PDF output; that way, the reader can view all of the
# most important columns. This step isn't necessary when the script is
# being run for HTML display, as the user can simply scroll horizontally
# to see all of the columns within the DataFrame.
df_variables[['Name', 'Label', 'Concept', 'Group']].head() if (
    render_for_pdf == True) else df_variables.head()
```

	Name	Label	Concept	Group
0	B01001_001E	Estimate!!Total:	SEX BY AGE	B01001
1	B01001_002E	Estimate!!Total:!!Male:	SEX BY AGE	B01001
2	B01001_003E	Estimate!!Total:!!Male:!!Under 5 years	SEX BY AGE	B01001
3	B01001_004E	Estimate!!Total:!!Male:!!5 to 9 years	SEX BY AGE	B01001
4	B01001_005E	Estimate!!Total:!!Male:!!10 to 14 years	SEX BY AGE	B01001

For reference, here's a look at `df_groups`: (this DataFrame is a simplified version of `df_variables` that shows each group code and corresponding concept only once.)

```
df_groups.head()
```

	Concept	Group
0	SEX BY AGE	B01001
1	SEX BY AGE (WHITE ALONE)	B01001A
2	SEX BY AGE (BLACK OR AFRICAN AMERICAN ALONE)	B01001B
3	SEX BY AGE (AMERICAN INDIAN AND ALASKA NATIVE ...	B01001C
4	SEX BY AGE (ASIAN ALONE)	B01001D

In order to find variables of interest, I recommend first searching for keywords of interest within the group table (which is much smaller in size) in order to identify relevant group IDs. Next, you can search for those group IDs inside the variables table in order to find the exact metrics to request from the Census API.

The following table stores variables for three separate groups: (1) the total population; (2) all males aged 25 to 29 years; and (3) all females aged 25 to 29 years. (The B01001 table that stores these values didn't have an entry for all people aged 25 to 29; we'll get around this limitation by retrieving sex-specific population totals within this age group, then adding those totals together.)

```
grad_destinations_variable_list = [
    'B01001_001E', 'B01001_011E',
    'B01001_035E']
```

8.7 Creating aliases and specifying retrieval years

The demographic columns in the Census API's output are labeled with their variable names (e.g. 'B01001_001E'). These names are concise, but you'll need a copy of the original variable list to interpret them. Therefore, I chose to replace these column names with a combination of the 'Label', 'Concept', and 'Name' entries in the original variable list. These column names are very long, but they do make the output easier to interpret (while also preserving the original names for reference).

In addition, if the description corresponding to a variable name happens to change from one year to another, the use of aliases will help you identify that change. (This will help prevent you from treating two different data types that happened to use the same variable code in different years as equal.)

The following function assists with this replacement by creating a dictionary whose keys are the original field names (e.g. 'B0101_001E') and whose values are the replacement names (e.g. 'Sex by Age_Estimate!!Total:_B01001_001E').

Creating our aliases:

```
grad_destinations_alias_dict = create_variable_aliases(
    df_variables=df_variables,
    variable_list=grad_destinations_variable_list)
grad_destinations_alias_dict
```

```
{'B01001_001E': 'SEX BY AGE_Estimate!!Total: (B01001_001E)',
 'B01001_011E': 'SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E)',
 'B01001_035E': 'SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)
 ↵'}
```

Next, we'll define a list of years for which we would like to retrieve Census data. In order to make this code easier to use in future years, I'll define these years as an offset of `acs5_year` rather than hardcoding them.

```
years_to_retrieve = [acs5_year - 12, acs5_year - 10,
                     acs5_year - 8,
                     acs5_year - 6,
                     acs5_year - 5,
                     acs5_year]
# American Community Survey 1-year estimates weren't available in 2020,
# so you'll want to remove that year from your list if it happens to be
# present. You can do so via the following code:
# if 2020 in years_to_retrieve:
#     years_to_retrieve.remove(2020)
# However, because I decided to use 5-year rather than 1-year estimates,
# I commented out this line.
years_to_retrieve
```

```
[2009, 2011, 2013, 2015, 2016, 2021]
```

At this point, it's a good idea to confirm that our three variable codes ('B01001_001E', 'B01001_011E', and 'B01001_035E') had the same meaning for all the years whose data we'll be retrieving. We can do so by running the following code, which retrieves these variables and their corresponding descriptions for all of the years in `years_to_retrieve`.

(Due to the size of the variables.html page, this code can take a while to run, so I commented it out below.)

```
# var_meanings_by_year_df_list = []
# for year in years_to_retrieve:
#     df_var_list = pd.read_html(
#         f'https://api.census.gov/data/{year}/acs/acss5/variables.html')[0][['Name', 'Label', 'Concept']].query(
#             "Name in @grad_destinations_variable_list")
#     df_var_list.insert(0, 'Year', year)
#     var_meanings_by_year_df_list.append(df_var_list)
# df_var_meanings_by_year = pd.concat(
#     [df for df in var_meanings_by_year_df_list])
# df_var_meanings_by_year.to_csv('var_meanings_by_year.csv', index=False)
# df_var_meanings_by_year
```

Here's a series of outputs from the saved .csv copy of this table that confirms that these codes had the same meaning in each of the years we're analyzing:

```
df_var_meanings_by_year = pd.read_csv(
    'var_meanings_by_year.csv')
for name in grad_destinations_variable_list:
    print(df_var_meanings_by_year.query("Name == @name"))
```

	Year	Name	Label	Concept
0	2009	B01001_001E	Estimate!!Total	SEX BY AGE
3	2011	B01001_001E	Estimate!!Total	SEX BY AGE
6	2013	B01001_001E	Estimate!!Total	SEX BY AGE
9	2015	B01001_001E	Estimate!!Total	SEX BY AGE
12	2016	B01001_001E	Estimate!!Total	SEX BY AGE
15	2021	B01001_001E	Estimate!!Total:	SEX BY AGE
	Year	Name	Label	Concept
1	2009	B01001_011E	Estimate!!Total!!Male!!25 to 29 years	SEX BY AGE
4	2011	B01001_011E	Estimate!!Total!!Male!!25 to 29 years	SEX BY AGE
7	2013	B01001_011E	Estimate!!Total!!Male!!25 to 29 years	SEX BY AGE
10	2015	B01001_011E	Estimate!!Total!!Male!!25 to 29 years	SEX BY AGE
13	2016	B01001_011E	Estimate!!Total!!Male!!25 to 29 years	SEX BY AGE
16	2021	B01001_011E	Estimate!!Total:!!Male:!!25 to 29 years	SEX BY AGE
	Year	Name	Label	Concept
2	2009	B01001_035E	Estimate!!Total!!Female!!25 to 29 years	SEX BY AGE
5	2011	B01001_035E	Estimate!!Total!!Female!!25 to 29 years	SEX BY AGE
8	2013	B01001_035E	Estimate!!Total!!Female!!25 to 29 years	SEX BY AGE
11	2015	B01001_035E	Estimate!!Total!!Female!!25 to 29 years	SEX BY AGE
14	2016	B01001_035E	Estimate!!Total!!Female!!25 to 29 years	SEX BY AGE
17	2021	B01001_035E	Estimate!!Total:!!Female:!!25 to 29 years	SEX BY AGE

8.8 Calling retrieve_census_data()

The `retrieve_census_data()` defined within ‘census_import_scripts.py’ simplifies the process of retrieving data from the Census API. It also enables the user to rename variable fields (e.g. ‘B01001_001E’) with aliases for those fields (e.g. ‘Sex by Age_Estimate!!Total: (B01001_001E)’) if desired. In addition, it allows more than 50 variables to be retrieved at the same time, thus simplifying the process of importing especially large datasets.

We’re now ready to retrieve our population totals for the years referenced in `years_to_retrieve`. We’ll do so by calling `retrieve_census_data()` for each of these years via a for loop, then adding their respective DataFrames together using `pd.concat()`.

(Note: Only 5825 results showed up when I requested ACS1 county-level data (as opposed to over 25,000 results for ACS5 data), so many counties were not getting incorporated within the ACS1 results. Therefore, the ACS5 will usually be the better of these two surveys for evaluating county-level growth.)

```
census_data_by_year_df_list = []
for year in years_to_retrieve:
    df_data = retrieve_census_data(
        survey='acs5', year=year,
        region='county',
        variable_list=grad_destinations_variable_list,
        rename_data_fields=True,
        field_vars_dict=grad_destinations_alias_dict, key=key)
    census_data_by_year_df_list.append(df_data)
df_growth_data_by_year = pd.concat(df for df in census_data_by_year_df_list)
# Removing Puerto Rico from our list of results so as to focus only on counties
# and county equivalents within the 50 US states and DC:
df_growth_data_by_year.query("state != '72'", inplace=True)
df_growth_data_by_year
```

```
0      Year county ... \
1      2009 091 ...
2      2009 093 ...
3      2009 095 ...
...
3141  2021 041 ...
3142  2021 043 ...
3143  2021 045 ...

0      SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E) \
1                           634
2                           531
3                          3264
...
3141                         574
3142                         175
3143                         146

0      SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)
1                           574
2                           352
3                          3900
...
3141                         567
3142                         200
3143                         112

[18856 rows x 7 columns]
```

8.8.1 Analyzing this data

The following cell adds together male and female population totals in order to calculate the total number of 25- to 29-year-olds within each county for each year. It also simplifies the original total population column name in order to improve its readability.

```
df_growth_data_by_year['Total_Pop_25_to_29'] = (df_growth_data_by_year[
    'SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E)'] +
df_growth_data_by_year[
    'SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)'])
df_growth_data_by_year.rename(
    columns={'SEX BY AGE_Estimate!!Total: (B01001_001E)':'Total_Pop'},
    inplace=True)
df_growth_data_by_year.drop([
    'SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E)',
    'SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)'],
    axis=1, inplace=True)
```

df_growth_data_by_year

	Year	county	...	Total_Pop	Total_Pop_25_to_29
0	2009	091	...	19695	1208
1	2009	093	...	11641	883
2	2009	095	...	95330	7164
3	2021	041	...	20514	1141
3141	2021	043	...	7768	375
3142	2021	045	...	6891	258
3143	2021	045	...	6891	258

[18856 rows x 6 columns]

Next, we'll apply `pd.pivot()` to place all population totals for a given county within the same row, thus making future growth calculations easier.

```
df_growth_data_by_year_pivot = df_growth_data_by_year.copy().pivot(
    columns='Year', index=['NAME', 'county', 'state']).reset_index()
# The values could be named explicitly, but since pivot() will infer them
# automatically, there's no need to do so. I thus removed the following
# code from the pivot() call:
# values=['Total_Pop',
#         'Total_Pop_25_to_29']

df_growth_data_by_year_pivot.head()
```

Year	NAME	county	...	Total_Pop_25_to_29	2016	2021
0	Abbeville County, South Carolina	001	...	1226.0	1292.0	
1	Acadia Parish, Louisiana	001	...	4240.0	3705.0	
2	Accomack County, Virginia	001	...	1819.0	1907.0	
3	Ada County, Idaho	001	...	30646.0	33625.0	
4	Adair County, Iowa	001	...	362.0	411.0	

[5 rows x 15 columns]

We'll now call `to_flat_index()` in order to group all column data within the same row:

```
df_growth_data_by_year_pivot.columns = (
    df_growth_data_by_year_pivot.columns.to_flat_index())

df_growth_data_by_year_pivot.head()
```

```
          (NAME, ) (county, ) ... \
0  Abbeville County, South Carolina      001 ...
1          Acadia Parish, Louisiana      001 ...
2        Accomack County, Virginia      001 ...
3            Ada County, Idaho      001 ...
4        Adair County, Iowa      001 ...

(Total_Pop_25_to_29, 2016)  (Total_Pop_25_to_29, 2021)
0                  1226.0              1292.0
1                  4240.0              3705.0
2                  1819.0              1907.0
3                 30646.0             33625.0
4                  362.0               411.0

[5 rows x 15 columns]
```

Next, we'll convert the tuple-based columns created by `to_flat_index()` to string-based ones by executing a list comprehension. This list comprehension will convert tuples that contain both a string and an integer (e.g. ('Total_Pop', 2011)) to a single string with an underscore separating the two elements. Tuples with an empty second entry (e.g. ('county',)) will get replaced with just the first entry (e.g. 'county').

```
df_growth_data_by_year_pivot.columns = [
    column[0] + '_' + str(column[1]) if isinstance(column[1], int)
    else column[0] for column in df_growth_data_by_year_pivot.columns]
df_growth_data_by_year_pivot.head()
```

```
          NAME county ... Total_Pop_25_to_29_2016 \
0  Abbeville County, South Carolina      001 ...           1226.0
1          Acadia Parish, Louisiana      001 ...           4240.0
2        Accomack County, Virginia      001 ...           1819.0
3            Ada County, Idaho      001 ...           30646.0
4        Adair County, Iowa      001 ...            362.0

Total_Pop_25_to_29_2021
0                  1292.0
1                  3705.0
2                  1907.0
3                 33625.0
4                  411.0

[5 rows x 15 columns]
```

8.9 Adding in comparison fields

Now that we have population data for several years in a relatively easy-to-parse format, we can call `create_comparison_fields()` to calculate nominal and percentage changes in population between certain years—along with corresponding percentile and rank information. In the following cell, this function will be called twice to create growth metrics for both total county populations and residents in the 25-29 age range.

```
for field_var in ['Total_Pop', 'Total_Pop_25_to_29']:
    create_comparison_fields(
        df=df_growth_data_by_year_pivot,
        field_var=field_var,
        year_list=years_to_retrieve,
        field_year_separator='_')
```

```
df_growth_data_by_year_pivot.head(5)
```

	NAME	county	...	\
0	Abbeville County, South Carolina	001	...	
1	Acadia Parish, Louisiana	001	...	
2	Accomack County, Virginia	001	...	
3	Ada County, Idaho	001	...	
4	Adair County, Iowa	001	...	

	2016-2021 Total_Pop_25_to_29 % Change Rank	\
0	1357.0	
1	2812.0	
2	1426.0	
3	869.0	
4	588.0	

	2016-2021 Total_Pop_25_to_29 % Change Percentile	
0	56.829035	
1	10.506208	
2	54.632283	
3	72.365489	
4	81.311684	

[5 rows x 75 columns]

8.10 Ranking counties by population growth

We'll now sort this dataset to answer a few questions:

1. Which counties had the highest (and lowest) total population growth rates between 2016 and 2021? (We'll evaluate this question for both counties with at least 100,000 residents in 2016 and those with at least one million residents).
2. Which counties experienced the highest (and lowest) 25-to-29-year-old population growth rates?

First, we'll specify a few variables that will be used within our analyses:

```
range_for_sorting = 5
range_start_year = acs5_year - range_for_sorting
```

(continues on next page)

(continued from previous page)

```
total_pop_var_for_sorting = 'Total_Pop'
total_pop_sort_column = f'{range_start_year}-{acs5_year} \
{total_pop_var_for_sorting} % Change Rank'
```

```
young_pop_var_for_sorting = 'Total_Pop_25_to_29'
young_pop_sort_column = f'{range_start_year}-{acs5_year} \
{young_pop_var_for_sorting} % Change Rank'
```

```
total_pop_col_root = f'{range_start_year}-{acs5_year} \
{total_pop_var_for_sorting}' # The 'root' from which other column entries
# will derive
```

```
young_pop_col_root = f'{range_start_year}-{acs5_year} \
{young_pop_var_for_sorting}'
```

The following function will help condense the output of our analyses, thus making them more readable.

```
def gen_display_cols(col_root):
    '''This function specifies which DataFrame columns to display within a
    given output. It does so by adding various suffixes to a col_root
    value (e.g. '2016-2021 Total_Pop').'''
    return ['NAME', f'{col_root} Change', f'{col_root} % Change',
            f'{col_root} % Change Rank', f'{col_root} % Change Percentile']
```

```
total_pop_display_cols = gen_display_cols(total_pop_col_root)
young_pop_display_cols = gen_display_cols(young_pop_col_root)
total_pop_display_cols
```

```
['NAME',
 '2016-2021 Total_Pop Change',
 '2016-2021 Total_Pop % Change',
 '2016-2021 Total_Pop % Change Rank',
 '2016-2021 Total_Pop % Change Percentile']
```

8.10.1 Sorting counties with at least 100K residents in 2016 by their 2016-2021 total population growth rates:

```
total_pop_sort_column # The following two sets of output will be
# sorted by percentage change (rather than nominal change) in the total
# population between 2016 and 2021.
```

```
'2016-2021 Total_Pop % Change Rank'
```

```
df_growth_data_by_year_pivot.query(
    f"Total_Pop_{range_start_year} >= 100000").sort_values(
        total_pop_sort_column).dropna(subset=total_pop_sort_column)[
    total_pop_display_cols].reset_index(drop=True)
```

```

      NAME 2016-2021 Total_Pop Change \
0      Hays County, Texas             48887.0
1      Comal County, Texas           32023.0
2      Kaufman County, Texas        28315.0
...
588     Wayne County, North Carolina -6755.0
589     Hinds County, Mississippi   -13824.0
590     Robeson County, North Carolina -15740.0

      2016-2021 Total_Pop % Change 2016-2021 Total_Pop % Change Rank \
0          26.327779               10.0
1          25.776358               12.0
2          25.319682               13.0
...
588         ...                   ...
589         ...                   ...
590         -11.695993              3056.0

      2016-2021 Total_Pop % Change Percentile
0                  99.713467
1                  99.649793
2                  99.617956
...
588                 ...
589                 ...
590                 2.737982

[591 rows x 5 columns]

```

The three counties with the highest 2016-2021 growth rates were all in Texas. Hays, Comal, and Kaufman Counties are located within the Austin, San Antonio, and DFW metropolitan areas, respectively (according to Wikipedia).

8.10.2 Sorting counties with at least *one million* residents in 2016 by their 2016-2021 total population growth rates:

Combining `concat()` with `head()` and `tail()` will allow us to display the counties with the highest and lowest growth rates within the same DataFrame.

```

df_1m_pop_pct_changes = df_growth_data_by_year_pivot.query(
    f"Total_Pop_{range_start_year} >= 1000000").sort_values(
    total_pop_sort_column).dropna(
    subset=total_pop_sort_column).copy().reset_index()
pd.concat([df_1m_pop_pct_changes.head(), df_1m_pop_pct_changes.tail()])[total_pop_display_cols]

```

	NAME	2016-2021 Total_Pop	Change	\
0	Orange County, Florida	153894.0		
1	Travis County, Texas	119619.0		
2	Hillsborough County, Florida	121300.0		
...		
40	Cuyahoga County, Ohio	4957.0		
41	St. Louis County, Missouri	1422.0		
42	Los Angeles County, California	-37520.0		

(continues on next page)

(continued from previous page)

```

2016-2021 Total_Pop % Change 2016-2021 Total_Pop % Change Rank \
0 12.252170 94.0
1 10.418176 140.0
2 9.168147 180.0
...
40 ... 1468.0
41 0.142120 1543.0
42 -0.373068 1687.0

2016-2021 Total_Pop % Change Percentile
0 97.039160
1 95.574658
2 94.301178
...
40 ...
41 50.907354
42 46.322827

[10 rows x 5 columns]

```

8.11 Repeating these steps in order to compare 25-to-29-year-old population growth rates

```

df_growth_data_by_year_pivot.query(
    f"Total_Pop_{range_start_year} >= 100000").sort_values(
        young_pop_sort_column).dropna(subset=young_pop_sort_column) [
    young_pop_display_cols].reset_index(drop=True)

```

	NAME	2016-2021 Total_Pop_25_to_29	Change	\
0	Douglas County, Colorado		4368.0	
1	Comal County, Texas		2035.0	
2	Kaufman County, Texas		2062.0	
...	
588	St. Lawrence County, New York		-745.0	
589	Caddo Parish, Louisiana		-2048.0	
590	Orleans Parish, Louisiana		-6371.0	
	2016-2021 Total_Pop_25_to_29 % Change	\		
0		31.155492		
1		31.116208		
2		30.602553		
...	...			
588		-10.792409		
589		-11.088852		
590		-16.970327		
	2016-2021 Total_Pop_25_to_29 % Change	Rank	\	
0		148.0		
1		150.0		
2		152.0		
...	...			
588		2755.0		

(continues on next page)

(continued from previous page)

```

589                      2765.0
590                      2898.0

    2016-2021 Total_Pop_25_to_29 % Change Percentile
0                         95.319962
1                         95.256288
2                         95.192614
..
588                      12.320917
589                      12.002547
590                      7.768227

[591 rows x 5 columns]

```

Comal and Kaufman counties ranked #2 and #3, respectively, among counties with 100K+ residents for both total population growth *and* 25-29 year-old population growth between 2016 and 2021.

```

df_1m_pop_pct_changes = df_growth_data_by_year_pivot.query(
    f"Total_Pop_{range_start_year} >= 1000000").sort_values(
    young_pop_sort_column).dropna(
    subset=young_pop_sort_column).copy().reset_index()
pd.concat([df_1m_pop_pct_changes.head(), df_1m_pop_pct_changes.tail()])[young_pop_display_cols]

```

	NAME	2016-2021 Total_Pop_25_to_29	Change	\
0	Pima County, Arizona		9395.0	
1	Wayne County, Michigan		16073.0	
2	King County, Washington		22087.0	
..			...	
40	Miami-Dade County, Florida		-11062.0	
41	Montgomery County, Maryland		-4043.0	
42	New York County, New York		-14977.0	
	2016-2021 Total_Pop_25_to_29 % Change	\		
0		15.212850		
1		13.496289		
2		12.478743		
..		...		
40		-5.717180		
41		-6.120379		
42		-7.548168		
	2016-2021 Total_Pop_25_to_29 % Change	Rank	\	
0		490.0		
1		590.0		
2		643.0		
..		...		
40		2504.0		
41		2521.0		
42		2608.0		
	2016-2021 Total_Pop_25_to_29 % Change	Percentile		
0		84.431710		
1		81.248010		
2		79.560649		

(continues on next page)

(continued from previous page)

```
..          ...
40          20.312003
41          19.770774
42          17.000955

[10 rows x 5 columns]
```

Although it's interesting to see which counties had the very highest and lowest growth rates, it's hard to gain an understanding of nationwide population trends via these tables alone. We can get a more comprehensive view of such trends via *choropleth* maps, which will allow us to present all counties within an easy-to-interpret graphic. We'll create examples of these maps within the Mapping section of Python for Nonprofits.

In order to allow our mapping code to access this data, we'll first save it as a .csv file:

```
df_growth_data_by_year_pivot.to_csv(
    f'Datasets/grad_destinations_acs_county_data.csv',
    index=False)
```

8.11.1 Retrieving state-level population growth data

Although county-level population growth provides a fascinating look at within-state trends, the broader view that state-level data provides will also prove useful. Therefore, we'll now repurpose the functions, variables, and code that we used earlier to import county-level data in order to create a state-level equivalent of `df_growth_data_by_year`.

```
census_data_by_year_df_list = []
for year in years_to_retrieve:
    df_data = retrieve_census_data(
        survey='acs5', year=year,
        region='state',
        variable_list=grad_destinations_variable_list,
        rename_data_fields=True, field_vars_dict=grad_destinations_alias_dict,
        key=key)
    census_data_by_year_df_list.append(df_data)
df_state_growth_data_by_year = pd.concat(df for df in census_data_by_year_df_list)
df_state_growth_data_by_year.query("state != '72'", inplace=True)
df_state_growth_data_by_year
```

```
0   Year state ... \
1   2009   01 ...
2   2009   02 ...
3   2009   04 ...
...
49  2021   54 ...
50  2021   55 ...
51  2021   56 ...

0   SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E) \
1                           155174
2                           28543
3                           251627
...
49                          ...
50                          55187
                               189562
```

(continues on next page)

(continued from previous page)

```

51                               19511
0   SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)
1                               157476
2                               24663
3                               228420
..                            ...
49                               52671
50                               180360
51                               17813

[306 rows x 6 columns]

```

```

df_state_growth_data_by_year['Total_Pop_25_to_29'] = (
    df_state_growth_data_by_year[
        'SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E)'] +
    df_state_growth_data_by_year[
        'SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)])
df_state_growth_data_by_year.rename(
    columns={'SEX BY AGE_Estimate!!Total: (B01001_001E)':'Total_Pop'},
    inplace=True)
df_state_growth_data_by_year.drop(
    ['SEX BY AGE_Estimate!!Total:!!Male:!!25 to 29 years (B01001_011E)',
     'SEX BY AGE_Estimate!!Total:!!Female:!!25 to 29 years (B01001_035E)'],
    axis=1, inplace=True)

```

```
df_state_growth_data_by_year
```

	Year	state	NAME	Total_Pop	Total_Pop_25_to_29
0	2009	01	Alabama	4633360	312650
1	2009	02	Alaska	683142	53206
2	2009	04	Arizona	6324865	480047
..
49	2021	54	West Virginia	1801049	107858
50	2021	55	Wisconsin	5871661	369922
51	2021	56	Wyoming	576641	37324

```
[306 rows x 5 columns]
```

```

df_state_growth_data_by_year_pivot = (
    df_state_growth_data_by_year.copy().pivot(
        columns = 'Year', index = ['NAME', 'state']).reset_index())

df_state_growth_data_by_year_pivot.columns = (
    df_state_growth_data_by_year_pivot.columns.to_flat_index())

df_state_growth_data_by_year_pivot.columns = [
    column[0] + '_' + str(column[1]) if isinstance(column[1], int)
    else column[0] for column
    in df_state_growth_data_by_year_pivot.columns]

df_state_growth_data_by_year_pivot.head()

```

```

      NAME state ... Total_Pop_25_to_29_2016 Total_Pop_25_to_29_2021
0    Alabama    01 ...           319177            331435
1     Alaska    02 ...           61185             59218
2   Arizona    04 ...          456680            496623
3  Arkansas    05 ...          194179            197068
4 California    06 ...         2918435            2992210

[5 rows x 14 columns]

```

```

for field_var in ['Total_Pop', 'Total_Pop_25_to_29']:
    create_comparison_fields(
        df=df_state_growth_data_by_year_pivot,
        field_var=field_var,
        year_list=years_to_retrieve,
        field_year_separator='_')

```

Saving this dataset to a .csv file so that it too can be used as a basis for state-level choropleth maps:

```

df_state_growth_data_by_year_pivot.to_csv(
    f'Datasets/grad_destinations_acs_state_data.csv',
    index=False)

```

Creating a DataFrame that shows the 5 states with the highest and lowest total population growth rates during the latest 5 years in the dataset:

```

df_state_growth_data_by_year_pivot.sort_values(
    total_pop_sort_column, inplace=True)
df_state_growth_data_by_year_pivot.reset_index(drop=True, inplace=True)
pd.concat([
    df_state_growth_data_by_year_pivot.head(5),
    df_state_growth_data_by_year_pivot.tail(5)])[
total_pop_display_cols]

```

	NAME	2016–2021 Total_Pop	Change	2016–2021 Total_Pop	% Change	\
0	Idaho	176134			10.769540	
1	Utah	282943			9.596405	
2	Nevada	220066			7.751063	
..	
48	Mississippi	-22169			-0.741639	
49	Wyoming	-6388			-1.095657	
50	West Virginia	-45043			-2.439911	
	2016–2021 Total_Pop	% Change	Rank	2016–2021 Total_Pop	% Change	Percentile
0	1.0			100.000000		
1	2.0			98.039216		
2	3.0			96.078431		
..		
48	49.0			5.882353		
49	50.0			3.921569		
50	51.0			1.960784		

[10 rows x 5 columns]

Creating a similar DataFrame that displays 25-to-29-year-old population growth rates:

```
df_state_growth_data_by_year_pivot.sort_values(
    young_pop_sort_column, inplace=True)
df_state_growth_data_by_year_pivot.reset_index(drop=True, inplace=True)
pd.concat([
    df_state_growth_data_by_year_pivot.head(5),
    df_state_growth_data_by_year_pivot.tail(5)]) [
young_pop_display_cols]
```

	NAME	2016-2021 Total_Pop_25_to_29	Change	\
0	Utah		29737	
1	Washington		58422	
2	Tennessee		46477	
..	
48	Alaska		-1967	
49	Louisiana		-15558	
50	Wyoming		-3359	
		2016-2021 Total_Pop_25_to_29 % Change	\	
0		13.733051		
1		11.363760		
2		10.661207		
..		
48		-3.214840		
49		-4.581151		
50		-8.256520		
		2016-2021 Total_Pop_25_to_29 % Change Rank	\	
0		1.0		
1		2.0		
2		3.0		
..		
48		49.0		
49		50.0		
50		51.0		
		2016-2021 Total_Pop_25_to_29 % Change Percentile		
0		100.000000		
1		98.039216		
2		96.078431		
..		
48		5.882353		
49		3.921569		
50		1.960784		

[10 rows x 5 columns]

8.12 Retrieving data for our education/income regression analyses

The following code creates tables of populations, median incomes, median home values, and regions for each US county and state. This data can then get incorporated into regression analyses later in Python for Nonprofits.

8.12.1 Importing US Census data on educational attainment and median earnings

```
# Retrieving a new list of ACS5 variables for the most recent year with
# ACS5 data (as of February 2025):
if download_new_variable_list == True:
    download_variable_list(acss5_latest_year, 'acs5')

# Reading this dataset into our script:
df_variables_latest = pd.read_csv(
    f'Datasets/acss5_{acss5_latest_year}_variables.csv')

regression_variable_list = ['B01001_001E', 'B15003_001E', 'B15003_022E',
    'B15003_023E', 'B15003_024E', 'B15003_025E', 'B20004_001E',
    'B20004_002E', 'B20004_003E', 'B20004_004E', 'B20004_005E', 'B20004_006E']

regression_alias_dict = create_variable_aliases(
    df_variables=df_variables_latest,
    variable_list=regression_variable_list)

df_regression_data_county = retrieve_census_data(
    survey='acs5', year=acss5_latest_year,
    region='county',
    variable_list=regression_variable_list,
    rename_data_fields=True, field_vars_dict=regression_alias_dict,
    key=key)

df_regression_data_state = retrieve_census_data(
    survey='acs5', year=acss5_latest_year,
    region='state',
    variable_list=regression_variable_list,
    rename_data_fields=True, field_vars_dict=regression_alias_dict,
    key=key)

# Retrieving national-level data (which will be useful for another
# data analysis that I'll include within the regressions section
# of Python for Nonprofits):
df_regression_data_national = retrieve_census_data(
    survey='acs5', year=acss5_latest_year,
    region='us',
    variable_list=regression_variable_list,
    rename_data_fields=True, field_vars_dict=regression_alias_dict,
    key=key)

df_regression_data_state
```

	Year	state	...	\
1	2023	01	...	
2	2023	02	...	
3	2023	04	...	

(continues on next page)

(continued from previous page)

```

...     ...     ...
50 2023    55 ...
51 2023    56 ...
52 2023    72 ...

0 Median Earnings in the Past 12 Months (in 2023 Inflation-Adjusted Dollars) by_<u>
<u>Sex by Educational Attainment for the Population 25 Years and Over_Estimate!!</u>
<u>Total:!!Bachelor's degree (B20004_005E) \</u>
1                               59318
2                               68158
3                               65087
...
50                             ...
51                             64443
51                             58411
52                             28988

0 Median Earnings in the Past 12 Months (in 2023 Inflation-Adjusted Dollars) by_<u>
<u>Sex by Educational Attainment for the Population 25 Years and Over_Estimate!!</u>
<u>Total:!!Graduate or professional degree (B20004_006E)</u>
1                               72667
2                               90606
3                               82033
...
50                             ...
51                             78648
51                             73535
52                             41197

[52 rows x 15 columns]

```

The following code calculates the percentage of residents within a given row who have at least a bachelor's degree; it also dramatically condenses many column names. (I converted it into a function so that the same set of code could be used to modify our state-level, county-level, and national datasets.)

```

def rename_and_calc_dataset(df, year):
    '''This dataset renames certain median earnings columns within
    a copy of census results and also estimates the percentage of
    respondents within each row who have at least a bachelor's degree.

    df: the DataFrame containing census results to process.

    year: the year in which the ACS was conducted. (This year shows
          up in many median earnings columns.'''
    df.rename(columns={
        'Median Earnings in the \
        Past 12 Months (in 2023 Inflation-Adjusted Dollars) by Sex by Educational \
        Attainment for the Population 25 Years and Over_Estimate!!\\
        Total:!!Less than high school graduate (B20004_002E)': \
        'Median_Earnings_Less_Than_HS', \
        'Median Earnings in the Past 12 Months (in 2023 Inflation-Adjusted \
        Dollars) by Sex by Educational Attainment for the Population 25 Years \
        and Over_Estimate!!Total:!!High school graduate \
        (includes equivalency) (B20004_003E)':'Median_Earnings_HS', \
        "Median Earnings in the Past 12 Months (in 2023 Inflation-Adjusted \
        Dollars) by Sex by Educational Attainment for the Population 25 Years \
        and Over_Estimate!!Total:!!Bachelor's degree (B20004_005E)": \
        'Median_Earnings_Bachelors'
    })

```

(continues on next page)

(continued from previous page)

```
Dollars) by Sex by Educational Attainment for the Population 25 Years \
and Over_Estimate!!Total:!!Some college or associate's \
degree (B20004_004E) ":"Median_Earnings_Some_College",

"Median Earnings in the Past 12 Months (in 2023 Inflation-Adjusted \
Dollars) by Sex by Educational Attainment for the Population 25 \
Years and Over_Estimate!!Total:!!Bachelor's degree (B20004_005E) ":" \
"Median_Earnings_Bachelors_Degree",

"Median Earnings in the Past 12 Months (in 2023 \
Inflation-Adjusted Dollars) by Sex by Educational Attainment \
for the Population 25 Years and Over_Estimate!!Total:!!Graduate \
or professional degree (B20004_006E) ":"Median_Earnings_Postgraduate",
"Median Earnings in the Past 12 Months (in 2023 \
Inflation-Adjusted Dollars) by Sex by Educational Attainment \
for the Population 25 Years and Over_Estimate!!Total: (B20004_001E) ":" \
"Median_Earnings_for_Total_25plus_Population",

'Sex by Age_Estimate!!Total: (B01001_001E) ':'Total_Population',
},
    inplace=True)
df["Pct_With_Bachelors_Degree"] = (100*(
    df["Educational Attainment for the Population \
25 Years and Over_Estimate!!Total:!!Bachelor's degree (B15003_022E)"] +
    df["Educational Attainment for the Population 25 \
Years and Over_Estimate!!Total:!!Master's degree (B15003_023E)"] +
    df["Educational Attainment for the Population \
25 Years and Over_Estimate!!Total:!!Professional school degree \
(B15003_024E)"] +
    df["Educational Attainment for the Population 25 \
Years and Over_Estimate!!Total:!!Doctorate degree (B15003_025E)"])
    / df["Educational Attainment for the \
Population 25 Years and Over_Estimate!!Total: (B15003_001E)"])
```

```
rename_and_calc_dataset(df = df_regression_data_county,
                       year = acs5_latest_year)
rename_and_calc_dataset(df = df_regression_data_state,
                       year = acs5_latest_year)
rename_and_calc_dataset(df = df_regression_data_national,
                       year = acs5_latest_year)
df_regression_data_state
```

0	Year	state	...	Median_Earnings_Postgraduate	Pct_With_Bachelors_Degree
1	2023	01	...	72667	27.755139
2	2023	02	...	90606	31.242276
3	2023	04	...	82033	32.610332
...
50	2023	55	...	78648	32.758344
51	2023	56	...	73535	29.920304
52	2023	72	...	41197	29.130778

[52 rows x 16 columns]

8.12.2 Mapping each county and state to a geographical region

As part of our regression analysis, we'll want to determine whether the relationship between the prevalence of bachelor's degrees and median earnings is influenced by the region in which a particular county or state is located. I'll first merge in this data for our county-level dataset; next, I'll do the same for the state-level table.

In order to add region information to our county-level dataset, we'll first need to figure out the state corresponding to each county; we can then map these states to their respective regions. Our dataset already contains numerical values for each state in the form of FIPS codes; however, to make sense of these codes, we'll also need to import a table that shows the actual state name for each code. Such a table is available on the Census website at https://www2.census.gov/geo/docs/reference/codes2020/national_state2020.txt.

I copied and pasted the contents of this table into `state_fips_codes_from_census.txt`, which can be found in the same folder as this script. The following code reads this table into a DataFrame. Note that (1) you can apply `read_csv()` on more than just .csv files and that (2) because the entries in this table were separated by the '|' character, I needed to specify *that* character as my separator.

```
df_fips = pd.read_csv('state_fips_codes_from_census.txt',
                      sep='|')

# Renaming columns to help facilitate an upcoming merge:
df_fips.rename(columns={'STATE':'State_Abbrev',
                       'STATEFP':'state'}, inplace=True)

# Converting the 'state' column into a zero-filled string
# so that its values can get matched with the corresponding
# FIPS codes in df_regression_data_county:
df_fips['state'] = df_fips['state'].astype('str').str.zfill(2)
df_fips
```

	State_Abbrev	state	STATENS	STATE_NAME
0	AL	01	1779775	Alabama
1	AK	02	1785533	Alaska
2	AZ	04	1779777	Arizona
..
54	PR	72	1779808	Puerto Rico
55	UM	74	1878752	U.S. Minor Outlying Islands
56	VI	78	1802710	United States Virgin Islands

[57 rows x 4 columns]

```
df_regression_data_county = df_regression_data_county.merge(
    df_fips[['State_Abbrev', 'state']],
    on = 'state', how = 'left')
df_regression_data_county
```

	Year	county	...	Pct_With_Bachelors_Degree	State_Abbrev
0	2023	001	...	28.282680	AL
1	2023	003	...	32.797637	AL
2	2023	005	...	11.464715	AL
..
3219	2023	149	...	22.481068	PR
3220	2023	151	...	18.360524	PR
3221	2023	153	...	27.116371	PR

(continues on next page)

(continued from previous page)

```
[3222 rows x 18 columns]
```

Now that we know each county's state abbreviation, we can merge region data into our table as well. We'll do so by importing state_regions.csv, a file that stores the state-region relationships found in this handy PDF file: https://www2.census.gov/geo/pdfs/maps-data/maps/reference/us_regdiv.pdf

```
df_state_regions = pd.read_csv('state_regions.csv')
df_state_regions
```

```
   State_Abbrev      Region
0          WA        West
1          OR        West
2          CA        West
...
48         VT  Northeast
49         NH  Northeast
50         ME  Northeast
```

```
[51 rows x 2 columns]
```

```
df_regression_data_county = df_regression_data_county.merge(
    df_state_regions, on='State_Abbrev', how='left')
df_regression_data_county
```

```
      Year county  ...  State_Abbrev Region
0    2023     001  ...
1    2023     003  ...
2    2023     005  ...
...
3219  2023     149  ...
3220  2023     151  ...
3221  2023     153  ...
```

```
[3222 rows x 19 columns]
```

8.12.3 Merging region information into our state-level dataset

Although the state-level Census dataset contains full state names, I'll still use the same approach that I took for the county level dataset, as that reduces the amount of additional code that I need to write. As the following cell demonstrates, it's possible to chain multiple `merge()` calls together; this allows for more concise code, though if anything goes wrong, you may need to call each `merge` operation individually for debugging purposes.

```
df_regression_data_state = df_regression_data_state.merge(
    df_fips[['State_Abbrev', 'state']],
    on = 'state', how = 'left').merge(
    df_state_regions, on = 'State_Abbrev', how = 'left')
df_regression_data_state
```

```
      Year state  ...  State_Abbrev Region
0    2023     01  ...          AL    South
```

(continues on next page)

(continued from previous page)

```

1 2023 02 ...
2 2023 04 ...
...
49 2023 55 ...
50 2023 56 ...
51 2023 72 ...

[52 rows x 18 columns]

```

Saving these DataFrames to .csv files so that they can get incorporated into the regressions section of Python for Non-profits:

```

df_regression_data_county.to_csv(
    'Datasets/education_and_earnings_county.csv', index = False)

df_regression_data_state.to_csv(
    'Datasets/education_and_earnings_state.csv', index = False)

df_regression_data_national.to_csv(
    'Datasets/education_and_earnings_national.csv', index = False)

```

8.13 Appendix

8.13.1 1: The Census Python Library

It's worth noting that there is also a 'census' Python library (available via pypi and conda) that helps simplify the process of requesting API data. You may choose to use it for your own Census research, but I ended up not needing it for the data retrieval tasks shown above. In addition, foregoing the library allowed me to demonstrate how to retrieve data directly from an API, which you may find helpful when working with APIs that don't have a corresponding Python library.

Here's an example of the Census library in use:

```

## Example of reading data from the Census library into a
# Pandas DataFrame:
from census import Census
c = Census(key)
pd.DataFrame(c.acs5.get(('NAME', 'B01001_001E'),
{'for': 'county:*'}))

```

	NAME	B01001_001E	state	county
0	Autauga County, Alabama	58761.0	01	001
1	Baldwin County, Alabama	233420.0	01	003
2	Barbour County, Alabama	24877.0	01	005
...
3219	Villalba Municipio, Puerto Rico	21984.0	72	149
3220	Yabucoa Municipio, Puerto Rico	30313.0	72	151
3221	Yauco Municipio, Puerto Rico	33988.0	72	153

[3222 rows x 4 columns]

8.13.2 2: The requests library

We can also use Python's *requests* library to retrieve data from the Census API, then convert it to JSON format:

```
# The following code borrows from the requests library documentation at
# https://docs.python-requests.org/en/latest/index.html
import requests
r = requests.get(f'https://api.census.gov/data/{acs5_year}/\
acs/acs5?get=NAME,B01001_001E&for=county:*&key={key}')
# Printing the first 300 characters of this output:
print("r.text:\n",r.text[0:300],'\n')
# Printing the first 5 lines of r.json:
print("r.json:\n",r.json()[0:5],'\n')
```

```
r.text:
[["NAME", "B01001_001E", "state", "county"],
 ["Autauga County, Alabama", "58239", "01", "001"],
 ["Baldwin County, Alabama", "227131", "01", "003"],
 ["Barbour County, Alabama", "25259", "01", "005"],
 ["Bibb County, Alabama", "22412", "01", "007"],
 ["Blount County, Alabama", "58884", "01", "009"],
 ["Bullock County, Ala

r.json:
[['NAME', 'B01001_001E', 'state', 'county'], ['Autauga County, Alabama', '58239',
 ↪'01', '001'], ['Baldwin County, Alabama', '227131', '01', '003'], ['Barbour
↪County, Alabama', '25259', '01', '005'], ['Bibb County, Alabama', '22412', '01',
 ↪'007']]
```

Converting our response to JSON allows it to be easily read into a Pandas DataFrame, as shown below:

```
pd.DataFrame(r.json()).head()
# Note that pd.DataFrame(r.text) would produce the following error:
# "ValueError: DataFrame constructor not properly called!"
```

	0	1	2	3
0	NAME	B01001_001E	state	county
1	Autauga County, Alabama	58239	01	001
2	Baldwin County, Alabama	227131	01	003
3	Barbour County, Alabama	25259	01	005
4	Bibb County, Alabama	22412	01	007

I included this approach in the appendix because you may find the *requests* library useful for other online data retrieval tasks. However, our use of `pd.read_json()` to import Census data rendered an explicit call to the *requests* library unnecessary.

8.13.3 3: Importing 51+ variables

I tested out the `retrieve_census_data()` function's ability to import data for 51 or more variables via the following code. Feel free to uncomment and run it yourself to test out this functionality.

```
# test_list = list(df_variables['Name'][0:151])

# test_alias_dict = create_variable_aliases(
#     df_variables=df_variables,
#     variable_list=test_list)

# test_acs5_data = retrieve_census_data(
#     survey='acs5', year=acs5_year, region='county',
#     variable_list=test_list,
#     rename_data_fields=True,
#     field_vars_dict=test_alias_dict, key=key)

# test_acs5_data
```

```
program_end_time = time.time()
run_time = round(program_end_time - program_start_time, 3)
print(f"Finished running script in {run_time} seconds.")
```

Finished running script in 13.447 seconds.

8.14 Conclusion

We imported a great deal of data within this script! Although we performed a few basic analyses as well, the real value of the data we retrieved will be manifested within the Mapping and Regressions sections of Python for Nonprofits,

Next, I'll introduce the '`census_import_scripts.py`' file, whose functions helped simplify much of the import and analysis code found within this section. I do recommend reading through its documentation also, as that will help you modify its functions for your own use cases.

8.15 census_import_scripts.py

```
# Census Data Import Scripts

# This Python file stores functions for creating lists of American
# Community Survey variables; generating aliases for variable codes;
# using the Census API to download data; and adding certain statistical
# fields to tables.

import pandas as pd
from iteration_utilities import duplicates

def download_variable_list(year, survey):
    '''This function imports a list of all variables from the Census
    website, thus allowing variable codes to get mapped to names in
    subsequent analyses.
```

(continues on next page)

(continued from previous page)

```
year: the year for which to retrieve variables.

survey: the ACS type for which to retrieve variables (e.g.
'acs5' or 'acs1' for the 5-year and 1-year ACS estimates,
respectively.)
'''

print(f"Importing {survey} variables from {year}.")
df_variables_page = pd.read_html(
    f'https://api.census.gov/data/\n{year}/acs/{survey}/variables.html')[0]
# [0] selects the first HTML table found on this page.
# See https://pandas.pydata.org/pandas-docs/stable/reference/api/
# pandas.read_html.html
# for more information on pd.read_html().

# Some rows in this table contain items other than demographic
# variables (e.g. region names). We can exclude them by selecting
# only rows that begin with 'Estimate'. (Another option would have
# been to filter out rows with N/A 'Group' entries (i.e.
# df_variables.query("Group.isna() == False")),
# but this would have left a couple non-variable rows in place.

df_variables = df_variables_page[
    df_variables_page['Label'].str[0:8] == 'Estimate'].copy()
    .reset_index(drop=True)
# Removing an extraneous column from our output
if 'Unnamed: 8' in df_variables.columns:
    df_variables.drop('Unnamed: 8', axis=1, inplace=True)
# Saving this table to a local .csv file:
df_variables.to_csv(f'Datasets/{survey}_{year}_variables.csv',
                     index=False)

# Given that there are tens of thousands of individual variables
# within the ACS, it could take a very long time to identify
# the items you'd like to retrieve from this dataset.
# The following code makes this search process somewhat easier by
# creating a separate *groups* table that shows only unique group
# names and their written descriptions (e.g. 'Sex by Age').

df_groups = df_variables.drop_duplicates(
    ['Group'])[['Concept', 'Group']].copy()
    .reset_index(drop=True)
df_groups.head()
df_groups.to_csv(f'Datasets/{survey}_{year}_groups.csv',
                 index=False)
print(f"Finished saving variable and group tables to .csv files.")

def create_variable_aliases(df_variables, variable_list):
    '''This function creates a dictionary whose keys are
    the original 'Name' values (e.g. 'B001_001E') within a variable
    list on the Census API website and whose values are the replacement
    names (e.g. 'Sex by Age_Estimate!!Total:_B01001_001E').
    This resulting dictionary can then be passed to a df.rename() call
```

(continues on next page)

(continued from previous page)

```

within retrieve_census_data() in order to make the output of that
function easier to interpret.

df_variables: A DataFrame containing a list of Census variables. For
an example of this list for the 2021 American Community Survey (5-Year
Estimates), visit:
https://api.census.gov/data/2021/acs/acs5/examples.html .

variable_list: The list of variables to rename
(e.g. ['B01001_001E', 'B01001_002E']).
'''

# Creating a DataFrame that contains the information needed for the
# updated column names:
df_aliases = df_variables.query(
    "Name in @variable_list")[['Name', 'Label', 'Concept']].copy()
# Creating a new 'Description' column that will replace the original
# output field names:
df_aliases['Description'] = (df_aliases['Concept']
    + '_' + df_aliases['Label']
    + ' (' + df_aliases['Name'] + ')')
# Creating a dictionary whose keys are the original field names and
# whose values are the new 'Description' entries that were
# just created:
alias_dict = df_aliases.set_index('Name').to_dict()['Description']
# See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\_dict.html
return alias_dict

```



```

def retrieve_census_data(survey, year, region, key, variable_list,
                        rename_data_fields=False,
                        field_vars_dict={}):
    '''This function retrieves data from the US
    Census API.

    survey: the survey from which to retrieve data. The only arguments
    currently supported are 'acs5' and 'acs1' (for the American Community
    Survey 5-Year and 1-Year estimates, respectively).

    year: the year for which you wish to retrieve survey data. Note that,
    When region is set to 'acs5', the survey results will include data
    for the 5 years leading up to (and including) the 'year' argument.
    (For example, if you set 'year' to 2021, you'll retrieve ACS5 data
    from 2017 to 2021 (inclusive).)

    region: The geographic level at which you wish to retrieve data.
    Examples include 'us', 'state', 'county', 'zip', 'msa'
    (for metropolitan/micropolitan statistical area data), and 'csa'
    (for combined statistical area data); however, other regions are
    supported as well. Consult your survey's API examples page for
    other options.
    (For instance, if you wanted to
    retrieve data by urban area within the 2021 ACS5, you could go to
    https://api.census.gov/data/2021/acs/acs5/examples.html, then search
    for 'urban area.' The Urban Area URL ends with

```

(continues on next page)

(continued from previous page)

```
'&for=urban%20area:&key=YOUR_KEY_Goes_Here'. Therefore, you'd want to  
use 'urban%20area' as your 'region' argument.)
```

(Note: 'zip' will retrieve results by Zip Code Tabulation Area, which are similar to (but not identical to) # zip codes. See # https://en.wikipedia.org/wiki/ZIP_Code_Tabulation_Area # for more information.

variable_list: The list of variables for which to retrieve data. This function allows for the retrieval of data for more than 50 variables; it does so by making multiple calls to the Census API, then merging the results of those calls together.

key: your personal Census API key.

rename_data_fields: set to True to replace column names in your dataset with new entries of your choice.

field_vars_dict: A dictionary that stores the original variable names retrieved by the Census (e.g. 'B01001_001E' as keys and your desired replacements as values. Example:

```
{'B01001_001E': 'Sex by Age_Estimate!!Total:_B01001_001E',  
 'B01001_002E': 'Sex by Age_Estimate!!Total:!!Male:_B01001_002E'}'.  
 I suggest that you use the output of a create_variable_aliases() call  
 as the argument for this parameter.
```

'''

```
# Using the iteration_utilities library to check for duplicate  
# values within variable_list (which could cause issues later on):  
# The following code is based on  
# https://iteration-utilities.readthedocs.io/en/latest/generated/#\_duplicates.html  
duplicate_variables = list(duplicates(variable_list))
```

```
if len(duplicate_variables) > 0:  
    raise ValueError(f"The following variables appear more than once \\  
in your variable list: {duplicate_variables}")
```

```
if survey == 'acs5':  
    survey_string = 'acs/acss5'
```

```
elif survey == 'acs1':  
    survey_string = 'acs/acss1'
```

```
else:  
    raise ValueError("This survey type is not currently supported by \\  
the function.")
```

```
# Converting simplified region names into strings that the Census API  
# will recognize:  
if region == 'zip':  
    region = 'zip%20code%20tabulation%20area' # Based on  
    # the ZCTA example within
```

(continues on next page)

(continued from previous page)

```

# https://api.census.gov/data/2021/acs/acs5/examples.html

if region == 'csa':
    region = 'combined%20statistical%20area'

if region == 'msa':
    region = 'metropolitan%20statistical\
%20area/micropolitan%20statistical%20area'

# Only 50 variables can be retrieved from the Census API at a time
# using the approach shown in this function. The following code
# accommodates this limitation by splitting variable_list into
# sublists of up to 45 variables. The data retrieved for the variables
# in these sublists will then get merged back together.
# (45 variables are retrieved at a time instead of 50 because certain
# geographic variables, such as 'NAME', also appear to count toward
# the 50-variable limit--and more than one geographic variable
# may be present depending on which region type was selected.)

i = 0

while i < len(variable_list): # i.e. while there
    # are still more variables to iterate through
    variable_sublist = variable_list[i:i+45] # This line reads the
    # next 45 variables from variable_list into a sublist that can
    # then be\ passed to the API
    # print("variable_sublist:", variable_sublist)
    # Converting the list of variables into a string that can be
    # passed to the API call:
    # (The Census API guide at
    # https://www.census.gov/content/dam/Census/data/developers/
    # api-user-guide/api-guide.pdf
    # demonstrates how to call multiple census variables at once.)
    variable_string = ','.join(variable_sublist)
    # print("variable_string:",variable_string)

# Retrieving data via the Census API:
# This line was originally based on an example found in
# https://api.census.gov/data/2022/acs/acs5/examples.html .

# read_json documentation:
# https://pandas.pydata.org/pandas-docs/stable/reference/api/
# pandas.read_json.html

api_url = f'https://api.census.gov/data/{year}/\
{survey_string}?get=NAME,{variable_string}&for={region}:*&key={key}'
    # print(api_url)

df_results = pd.read_json(api_url)

# At this point, the DataFrame's columns are a list of integers;
# the desired column names are stored within the first row.
# The following code resolves this issue by setting these row
# values as the column values and then deleting this row.

```

(continues on next page)

(continued from previous page)

```
df_results.columns = df_results.iloc[0]
df_results.drop(0, inplace=True)

# Determining which merge keys to use when combining API results
# for different sublists together:
# This is made more complicated by the fact that results for
# different regions will have different identifier
# columns (e.g. 'NAME', 'county', and 'state' for county data but
# only 'NAME' and 'state' for state data). However, we can
# accommodate this behavior by simply initializing our list of
# merge keys as the set of all columns that are *not* also
# variable columns.
if i == 0: # This step only needs to be performed for our first
    # sublist of variables, since merge keys for other sublists
    # will be identical.
    merge_keys = list(set(df_results.columns)
                      - set(variable_sublist))
    # print("merge_keys:", merge_keys)

if i == 0: # Since this is the first set
    # of results, we can initialize df_combined_results
    # as a copy of df_results.
    df_combined_results = df_results.copy()
else: # Merging our latest set of results into df_results:
    df_combined_results = df_combined_results.merge(
        df_results, on=merge_keys,
        how='outer').copy()
    # Added .copy() here in response to a data fragmentation
# warning

i += 45 # Allows the function to iterate through the next
# 45 variables within variable_list

# Converting variable columns to numeric data types:
for column in variable_list:
    # print(f"Now converting {column} to a numeric type.")
    df_combined_results[column] = pd.to_numeric(
        df_combined_results[column])
    # pd.to_numeric() allows for either integer or float outputs
    # depending on the nature of the original data.
    # See https://pandas.pydata.org/pandas-docs/stable/reference/api/
    # pandas.to_numeric.html

# Replacing column names with aliases if requested:
if rename_data_fields == True:
    df_combined_results.rename(
        columns=field_vars_dict, inplace=True)

# The following for loop moves all of the merge keys (e.g. geographic
# identifiers) to the left side of the table. This is particularly
# useful when retrieving longer lists of variables, as otherwise,
# certain keys can get buried in the middle of the dataset.

# I think I got the idea of chaining .insert() and .pop() together
```

(continues on next page)

(continued from previous page)

```

# from a StackOverflow answer like this one from Marc Maxmeister :
# https://stackoverflow.com/a/77463008/13097194 )

for i in range(len(merge_keys)):
    df_combined_results.insert(
        i, merge_keys[i],
        df_combined_results.pop(merge_keys[i]))

# Adding a 'Year' column to the left of all existing DataFrame columns:
# (this will prove particularly
# helpful when comparing data from different years.)
df_combined_results.insert(0, 'Year', year)

return df_combined_results

def create_comparison_fields(df, field_var, year_list,
                             field_year_separator='_'):
    '''This function calculates nominal and percentage changes
    between the last year in a list and all years leading up to that year.
    It also calculates both rank and percentile information for these
    changes.

    df: the DataFrame that will be updated with comparisons between years.
    This function assumes that the fields within df that contain
    data for a particular year use the format
    {field_var}{field_year_separator}{latest_year} (e.g. 'Total_Pop_2009',
    'Total_Pop_2015', etc.).

    field_var: A string representing the variable
    whose values should be compared (e.g. 'Total_Pop' within the field
    'Total_Pop_2009').

    year_list: A list of years to compare. The function will compare
    all years leading up to the last year with the last year. For example,
    if year_list equals [2005, 2009, 2015], the function will create
    comparisons between (1) 2005 and 2015 and (2) 2009 and 2015 (but not
    2005 and 2009).
    year_list should be sorted chronologically in order to avoid
    unexpected/undesired outputs.

    field_var data for each of these years should be stored within
    the DataFrame. For instance, if year_list is equal to the example
    shown above, field_var is 'Total_Pop', and field_year_separator (see
    below) is '_', the script will expect to see the following fields
    within the DataFrame:
    'Total_Pop_2005', 'Total_Pop_2009', 'Total_Pop_2015'

    field_year_separator: The character (e.g. a space, an underscore,
    etc.) separating the field_var and year values within the DataFrame's
    fields.

    ...
    latest_year = year_list[-1]
    for year in year_list[:-1]: # E.g. for all years leading up
        # to (but not including) latest_year

```

(continues on next page)

(continued from previous page)

```
# Calculating the nominal change between the two years:  
df[f'{year}-{latest_year} {field_var} Change'] = (  
    df[f'{field_var}{field_year_separator}{latest_year}']  
    - df[f'{field_var}{field_year_separator}{year}'])  
  
# Calculating the percentage change:  
df[f'{year}-{latest_year} {field_var} % Change'] = 100*((  
    df[f'{field_var}{field_year_separator}{latest_year}']  
    / df[f'{field_var}{field_year_separator}{year}']) - 1)  
  
# Calculating ranks and percentiles for both the nominal  
# change and % change columns:  
# Note that field_var still needs to be included within these  
# columns in order to specify what change, exactly, is being  
# analyzed. (This is particularly important when this function  
# gets called for multiple field_var entries.)  
  
# Nominal change rank and percentile calculations:  
df[f'{year}-{latest_year} {field_var} Change Rank'] = df[  
    f'{year}-{latest_year} {field_var} Change'].rank(  
    ascending=False, method='min')  
  
df[f'{year}-{latest_year} {field_var} Change Percentile'] = (  
    100 * df[  
        f'{year}-{latest_year} {field_var} Change'].rank(  
        pct=True, ascending=True, method='max'))  
  
# Percentage change rank and percentile calculations:  
df[f'{year}-{latest_year} {field_var} % Change Rank'] = (  
    df[f'{year}-{latest_year} {field_var} % Change'].rank(  
    ascending=False, method='min'))  
  
df[f'{year}-{latest_year} {field_var} % Change Percentile'] = (  
    100 * df[  
        f'{year}-{latest_year} {field_var} % Change'].rank(  
        pct=True, ascending=True, method='max'))
```

Part IV

Visualizing Data

CREATING INTERACTIVE AND STATIC CHARTS WITH PLOTLY

By Kenneth Burchfiel

Released under the MIT License

Now that we've learned how to import, clean, and analyze data, we can finally turn to one of my favorite uses of Python: *visualizations*. In this section, we'll learn how to use the Plotly library to create bar and line graphs; scatter plots; histograms; and treemaps. Our main data source will be NVCU survey data, though we'll also make use of current enrollment data for our treemaps and bookstore transaction data for our scatter plots and histograms. We'll then save these charts to both interactive HTML files and static .png files.

When I was first learning Python, I mainly used Matplotlib for my charts. Matplotlib is a great library as well, but I switched over to Plotly for as my default chart library for several reasons:

1. Plotly makes it easy to produce both interactive and static versions of a given chart. These interactive versions, which work great on websites (but can also be opened as standalone HTML files), allow users to access more information than they could via static .png or .jpg copies.
2. Plotly integrates very well with Dash, a Python library that allows you to build interactive web apps within Python. (A later section of Python for Nonprofits will demonstrate how to build both simple and more complex Dash web apps).
3. My general sense is that Plotly tends to require less code to produce my desired result than does Matplotlib, although I don't have any hard data on which to base this point.

Note: this script will create graphs using Plotly Express (<https://plotly.com/python/plotly-express/>), which allows you to create detailed charts with relatively few lines of code. For more complex charts, you may need to instead work with Plotly *graph objects*, which allow for more customization: however, I'd recommend using Plotly Express code within your scripts when possible. After all, as the page linked to above notes, "Any figure created in a single function call with Plotly Express could be created using graph objects alone, but with between 5 and 100 times more code."

```
import pandas as pd
import numpy as np
import plotly.express as px

from sqlalchemy import create_engine
e = create_engine('sqlite:///../Appendix/nvcu_db.db')

import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook
display_type = config_notebook(display_max_columns=8)

from IPython.display import Image # Based on
# a StackOverflow answer from 'zach' at
# https://stackoverflow.com/a/11855133/13097194 .
```

9.1 Creating bar graphs

(For more details on bar charts in Plotly, consult the `px.bar()` documentation (<https://plotly.com/python/bar-charts/>)).

In order to create bar graphs of NVCU survey data, we'll first import the pivot tables of this data created within PFN's Descriptive Stats section, then feed them into our Plotly graphing script.

```
df_survey_results_by_college_long = pd.read_csv(  
    '../Descriptive_Stats/survey_results_by_college_long.csv')  
# Capitalizing all column names will allow our charts' x and y axis labels  
# to appear capitalized by default.  
df_survey_results_by_college_long.columns = [  
    column.title() for column in  
    df_survey_results_by_college_long.columns]  
df_survey_results_by_college_long
```

```
Starting_Year College Season Score  
0 2023 STB Fall 69.797119  
1 2023 STB Winter 64.406522  
.. ... ... ... ...  
10 2023 STM Winter 64.318689  
11 2023 STM Spring 76.639004  
  
[12 rows x 4 columns]
```

9.1.1 Simple bar graphs

We'll begin by creating a bar chart that shows spring survey results by college:

```
fig_spring_survey_results_by_college = px.bar(  
    df_survey_results_by_college_long.query(  
        "Season == 'Spring'"), x='College', y='Score',  
    title='Spring Survey Scores by College',  
    color='College', text_auto='.0f')  
# See https://plotly.com/python/bar-charts/  
  
# The text_auto argument allows you to easily add bar labels using  
# a format of your choice. '.0f' rounds labels to the nearest integer;  
# to show values with 2 decimal points, use '.2f' instead. You can also  
# convert proportions into percentages by replacing 'f' with '%'.  
  
fig_spring_survey_results_by_college if display_type == '.html' else None
```

Plotly charts are generated in HTML format by default; this lets them offer a range of interactive features, such as tooltips (extra information that appears when you hover over a chart element) and panning/zooming options. You can also filter the x axis elements by clicking once on a legend element to remove it and clicking twice to keep only that element.

These interactive charts can easily be saved as HTML files via the `write_html()` function:

```
fig_spring_survey_results_by_college.write_html(  
    'output/spring_results_by_college.html')
```

This .html file can now be opened in a web browser as a standalone document or embedded in a website for public display.

However, it's also convenient to create static copies of these charts (e.g. for use within a slide deck or a web page that doesn't allow certain HTML elements to be displayed). Thankfully, it's very simple to create static copies of these charts. We simply need to call `write_image()` instead of `write_html()`.

However, the default settings for `write_image()` produce, in my view, a relatively blurry image that isn't suitable for presenting. Therefore, the following cell include arguments for three optional parameters for `write_image()`: `height`, `width`, and `scale`. The settings shown below produce a chart with 4K resolution (3840 * 2160 pixels) whose text is also large enough to be easily readable.

```
height = 405
aspect_ratio = 16/9
width = height * aspect_ratio
scale = 2160 / height

# The above approach allows the height setting to automatically adjust
# the width and scale setting while preserving a resolution of
# 3840 * 2160.

# The chart's height and width (in pixels) will be height * scale
# and width * scale, respectively. Smaller height and width values will
# result in larger (and thus more readable) text, but in order to keep
# the resulting file sharp, I increased the scale to compensate. For
# instance, a height of 540 will result in a scale of 4 (2160 / 540);
# a height of 360, on the other hand, will result in a scale of 6.

# You may want to experiment with different settings in order to find the
# result that best meets your needs.
# The 16/9 aspect ratio is used in many modern TVs and monitors, thus
# making it ideal for full-screen displays of charts.

print("Height:",height, "\nWidth:", width, "\nScale:",scale)

fig_spring_survey_results_by_college.write_image(
    'output/spring_results_by_college.png', height=height,
    width=width, scale=scale)
# For more information about write_image, see:
# https://plotly.com/python-api-reference/generated/plotly.io.write\_image.html
```

```
Height: 405
Width: 720.0
Scale: 5.333333333333333
```

Note: in order for the above code to work, you'll most likely need to have installed the `kaleido` library beforehand. When using the `conda` package manager on Windows, I've found that I've needed to request version 0.1.0 of `kaleido` when installing the library (e.g. by calling `conda install python-kaleido=0.1.0` within my command prompt); this is because the later version appears to cause image generation scripts to hang indefinitely. (I found this solution via this GitHub post: <https://github.com/plotly/Kaleido/issues/134#issuecomment-1781183798>).

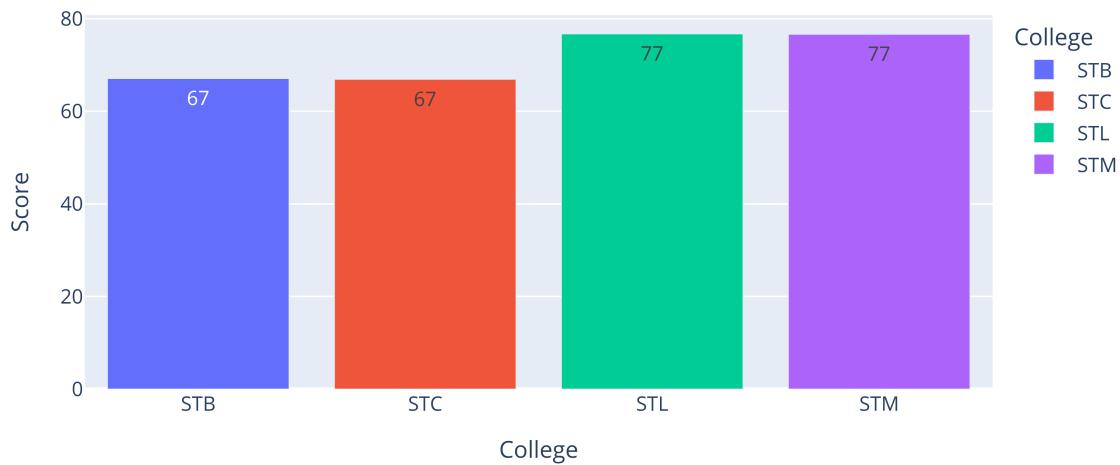
I haven't encountered this problem on Linux, so on that operating system (and presumably on Mac), `conda install python-kaleido` should work fine.

The `kaleido` library makes creating static versions of HTML-based Plotly graphs very straightforward. In the Mapping section of Python for Nonprofits, we'll go over a more manual approach for creating static versions of HTML files (as that section will utilize a library that doesn't yet have `kaleido` support).

To display this image within this notebook, we can call `Image()` from `IPython.display`:

```
Image('output/spring_results_by_college.png', width=720)
# Based on the StackOverflow answer from 'zach'
# at https://stackoverflow.com/a/11855133/13097194 ;
# See also:
# https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.
# html#IPython.display.Image
```

Spring Survey Scores by College



9.1.2 Importing a function to simplify the process of saving and displaying charts

We can more easily save our Plotly charts to HTML and PNG files, then display a copy of the output by importing a function that takes care of all three operations for us. This function, `wadi()`, will get applied within other sections of Python for Nonprofits also. I highly recommend reading its source code, which can be found inside the Appendix folder's `helper_funcs.py` file, so that you understand what the function does and how to modify it for your own needs.

```
from helper_funcs import wadi
```

9.1.3 More detailed bar graphs

The bar graph created above shows that STL and STM had better survey results than did STB and STC. However, NVCU's administrators will also want to know how each college's survey results changed over the course of the year. We can visualize these changes by adding `season` as our argument for the `color` variable within `px.bar()`, thus creating a chart that shows separate color-coded bars for each college/season pair. (We'll also add `barmode='group'` in order to display grouped bars rather than stacked ones.)

9.1.4 Sorting DataFrames using a custom key

Note that, because ‘Spring’ precedes ‘Winter’ alphabetically, these charts will show winter results before spring results by default. We can display them in chronological order instead by sorting our DataFrame to have winter rows precede spring ones.

The easiest way to accomplish this sort would be to first add a separate ‘season_sort_order’ column that contains values of 0, 1, and 2 for ‘Fall’, ‘Winter’, and ‘Spring’ rows, respectively. Once this column is in place, we could then sort the DataFrame by it. However, for demonstration purposes, the following cell shows how this sort can be accomplished without adding an additional column. The approach shown below relies on the `key` argument available within `sort_values()` together with Pandas’ `map()` function.

```
df_survey_results_by_college_long.sort_values(
    'Season', key=lambda col: col.map(
        {'Fall':0,'Winter':1,'Spring':2}), inplace=True)
# See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.
# DataFrame.sort_values.html
df_survey_results_by_college_long
```

	Starting_Year	College	Season	Score
0	2023	STB	Fall	69.797119
3	2023	STC	Fall	69.568665
..
8	2023	STL	Spring	76.727809
11	2023	STM	Spring	76.639004

[12 rows x 4 columns]

Now that the DataFrame is sorted correctly, we can create our grouped bar graph:

```
fig_survey_results_by_college_and_season = px.bar(
    df_survey_results_by_college_long,
    x='College', y='Score',
    title='Survey Scores by College and Season',
    color='Season', text_auto='.0f', barmode='group')
```

Calling `wadi` to save static and interactive copies of this chart, then display the format specified by `display_type`:

`display_type`

`'png'`

```
wadi(fig=fig_survey_results_by_college_and_season,
      file_path='output/results_by_college_and_season',
      display_type=display_type)
```

Survey Scores by College and Season

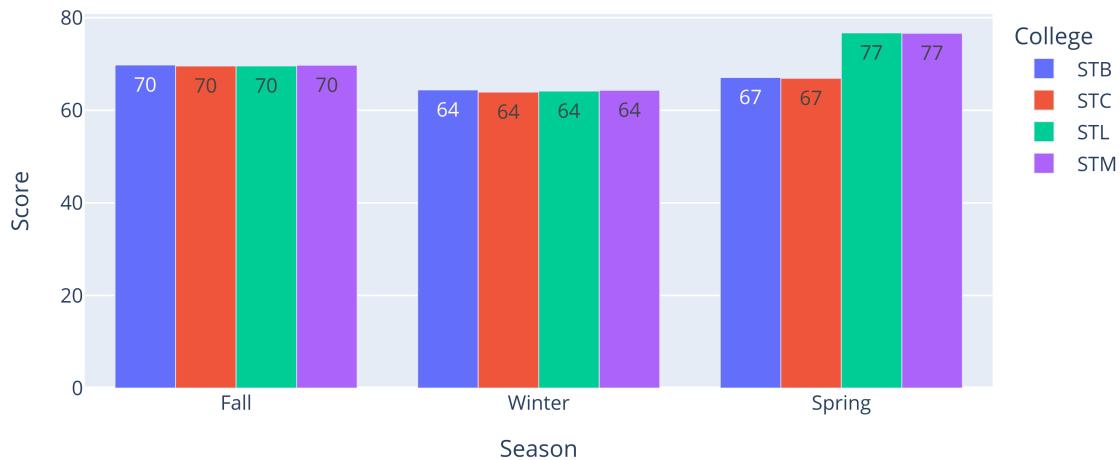


This graph clearly shows how each school's scores changed over time. However, in order to more easily compare results among schools for each season, we can switch the `x` and `color` arguments:

```
fig_survey_results_by_season_and_college = px.bar(
    df_survey_results_by_college_long,
    x='Season', y='Score',
    title='Survey Scores by Season and College',
    color='College', text_auto='.0f', barmode='group')

wadi(fig=fig_survey_results_by_season_and_college,
      file_path='output/results_by_season_and_college',
      display_type=display_type)
```

Survey Scores by Season and College



This chart demonstrates that each college had an average fall score around 70 followed by a 6-point drop to 64 during the winter. Although all colleges demonstrated growth from the fall to the winter, STL and STM clearly outperformed STB and STC (whose spring scores failed to match their fall scores).

9.1.5 Graphing wide data

The previous two graphs used long-formatted data, meaning each row showed only one metric (in this case, one survey score). However, you may often encounter (or choose to create) tables in which multiple metrics are found within the same row. These datasets would be considered ‘wide.’ Here’s an alternative to `df_survey_results_by_college_long` that uses a wide format:

```
df_survey_results_by_college_wide = pd.read_csv(
    '../Descriptive_Stats/survey_results_by_college_wide.csv')
df_survey_results_by_college_wide.columns = [
    column.title() for column in
    df_survey_results_by_college_wide.columns]
df_survey_results_by_college_wide.drop('Starting_Year', axis=1)
```

	College	Fall	Winter	Spring	Fall-Spring Change	\
0	STB	69.797119	64.406522	67.077551	-2.719568	
1	STC	69.568665	63.934845	66.911444	-2.657221	
2	STL	69.596675	64.146248	76.727809	7.131134	
3	STM	69.735685	64.318689	76.639004	6.903320	
		Fall-Winter Change	Winter-Spring Change			
0		-5.390596	2.671029			
1		-5.633820	2.976599			
2		-5.450427	12.581561			
3		-5.416995	12.320315			

This dataset contains the same score data as `df_survey_results_by_college_long`, but it places those each college’s seasonal scores on the same row. (This made it easier to calculate changes in scores over time, which this dataset also features.)

Here's another look at df_survey_results_by_college_long for comparison:

```
df_survey_results_by_college_long
```

```
Starting_Year College Season Score
0           2023    STB   Fall  69.797119
3           2023    STC   Fall  69.568665
...
8           2023    STL Spring 76.727809
11          2023   STM Spring 76.639004
[12 rows x 4 columns]
```

Plotly makes graphing wide data like this very straightforward. Instead of passing season to the 'color' argument (as we did with the long-formatted table), we can simply add each season to a list that will then serve as our y argument:

```
fig_survey_results_by_college_and_season_wide = px.bar(
    df_survey_results_by_college_wide, x='College',
    y=['Fall', 'Winter', 'Spring'], barmode='group',
    title='Survey Scores by College and Season',
    text_auto='.0f')
# See:
# https://plotly.com/python/bar-charts/#bar-charts-with-wide-format-data

# Because we are displaying multiple y values rather than just one, Plotly
# sets the y axis and legend titles as 'value' and 'variables' by default.
# We can change these to 'Score' and 'Season', respectively, via the
# following code:
fig_survey_results_by_college_and_season_wide.update_layout(
    yaxis_title='Score', legend_title='Season')
# This code was based on https://plotly.com/python/figure-labels/
wadi(fig=fig_survey_results_by_college_and_season_wide,
      file_path='output/results_by_college_and_season_wide',
      display_type=display_type)
```

Survey Scores by College and Season



Note that this graph, though based on a ‘wide’ rather than ‘long’ dataset, is equivalent to `fig_survey_results_by_college_and_season`.

9.1.6 Bar charts with three or more variables

If you need to differentiate between three different variables (such as college, level, and season), you have a few different options. One would be to represent the third variable via Plotly’s `pattern_shape` argument, which adds different patterns to bars to represent different variable values. Another would be to use `facet_row` or `facet_col` to create a separate plot for each value within this third variable. Finally, you could consider grouping two variables into one via Pandas, then passing that grouped column to `px.bar()`’s `x` or `color` parameters. The following code will demonstrate each of these three options.

Reading in a dataset that shows results by school, level, *and* college (hence the use of ‘slc’ in the DataFrame name):

```
df_survey_results_slc = pd.read_csv(
    '../Descriptive_Stats/survey_results_slc_long.csv')
df_survey_results_slc.columns = [
    column.title() for column in
    df_survey_results_slc.columns]

# In order for our graphs to display levels and seasons in the correct
# order, we'll add in a 'Season_For_Sorting' column (an equivalent for
# levels already exists), then sort the DataFrame by these two columns.
# (I'm using this approach rather than the key-based one shown earlier
# because I don't believe that that one works with more than one sort
# value.)

df_survey_results_slc['Season_For_Sorting'] = df_survey_results_slc[
    'Season'].map({'Fall':0,'Winter':1,'Spring':2})
df_survey_results_slc.sort_values(['Season_For_Sorting',
    'Level_For_Sorting'], inplace=True)
df_survey_results_slc.drop('Season_For_Sorting', axis = 1).head()
```

	Starting_Year	College	Level_For_Sorting	Level	Season	Score
0	2023	STB		0	Fr	Fall 69.593583
12	2023	STC		0	Fr	Fall 69.598913
24	2023	STL		0	Fr	Fall 69.585030
36	2023	STM		0	Fr	Fall 69.650503
3	2023	STB		1	So	Fall 70.377306

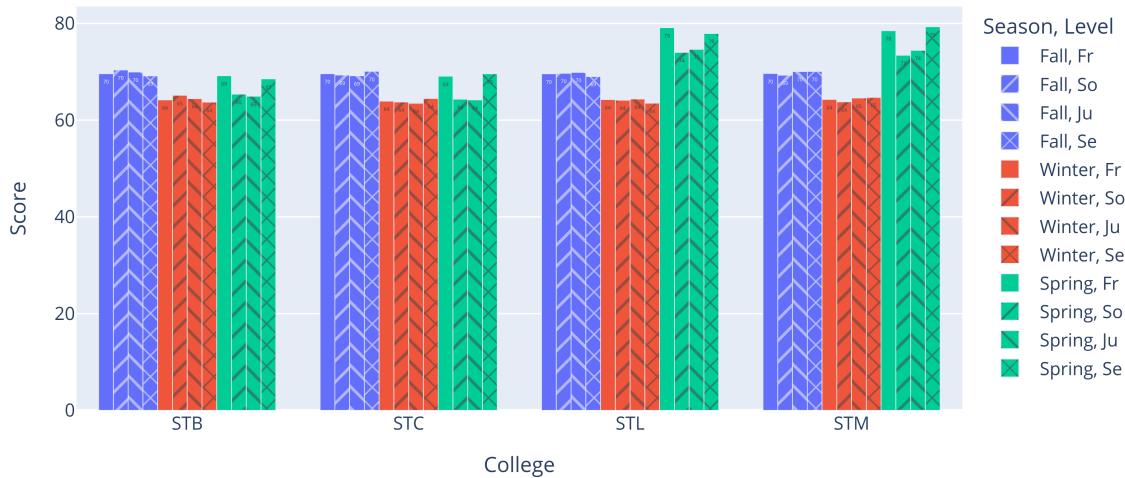
9.1.7 Graphing results by college, level, and season

I'll first use the pattern_shape argument to represent 'Level' values:

```
fig_survey_results_slc_pattern_shape = px.bar(
    df_survey_results_slc, x='College',
    color='Season', y='Score', pattern_shape='Level',
    barmode='group',
    title='Survey Scores by College, Season, and Level',
    text_auto='.0f')

wadi(fig=fig_survey_results_slc_pattern_shape,
     file_path='output/results_slc_pattern_shape',
     display_type=display_type, height=450)
```

Survey Scores by College, Season, and Level



This graph, as shown within the image above, is awfully cluttered and difficult to interpret. Your outcomes with pattern_shape might be better if you have, say, only two different values to pass to that argument (e.g. Fall and Spring results), but even then, your visualization may not be very intuitive.

As an alternative to pattern_shape, you might prefer to display results for a third comparison variable via the facet_col or facet_row arguments instead. Plotly will then create a separate chart for each value within the variable passed to those arguments, then display them together in the same graphic.

Here's what we get if we pass 'Level' to facet_col rather than to pattern_shape:

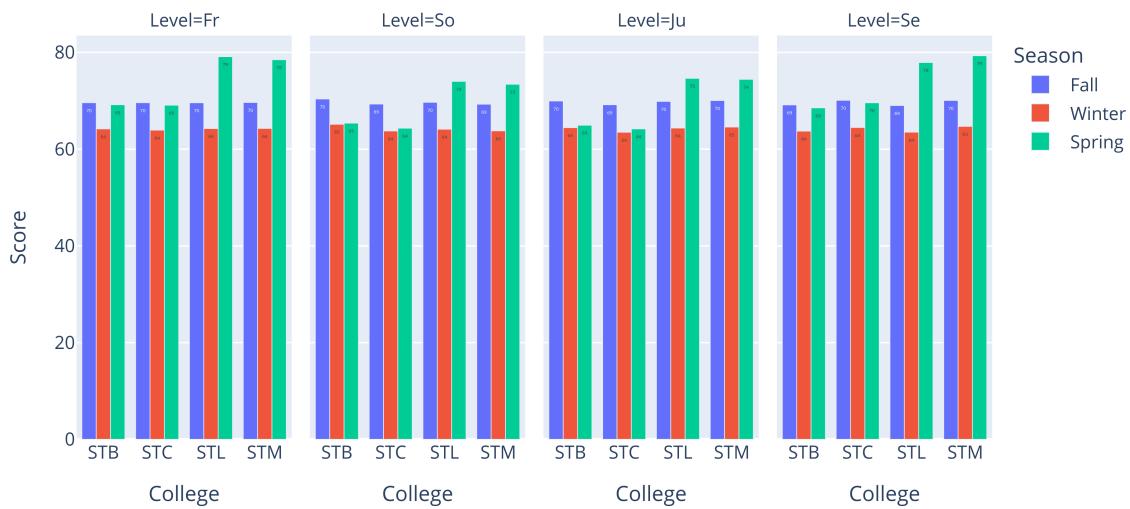
```

fig_survey_results_slc_facet_col = px.bar(
    df_survey_results_slc, x='College',
    color='Season', y='Score', facet_col='Level',
    barmode='group',
    title='Survey Scores by College, Season, and Level',
    text_auto='.0f')

wadi(fig=fig_survey_results_slc_facet_col,
     file_path='output/results_slc_facet_col',
     display_type=display_type, height=450)

```

Survey Scores by College, Season, and Level



Note that each of the four graphs shows data for one particular level. These graphs are displayed within separate columns inside the same row; if we instead selected `facet_row`, they would be shown within separate rows inside the same column.

A third option is to combine two fields (such as `College` and `Level`) into one, then pass that combined field to our graphing code:

```

df_survey_results_slc['College_and_Level'] = (
    df_survey_results_slc['College']
    + ' ' + df_survey_results_slc['Level'])
df_survey_results_slc.head()

```

	Starting_Year	College	Level_For_Sorting	Level	Season	Score	\
0	2023	STB		0	Fr	Fall	69.593583
12	2023	STC		0	Fr	Fall	69.598913
24	2023	STL		0	Fr	Fall	69.585030
36	2023	STM		0	Fr	Fall	69.650503
3	2023	STB		1	So	Fall	70.377306
Season_For_Sorting College and Level							
0		0	STB	Fr			

(continues on next page)

(continued from previous page)

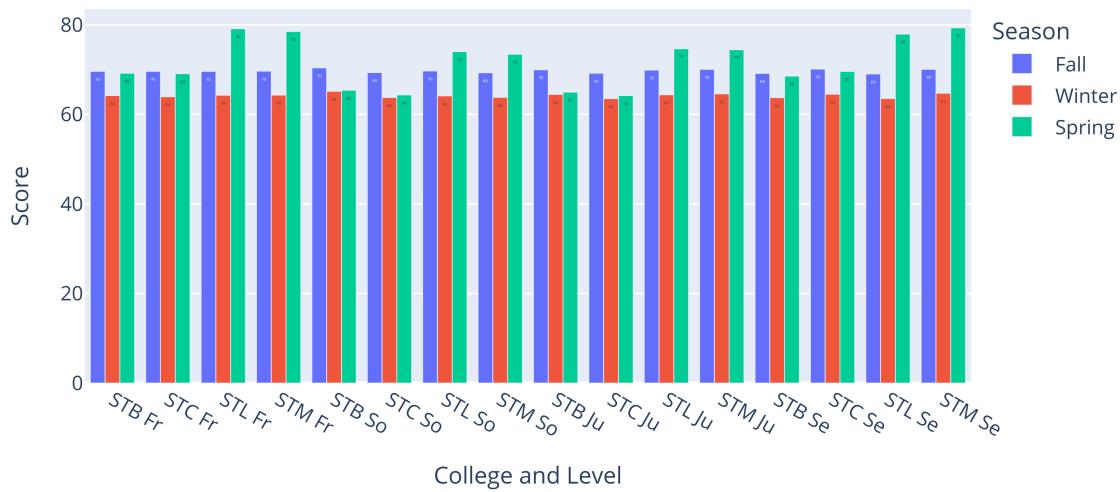
12	0	STC Fr
24	0	STL Fr
36	0	STM Fr
3	0	STB So

Now that we've combined College and Level into our `x` variable, we no longer need a `pattern_shape` or `facet_row/facet_col` argument:

```
fig_survey_results_slc_combined_x = px.bar(
    df_survey_results_slc, x='College and Level',
    color='Season', y='Score', barmode='group',
    title='Survey Scores by College, Season, and Level',
    text_auto='.0f')

wadi(fig=fig_survey_results_slc_combined_x,
      file_path='output/results_slc_combined_x',
      display_type=display_type)
```

Survey Scores by College, Season, and Level



I find this chart to be somewhat more intuitive than the one that incorporated a `pattern_shape` argument, though it's still quite cluttered. However, users can simplify their view of the interactive version of this chart via Plotly's Box Select feature, which allows them to zoom in on a specific section.

If you have four or more variables to display, you could combine them via the same method used to generate the 'College and Level' column above. However, you may conclude that limiting your chart to three (or ideally two) comparison variables is a better option.

9.2 Creating line graphs

Line graphs are an excellent choice for displaying changes in data over time. The following code will provide a basic overview of building line charts within Plotly; for more detailed instructions, I recommend referencing the Plotly Express documentation for line charts (<https://plotly.com/python/line-charts/>) and the `plotly.express.line` reference (<https://plotly.com/python-api-reference/generated/plotly.express.line>).

First, we'll create a simple line chart that shows how survey scores changed over time. This chart will show the same data as `fig_survey_results_by_college_and_season`, a bar chart that we created earlier; however, it will also make the divergence in spring scores between STB/STM and STC/STL more readily apparent.

Here's another look at `df_survey_results_by_college_long`, the basis for this chart, for reference:

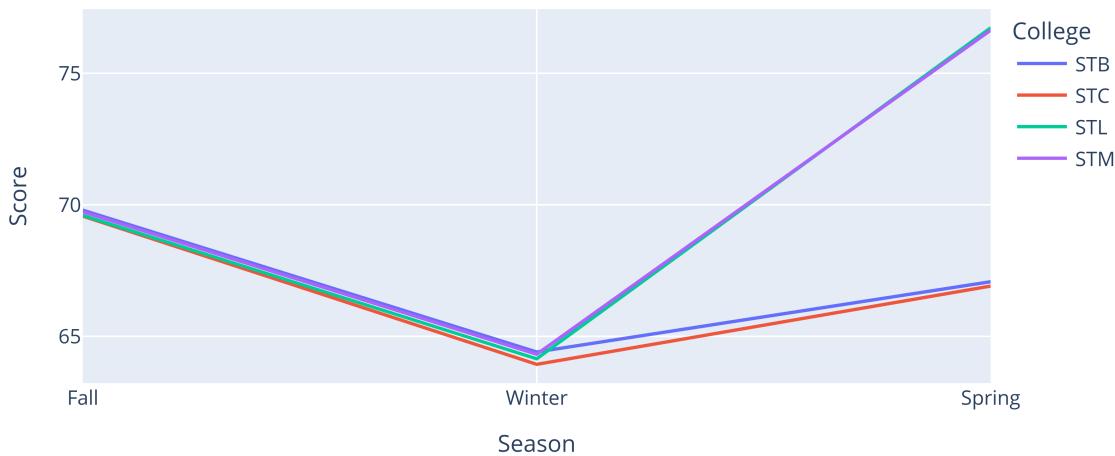
```
df_survey_results_by_college_long
```

	Starting_Year	College	Season	Score
0	2023	STB	Fall	69.797119
3	2023	STC	Fall	69.568665
..
8	2023	STL	Spring	76.727809
11	2023	STM	Spring	76.639004

[12 rows x 4 columns]

```
fig_survey_results_by_college_and_season_line = px.line(
    df_survey_results_by_college_long, x='Season', y='Score',
    color='College', title='Survey Scores by College and Season')
wadi(fig_survey_results_by_college_and_season_line,
     file_path='output/results_by_college_and_season_line',
     display_type=display_type)
```

Survey Scores by College and Season



9.2.1 Representing multiple variables within line charts

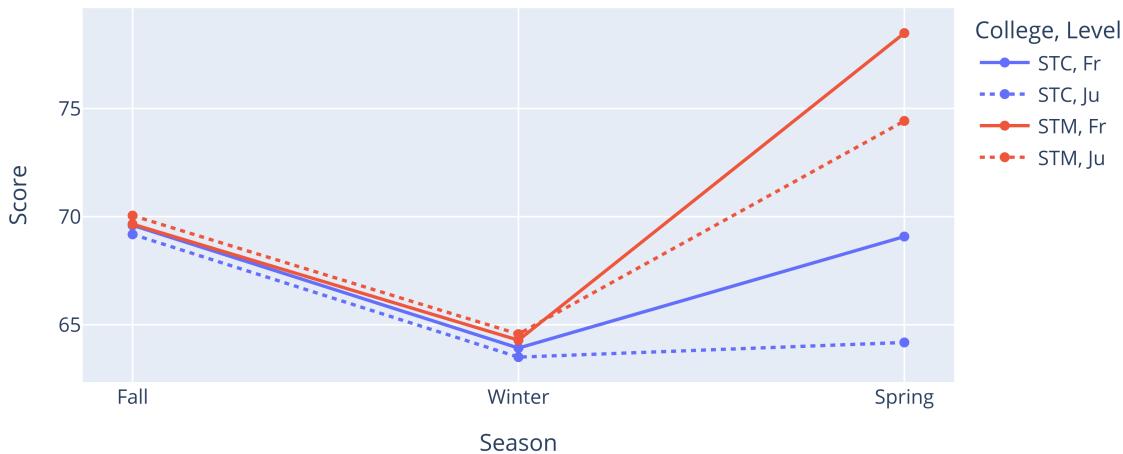
If you need to represent more than one comparison variable in your line chart, consider assigning one to the `color` argument (as shown in the above chart) and the other one to the `line_dash` argument (as shown in the following chart). However, this approach works better if your total variable count remains quite limited.

The following chart shows how survey scores changed over time for freshmen and juniors in STC and STM. (I purposefully reduced the number of possible combinations from 16 to 4 so that the chart would remain readable. However, if you're producing an interactive chart, you could consider showing a larger number of color and dash combinations than you would within a static chart. This is because your viewers can click and double-click within legend entries in order to specify which items to keep, *and* because they can hover over specific points to verify the variables that it represents.)

This chart also adds in markers to draw readers' attention to the actual points being graphed. (You can also vary the marker type for each line (using `px.line()`'s `symbol` argument), rather than the line style, in order to designate which variable is represented by a given line.)

```
fig_selected_survey_results_line = px.line(
    df_survey_results_slc.query("College in ['STM', 'STC'] and Level \
in ['Fr', 'Ju']"), x='Season', y='Score',
    color='College', line_dash='Level', markers=True,
    title='STM and STC Survey Scores by Season for Freshmen and Juniors')
wadi(fig_selected_survey_results_line,
    file_path='output/selected_survey_results_line',
    display_type=display_type)
```

STM and STC Survey Scores by Season for Freshmen and Juniors



9.2.2 Adding labels to points

Adding labels to line charts is relatively straightforward; however, making them readable can be a bit more challenging. For instance, if two lines overlap, their labels will also overlap by default.

Let's try adding labels to a chart that shows changes in survey scores for two schools: STB and STC. First, we'll create a new 'Rounded_Score' field within `df_survey_results_by_college_long` that we can use as text labels. (Using the original 'Score' field for our labels would display far more decimal points for each point than we need.)

```
df_survey_results_by_college_long['Rounded_Score'] = (
    df_survey_results_by_college_long['Score'].round(2))
df_survey_results_by_college_long
```

	Starting_Year	College	Season	Score	Rounded_Score
0	2023	STB	Fall	69.797119	69.80
3	2023	STC	Fall	69.568665	69.57
..
8	2023	STL	Spring	76.727809	76.73
11	2023	STM	Spring	76.639004	76.64

[12 rows x 5 columns]

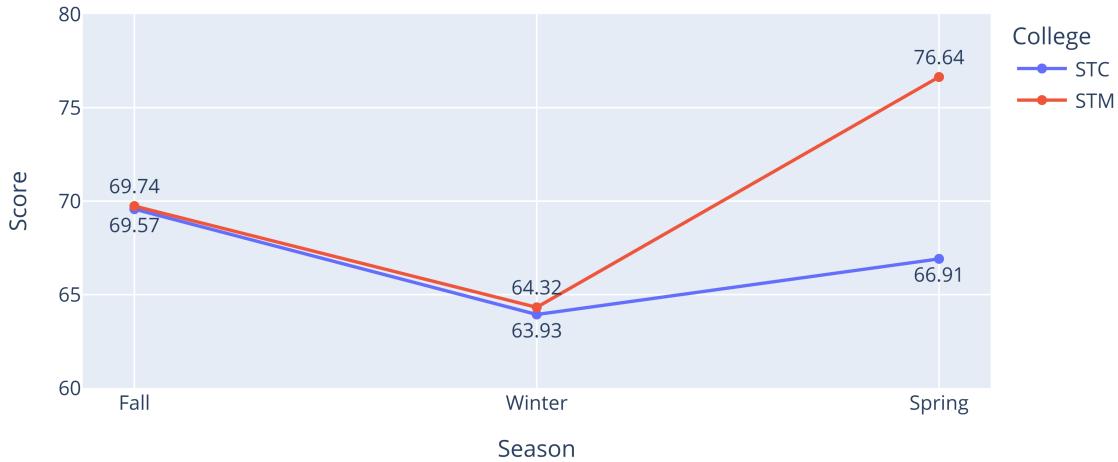
We can add labels to our chart using `px.line()`'s `text` argument (rather than `text_auto`, which is supported for bar charts but not line charts). By default, these labels will appear directly over each point, which I find makes the chart less readable. Fortunately, we can use the `textposition` argument of `update_traces()` to provide an offset between each point and its corresponding label.

However, as shown within the first line chart we produced, STM and STC's fall and winter scores were almost identical. This means that, by default, the labels for these text options will overlap. To avoid this issue, we'll shift STC's and STM's labels *above* and *below* their corresponding points, respectively. This can be accomplished by calling `update_traces` twice, once for each line; we can then specify the line whose `textposition` value we wish to update via the `selector` argument of `update_traces()`. (For more on this argument, visit <https://plotly.com/python/reference/scatter/#scatter>. Special thanks to python-trader, akroma, and adamschroeder in this thread (<https://community.plotly.com/t/need-to-target-a-specific-trace-by-name-when-use-update-traces-s-selector/63151/2>) for helping me figure out this solution.)

```
fig_stc_stm_survey_results_line = px.line(
    df_survey_results_by_college_long.query("College in ['STM', 'STC']"),
    x='Season', y='Score',
    color='College', markers=True,
    text='Rounded_Score',
    title='STM and STC Survey Scores by Season').update_traces(
    textposition='top center', selector={'name': 'STM'}).update_traces(
    textposition='bottom center',
    selector={'name': 'STC'}).update_layout(yaxis_range=[60, 80])

wadi(fig_stc_stm_survey_results_line,
      file_path='output/stc_stm_survey_results_line',
      display_type=display_type)
```

STM and STC Survey Scores by Season



To figure out what item to pass to the 'name' argument (or another argument of your choice) of `selector`, try viewing the `.data` attribute of your image. Note that 'STC' and 'STM' are listed below as the names of each trace.

```
fig_stc_stm_survey_results_line.data
```

```
(Scatter({
    'hovertemplate': 'College=STC<br>Season=%{x}<br>Score=%{y}<br>Rounded_Score=%
    <text></extra></extra>',
    'legendgroup': 'STC',
    'line': {'color': '#636efa', 'dash': 'solid'},
    'marker': {'symbol': 'circle'},
    'mode': 'lines+markers+text',
    'name': 'STC',
    'orientation': 'v',
    'showlegend': True,
    'text': array([69.57, 63.93, 66.91]),
    'textposition': 'bottom center',
    'x': array(['Fall', 'Winter', 'Spring'], dtype=object),
    'xaxis': 'x',
    'y': array([69.56866485, 63.9348451 , 66.91144414]),
    'yaxis': 'y'
}),
Scatter({
    'hovertemplate': 'College=STM<br>Season=%{x}<br>Score=%{y}<br>Rounded_Score=%
    <text></extra></extra>',
    'legendgroup': 'STM',
    'line': {'color': '#EF553B', 'dash': 'solid'},
    'marker': {'symbol': 'circle'},
    'mode': 'lines+markers+text',
    'name': 'STM',
    'orientation': 'v',
    'showlegend': True,
    'text': array([69.74, 64.32, 76.64]),
    'textposition': 'bottom center',
    'x': array(['Fall', 'Winter', 'Spring'], dtype=object),
    'xaxis': 'x',
    'y': array([69.56866485, 63.9348451 , 66.91144414]),
    'yaxis': 'y'
}),
```

(continues on next page)

(continued from previous page)

```
'textposition': 'top center',
'x': array(['Fall', 'Winter', 'Spring'], dtype=object),
'xaxis': 'x',
'y': array([69.73568465, 64.31868932, 76.63900415]),
'yaxis': 'y'
}))
```

9.3 Scatter plots

(For a more detailed overview of scatter plots, see <https://plotly.com/python/line-and-scatter/>.)

Scatter plots represent relationships between two variables by visualizing paired copies of those variables. For instance, if we wish to see how fall and spring NVCU bookstore spending are correlated, we could create a scatter plot in which each student is represented by a point; each point's x value represents fall spending; and each point's y value represents spring spending. That's exactly what we'll do in this section!

First, we'll load in our sales data:

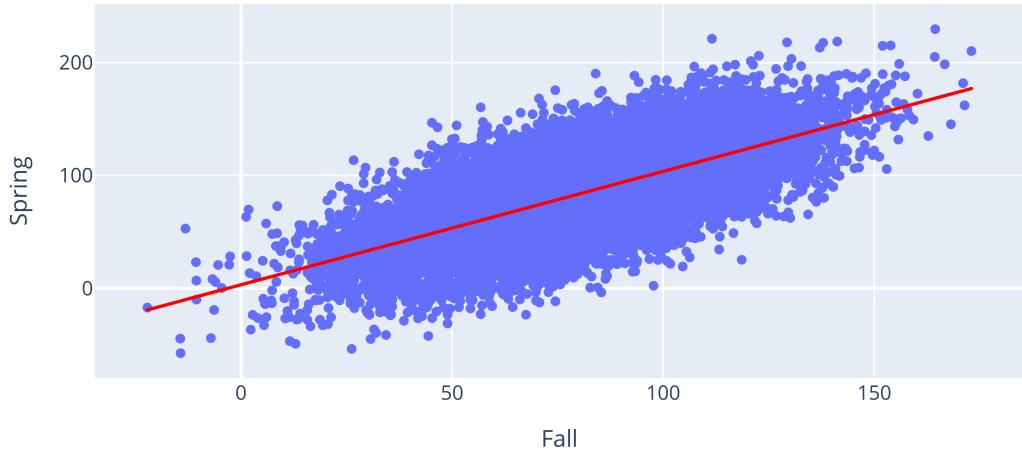
```
df_bookstore_sales = pd.read_sql("Select * from bookstore_sales",
                                 con=e)
df_bookstore_sales.head()
```

	student_id	gender	college	level	Fall	Spring	Fall_Spring_Change
0	2020-1	F	STC	Se	66.80	58.24	-8.56
1	2020-2	F	STM	Se	104.67	151.90	47.23
2	2020-3	F	STC	Se	46.17	16.56	-29.61
3	2020-4	F	STC	Se	58.68	73.77	15.09
4	2020-5	F	STM	Se	99.73	108.78	9.05

We'll now create a scatter plot that compares Fall and Spring spending for each student. We'll also include a linear trend line that helps make sense of the relationship between these two values. (You can modify the color of this line via the `trendline_color_override` argument as noted at <https://plotly.com/python/linear-fits/>).

```
fig_scatter = px.scatter(df_bookstore_sales,
                         x='Fall', y='Spring',
                         trendline='ols', trendline_color_override='red',
                         title='Fall vs. Spring Bookstore Sales')
wadi(fig_scatter, 'output/sales_scatter', display_type=display_type)
```

Fall vs. Spring Bookstore Sales



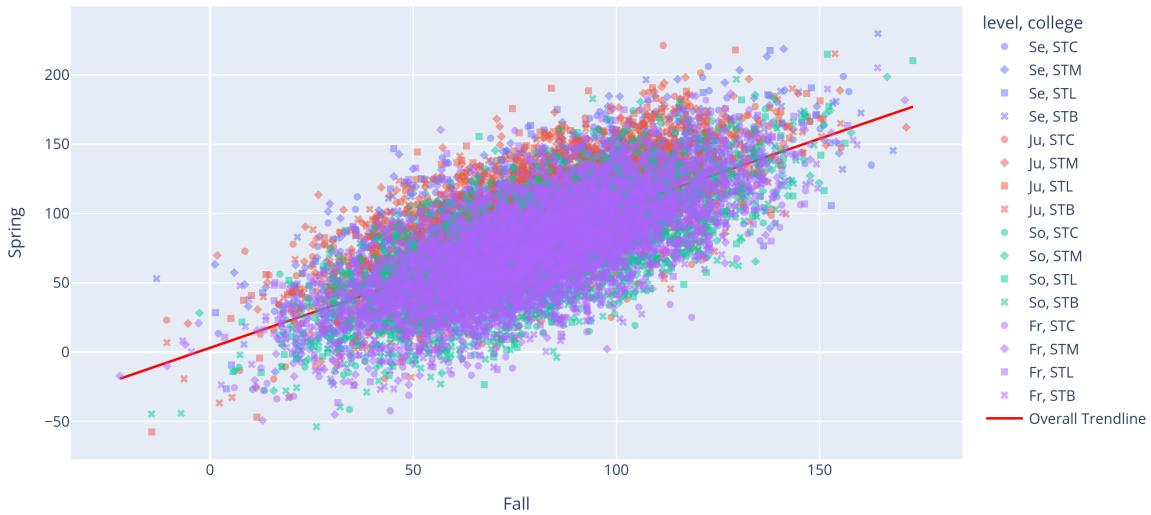
We can visualize additional categories using the `color` and `symbol` arguments of `px.scatter()`. The following plot uses different colors to represent levels and different symbols to represent different colleges.

To be honest, I don't think that these added elements contribute much to this particular chart; with so many points present, it's still hard to make out anything but the overall trend (that fall and spring spending are positively correlated). However, if your scatter plot has fewer points, you may find these arguments more useful.

```
fig_scatter_with_color = px.scatter(df_bookstore_sales,
                                      x='Fall', y='Spring', color='level', symbol='college',
                                      opacity=0.5,
                                      trendline='ols', trendline_color_override='red',
                                      trendline_scope='overall',
                                      title='Fall vs. Spring Bookstore Sales')

# I chose a higher height value here so that all legend entries would
# appear within the static version of the chart.
wadi(fig_scatter_with_color, 'output/sales_scatter_with_color',
      height=550, display_type=display_type)
```

Fall vs. Spring Bookstore Sales

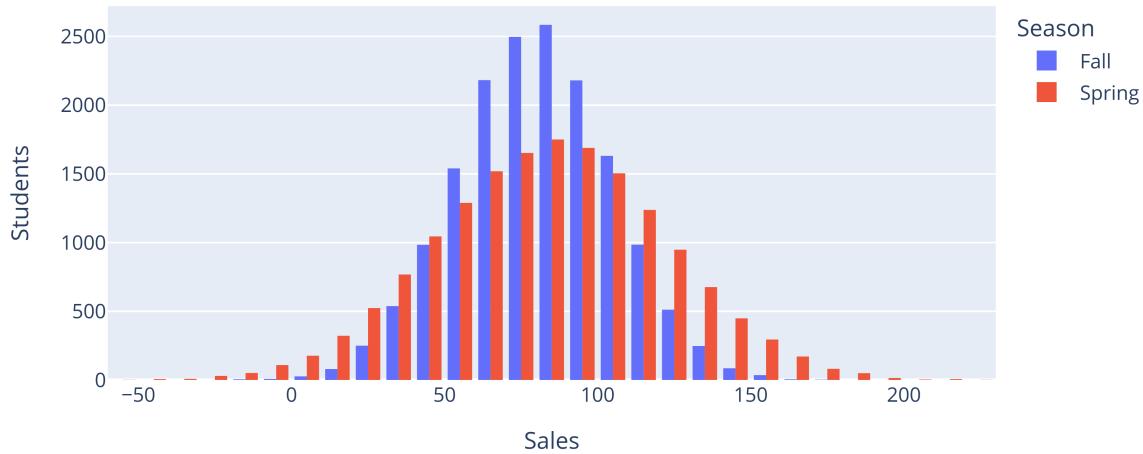


9.4 Histograms

Histograms allow you to view the number of entries in a dataset that fall into a particular range. For instance, the following histogram shows the number of students who accumulated a certain range of bookstore sales during the fall and spring semesters.

```
fig_sales_histogram = px.histogram(df_bookstore_sales,
                                    x=['Fall', 'Spring'], barmode='group', nbins=30,
                                    title='Distributions of fall and spring sales').update_layout(
                                    xaxis_title='Sales', yaxis_title='Students',
                                    legend_title='Season')
wadi(fig_sales_histogram, 'output/sales_histogram',
      display_type=display_type)
```

Distributions of fall and spring sales



This histogram indicates that students' spending varied more during the spring than during the fall. It also looks like students spent more during the spring, on average, than they did during the fall.

We can confirm these hunches by calling `describe` on our underlying dataset: ('std' here stands for 'standard deviation', a measure of how dispersed a given set of values is.)

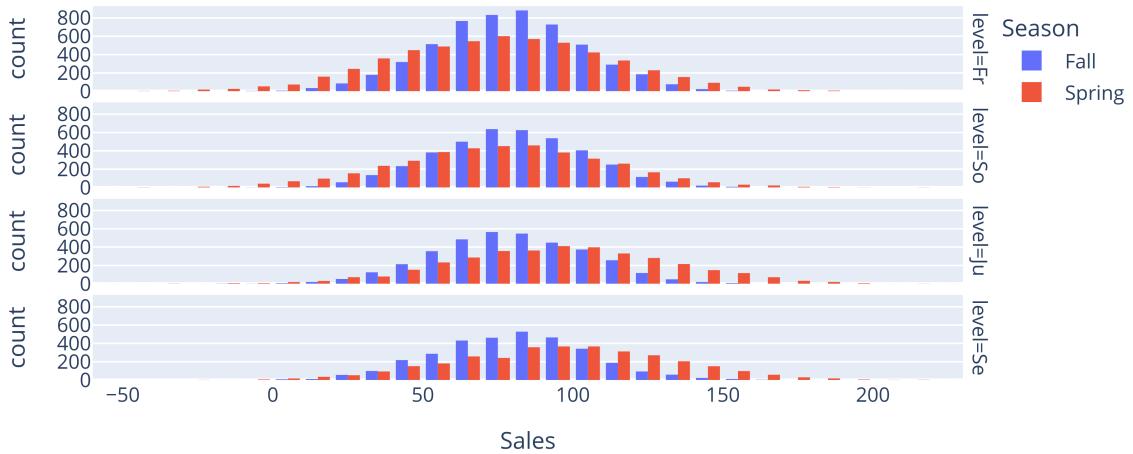
```
df_bookstore_sales.describe().loc[['count', 'mean', 'std']]
```

	Fall	Spring	Fall_Spring_Change
count	16384.000000	16384.000000	16384.000000
mean	80.091662	83.644286	3.552625
std	25.023476	37.297291	27.534548

To see whether these distributions varied by level, we can pass that variable as an argument to the `facet_row` parameter:

```
fig_sales_by_level = px.histogram(df_bookstore_sales.sort_values(
    'level', key=lambda col: col.map({'Fr':0,'So':1,'Ju':2, 'Se':3})),
    x=['Fall', 'Spring'], barmode='group', nbins=30,
    facet_row='level',
    title='Fall/spring sales distributions \
by level').update_layout(
    xaxis_title='Sales', legend_title='Season')
wadi(fig_sales_by_level, 'output/sales_by_level_histogram',
    display_type=display_type)
```

Fall/spring sales distributions by level

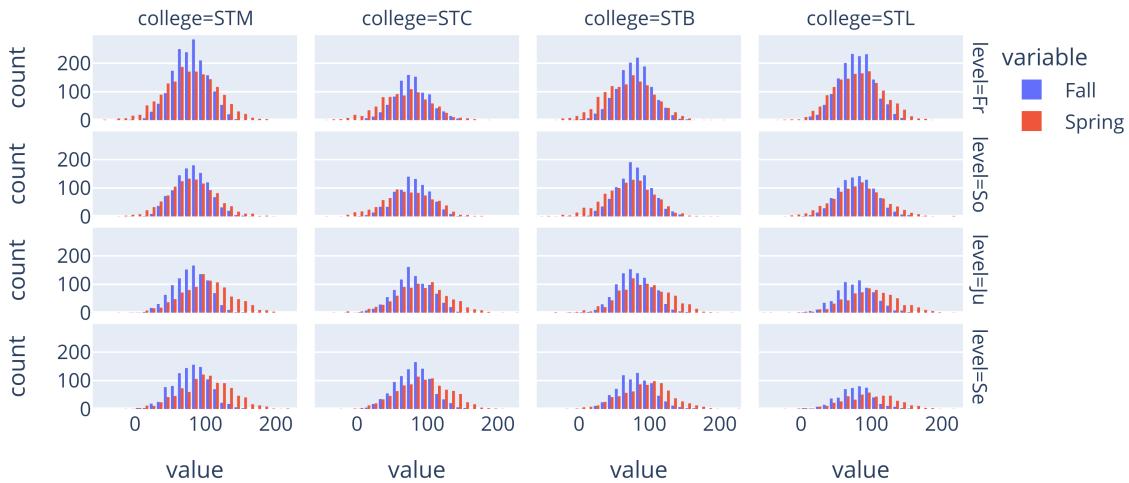


Note that, in the output above, the spring bars appear to be shifted to the right of the fall bars for juniors and seniors, but not for freshmen and sophomores. This suggests that fall-to-spring spending growth was stronger for upperclassmen than it was for underclassmen.

If we wanted to show fall and spring spending distributions for each level/college combination, we could update our previous chart by assigning ‘college’ to the `facet_col` parameter:

```
fig_sales_by_college_and_level = px.histogram(
    df_bookstore_sales.sort_values(
        'level', key=lambda col: col.map({'Fr':0, 'So':1, 'Ju':2, 'Se':3})),
    x=['Fall', 'Spring'], barmode='group', nbins=30,
    facet_row='level',
    facet_col='college',
    title='Fall/Spring Sales Distributions by College and Level')
wadi(fig_sales_by_college_and_level,
     'output/sales_by_college_and_level_histogram',
     display_type=display_type)
```

Fall/Spring Sales Distributions by College and Level



However, this type of chart will cause many readers' eyes to glaze over; the more meme-aware members of your audience might respond with "I Ain't Reading All That" (<https://knowyourmeme.com/memes/i-aint-reading-all-that>). As the saying goes, *Just because you can doesn't mean you should*.

What would be a simpler way to visualize these trends? In the Regressions section of PFN, we'll create a grouped bar chart that compares average fall/spring sales growth by college and level. By replacing individual fall and spring distributions with a single growth metric, we can display all 16 level and college combinations within the same graphic.

Once you make it to the Regressions section, compare that chart (titled 'Average Fall-Spring Sales Growth by College and Level' to this one; I think you'll agree that the latter is much easier to interpret than the matrix of histograms shown above.)

9.5 Treemaps

You might have expected to see pie charts covered in this section, and Plotly certainly supports them (<https://plotly.com/python/pie-charts/>). However, I would like to humbly suggest that you use *treemaps* as opposed to pie charts. Why? As Joel Abrams notes at <https://theconversation.com/heres-why-you-should-almost-never-use-a-pie-chart-for-your-data-214576>, it can be hard to estimate area within pie charts alone; in addition, when many different categories are present, it can be quite hard to figure out the value corresponding to one specific category.

Abrams suggests that you consider using bar charts instead, which is perfectly fine; however, treemaps are a worthy alternative to consider as well. Because they show data in rectangular (rather than wedge-shaped form), they're much easier to interpret; in addition, they do a good job of displaying hierarchical data. (I'll also add a bar chart in this section that shows equivalent data as the tree chart, thus allowing you to compare the two.)

Loading our NVCU current enrollment table into the script:

```
df_curr_enrollment = pd.read_sql("Select * from curr_enrollment", con=e)
df_curr_enrollment['Count'] = 1
```

```
df_curr_enrollment.head()
```

	first_name	last_name	gender	matriculation_year	...	class_of	level	\
0	Amanda	Murphy	F	2020	...	2024	Se	
1	Terri	Washington	F	2020	...	2024	Se	
2	Crystal	Davis	F	2020	...	2024	Se	
3	Theresa	Joseph	F	2020	...	2024	Se	
4	Chelsea	Martin	F	2020	...	2024	Se	
	level_for_sorting	Count						
0		3	1					
1		3	1					
2		3	1					
3		3	1					
4		3	1					

[5 rows x 12 columns]

I'll now create a 'college/level' field that will prove useful for one of our treemaps.

```
df_curr_enrollment['college/level'] = (
    df_curr_enrollment['college'] + ' (' +
    + df_curr_enrollment['level'] + ')')

# Now that I've created this field, I'll replace the original 'level'
# values with their non-abbreviated equivalents. (The abbreviated versions
# will fit better within the combined college/level field, so I waited
# to make these changes until that field had been created.)
df_curr_enrollment['level'] = df_curr_enrollment['level'].map(
    {'Fr':'Freshman', 'So':'Sophomore', 'Ju':'Junior','Se':'Senior'})

df_curr_enrollment
```

	first_name	last_name	gender	matriculation_year	...	level	\
0	Amanda	Murphy	F	2020	...	Senior	
1	Terri	Washington	F	2020	...	Senior	
...
16382	Phillip	Benitez	M	2023	...	Freshman	
16383	Nicholas	Wheeler	M	2023	...	Freshman	
	level_for_sorting	Count	college/level				
0		3	1	STC (Se)			
1		3	1	STM (Se)			
...				
16382		0	1	STB (Fr)			
16383		0	1	STM (Fr)			

[16384 rows x 13 columns]

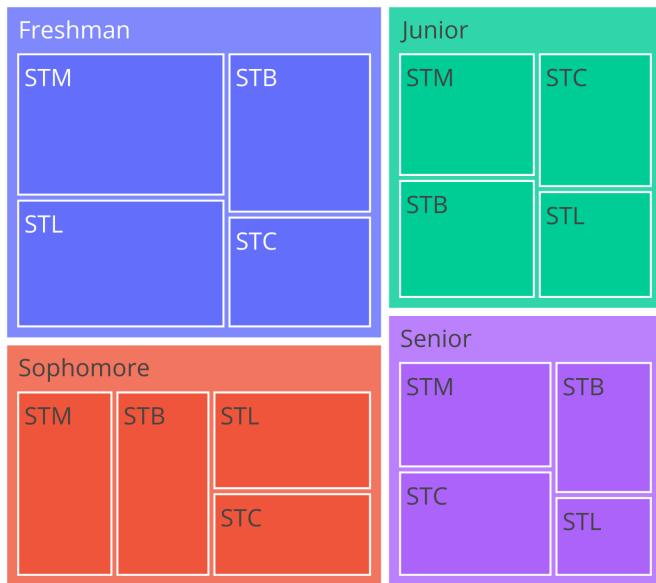
The following treemap shows NVCU's current enrollment by level and by college. The outer rectangles show the total enrollment for each level, while the inner rectangles show the enrollment distribution, by college, for all students within that level.

Note that the Freshman rectangle has the largest area and that the Senior rectangle has the smallest; this suggests that NVCU's enrollment is growing (though it could also indicate an attrition problem—which, fortunately for NVCU, isn't the case.) Also note that, while STL's rectangle within the Senior field is relatively small relative to those for the other

schools, its relative size within the Freshman rectangle is much larger. This suggests that STL has become more popular (relative to other colleges) over time, though it could also indicate that students tend to transfer out of it between their first and last years at NVCU.

```
fig_level_college_treemap = px.treemap(  
    df_curr_enrollment, path=['level', 'college'],  
    values='Count', branchvalues='total',  
    title='Enrollment by Level and College')  
  
# In the following wadi() call, setting aspect ratio to 1 makes our  
# chart square and thus a bitmore compact. (I'll use this same ratio for  
# the other treemaps in this script.)  
wadi(fig_level_college_treemap, 'output/treemap_level_college',  
    height=500, aspect_ratio=1,  
    display_type=display_type, display_width=500)
```

Enrollment by Level and College



In this treemap, college enrollments are nested within level-wide enrollments. However, as shown below, we can use the ‘college/level’ field within `df_curr_enrollment` to create an ‘ungrouped’ equivalent of this chart. We’ll color the rectangles within this chart by college to make our output, which is sorted by size, somewhat more intuitive.

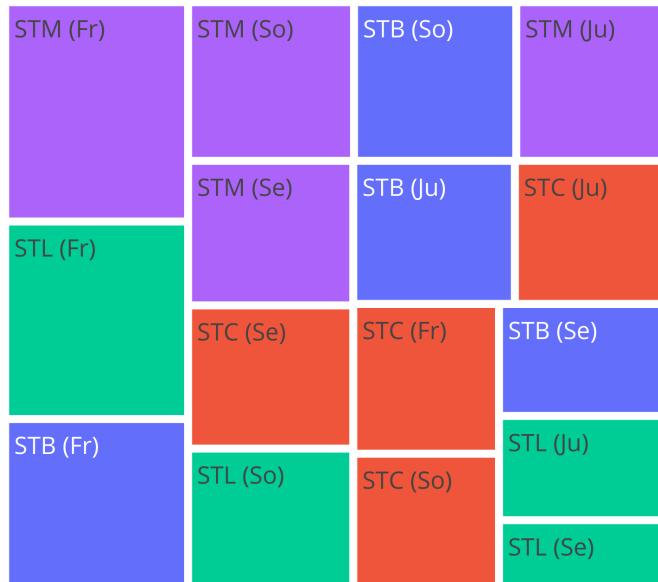
```
fig_level_college_treemap_ungrouped = px.treemap(  
    df_curr_enrollment, path=['college/level'],
```

(continues on next page)

(continued from previous page)

```
values='Count', color='college',
width=600, height=600,
title='Enrollment by College and Level')
wadi(fig_level_college_treemap_ungrouped,
    'output/treemap_level_college_ungrouped',
    height=500, aspect_ratio=1,
    display_type=display_type, display_width=500)
```

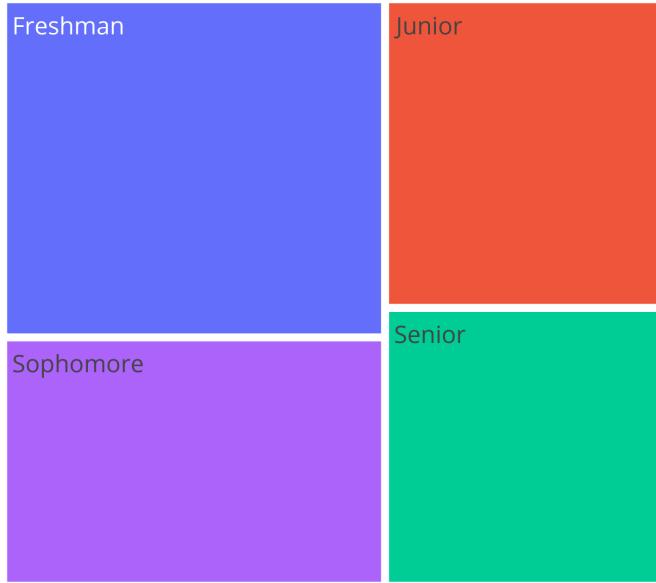
Enrollment by College and Level



We can view the relative size of our enrollments for each level a bit more clearly within the following treemap, which groups students by level only.

```
fig_level_treemap = px.treemap(
    df_curr_enrollment, path=['level'],
    values='Count', color='level',
    title='Enrollment by Level')
wadi(fig_level_treemap, 'output/treemap_level',
    height=500, aspect_ratio=1,
    display_type=display_type, display_width=500)
```

Enrollment by Level



Getting data labels inserted into treemaps created with Plotly Express is trickier than it is for bar charts, as there's no `text_auto` or `text` argument within the `px.treemap()` function. (See <https://plotly.com/python-api-reference/generated/plotly.express.treemap> for reference.)

If your chart is HTML-based (i.e. interactive), users will still be able to view the actual values corresponding to a rectangle when they hover over it. However, if you need these labels to show up within a static chart, you could create a new field that shows both categories *and* their values, then pass *that* field to the `path` argument of `px.treemap()`. The following cells provide an example of this approach.

We'll first create a pivot table that shows enrollment by level, then add a new field to it that contains both level names and enrollment totals.

```
df_enrollment_by_level = df_curr_enrollment.pivot_table(  
    index=['level_for_sorting', 'level'], values='Count',  
    aggfunc='sum').reset_index().rename(  
    columns={'Count':'Enrollment', 'level':'Level'})  
  
# Creating a column that contains both enrollment totals by level  
# and the name of that level:  
# (These levels will be converted to plural forms by appending 's' to  
# them. The script will also add in a correction for the plural version  
# of 'freshman.')  
df_enrollment_by_level['Labeled Enrollment'] = (
```

(continues on next page)

(continued from previous page)

```
df_enrollment_by_level['Enrollment'].astype('str')
+ ' ' + df_enrollment_by_level['Level'] + 's').str.replace(
    'Freshmans', 'Freshmen')

df_enrollment_by_level
```

	level_for_sorting	Level	Enrollment	Labeled Enrollment
0	0	Freshman	5443	5443 Freshmen
1	1	Sophomore	3999	3999 Sophomores
2	2	Junior	3653	3653 Juniors
3	3	Senior	3289	3289 Seniors

Next, we'll incorporate this 'Labeled Enrollment' field into a new treemap:

```
fig_labeled_level_treemap = px.treemap(
    df_enrollment_by_level, path=['Labeled Enrollment'],
    values='Enrollment', color='Level',
    title='Enrollment by Level')
wadi(fig_labeled_level_treemap, 'output/treemap_labeled_levels',
      height=500, aspect_ratio=1,
      display_type=display_type, display_width=500)
```

Enrollment by Level



9.6 Comparing these treemaps to bar charts

As Joel Abrams pointed out, you can use bar charts to display totals as well. Let's create a bar-chart equivalent of our 'Enrollment by Level and College' treemap so that you can more easily compare these two visualization types.

First, we'll create a pivot table that can serve as the basis for this chart:

```
df_enroll_pivot = df_curr_enrollment.pivot_table(  
    index=['level_for_sorting', 'level', 'matriculation_year',  
           'college'], values='Count',  
    aggfunc='sum').reset_index()  
df_enroll_pivot.rename(columns = {'matriculation_year':  
                                'Matriculation Year',  
                                'college':'College',  
                                'Count':'Enrollment',  
                                'level':'Level'}, inplace=True)  
  
# Converting the Matriculation Year to a string will ensure that these  
# values, when used as x axis labels within an upcoming line chart,  
# will appear as distinct items rather than a range. We could also  
# achieve this outcome after creating a chart by applying  
# the following code:  
# fig.update_layout(xaxis_type='category')  
  
df_enroll_pivot['Matriculation Year']  
df_enroll_pivot['Matriculation Year'] = (  
    df_enroll_pivot['Matriculation Year'].astype('str'))  
df_enroll_pivot
```

level_for_sorting	Level	Matriculation Year	College	Enrollment	
0	0	Freshman	2023	STB	1309
1	0	Freshman	2023	STC	920
..
14	3	Senior	2020	STL	488
15	3	Senior	2020	STM	1007

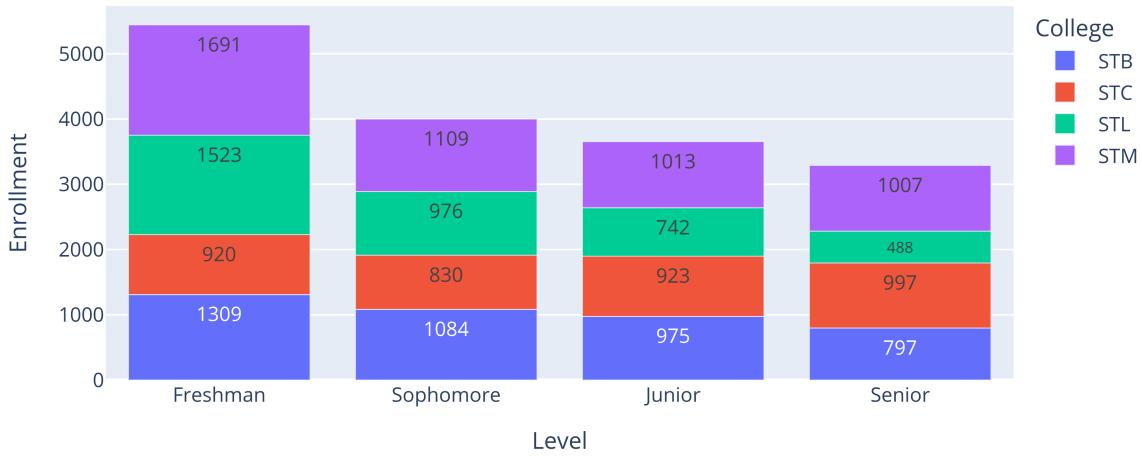
[16 rows x 5 columns]

Next, we'll create our bar chart. Note that the output shows the same values as our earlier 'Enrollment by Level and College' treemap; these values are simply arranged differently.

I'll admit that I *do* prefer this bar chart over its corresponding treemap, as the former does a better job of highlighting the differences in enrollment totals between each level.

```
fig_enroll_bar = px.bar(  
    df_enroll_pivot,  
    x='Level', y='Enrollment',  
    color='College', text_auto='.0f',  
    title='Enrollment by Level and College')  
wadi(fig_enroll_bar, 'output/enroll_bar', display_type=display_type)
```

Enrollment by Level and College

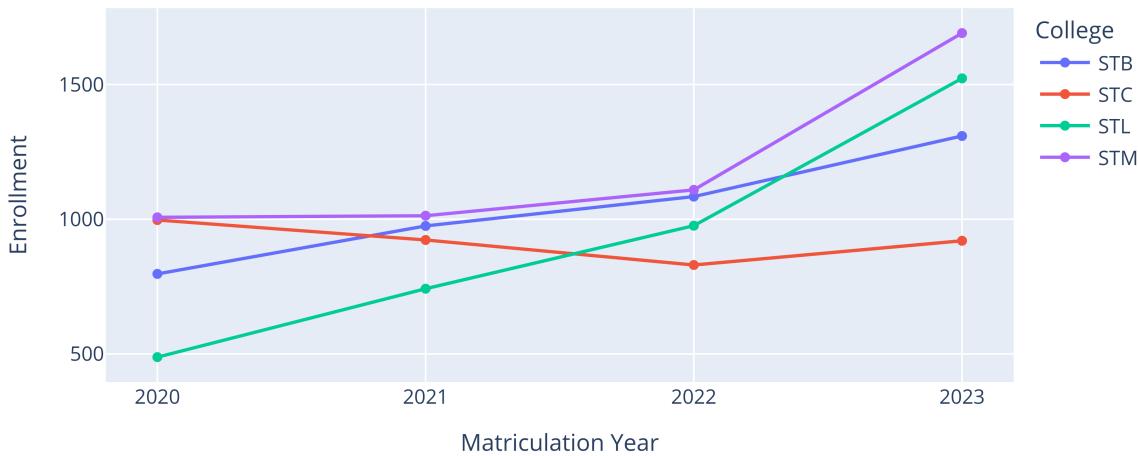


If we replace Level as our x-axis value with Matriculation Year, we can also create a line graph that indicates how freshman enrollment at each college has grown over time. (Most of the seniors in our current enrollment dataset matriculated in 2020, and would thus be considered freshmen in that year; juniors matriculated in 2021; sophomores matriculated in 2022; and freshmen matriculated in 2023.)

**(I say ‘indicates’ here because we’re working off a current enrollment dataset that does not take transfers or attrition into account; if NVCU had lots of attrition for a given year, or many transfers, this chart would thus underestimate or overestimate freshman enrollment for that year, respectively.)*

```
fig_enroll_line = px.line(df_enroll_pivot.sort_values(
    'Matriculation Year'),
    x='Matriculation Year', y='Enrollment',
    color='College',
    title='Enrollment by Matriculation Year and College',
    markers=True)
wadi(fig_enroll_line, 'output/enroll_line', display_type=display_type)
```

Enrollment by Matriculation Year and College



This chart shows some intriguing patterns. First, it's evident that freshman enrollment has grown over time (barring the caveats about this dataset that I explained earlier). However, this growth is not uniform across colleges: while STL has had particularly strong growth in freshman enrollment, STC's numbers are actually lower now than they were back in 2020.

That does it for the first part of the Graphing section. In the following Pivot and Graph Functions notebook, you'll learn how to speed up the process of creating pivot tables and charts—a skill that will prove especially useful when we begin creating interactive online visualizations.

PIVOT AND GRAPH FUNCTIONS

This chapter will demonstrate how to use functions to simplify the Plotly graphing process. These functions will also prove useful within the online visualizations section of Python for Nonprofits.

10.1 The motivation behind this chapter

As shown in the Graphing chapter, pivot tables play a crucial role in generating Plotly charts. They allow you to turn a raw dataset into a table of aggregate values that can easily be rendered as a bar graph, line graph, or some other chart type.

Thanks to the powerful Pandas library, these pivot tables are easy to produce within Python. However, creating separate pivot tables for each graph still requires a decent amount of work. In addition, there are times when coding these pivot tables would be impractical.

Suppose, for instance, that you're putting together an online dashboard that visualizes mean NVCU survey result scores in bar chart form. You would like to allow users to compare these scores by up to 6 potential variables: starting year, season, gender, graduation Year, college, and level. (In other words, they should be able to view these results by season and college; college, level, and season; all six variables together; and even no variables at all—in which case the mean survey score will be displayed within a single bar.)

If you wanted to create a pivot table for each of these comparison options, you'd end up having to construct over 60 tables. (There are 6 comparison options, each of which can either be present or absent—resulting in a total of $2^6 = 64$ tables.) In order to avoid the time-consuming task of pre-rendering all of these tables, we'll need to utilize **an ‘autopivot’ function that, given a list of comparison variables, a y variable, and a color variable, returns a pivot table, x and y variables, and a color variable**. We will demonstrate such a function within this notebook.

Although this pivot table function will play a pivotal (pun intended) role in our final graph, we'll also need to create a corresponding **‘autobar’ function that converts the output of the autopivot function into a graph**. (We could add this code into our main pivot table function, but separating the two allows other graph types—such as line charts—to be created rather than bar charts, thus making the code more versatile.)

```
import pandas as pd
import plotly.express as px
from sqlalchemy import create_engine
e = create_engine('sqlite:///../Appendix/nvcu_db.db')

import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, wadi
display_type = config_notebook(display_max_columns = 5)
```

10.2 Creating a detailed survey results dataset

This dataset will include data from both our survey results *and* enrollment tables, thus allowing for additional comparison options.

10.2.1 Importing each dataset:

```
df_enrollment = pd.read_sql(  
    'select * from curr_enrollment', con=e)  
df_enrollment.head()
```

```
   first_name  last_name  ... level  level_for_sorting  
0      Amanda     Murphy  ...   Se            3  
1       Terri  Washington  ...   Se            3  
2     Crystal      Davis  ...   Se            3  
3    Theresa     Joseph  ...   Se            3  
4    Chelsea     Martin  ...   Se            3
```

[5 rows x 11 columns]

```
df_survey_results = pd.read_sql(  
    'select * from survey_results', con=e)  
df_survey_results.head()
```

```
  student_id  starting_year  season  score  
0      2020-1          2023   Fall    88  
1      2020-2          2023   Fall    37  
2      2020-3          2023   Fall    54  
3      2020-4          2023   Fall    56  
4      2020-5          2023   Fall    77
```

10.2.2 Merging these datasets together:

```
df_se = df_survey_results.merge(  
    df_enrollment, on='student_id', how='left')  
# The 'se' in df_se stands for 'survey and enrollment.'  
  
# Removing columns that we're not interested in using for our  
# comparisons:  
df_se.drop(  
    ['student_id', 'first_name', 'last_name',  
     'date_of_birth', 'matriculation_number'],  
    axis=1, inplace=True)  
  
# Reformatting our column names so that they'll show up better  
# within graphs:  
  
df_se.columns = [  
    column.replace('_', ' ').title()  
    for column in df_se.columns]
```

(continues on next page)

(continued from previous page)

```
df_se['Season For Sorting'] = (
    df_se[
        'Season'].map({'Fall':0,'Winter':1,'Spring':2}))
df_se.sort_values(
    ['Season For Sorting', 'Level For Sorting'],
    inplace=True)

df_se.head()
```

	Starting Year	Season	...	Level For Sorting	Season For Sorting
10941	2023	Fall	...	0	0
10942	2023	Fall	...	0	0
10943	2023	Fall	...	0	0
10944	2023	Fall	...	0	0
10945	2023	Fall	...	0	0

[5 rows x 10 columns]

```
df_se['Responses'] = 1
df_se_pivot = df_se.pivot_table(
    index=['Starting Year', 'Season', 'Gender',
           'Matriculation Year', 'College', 'Class Of',
           'Level', 'Level For Sorting', 'Season For Sorting'],
    values=['Score', 'Responses'],
    aggfunc={'Score':'mean', 'Responses':'sum'}).reset_index()
df_se_pivot
```

	Starting Year	Season	...	Responses	Score
0	2023	Fall	...	408	69.583333
1	2023	Fall	...	519	69.697495
..
62	2023	Spring	...	756	79.120370
63	2023	Spring	...	858	78.083916

[64 rows x 11 columns]

10.3 A basic autopivot function

The following code shows a simple version of an ‘autopivot’ function that can automatically create both pivot tables to be graphed and corresponding graphing variables. (This table and corresponding variables can then be sent to a graphing function like `px.bar()`.) One key variable generated by this function is a field that groups together all of the individual comparison values passed to `x_vars[]` and can thus serve as the ‘`x`’ argument within a Plotly graph call.

This function has a number of limitations: for instance, it doesn’t take color options into account, and it also can’t accommodate variables that should be included in the *pivot table* function (e.g. for sorting purposes) but excluded from the *x variable column*. However, I chose to include it here in order to highlight the core concepts of this ‘autopivot’ function. (The final autopivot function will be somewhat more complicated.)

```
def autopivot_simple(df, y, aggfunc, x_vars=[], overall_data_name='All Data'):
```

(continues on next page)

(continued from previous page)

```
'''This is a simple version of an 'autopivot' chart. Given a
DataFrame, a y variable, an aggregate function, and an arbitrary list
of variables, it will return a pivot table, an x variable,
and a y variable that can then get fed into a graphing function.

For more documentation on these steps,
see the full autopivot function within auto_pivot_and_graph.py,
which is located within the PFN_Dash_App_Demo section of PFN.
(I excluded that documentation from this code in order to
avoid redundancy.)'''

df_for_pivot = df.copy()
df_for_pivot

# Handling a situation in which no comparison variables were provided:
if len(x_vars) == 0:
    df_for_pivot[overall_data_name] = overall_data_name
    df_pivot = df_for_pivot.pivot_table(
        index=overall_data_name, values=y,
        aggfunc=aggfunc).reset_index()
    x_val_name = overall_data_name
    color = overall_data_name
    barmode = 'relative'
    index = [overall_data_name]

else:
    print("x_vars:", x_vars)

    # Creating a pivot table using the arguments provided:
    df_pivot = df_for_pivot.pivot_table(
        index=x_vars, values=y, aggfunc=aggfunc).reset_index()

    # Creating the x variable's column name:
    # (This name can then get fed into a graphing function.)
    x_val_name = ('/').join(x_vars)

    # Creating the values for the x_val_name column:
    # (The function does so by combining each row's values for
    # the fields represented in x_vars into a single string.)
    # It starts with the first x variable provided, then loops
    # through the rest of x_vars to add in the others.
    df_pivot[x_val_name] = df_pivot[x_vars[0]].astype('str')
    for i in range(1, len(x_vars)):
        df_pivot[x_val_name] = (
            df_pivot[x_val_name]
            + ' / ' + df_pivot[x_vars[i]].astype('str'))

return df_pivot, x_val_name, y, aggfunc
```

10.4 A basic autobar function

The following function uses the output of `autopivot_simple()` as inputs for a bar chart. It's quite limited relative to the final `autobar` function that will be applied later in this code, but it still demonstrates the utility of the `autopivot_simple()` function.

```
def autopbar_simple(df_pivot, x_val_name, y, aggfunc):
    '''This function creates a bar graph of a pivot table (such as one
    created by autopivot_simple.
    For more documentation on this function, view the full autopbar()
    function within within auto_pivot_and_graph.py.
    '''

    fig = px.bar(df_pivot, x=x_val_name, y=y, text_auto='%.0f',
                 title=f'{aggfunc.title()} {y} by {x_val_name}')
    return fig
```

10.5 Testing out `autopivot_simple()` and `autobar_simple()`

Let's try using these functions to graph average survey results by level and season. First, we'll call `autopivot_simple()` to generate inputs for `autobar_simple()`:

```
df_pivot, x_val_name, y, aggfunc = autopivot_simple(
    df_se, y='Score', aggfunc='mean', x_vars=['Level', 'Season'])

print(x_val_name, y, df_pivot.head())
```

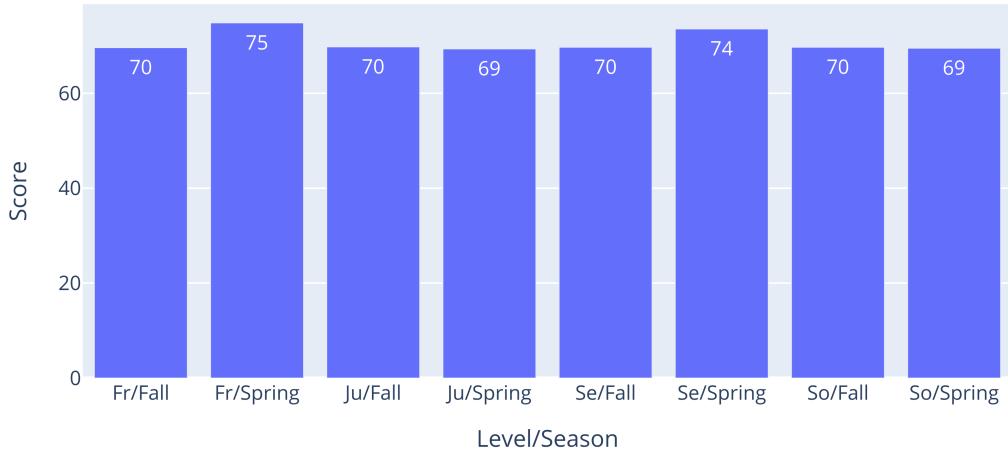
	x_vars: ['Level', 'Season']	Level/Season	Score	Level	Season	Score	Level/Season
0	Fr Fall	69.609774		Fr/Fall			
1	Fr Spring	74.833180		Fr/Spring			
2	Ju Fall	69.768957		Ju/Fall			
3	Ju Spring	69.350671		Ju/Spring			
4	Se Fall	69.698085		Se/Fall			

Next, we'll call `autobar_simple()` to convert these values into a bar chart:

```
fig = autobar_simple(
    df_pivot=df_pivot, x_val_name=x_val_name, y=y,
    aggfunc=aggfunc)

wadi(fig=fig, file_path='output/mean_score_by_level_and_season',
      display_type=display_type)
```

Mean Score by Level/Season



Note that we only needed to specify our chart parameters within `autopivot_simple()`; its output could then get passed to `autobar_simple()` without any further tweaking.

Let's try simplifying the process of creating these charts even more by defining a function called `autopivot_plus_bar_simple()`. This function, given the same list of inputs that we passed to `autopivot_simple()`, will call `autopivot_simple()` to create a pivot table, then immediately call `autobar_simple()` to generate a corresponding bar chart. It will then return both outputs for further use.

```
def autopivot_plus_bar_simple(df, y, aggfunc, x_vars=[], overall_data_name='All Data'):
    '''This function calls both autopivot_simple() and autobar_simple()
    in order to create both a pivot table and a bar.'''
    df_pivot, x_val_name, y, aggfunc = autopivot_simple(
        df=df, y=y, aggfunc=aggfunc, x_vars=x_vars,
        overall_data_name=overall_data_name)
    fig = autobar_simple(df_pivot=df_pivot, x_val_name=x_val_name, y=y,
                          aggfunc=aggfunc)
    return df_pivot, fig
```

We can now create the same figure shown above, along with its corresponding pivot table, using a single function call. The following output shows what this pivot table looks like:

```
df_pivot, fig = autopivot_plus_bar_simple(
    df_se, y='Score', aggfunc='mean', x_vars=['Level', 'Season'])
df_pivot.head()
```

```
x_vars: ['Level', 'Season']
```

Level	Season	Score	Level/Season
0 Fr	Fall	69.609774	Fr/Fall
1 Fr	Spring	74.833180	Fr/Spring
2 Ju	Fall	69.768957	Ju/Fall

(continues on next page)

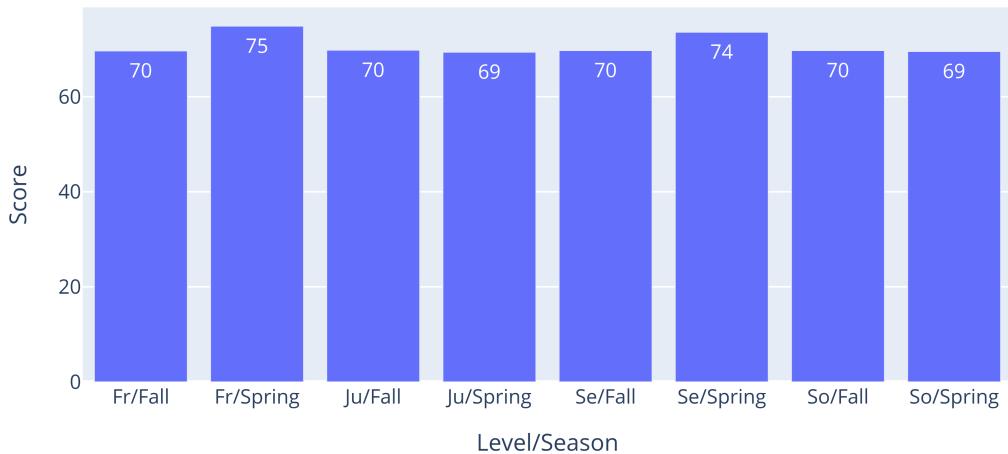
(continued from previous page)

3	Ju	Spring	69.350671	Ju/Spring
4	Se	Fall	69.698085	Se/Fall

And here's the image—which is identical to the one we created earlier.

```
wadi(fig=fig, file_path='output/mean_score_by_level_and_season',
     display_type=display_type)
```

Mean Score by Level/Season



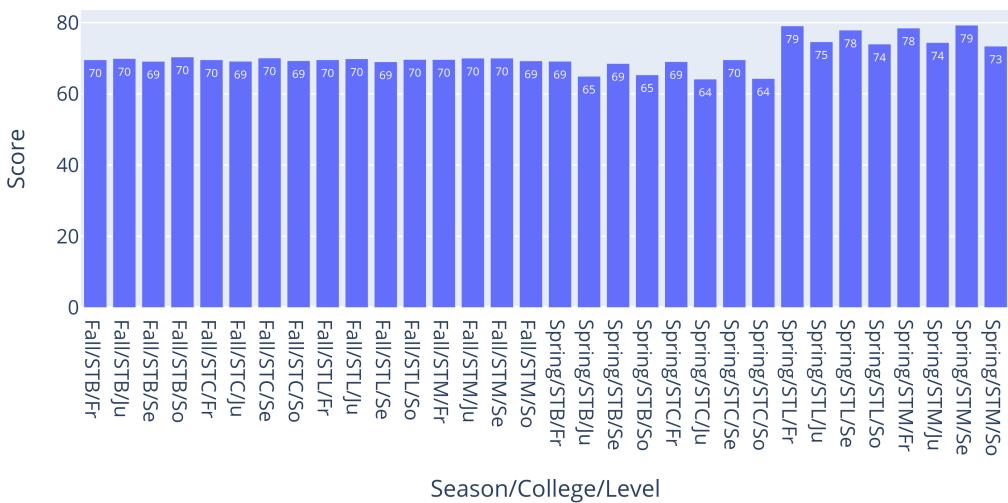
Next, we'll call `autopivot_plus_bar_simple()` to display average scores by season, college, and level:

```
df_pivot, fig = autopivot_plus_bar_simple(
    df_se, y='Score', aggfunc='mean', x_vars=[
        'Season', 'College', 'Level'])

wadi(fig=fig, file_path='output/mean_score_by_season_college_and_level',
      display_type=display_type)
```

```
x_vars: ['Season', 'College', 'Level']
```

Mean Score by Season/College/Level



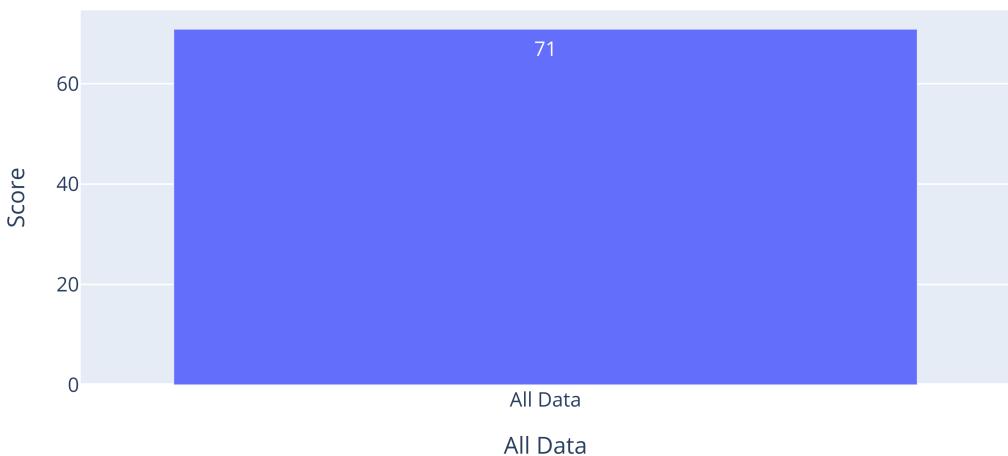
This chart would be both easier to interpret and more aesthetically pleasing with some color. Our simple autopivot and autobar functions don't handle color values, but their full-fledged versions, which we'll import shortly, do.

Finally, we'll create a chart that groups all data together:

```
df_pivot, fig = autopivot_plus_bar_simple(
    df_se, y='Score', aggfunc='mean', x_vars=[])

wadi(fig=fig, file_path='output/mean_score_by_all_data',
    display_type=display_type)
```

Mean Score by All Data



The above examples demonstrate how the `autopivot_simple()` and `autobar_simple()` functions can work together to generate all kinds of different table and chart combinations, thus eliminating the need to write pivot table and bar chart code for each set of comparison variables a user might want to analyze.

However, the basic charts that they produce leave much to be desired. There are no colors: the titles could use some work; and levels are sorted alphabetically rather than chronologically. Therefore, we'll now import more robust versions of these functions.

10.6 Importing autopivot and autobar functions

These functions are stored within the 'auto_pivot_and_graph.py' file within the PFN_Dash_App_Demo component of PFN's Online Visualizations section. The Dash App Demo project applies them to automatically create new pivot tables—and corresponding charts—based on the input variables requested by dashboard visitors.

I had originally kept a copy of these functions within this section as well, but that resulted in two redundant Python files that I would need to manually synchronize. Therefore, I decided to delete this section's copy and instead direct readers to the Online Visualizations section if they needed to reference the code.

At this point, I *do* suggest that you take some time to look through the `auto_pivot_and_graph.py` file; it contains detailed commentary to help you understand what's going on within each step.

```
sys.path.insert(2, '../Online_Visualizations/PFN_Dash_App_Demo')
from auto_pivot_and_graph import autopivot, autobar, autopivot_plus_bar
```

10.6.1 Demonstrating these functions

Let's try using these functions to graph average scores by the college, level, and season fields—just as we did in our previous chart. In order to simplify our x axis values (and improve the chart's appearance), we'll pass `Season` to this function's `color` parameter. In addition, although we're passing `Level For Sorting` to our `x_vars` argument (so that levels will appear within the correct order), we won't want that argument to show up within our x axis labels, so we'll add that value to `x_vars_to_exclude`.

First, here's a look at the output of `autopivot()` on its own:

```
(df_pivot, x_val_name, y, color, barmode,
x_var_count, index, aggfunc) = autopivot(
    df=df_se, y='Score',
    x_vars=['College', 'Level For Sorting', 'Level'],
    color='Season',
    x_vars_to_exclude=['Level For Sorting'],
    aggfunc='mean')
print(x_val_name, y, color, barmode)
df_pivot.head(5)
```

```
x_vars: ['College', 'Level For Sorting', 'Level']
index prior to pivot_table() call: ['College', 'Level For Sorting', 'Level',
                                     'Season']
College/Level Score Season group
```

College	Level For Sorting	...	Score	College/Level
0	STB	0	...	69.593583 STB/Fr
1	STB	0	...	69.177235 STB/Fr

(continues on next page)

(continued from previous page)

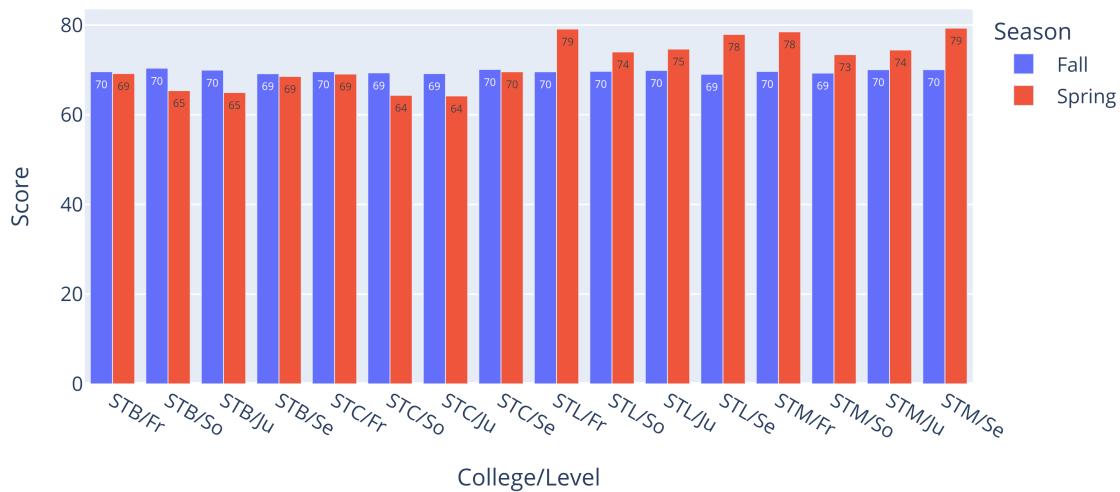
2	STB	1	...	70.377306	STB/So
3	STB	1	...	65.377306	STB/So
4	STB	2	...	69.950769	STB/Ju
[5 rows x 6 columns]					

And here's what `autobar()` produces when provided the arguments generated within `autopivot()`:

```
fig = autobar(
    df_pivot=df_pivot,
    x_val_name=x_val_name, y=y,
    color=color, barmode=barmode,
    x_var_count=x_var_count, index=index,
    aggfunc=aggfunc,
    text_auto='0f')

wadi(fig=fig,
      file_path='output/mean_score_by_college_level_and_season_autopivot',
      display_type=display_type)
```

Mean Score by College, Level, and Season



This chart is certainly an improvement over our previous one. The levels now show up in the correct order (Fr, So, Ju, and Se), and the Season variable no longer appears within the x axis labels. (Since it's already featured within the legend, its presence there would be redundant.) In addition, our title now features better formatting also.

We can simplify this code even further by calling `autopivot_plus_bar()`:

```
fig = autopivot_plus_bar(
    df=df_se, y='Score',
    x_vars=['College', 'Level For Sorting', 'Level'],
    color='Season',
    x_vars_to_exclude=['Level For Sorting', 'Season For Sorting'],
```

(continues on next page)

(continued from previous page)

```

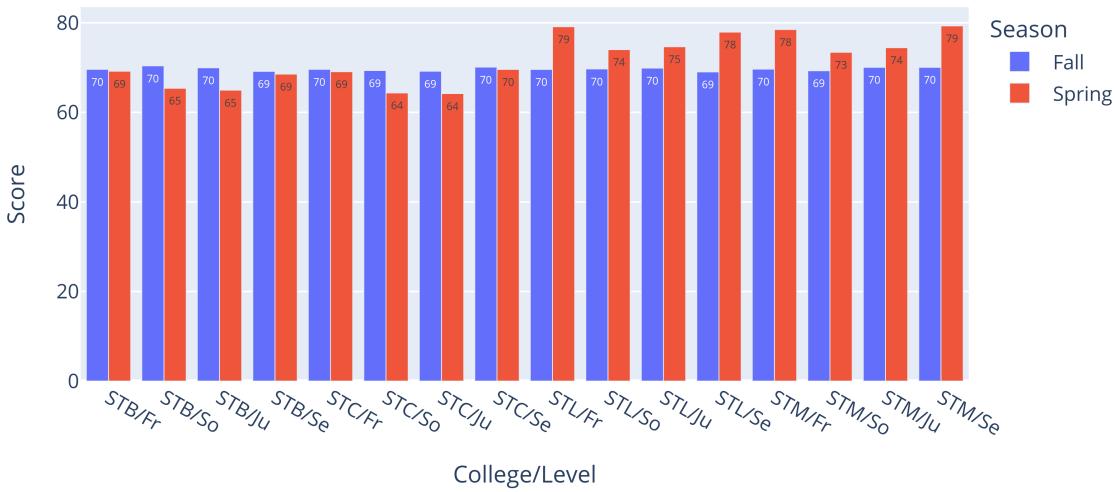
aggfunc='mean',
text_auto='0f')

wadi(fig=fig,
      file_path='output/mean_score_by_college_level_and_\
season_autopivot_v2', display_type=display_type)

x_vars: ['College', 'Level For Sorting', 'Level']
index prior to pivot_table() call: ['College', 'Level For Sorting', 'Level',
→'Season']

```

Mean Score by College, Level, and Season



Here's an alternative chart with just 2 comparison variables: (This chart demonstrates how to use autopivot_plus_bar()'s text_auto argument to add extra precision to our text labels—though this level of detail is overkill in this case.

```

fig = autopivot_plus_bar(
    df=df_se, y='Score',
    x_vars=['Level', 'Season For Sorting'],
    color='Season',
    x_vars_to_exclude=['Season For Sorting'],
    aggfunc='mean',
    text_auto='0.3f')

wadi(fig=fig,
      file_path='output/mean_score_by_level_and_season_autopivot',
      display_type=display_type)

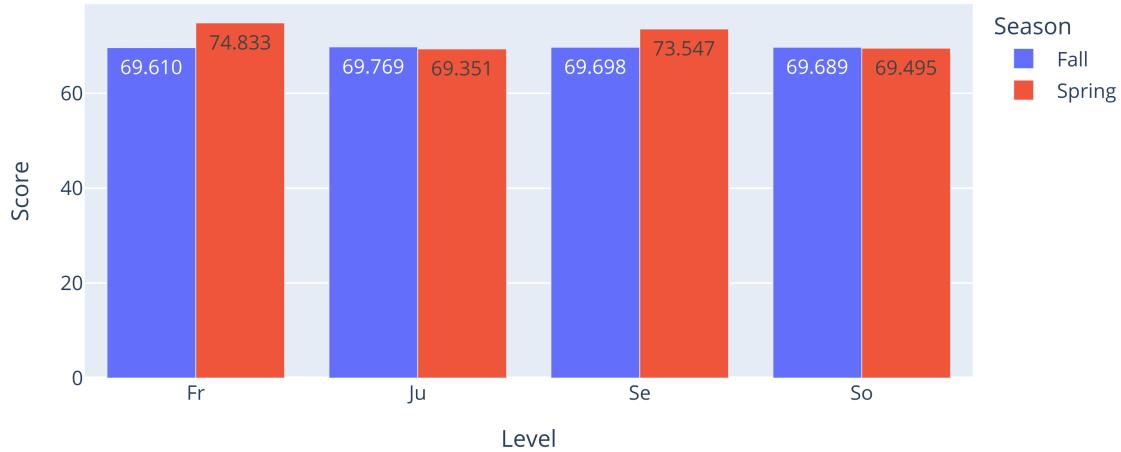
```

```

x_vars: ['Level', 'Season For Sorting']
index prior to pivot_table() call: ['Level', 'Season For Sorting', 'Season']

```

Mean Score by Level and Season

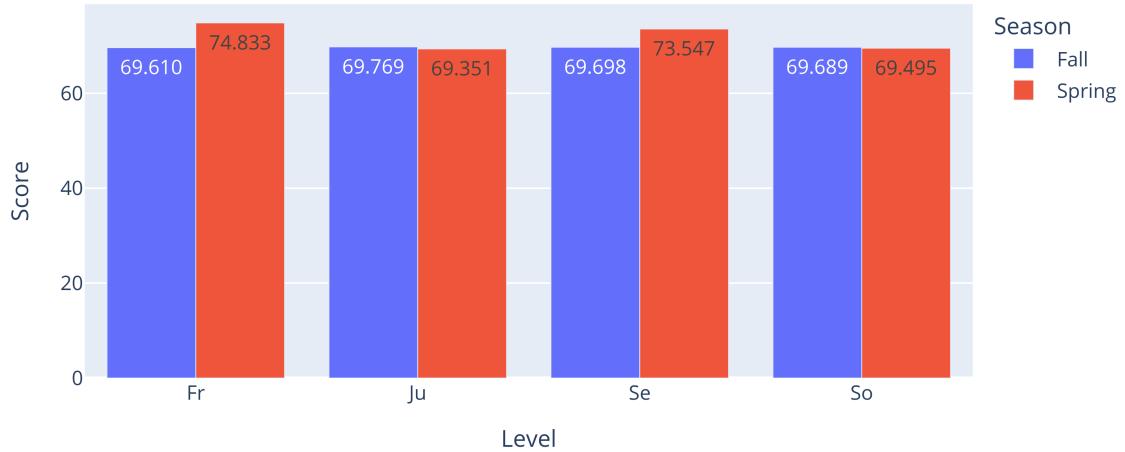


Note that the above output is identical to the output of the following cell, which uses an aggregated dataframe (df_se_pivot) as its data source. This identical output was achieved by adding in a weight column (Responses).

```
fig = autopivot_plus_bar(  
    df=df_se_pivot, y='Score',  
    x_vars=['Level'],  
    color='Season',  
    x_vars_to_exclude=['Season For Sorting'],  
    aggfunc='mean', weight_col='Responses',  
    text_auto='.3f')  
  
wadi(fig=fig,  
    file_path='output/mean_score_by_level_and_season_autopivot_v2',  
    display_type=display_type)
```

```
x_vars: ['Level']  
index prior to pivot_table() call: ['Level', 'Season']
```

Mean Score by Level and Season



In contrast, the following output, which uses `df_se_pivot` as well but does *not* reference a weight column, differs slightly from the above two graphs. This is because it is falling into the ‘average of averages’ error noted within Descriptive Stats: Part 2.

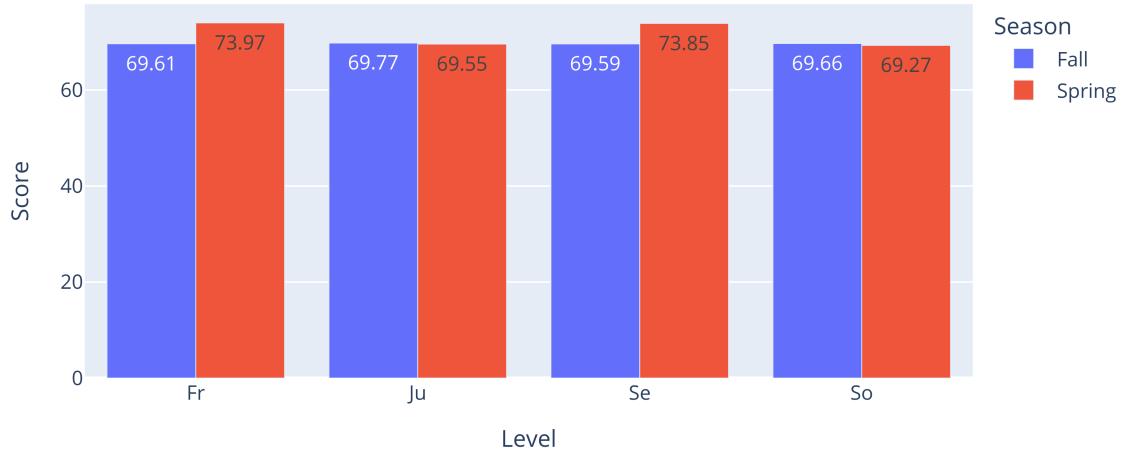
```
fig = autopivot_plus_bar(
    df=df_se_pivot, y='Score',
    x_vars=['Level'],
    color='Season',
    x_vars_to_exclude=['Season For Sorting'],
    aggfunc='mean').update_layout(title='Mean Score by Level and \
Season (with incorrect averages)')

# As shown by this update_layout() call, further modifications can be
# made as needed to figures returned by autopivot_plus_bar().
```

wadi(fig=fig,
 file_path='output/mean_score_by_level_and_season_autopivot_aoa_error',
 display_type=display_type)

```
x_vars: ['Level']
index prior to pivot_table() call: ['Level', 'Season']
```

Mean Score by Level and Season (with incorrect averages)

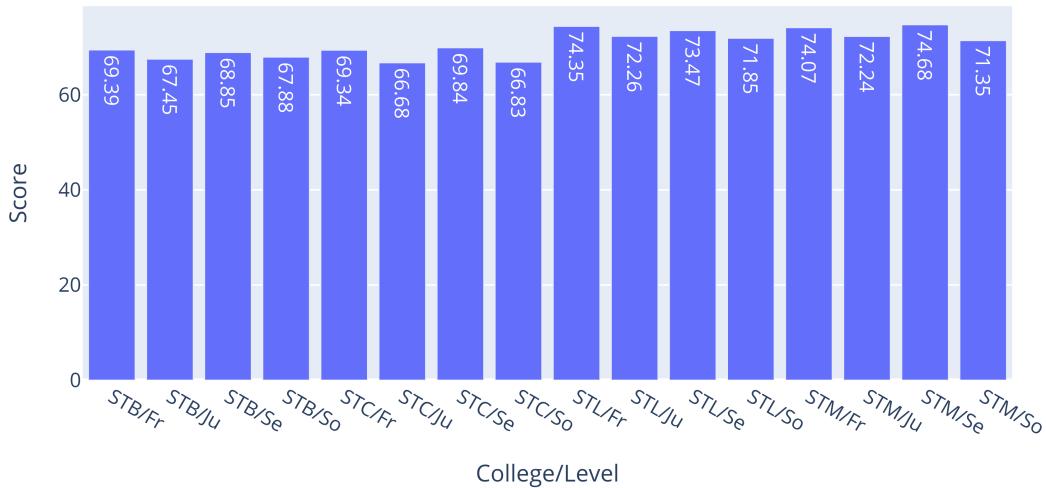


These functions also allow the color argument to be skipped, though the result is rather plain:

```
fig = autopivot_plus_bar(  
    df=df_se, y='Score',  
    x_vars=['College', 'Level'],  
    aggfunc='mean')  
  
wadi(fig=fig,  
      file_path='output/mean_score_by_college_and_level_\  
autopivot_colorless',  
      display_type=display_type)
```

```
x_vars: ['College', 'Level']  
index prior to pivot_table() call: ['College', 'Level']
```

Mean Score by College and Level

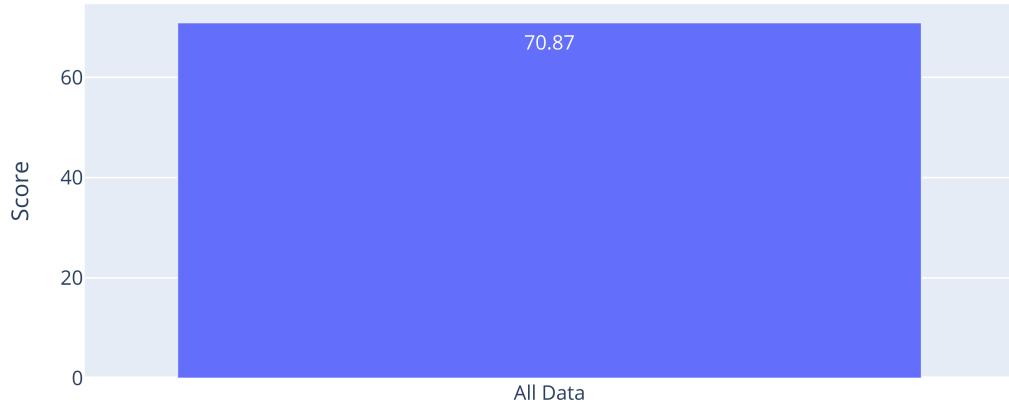


They also allow an empty list of comparison values to get passed to x_vars, thus producing a single bar that represents the average of all values in the DataFrame.

```
fig = autopivot_plus_bar(
    df=df_se, y='Score',
    x_vars=[],
    color=None,
    x_vars_to_exclude=[],
    aggfunc='mean')

wadi(fig=fig,
      file_path='output/overall_mean_score_autopivot',
      display_type=display_type)
```

Overall Mean Score



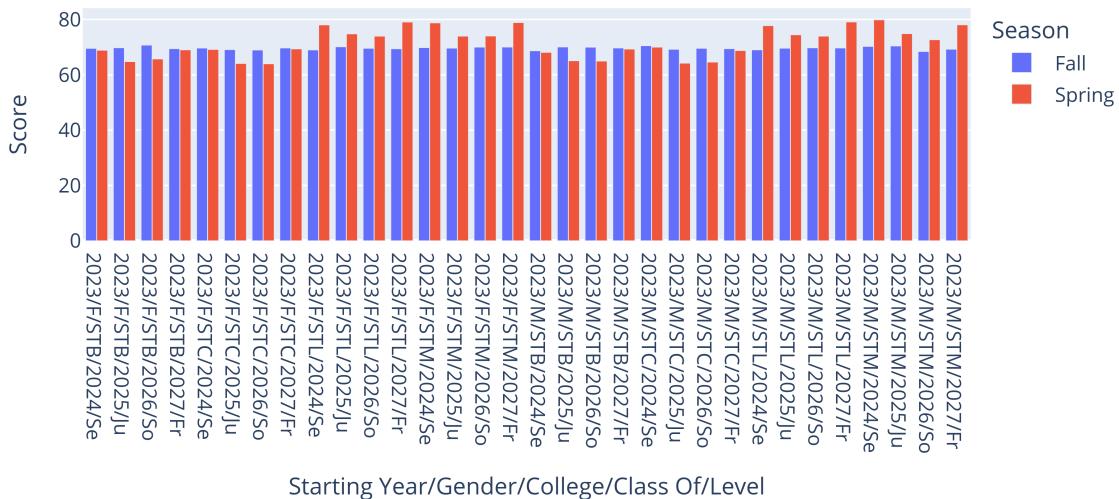
At the opposite extreme, we can graph 6 comparison variables within the same chart (though the result is not the easiest to read):

```
fig = autopivot_plus_bar(
    df=df_se, y='Score',
    x_vars=[
        'Starting Year', 'Season', 'Gender', 'College', 'Class Of',
        'Level For Sorting', 'Level'],
    color='Season',
    x_vars_to_exclude=['Level For Sorting'],
    aggfunc='mean', text_auto=False)

wadi(fig=fig,
      file_path='output/mean_score_by_6_variables_autopivot',
      display_type=display_type)
```

```
x_vars: ['Starting Year', 'Gender', 'College', 'Class Of', 'Level For Sorting',
         ↴'Level']
index prior to pivot_table() call: ['Starting Year', 'Gender', 'College', 'Class Of',
         ↴', 'Level For Sorting', 'Level', 'Season']
```

Mean Score by Starting Year, Gender, College, Class Of, Level, and Season



10.7 Conclusion

The `autopivot()`, `autobar()`, and `autopivot_plus_bar()` functions shown here can help speed up the process of creating standalone Plotly charts. However, these functions will prove even more useful within the Online Visualizations section of Python for Nonprofits—as they'll allow us to turn a small set of user inputs into a wide variety of charts and tables.

Now that we've learned how to use Plotly to create various types of charts, we'll shift our focus to mapping—an area in which Plotly also excels.

MAPPING CENSUS DATA USING PLOTLY (WORK IN PROGRESS)

By Kenneth Burchfiel

Released under the MIT license

This script will demonstrate how to use Plotly's `px.choropleth()` and `px.choropleth_map()` functions to create *choropleth maps* that visualize population growth rates. (These growth rates were calculated using `census_data_imports_v2.ipynb` (https://github.com/kburchfiel/pfn/blob/main/Census_Data_Imports/census_data_imports_v2.ipynb) within PFN's Census Data Imports section. This notebook will also show how to improve maps' readability by basing choropleth colors on percentiles.

When running the notebook on your computer, I recommend setting the 'render_for_pdf' variable within the Appendix's helper_funcs.py file to False in order to allow interactive, rather than static maps, to appear. (This will also make more DataFrame columns visible within the notebook's output.

A primer on choropleth maps

Choropleth maps assign different colors to different regions depending on some underlying data point. Perhaps the most famous example of choropleth maps, at least in the US, are presidential election maps. Recent versions of these maps assign red and blue colors to states that are won by the Republican and Democratic nominees for president, respectively. They're a ubiquitous sight on election nights, as they provide useful overviews of which states have been called for a particular candidate so far.

Our net migration maps will differ from these election maps in that colors will be assigned based on numerical data (population growth rates) rather than categorical data (presidential parties). States and counties with the highest growth rates will be colored orange whereas those with the lowest rates will be colored purple.

11.1 A note on Plotly vs. Folium

I used to use Folium (<https://python-visualization.github.io/folium/latest/>) as my main Python mapping library. However, I decided to switch to Plotly for two main reasons:

1. The process of creating static copies of maps within Python is much, much simpler with Plotly than with Folium. The latter process is absolutely doable, but I found it challenging to create matching static and interactive copies of the same map.
2. Plotly makes it easy to render Alaska and Hawaii near the lower 48 US states; this greatly simplifies the process of creating static choropleth maps that show the entire US.

However, in your own work, you may want (or need) to use Folium rather than Plotly. In that case, make sure to check out the `choropleth_maps_with_folium.ipynb` notebook and its corresponding `folium_choropleth_map_functions.py` Python file.

```
import time
start_time = time.time()
import pandas as pd
pd.set_option('display.max_columns', 1000)
import geopandas
import plotly.express as px
import plotly.graph_objects as go
import numpy as np
from IPython.display import Image # Source: StackOverflow user 'zach' at
# https://stackoverflow.com/a/11855133/13097194

import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, wadi
display_type = config_notebook(display_max_columns = 5)

from plotly_choropleth_map_functions import update_and_save_plotly_map, \
gen_choropleth
```

11.2 Importing data to be mapped

Choropleth maps require two sets of data: (1) boundary data that allows outlines of regions to get drawn and (2) a metric that determines how those boundaries should get colored. The following cells will import county-level boundary data and population growth metrics, thus allowing us to visualize population increases and decreases by county.

11.2.1 Importing 2021 County Shapefiles

These shapefiles show the outlines of each US county. They can be accessed by downloading the shapefile folder located at https://www2.census.gov/geo/tiger/GENZ2021/shp/cb_2021_us_county_500k.zip. (If this link doesn't work, try navigating to <https://www.census.gov/geographies/mapping-files/time-series/geo/cartographic-boundary.2021.html>; scrolling down to the **Counties** header; and then clicking on the 'shapefile' link that appears to the right of the "1 : 500,000 (national)" text.)

This shapefile folder contains unzipped 1:500,000-scale versions of the county shapefiles available on the Census website. (I could also have downloaded less detailed shapefiles, which feature smaller file sizes, but it's easy to create simplified versions of these outlines via Python—which I'll demonstrate in the following cell.)

Note that each folder contains a number of files. My experience has been that *all* of these files need to be present in the folder containing a given shapefile in order for that shapefile to work correctly within Python. Your experience may vary, though.

Note: I chose to import 2021 shapefiles because copies for later years did not contain the original Connecticut counties found within my corresponding population growth dataset. If your own dataset references Connecticut's planning regions (https://en.wikipedia.org/wiki/Councils_of_governments_in_Connecticut) rather than counties, you'll most likely want to use a county dataset from 2022 or later.

```
gdf_counties = geopandas.read_file(
    'shapefiles/cb_2021_us_county_500k/cb_2021_us_county_500k.shp')
gdf_counties['geometry'] = gdf_counties['geometry'].simplify(
    tolerance = 0.005)
# Creating a column that combines county names and long state names:
# (This column can then serve as a key for an upcoming merge.)
```

(continues on next page)

(continued from previous page)

```
# Note that 'STATE_NAME' is used rather than 'STUSPS' for the state
# component of this column in order to match the format used within
# our demographic dataset.
gdf_counties.insert(
    0, 'County/State',
    gdf_counties['NAMELSAD'] + ', ' + gdf_counties['STATE_NAME'])
gdf_counties.head()
```

	County/State	STATEFP	...	AWATER	\
0	Riley County, Kansas	20	...	32047392	
1	Ringgold County, Iowa	19	...	8723135	
2	Carbon County, Montana	30	...	35213028	
3	Bear Lake County, Idaho	16	...	191364281	
4	Buffalo County, Wisconsin	55	...	87549529	

	geometry
0	POLYGON ((-96.96169 39.22008, -96.95872 39.566...)
1	POLYGON ((-94.47121 40.57082, -94.47078 40.899...)
2	POLYGON ((-109.79867 45.16734, -109.68779 45.1...)
3	POLYGON ((-111.63452 42.57034, -111.60312 42.5...)
4	POLYGON ((-92.08384 44.412, -92.04391 44.51238...)

[5 rows x 14 columns]

```
gdf_counties.query("STATEFP == '09'") # Confirming that this dataset
# contains Connecticut's former counties rather than planning regions.
# (Our population growth dataset uses these counties as well, so we'll
# be able to merge the county data and boundaries together without
# any issues.)
```

	County/State	STATEFP	...	AWATER	\
60	New London County, Connecticut	09	...	276863144	
306	Fairfield County, Connecticut	09	...	549280913	
...	
2936	Litchfield County, Connecticut	09	...	62350378	
3022	Hartford County, Connecticut	09	...	40617843	

	geometry
60	MULTIPOLYGON ((((-72.2239 41.29346, -72.22523 4...
306	MULTIPOLYGON ((((-73.21763 41.14235, -73.21499 ...
...	...
2936	POLYGON ((-73.51795 41.67086, -73.48731 42.049...)
3022	POLYGON ((-73.02954 41.96661, -73.00876 42.038...))

[8 rows x 14 columns]

11.2.2 Importing growth data for US counties

The following table was created within the Census Data Imports section of Python for Nonprofits.

```
df_acs_county_growth = pd.read_csv(
    f'../Census_Data_Imports/Datasets/\\
grad_destinations_acs_county_data.csv')
# Renaming the NAME column in order to avoid a conflict with our shapefile
# dataset's NAME column:
df_acs_county_growth.rename(
    columns = {'NAME':'County/State'},
    inplace = True)
df_acs_county_growth.head()
```

	County/State	county	...	\
0	Abbeville County, South Carolina	1	...	
1	Acadia Parish, Louisiana	1	...	
2	Accomack County, Virginia	1	...	
3	Ada County, Idaho	1	...	
4	Adair County, Iowa	1	...	

	2016-2021 Total_Pop_25_to_29 % Change Rank	\
0	1357.0	
1	2812.0	
2	1426.0	
3	869.0	
4	588.0	

	2016-2021 Total_Pop_25_to_29 % Change Percentile	
0	56.829035	
1	10.506208	
2	54.632283	
3	72.365489	
4	81.311684	

[5 rows x 75 columns]

Merging our Census data into our county shapefiles:

(We could also have used state and county codes for this merge.)

```
gdf_counties_and_growth_stats = gdf_counties.merge(
    df_acs_county_growth, on = 'County/State', how = 'inner')
# Limiting the results to just the 50 US states plus DC:
gdf_counties_and_growth_stats.query("STUSPS != 'PR'", inplace = True)
gdf_counties_and_growth_stats.head()
```

	County/State	STATEFP	...	\
0	Riley County, Kansas	20	...	
1	Ringgold County, Iowa	19	...	
2	Carbon County, Montana	30	...	
3	Bear Lake County, Idaho	16	...	
4	Buffalo County, Wisconsin	55	...	

	2016-2021 Total_Pop_25_to_29 % Change Rank	\
0	2912.0	

(continues on next page)

(continued from previous page)

```

1                      2979.0
2                      2323.0
3                      2684.0
4                      2289.0

2016-2021 Total_Pop_25_to_29 % Change Percentile
0                  7.322509
1                  5.189430
2                 26.074499
3                 14.581344
4                 27.156956

[5 rows x 88 columns]

```

In order to create our map, we'll group a chosen metric (e.g. nominal population growth over a 10-year period) into a specific number of bins. The following line specifies how many bins we'd like to use:

Determining which set of growth data to visualize within our map:

```

year_range_end = 2021
year_range_start = year_range_end - 10
year_range = f'{year_range_start}-{year_range_end}'
year_range

```

```
'2011-2021'
```

Filtering our DataFrame to include only those fields that we'll need for our map and removing rows that lack entries for the field to be visualized:

```

data_col = f'{year_range} Total_Pop % Change'
# Specifying an additional rows that will get included within the maps'
# hover data (e.g. information that pops up when viewers move their mouse
# over a particular region):
# Percentile information will get added in automatically by a mapping
# function that we'll use later within this script (as long as its
# 'percentile' argument is set to True), so we won't add that as an option
# here.
extra_hover_data = [
    f'{data_col} Rank']

gdf_counties_for_map = gdf_counties_and_growth_stats.copy().dropna(
    subset = data_col).sort_values(data_col, ascending = False).set_index(
    'County/State')[[
        'geometry', data_col, f'{year_range} Total_Pop % Change Percentile']
    + extra_hover_data].copy()
gdf_counties_for_map

```

County/State	geometry
McKenzie County, North Dakota	POLYGON ((-104.04531 47.33014, -104.04412 47.9...))
Williams County, North Dakota	POLYGON ((-104.04412 47.99608, -104.04758 48.6...))
...	...
Blaine County, Nebraska	POLYGON ((-100.26762 42.08615, -99.68696 42.08...))
Kenedy County, Texas	MULTIPOLYGON ((((-97.39839 26.86789, -97.38686 ...)))

(continues on next page)

(continued from previous page)

```

2011-2021 Total_Pop % Change \
County/State
McKenzie County, North Dakota           119.770042
Williams County, North Dakota           74.453416
...
Blaine County, Nebraska                 -40.854701
Kenedy County, Texas                   -48.000000

2011-2021 Total_Pop % Change Percentile \
County/State
McKenzie County, North Dakota           100.000000
Williams County, North Dakota           99.968122
...
Blaine County, Nebraska                 0.063755
Kenedy County, Texas                   0.031878

2011-2021 Total_Pop % Change Rank
County/State
McKenzie County, North Dakota           1.0
Williams County, North Dakota           2.0
...
Blaine County, Nebraska                 3136.0
Kenedy County, Texas                   3137.0

[3137 rows x 4 columns]

```

Important: In order for your mapping code to work correctly, it is crucial that your indices contain the location values that you wish to use (e.g. county or state names). If you instead use a regular numerical index (e.g. 0 ... n), it appears that px.choropleth() will interpret those index names as state or county FIPS codes. This will likely cause *the wrong data to get shown for certain locations*.

Defining a custom color scale: (This step is optional, but I wanted to demonstrate how you can send a custom scale to your map as needed. Plotly will interpolate between these colors as needed when creating the map.)

```

custom_color_scale = ['#800080', '#ffffff', '#ffa500'] # Purple,
# white, and orange. See https://en.wikipedia.org/wiki/Web\_colors
# for more details on basic html colors like these.

```

Specifying the default static and html folder paths for maps:

```

static_file_folder = 'map_screenshots'
html_file_folder = 'maps'

```

11.3 Creating an initial choropleth map with a linear scale

The following code successfully creates a choropleth map using Plotly; however, there's a significant problem with this map, which I'll discuss following its render.

```

fig = px.choropleth(
    gdf_counties_for_map,
    geojson = gdf_counties_for_map['geometry'],

```

(continues on next page)

(continued from previous page)

```

locations = gdf_counties_for_map.index,
color = data_col,
scope = 'usa',
color_continuous_scale = custom_color_scale,
basemap_visible = False,
title = f'Population % Change by County from {year_range_start} \
to {year_range_end}'

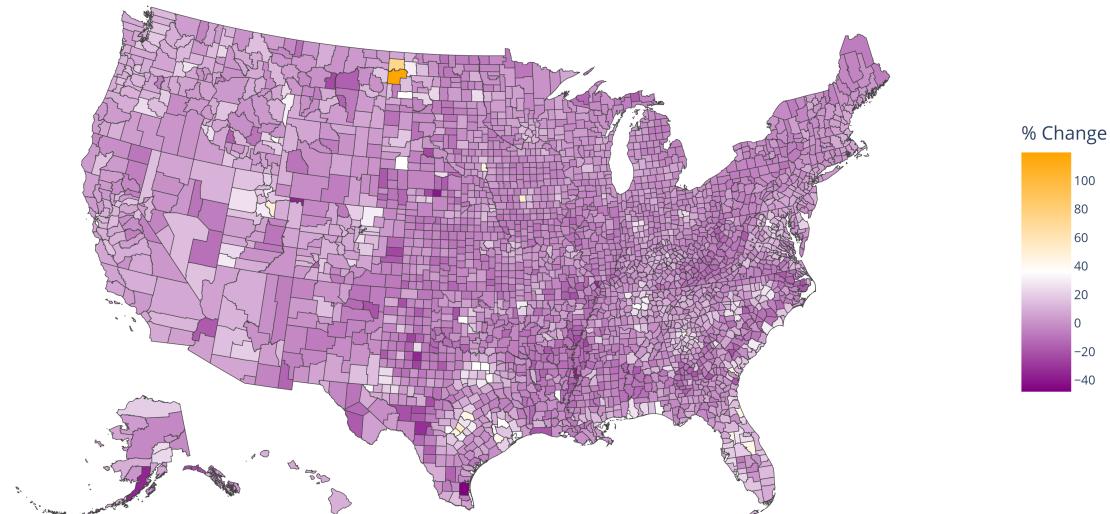
# Shortening our colorbar title:
fig.update_coloraxes(colorbar_title = "% Change")
# (This code was based on the documentation found at
# https://plotly.com/python/reference/layout/coloraxis/)

# I found that the basemap's presence made certain coastal county borders
# appear a bit 'messy', so I decided to disable it.
# Saving this map as an image file: (
# Consult the update_and_save_plotly_map() documentation within
# plotly_choropleth_map_functions.py for more details on the function
# called below.)
filename = 'sample_linear_map'
update_and_save_plotly_map(
    fig, static_file_folder = static_file_folder, filename = filename,
    save_html = False)

# Calling wadi() to display the map we just created:
# (Because a .png version of this image has already been created,
# we'll set generate_image to False.)
wadi(fig, static_file_folder+'/'+filename, generate_image = False,
     display_type = display_type)

```

Population % Change by County from 2011 to 2021



The problem, as you might have noticed already, is that the colors within this map appear rather uniform. That's because the outliers within the map (e.g. counties with unusually high population growth or declines) have forced most of the other counties to share a similar color.

11.4 Creating map with a percentile scale

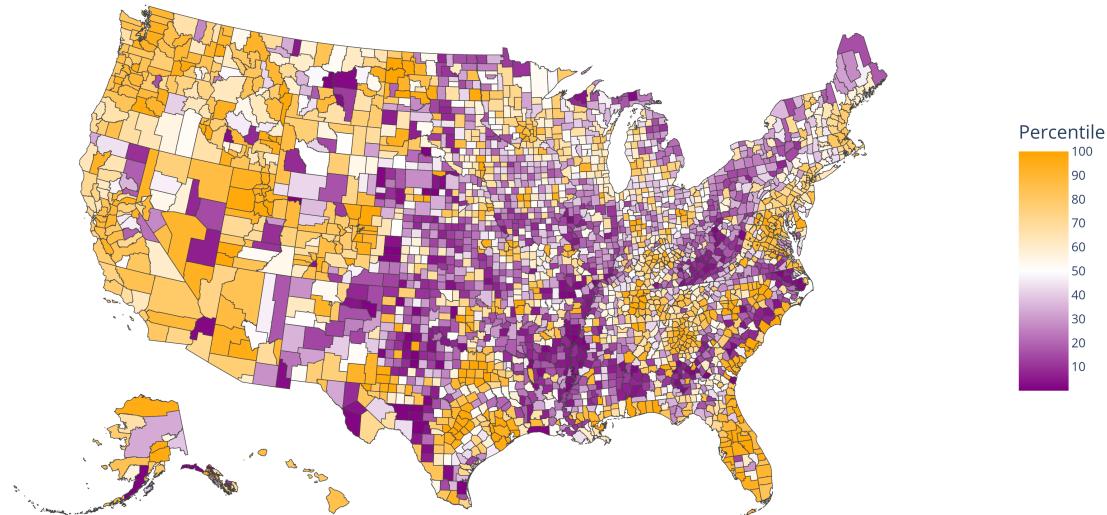
In this case, a better approach will be to base our colors on *percentiles* rather than actual county growth values. Because percentiles are distributed pretty evenly, we'll see a nice range of colors within our map.

The following map demonstrates this approach. We'll simply replace the existing `color` argument (`2011-2021 % Change Percentile`) with its corresponding percentile column:

```
fig = px.choropleth(
    gdf_counties_for_map,
    geojson = gdf_counties_for_map['geometry'],
    locations = gdf_counties_for_map.index,
    color = '2011-2021 Total_Pop % Change Percentile',
    hover_data = [data_col] + extra_hover_data,
    color_continuous_scale = custom_color_scale,
    # color_discrete_sequence = color_list,
    scope = 'usa',
    title = f'Population % Change by County from {year_range_start} \
    to {year_range_end}',
    basemap_visible = False)
fig.update_coloraxes(colorbar_title = "Percentile")

filename = 'sample_percentile_map'
update_and_save_plotly_map(
    fig, static_file_folder = static_file_folder, filename = filename,
    save_html = False)
wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)
```

Population % Change by County from 2011 to 2021



This map is much more colorful—and, as a result, easier to interpret—than the version that used a linear color scale. However, a significant drawback is that the legend no longer shows the actual percentage change values on which the percentiles are based. Viewers can hover over counties to identify these values, but this approach is more cumbersome and won't work at all when the map is converted to a static image.

Thankfully, using Plotly's `update_coloraxes()` function, we can replace the **percentiles in the colorbar** with **the actual percentage change values to which they correspond**. We'll do so by modifying two properties of the colorbar:

1. `colorbar_tickvals`, which specifies where along the colorbar (e.g. from the lowest percentile to the highest) we'd like to add text labels. (We'll set this property to specific percentile ranks within our dataset.)
2. `colorbar_ticktext`, which specifies what text we'd like to place at those values. (We'll set this property to the actual values within our dataset that correspond to the percentile ranks that we passed to `colorbar_tickvals`.)

Making changes will require some additional code, but the result will be well worth it.

11.4.1 Specifying how many data points we'd like to include within our color scale

```
colorscale_tick_count = 11
quantile_increment = 1/(colorscale_tick_count - 1)
# This variable will determine
# the distance between increments within np.arange(), which we'll call
# below in order to determine which quantiles to retrieve from
# the DataFrame.
# I subtracted 1 from the denominator so that the final number of
# quantiles (which will include both a minimum and maximum value)
# will match the 'quantile quantity' specified within
# colorscale_tick_count.
quantile_increment
```

0.1

We've now designated how many values to show within the colorbar. Next, we'll want to figure out which actual percentile ranks within our dataset can function as relatively equally-spaced colorscale label locations.

11.4.2 Using `quantile_increment` to specify which quantiles to retrieve

`quantile_range` will increase from 0 to 1 by the amount specified in `quantile_increment`. `quantile_increment/2` is added to 1 in the function call in order to ensure that the output will include, but also stop at, 1.

```
quantile_range = np.arange(0, 1+quantile_increment/2,
                         quantile_increment)
quantile_range
```

array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.])

11.4.3 Finding the actual percentile ranks within our DataFrame that correspond to these quantiles:

(A quantile of 0 refers to the smallest percentile rank in the dataset, while a quantile of 1 refers to the highest percentile rank; a quantile of 0.5, if specified, refers to the median percentile rank.)

Note: in an earlier version of this code, I created revised colorbars using the following steps:

1. I used `quantile()` to figure out which data column values to pass to the `colorbar_ticktext` property discussed earlier.

2. I then multiplied quantile_range() by 100 to create a range of percentile ranks (called percentile_range) that could be passed to the `colorbar_tickvals` property.

However, this approach had a significant flaw: because datasets often wouldn't have an exact copy of one of the percentile ranks in percentile_range, the scores that replaced those percentile ranks often corresponded to a slightly different percentile rank. (In one actual example, Pandas calculated the quantile of 0.5 (e.g. the 50.000th percentile) as 6.289%, but this percentage actually represented the 50.98th percentile. It would therefore be inaccurate to replace the 50th-percentile marker in the colorbar with this value.)

In order to avoid this issue, I revised the code so that it would find the quantiles of the actual *percentile ranks* in our dataset. This way, all of the percentile ranks passed to colorbar_tickvals would have actual corresponding values that could be passed to colorbar_ticktext.

```
percentile_col = f'{year_range} Total_Pop % Change Percentile'  
# Finding the quantiles of these percentiles: (See notes above for  
# clarification on this step.)  
gdf_county_percentile_quantiles = gdf_counties_for_map[  
    percentile_col].quantile(  
        quantile_range, interpolation = 'lower').sort_values(  
        ascending = False)  
# The 'lower' interpretation method will help ensure that only  
# actual percentile ranks present in the DataFrame get retrieved.  
# This is a necessary prerequisite for using df.query() to locate  
# the scores that match these percentile ranks, which we'll do shortly.  
print(f"len(gdf_county_percentile_quantiles) {len(gdf_county_percentile_quantiles)} quantiles created.")  
gdf_county_percentile_quantiles
```

11 quantiles created.

```
1.0      100.000000  
0.9      89.990437  
...  
0.1      10.009563  
0.0      0.031878  
Name: 2011-2021 Total_Pop % Change Percentile, Length: 11, dtype: float64
```

```
gdf_county_percentile_quantile_list = (  
    gdf_county_percentile_quantiles.to_list())  
gdf_county_percentile_quantile_list
```

```
[100.0,  
 89.99043672298374,  
 79.98087344596748,  
 70.00318775900541,  
 59.99362448198916,  
 50.01593879502709,  
 40.00637551801084,  
 29.99681224099458,  
 20.019126554032518,  
 10.009563277016255,  
 0.0318775900541919]
```

Determining the actual data_col values within the dataset that correspond to these percentile ranks:

Note that only one row will be retained for each percentile rank; this will ensure that the lengths of the percentile rank and percentile score lists match. (If these lengths differed, we could encounter issues when trying to replace the former

with the latter within our colorbar.)

```
df_county_percentile_rank_score_pairs = gdf_counties_for_map.query(
    f" ` {percentile_col} ` in @gdf_county_percentile_quantile_list "
).drop_duplicates(percentile_col).copy()
df_county_percentile_rank_score_pairs
```

County/State	geometry
McKenzie County, North Dakota	POLYGON ((-104.04531 47.33014, -104.04412 47.9...)
Lubbock County, Texas	POLYGON ((-102.08573 33.82468, -101.56358 33.8...)
...	...
Cimarron County, Oklahoma	POLYGON ((-103.00243 36.5004, -103.0022 37.000...)
Kenedy County, Texas	MULTIPOLYGON ((((-97.39839 26.86789, -97.38686 ...
	2011-2021 Total_Pop % Change \
County/State	
McKenzie County, North Dakota	119.770042
Lubbock County, Texas	12.473074
...	...
Cimarron County, Oklahoma	-8.648431
Kenedy County, Texas	-48.000000
	2011-2021 Total_Pop % Change Percentile \
County/State	
McKenzie County, North Dakota	100.000000
Lubbock County, Texas	89.990437
...	...
Cimarron County, Oklahoma	10.009563
Kenedy County, Texas	0.031878
	2011-2021 Total_Pop % Change Rank
County/State	
McKenzie County, North Dakota	1.0
Lubbock County, Texas	315.0
...	...
Cimarron County, Oklahoma	2824.0
Kenedy County, Texas	3137.0

[11 rows x 4 columns]

Rounding these percentile scores (so that they'll display better within our colorbar), then saving them to a list:

```
round_val = 1
```

```
gdf_county_score_series = df_county_percentile_rank_score_pairs[
    data_col]
if round_val is not None:
    gdf_county_score_series = gdf_county_score_series.round(round_val)
gdf_county_score_list = gdf_county_score_series.to_list()
gdf_county_score_list
```

[119.8, 12.5, 7.4, 4.1, 1.8, -0.2, -1.9, -3.8, -5.6, -8.6, -48.0]

11.4.4 Rendering an improved percentile-based choropleth map

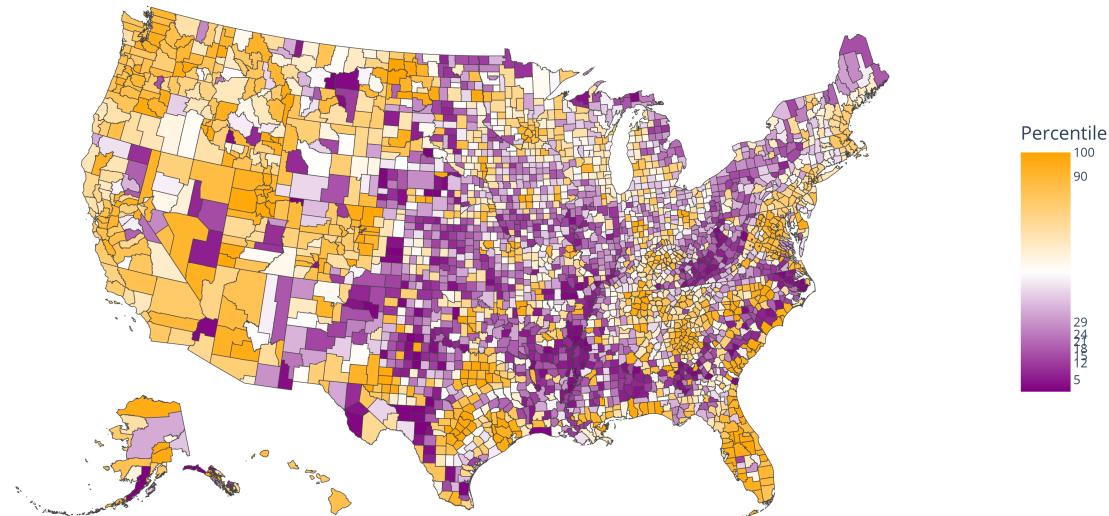
This percentile map will incorporate both a percentile-based scale *and* colorbar labels that show the values corresponding to those percentiles.

First, I'll use some simulated data to demonstrate that `colorbar_tickvals` determines where items are placed on the colorbar, whereas `colorbar_ticktext` shows what their values will be. This example will hopefully make our actual map updates clearer.

Modifying `colorbar_tickvals` to designate where colorbar labels will be placed:

```
sample_colorbar_tickvals = [5, 12, 15, 18, 21, 24, 29, 90, 100]
fig.update_coloraxes(colorbar_tickvals = sample_colorbar_tickvals)
filename = 'sample_tickval_change'
update_and_save_plotly_map()
    fig, static_file_folder = static_file_folder, filename = filename,
    save_html = False)
wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)
```

Population % Change by County from 2011 to 2021

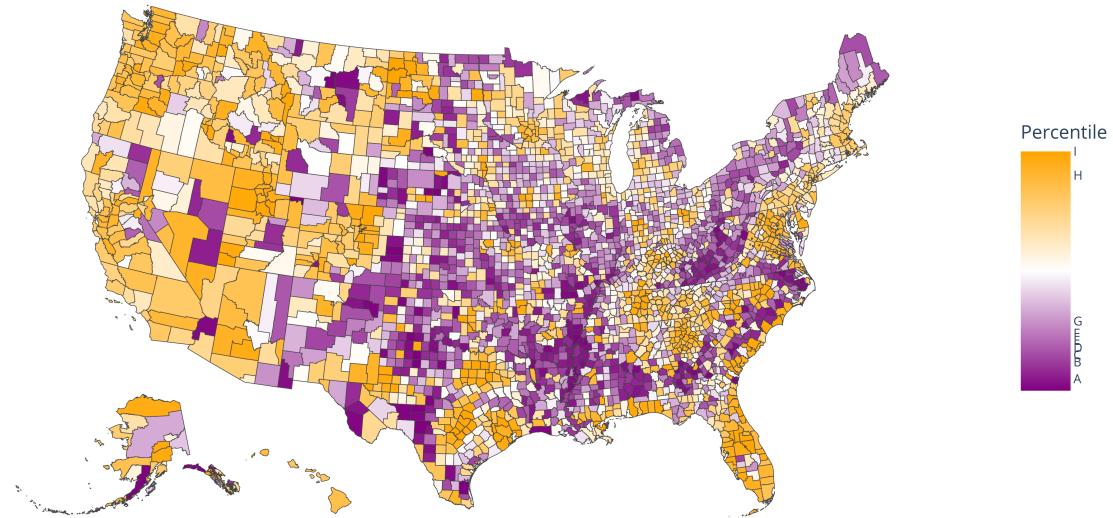


There are two important things to note here: (1) the colorbar labels are now located at the percentiles specified by `sample_colorbar_tickvals`, and (2) the labels themselves have been updated to match these percentiles.

Modifying `colorbar_ticktext` to replace these labels with letters of the alphabet: (this is simply to demonstrate that we can change the text entries for colorbar labels without also modifying their position.)

```
sample_colorbar_ticktext = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
fig.update_coloraxes(colorbar_ticktext = sample_colorbar_ticktext)
filename = 'sample_ticktext_change'
update_and_save_plotly_map()
    fig, static_file_folder = static_file_folder, filename = filename,
    save_html = False)
wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)
```

Population % Change by County from 2011 to 2021



The positions of the labels haven't changed, but their values have.

Now that we've gotten this example out of the way, let's *finally* update the map to incorporate our real-life replacements for the `colorbar_tickvals` and `colorbar_tickmode` arguments.

```
# The following code updates the colorbar to show the values
# corresponding to each percentile rather than the percentiles
# themselves. It does so by (1)
# selecting the percentile quantiles calculated earlier
# as the colorbar tick locations; (2) selecting these quantiles'
# corresponding percentile scores as the colorbar values;
# and (3) changing the title of the colorbar from 'Percentile' to reflect
# the data being displayed within the tick text entries.

fig.update_coloraxes(
    colorbar_tickvals = gdf_county_percentile_quantile_list,
    colorbar_tickmode = 'array',
    colorbar_ticktext = gdf_county_score_list,
    colorbar_title = "% Change",
    colorbar_title_side = 'bottom')

# Note: I chose to place the colorbar title below the colorbar (via
# colorbar_title_side = 'bottom' because it ended up pretty close to
# the topmost tick when the default setting ('top') was used.)

# The documentation at https://plotly.com/python/reference/layout/coloraxis/
# proved indispensable when drafting this code.

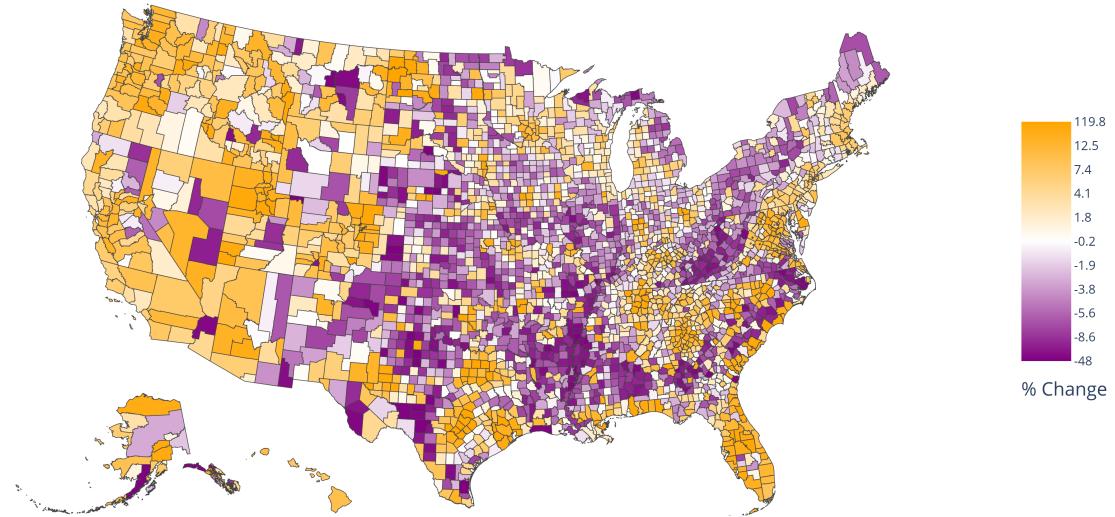
filename = f'county_pop_pct_growth_{year_range_start}\ \
-{year_range_end}'
update_and_save_plotly_map(
    fig, static_file_folder = static_file_folder,
    html_file_folder=html_file_folder, filename = filename)
```

(continues on next page)

(continued from previous page)

```
wadi(fig, static_file_folder+'/' +filename, generate_image = False,
      display_type = display_type)
```

Population % Change by County from 2011 to 2021



```
young_data_col = f'{year_range} Total_Pop_25_to_29 % Change'
young_extra_hover_data = [
    f'{young_data_col} Rank']
young_data_col

gdf_young_counties_for_map = gdf_counties_and_growth_stats.copy().dropna(
    subset = young_data_col).sort_values(
        young_data_col, ascending = False).set_index(
            'County/State')[['geometry', young_data_col] + young_extra_hover_data].copy()
gdf_young_counties_for_map
```

County/State	geometry \
Esmeralda County, Nevada	POLYGON ((-118.4278 37.89623, -118.35148 37.89...
Wheeler County, Oregon	POLYGON ((-120.49516 45.06828, -119.79074 45.06...
...	...
Edwards County, Texas	POLYGON ((-100.70039 30.28828, -99.75414 30.29...
Hamilton County, Kansas	POLYGON ((-102.04199 37.73854, -102.04451 38.2...
2011-2021 Total_Pop_25_to_29 % Change \	
County/State	
Esmeralda County, Nevada	460.000000
Wheeler County, Oregon	385.185185
...	...
Edwards County, Texas	-77.777778
Hamilton County, Kansas	-89.189189

(continues on next page)

(continued from previous page)

County/State	2011-2021 Total_Pop_25_to_29	% Change	Rank
Esmeralda County, Nevada		1.0	
Wheeler County, Oregon		2.0	
...		...	
Edwards County, Texas		3136.0	
Hamilton County, Kansas		3137.0	

[3137 rows x 3 columns]

```
test_list = [10, 9, 8]
test_list.sort(reverse=False)
test_list
```

[8, 9, 10]

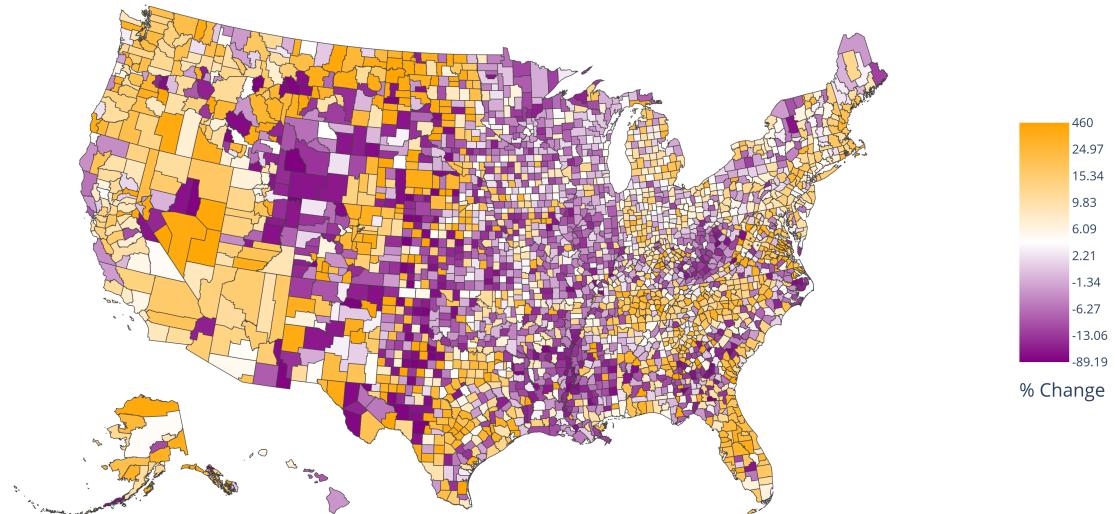
11.5 Mapping 25-to-29-year-old population growth at the county level

This code will call `gen_choropleth()`, a function stored within `plotly_choropleth_map_functions.py`, in order to produce *and* save a visualization of young adult population growth across US counties. This function is a more portable version of the mapping code presented earlier in this notebook.

```
filename = f'county_25-29_pop_pct_growth_{year_range_start}\'
- {year_range_end}'
fig = gen_choropleth(
    original_gdf=gdf_young_counties_for_map,
    geojson_col='geometry',
    data_col=young_data_col, extra_hover_cols=young_extra_hover_data,
    color_continuous_scale=custom_color_scale,
    scope='usa', title=f'25-29 Year-Old Population % Change by \
County from {year_range_start} to {year_range_end}',
    basemap_visible=False, colormap_type='percentile',
    colorscale_tick_count = 10,
    tick_round_value=2,
    static_file_folder=static_file_folder,
    html_file_folder=html_file_folder,
    filename = filename,
    custom_colorbar_title = '% Change',
    add_labels = False,
    label_round_value = 1)

wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)
```

25-29 Year-Old Population % Change by County from 2011 to 2021



11.6 Performing similar steps to map state-level growth data

```
gdf_states = geopandas.read_file(
    'shapefiles/cb_2023_us_state_500k/cb_2023_us_state_500k.shp')
gdf_states['geometry'] = gdf_states['geometry'].simplify(
    tolerance = 0.005)

gdf_states.head()
```

	STATEFP	STATENS	...	AWATER	\
0	35	00897535	...	726463919	
1	46	01785534	...	3387709166	
2	06	01779778	...	20291770234	
3	21	01779786	...	2384223544	
4	01	01779775	...	4582326383	

	geometry	
0	POLYGON	((-109.05004 31.3325, -109.04522 36.99...)
1	POLYGON	((-104.05788 44.9976, -104.03969 44.99...)
2	MULTIPOLYGON	(((-118.60442 33.47855, -118.5386...)
3	MULTIPOLYGON	(((-89.41728 36.49901, -89.37637 ...)
4	MULTIPOLYGON	(((-88.05338 30.50699, -88.03867 ...)

[5 rows x 10 columns]

```
df_acs_state_growth = pd.read_csv(
    f'../Census_Data_Imports/Datasets/\\
grad_destinations_acs_state_data.csv')
df_acs_state_growth.rename(
    columns = {'state':'state_code'},
```

(continues on next page)

(continued from previous page)

```
    inplace = True)
df_acs_state_growth.head()
gdf_states_and_growth_stats = gdf_states.merge(
    df_acs_state_growth, on = 'NAME', how = 'inner').rename(
    columns = {'NAME':'State'})
gdf_states_and_growth_stats.query("State != 'Puerto Rico'",  

                                    inplace = True)
gdf_states_and_growth_stats.head()
```

	STATEFP	STATENS	...	2016-2021 Total_Pop_25_to_29	% Change	Rank	\
0	35	00897535	...			36.0	
1	46	01785534	...			39.0	
2	06	01779778	...			30.0	
3	21	01779786	...			16.0	
4	01	01779775	...			25.0	

	2016-2021 Total_Pop_25_to_29	% Change	Percentile
0		31.372549	
1		25.490196	
2		43.137255	
3		70.588235	
4		52.941176	

[5 rows x 83 columns]

```
gdf_states_for_map = gdf_states_and_growth_stats.dropna(  

    subset = data_col).sort_values(  

        data_col, ascending = False).set_index('State')[  

['geometry', data_col] + extra_hover_data].copy()  

gdf_states_for_map.head()
```

	geometry	\
State		
Utah	POLYGON ((-114.0506 37.0004, -114.04148 41.993...)	
Idaho	POLYGON ((-117.24268 44.39655, -117.22698 44.4...)	
Texas	MULTIPOLYGON (((-94.71618 29.72778, -94.71651 ...	
North Dakota	POLYGON ((-104.04166 47.86228, -104.04874 48.9...)	
Colorado	POLYGON ((-109.06006 38.27549, -109.05151 39.1...))	

	2011-2021 Total_Pop	% Change	2011-2021 Total_Pop	% Change	Rank
State					
Utah	19.002541				1.0
Idaho	16.879496				2.0
Texas	16.502636				3.0
North Dakota	15.981361				4.0
Colorado	15.245785				5.0

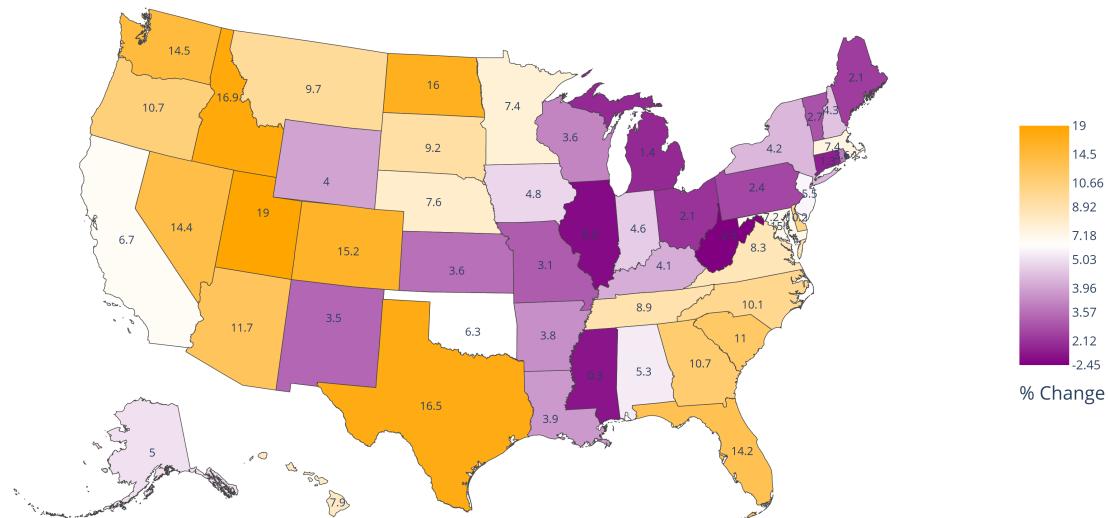
11.7 Rendering our state-level maps

We'll first create a map of total population growth; next, we'll map population growth for the 25-29 age range.

```
filename = f'state_pop_pct_growth_{year_range_start}\
-{year_range_end}'
fig = gen_choropleth(
    original_gdf=gdf_states_for_map, geojson_col='geometry',
    data_col=data_col, extra_hover_cols=extra_hover_data,
    color_continuous_scale=custom_color_scale,
    scope='usa', title=f'Population % Change by \
State from {year_range_start} to {year_range_end}',
    basemap_visible=False, colormap_type='percentile',
    colorscale_tick_count = 10,
    tick_round_value=2,
    static_file_folder=static_file_folder,
    html_file_folder=html_file_folder,
    filename = filename,
    custom_colorbar_title = '% Change',
    add_labels = True,
    label_round_value = 1,
    revise_state_label_points = True)

wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)
```

Population % Change by State from 2011 to 2021



```

geometry \
State
North Dakota      POLYGON ((-104.04166 47.86228, -104.04874 48.9...
Colorado          POLYGON ((-109.06006 38.27549, -109.05151 39.1...
Washington        MULTIPOLYGON (((-122.33164 48.02056, -122.3223...
New Hampshire     MULTIPOLYGON (((-70.61399 42.97335, -70.61457 ...
District of Columbia POLYGON ((-77.11976 38.93434, -77.041 38.99511...

2011-2021 Total_Pop_25_to_29 % Change \
State
North Dakota      26.479818
Colorado          21.190640
Washington        20.269558
New Hampshire     17.841358
District of Columbia 17.321342

2011-2021 Total_Pop_25_to_29 % Change Rank
State
North Dakota      1.0
Colorado          2.0
Washington        3.0
New Hampshire     4.0
District of Columbia 5.0

```

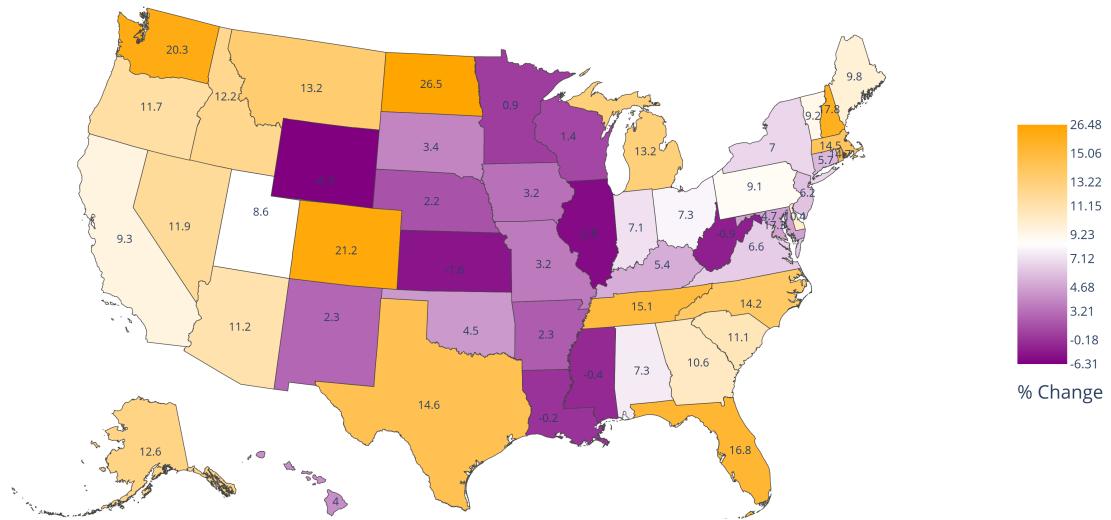
```

filename = f'state_25-29_pop_pct_growth_{year_range_start}\ \
-{year_range_end}'
fig = gen_choropleth(
    original_gdf=gdf_young_states_for_map, geojson_col='geometry',
    data_col=young_data_col, extra_hover_cols=young_extra_hover_data,
    color_continuous_scale=custom_color_scale,
    scope='usa', title=f'25-29 Year-Old Population % Change by \
State from {year_range_start} to {year_range_end}',
    basemap_visible=False, colormap_type='percentile',
    colorscale_tick_count = 10,
    tick_round_value=2,
    static_file_folder=static_file_folder,
    html_file_folder=html_file_folder,
    filename = filename,
    custom_colorbar_title = '% Change',
    add_labels = True,
    label_round_value = 1,
    revise_state_label_points = True)

wadi(fig, static_file_folder+'/'+filename, generate_image = False,
    display_type = display_type)

```

25-29 Year-Old Population % Change by State from 2011 to 2021



11.8 Creating tiled choropleth maps

The choropleth maps created thus far do not show any background images that can help users identify the regions specified within the maps. (These images are often referred to as ‘tiles.’) In order to include such tiles, we’ll need to use a separate Plotly function: `px.choropleth_map()`. (More information on this function can be found at https://plotly.com/python-api-reference/generated/plotly.express.choropleth_map.html).

Tiled maps have a few significant advantages:

1. They can make areas of the map easier to identify. For instance, if you want to figure out which cities are contained within a particular county in one of the 'tileless' maps created earlier, you'd need to open up a separate map in order to find out. With tiled maps, you can simply zoom in on the county in question and review the city names that appear within it.
 2. I have found that zooming and scrolling within HTML versions of tiled maps is a smoother process than it is for the non-tiled HTML-based maps shown above. I *think* this is because the non-tiled maps made use of the Albers USA projection, which might require more processing power than the tiled maps we'll create shortly.

The main advantage of non-tiled maps, at least for the US, is that they support the Albers USA projection; this projection makes it easy to include Alaska and Hawaii within static versions of the map. They also offer a simpler, cleaner display that works great for certain maps (such as state-based ones) for which tiles don't add that much explanatory value.

In general—at least for US-based maps—I would recommend using a non-tiled setup for static maps and a tiled one for HTML-based ones.

11.8.1 Creating a simple tiled map

Before we create a tiled *choropleth* map, let's first see what a standard Plotly tiled map looks like.

The following cell calls `px.choropleth_map()` directly to create a tiled map of the lower 48 US states. Within the `choropleth_map()` call, we'll specify our tile provider* as well with the starting zoom and center of our map. After creating the map, we'll also remove all margins from our map so that it takes up more of the screen. (I've found that this can be helpful for tiled maps.)

*There are other tile providers that you can use as well. However, you'll want to look into the terms of service for each provider in order to determine whether you need to pay, register, etc. in order to use the tiles for personal and/or commercial use.

Relevant resources for OpenStreetMap tile and data usage include:

- <https://operations.osmfoundation.org/policies/tiles/>
- <https://opendatacommons.org/licenses/odbl/>
- https://osmfoundation.org/wiki/Licence/Licence_and_Legal_FAQ

```
fig_simple_cm = px.choropleth_map(map_style='open-street-map', zoom=4.35,
    center={'lat':37.9, 'lon':-96}).update_layout(margin = {
        "r":0,"t":0,"l":0,"b":0})

# This update_layout() all was taken from
# https://plotly.com/python/map-configuration/ .

# When saving the file as an image, I'll choose a larger-than-usual
# height setting so that the tiles will appear more clearly.
# In later examples, I'll actually render tiled choropleth maps in
# UHD (3840 x 2160) format for even more clarity.

wadi(fig_simple_cm, file_path = 'simple_choropleth_map',
    generate_image = True, html_path_prefix = 'maps/',
    static_path_prefix = 'map_screenshots/', display_type = display_type,
    debug = False, height = 1080, scale = 1)
```



This map differs from the ones created earlier in that it shows map data in the form of tiles. (The previous maps showed state and county outlines from the Census, but there were no tiles underlying them—just a blank background.)

If you zoom in on an HTML copy of this map (available within `maps/simple_choropleth_maps.html`), you'll find that the map data becomes even more detailed: if you go far enough, you'll begin to see street names and the outlines of individual buildings.

However, while this map is neat, we can get essentially the same experience by visiting <https://www.openstreetmap.org/#map=5/37.9/-96>. Therefore, let's now make our Python based maps more useful—by overlaying choropleth data onto the tiles.

11.8.2 Creating tiled choropleth maps via a for loop

The following code will create four new tiled versions of our county-level population change maps. We can still use `gen_choropleth()` to create these maps; we'll just need to set its `show_tiles` argument to `True`. (This will instruct it to render the maps using `px.choropleth_map()` rather than `px.choropleth()`).

In order to reduce the amount of code required for this section, we'll generate these maps within a `for` loop. This loop will first create total population growth maps; it will then build growth maps for the 25-29-year-old population.

For each of these map types, the code will first create a map optimized for HTML display, followed by a map designed to be viewed as a screenshot. The HTML- and PNG-optimized maps will be saved only in HTML and PNG format, respectively. Configuration settings for these two map types can be found within `options_dict_list`.

A few additional notes on this code:

1. I updated the `screenshot_width` and `screenshot_height` settings to 3840 and 2160, respectively, because I found that static tiled maps appear clearer when higher width and height settings are applied.
2. In order to compensate for these wider dimensions, the PNG-optimized maps use higher zoom and font size settings than the standard ones.
3. The HTML-based maps do not include a title. I have found that the presence of these titles can make these maps harder to view when their underlying windows are compressed.

4. PNG-based maps will include a top margin (in order to make room for the title), but HTML-based maps will have all of their margins set to 0. (`gen_choropleth()` will make each margin 0 if `None` is passed to the `margin_list` argument and `show_tiles` is set to `True`.

```

for pop_type in ['', '25-29_']: # These variables refer to all residents
    # and residents aged 25-29, respectively. I chose these values
    # so that they could be incorporated within our map filenames.
    if pop_type == '':
        original_gdf = gdf_counties_for_map.copy()
        pop_type_for_title = ''
        tiled_data_col = data_col
        extra_hover_cols = extra_hover_data
    if pop_type == '25-29_':
        original_gdf = gdf_young_counties_for_map.copy()
        pop_type_for_title = '25-29 Year-Old '
        tiled_data_col = young_data_col
        extra_hover_cols = young_extra_hover_data

    filename = f'county_{pop_type}pop_pct_growth_{year_range_start}\ \
- {year_range_end}_tiled'

    options_dict_list = [{['zoom':4.3, 'save_static':False, 'save_html':True,
    'margin_list':None, 'title':''},
    {'zoom':5.4, 'save_html':False, 'save_static':True,
    'margin_list':[0, 200, 0, 0], 'title':f'{pop_type_for_title}\ \
Population % Change by County from {year_range_start} \ \
to {year_range_end}'}]

    for options_dict in options_dict_list:

        fig = gen_choropleth(
            original_gdf=original_gdf,
            geojson_col='geometry',
            data_col=tiled_data_col, extra_hover_cols=extra_hover_cols,
            color_continuous_scale=custom_color_scale,
            title=options_dict['title'],
            colormap_type='percentile',
            colorscale_tick_count = 10,
            tick_round_value=2,
            static_file_folder=static_file_folder,
            html_file_folder=html_file_folder,
            filename = filename,
            custom_colorbar_title = '% Change',
            add_labels = False, label_round_value = 1, show_tiles = True,
            zoom = options_dict['zoom'], screenshot_width = 3840,
            screenshot_height = 2160,
            screenshot_scale = 1,
            margin_list = options_dict['margin_list'],
            save_static = options_dict['save_static'],
            save_html = options_dict['save_html'],
            screenshot_title_font_size = 70,
            screenshot_colorbar_title_font_size = 50,
            screenshot_colorbar_tickfont_size = 40
        )

    # Displaying the static image created within the second (and last)
    # entry in the for loop:

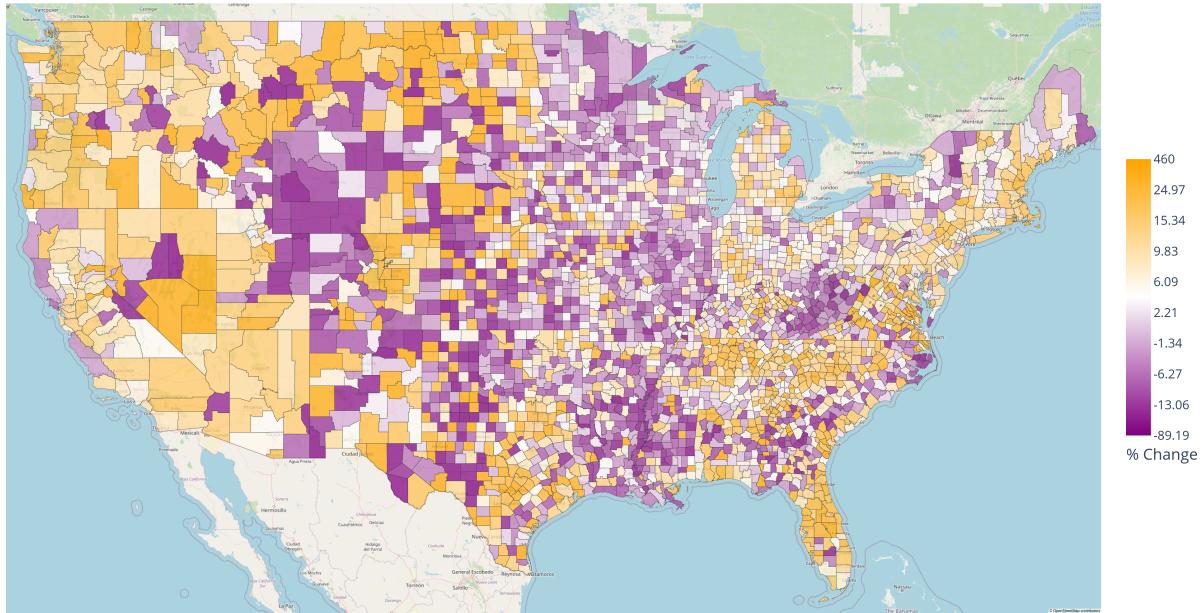
```

(continues on next page)

(continued from previous page)

```
wadi(fig, static_file_folder+'/'+filename, generate_image = False,  
display_type=display_type)
```

25-29 Year-Old Population % Change by County from 2011 to 2021



That's it for this notebook! We could have covered other map types as well, such as maps that show individual points, but the lessons that you've learned here—and the source code in `plotly_choropleth_map_functions.py`—should prove useful for all sorts of visualization projects.

```
end_time = time.time()  
run_time = end_time - start_time  
print(f"Finished running script in {round(run_time, 3)} seconds.")
```

Finished running script in 33.473 seconds.

11.9 plotly_choropleth_map_functions.py

```
import pandas as pd  
import geopandas  
import plotly.express as px  
import plotly.graph_objects as go  
import numpy as np  
  
def update_and_save_plotly_map(  
    fig, filename, static_file_folder = '', html_file_folder = '',  
    image_extension = 'png', save_html = True, save_static = True,  
    include_plotlyjs = 'cdn', screenshot_label_font_size = 18,  
    screenshot_width = 1920, screenshot_height = 1080,  
    screenshot_scale = 2, screenshot_title_y = 0.95,  
    screenshot_title_font_size = 40,
```

(continues on next page)

(continued from previous page)

```
screenshot_colorbar_thickness = 80, screenshot_colorbar_len = 0.5,
screenshot_colorbar_tickfont_size = 20,
screenshot_colorbar_title_font_size = 30):
```

'''This function assists with the process of saving a Plotly map as both an HTML file and a static image (if requested).

After saving an HTML version of the map, it will then increase the map's width and height so as to increase its relative size; next, it will enlarge certain text and colorbar values accordingly.

(These changes are performed in order to adjust for changes in the map's size when it gets converted to a screenshot.) Finally, the function will a static copy of the map.

fig: the map to be saved as an image.

static_file_folder and html_file_folder: The folder paths (either relative or absolute) in which to save static and interactive copies of the map, respectively. These paths should not include the desired filename, as that will get added in via the filename argument. In addition, they should not include any trailing forward or backward slashes.

*filename: The name to use when saving the file. This name should include the file name but *not* any image extensions; these will get added in manually.*

image_extension: the image extension (e.g. 'png', 'svg', etc) to use when saving a static copy of the map.

save_html and save_static: set to True to save HTML-based and static copies of the map, respectively.

include_plotlyjs: The argument to pass to the respective include_plotlyjs parameter within write_html(). The default setting allows for smaller HTML file sizes; however, if it's important for your maps to render offline, select True as your argument instead. For more details on these and other options, consult https://plotly.com/python-api-reference/generated/plotly.io.write_html.html.

screenshot_label_font_size: The font size in which data labels should appear within the screenshot.

screenshot_width and screenshot_height: The values to pass to the width and height arguments, respectively, of an update_layout() call.

*screenshot_scale: The scale to use when saving a static copy of the map. NOTE: the actual dimensions of the map will equal screenshot_width * scale and screenshot_height * scale. You can tweak these values as needed so that your screenshot has a sufficiently high resolution but remains readable.*

screenshot_title_y and screenshot_title_font_size: Arguments for tweaking the screenshot title's vertical location and font size, respectively.

(continues on next page)

(continued from previous page)

```
screenshot_colorbar_thickness, screenshot_colorbar_len,
screenshot_colorbar_tickfont_size, and
screenshot_colorbar_title_font_size: Arguments for modifying the
thickness, length, tick font size, and title font size,
respectively, of the screenshot's colorbar.
'''

if len(filename) == 0:
    raise ValueError(
        "Please specify a name to assign your map file(s).")

if save_html == True:
    # Adding a forward slash to html_file_folder in order to
    # distinguish it from the directory: (this step is unnecessary,
    # and will thus be skipped, if html_file_folder is
    # an empty string.)
    if len(html_file_folder) > 0:
        html_file_folder += '/'
    fig.write_html(html_file_folder + filename + '.html',
                   include_plotlyjs = include_plotlyjs)

if save_static == True:
    fig_for_chart = go.Figure(fig) # This method of creating a copy of
    # the original figure (suggested by StackOverflow user vestland
    # at https://stackoverflow.com/questions/58375026/how-to-make-a-
    # copy-of-a-plotly-figure-object/58375046#58375046)
    # ensures that the following changes won't have any effect on the
    # original figure.

    # Adjusting values within the HTML-based map in order to prepare
    # it for conversion to a static image:
    fig_for_chart.update_layout(
        width = screenshot_width, height = screenshot_height,
        title_font_size = screenshot_title_font_size,
        title_y = screenshot_title_y)
    fig_for_chart.update_coloraxes(
        colorbar_thickness = screenshot_colorbar_thickness,
        colorbar_len = screenshot_colorbar_len,
        colorbar_tickfont_size = screenshot_colorbar_tickfont_size,
        colorbar_title_font_size = screenshot_colorbar_title_font_size)
    fig_for_chart.update_traces(
        textfont_size=screenshot_label_font_size,
        selector=dict(type='scattergeo'))
    # The above line is based on the documentation found in
    # https://plotly.com/python/reference/scattergeo/ .

    if len(static_file_folder) > 0:
        static_file_folder += '/'

    fig_for_chart.write_image(
        file = static_file_folder + filename + '.' + image_extension,
        width = screenshot_width, height = screenshot_height,
        scale = screenshot_scale)

def gen_choropleth(
    original_gdf, geojson_col, data_col, extra_hover_cols = [],
    color_continuous_scale=None, scope=None, title=None,
```

(continues on next page)

(continued from previous page)

```
basemap_visible=False, colormap_type='linear',
colorscale_tick_count=10, tick_round_value=None,
custom_colorbar_title=None, add_labels=False,
label_round_value=None,
save_html=True, save_static=True, static_file_folder='',
html_file_folder='', filename='', image_extension='png',
include_plotlyjs='cdn', screenshot_label_font_size=18,
screenshot_width=1920, screenshot_height=1080,
screenshot_scale=2, screenshot_title_y=0.95,
screenshot_title_font_size=40, screenshot_colorbar_thickness=80,
screenshot_colorbar_len=0.5, screenshot_colorbar_tickfont_size=20,
screenshot_colorbar_title_font_size=30,
revise_state_label_points=False, debug=False, show_tiles=False,
tile_source='open-street-map', zoom=4.35,
starting_loc=[37.9, -96],
opacity=0.75, margin_list=None, colorbar_len=-1):
```

'''This function converts a GeoDataFrame into a choropleth map within Plotly, then saves that map (if requested by the caller) to HTML and image files. It also allows for percentile-based color ranges (which can better display data that contains outliers).

Note: this function assumes that the values to be passed to the choropleth map exist within the GeoDataFrame's index.

Make sure to consult the documentation for px.choropleth() available at (<https://plotly.com/python-api-reference/generated/plotly.express.choropleth.html>) as needed.

original_gdf: the GeoDataFrame from which regional boundaries and data will be retrieved. (The function will create a copy of this DataFrame so as not to modify it.)

geojson_col: the column within gdf that contains boundary data. This will be passed to the geojson argument of px.choropleth().

data_col: the column within gdf that contains data to be mapped.

*extra_hover_cols: A list of columns *in addition to* data_col that should be displayed when the user hovers over a given region.*

color_continuous_scale: A custom color scale to pass to the color_continuous_scale parameter of px.choropleth().

scope: the argument (e.g. 'usa') to pass to the 'scope' parameter of px.choropleth().

title: the title to use for the map.

basemap_visible: set to True to render the Plotly basemap and False to exclude it.

*colormap_type: set to 'percentile' in order to render map colors based on the *percentile ranks* of data_col values. Set to *linear**

(continues on next page)

(continued from previous page)

in order to base these colors directly on data_col values. Although both the region colors and colorbar color scale will reflect percentile ranks, the colorbar text entries will still show actual data_col values.

colorscale_tick_count: The number of ticks (and, in turn, text entries) to include within the map's colorbar. (Currently, this argument will only get applied if colormap_type is set to 'percentile'.)

tick_round_value: The value to pass to round() when rounding colorbar text entries. Set this value to 0 for integers, 1 for single decimal points, 2 for two decimal points, and so on. Set to None to prevent these entries from getting rounded. If colormap_type is not set to 'percentile', this argument will have no effect.

custom_colorbar_title: A custom title to use for the colorbar. If None is used as its argument, data_col will be used as the colorbar title by default. If colormap_type is not set to 'percentile', this argument will have no effect.

add_labels: set to True to add text labels to the map.

label_round_value: The value to pass to round() when rounding labels. Set this value to 0 for integers, 1 for single decimal points, 2 for two decimal points, and so on. Set to None to prevent these entries from getting rounded.

revise_state_label_points: set to True to shift the data label locations for Maryland; (so as not to overlap with DC's); Michigan (so that it appears within the 'hand' rather than the 'peninsula'); and Louisiana (to move it off of the Mississippi border). These operations will only work if you're creating labels for US states and have 'Maryland', 'Michigan', and 'Louisiana' as index entries.

save_html and subsequent arguments will get passed to their equivalent arguments within update_and_save_plotly_map(); see that function's documentation for further details on these entries.

debug: set to True to return both the figure and additional variables that can help with troubleshooting or extending the function; set to False to return just the figure.

show_tiles: Set to True to render the map using px.choropleth_map(), thus allowing tiles to appear behind the map. Keep False to render the map using px.choropleth() instead. Both options have their strengths and weaknesses for various use cases. For more on choropleth_map(), see https://plotly.com/python-api-reference/generated/plotly.express.choropleth_map.html.

tile_source: The map tile provider to use. (Will only have an effect if show_tiles is set to True.)

zoom: The starting zoom to use for a map. (Will only have an effect

(continues on next page)

(continued from previous page)

```

if show_tiles is set to True.)
```

starting_loc: A list of floats representing the starting latitude and longitude for the map. These should be represented in decimal degree format rather than degrees/minutes/seconds format. (Will only have an effect if show_tiles is set to True.)

(Note: the default zoom and starting_loc settings are tailored towards HTML-resolution maps of the lower 48 US states.)

opacity: The opacity level to use for choropleth regions. It will only have an effect if show_tiles is set to True.

margin_list: The right, top, left, and bottom margins to use for the map, respectively. If set to None, the map's default margins will be retained if show_tiles is False but set to 0 on all sides if show_tiles is True.

coloraxis_colorbar_len: The length to use for colorbars. If set to -1, this value will get changed within the function to 1 (if show_tiles is False) or 0.8 (if show_tiles is True). (A smaller colorbar can help prevent its title from getting cut off when the bottom margin is removed--which will occur by default when show_tiles is set to True.)

'''

```

# Creating a copy of the original DataFrame (so as not to modify
# it):
# This copy will also be sorted by the data column in descending
# order. This step helps ensure that, when a percentile-based
# colorbar is requested, the colorbar legend entries (which are
# retrieved from the sort_list variable created within this function)
# will appear within the same order as the percentile_quantile_list
# values. The latter are sorted explicitly within the function,
# but the former are not--so sorting the DataFrame at the outset
# helps keep them synchronized.

gdf = original_gdf.copy().sort_values(data_col, ascending = False)

if colormap_type == 'percentile':
    # In order to accommodate percentile-based color ranges,
    # the following code will (1) generate percentile ranks
    # for the values in data_col, then (2) create lists of selected
    # ranks along with their corresponding data_col values.
    # These rank and value lists will then be passed to the
    # colorbar_tickvals and colorbar_ticktext arguments
    # of an update_coloraxes() call.

    percentile_col = data_col+'_percentiles_for_map'
    if percentile_col in gdf.columns:
        raise ValueError(f"The name of the column that will be used \
to store data_col percentiles ({percentile_col}) is already present \
within gdf. Rename this column before calling the function to prevent \
any conflicts.")
```

(continues on next page)

(continued from previous page)

```
gdf[percentile_col] = 100 * gdf[
    data_col].rank(pct=True, ascending=True, method='max') # See
# https://pandas.pydata.org/pandas-docs/stable/reference/api/
# pandas.DataFrame.rank.html
# for more information on the use of df.rank() to create
# percentile ranks.

quantile_increment = 1/(colorscale_tick_count - 1)
# This variable will determine
# the distance between increments within np.arange(),
# which we'll call below in order to determine which
# quantiles to retrieve from the DataFrame.
# I subtracted 1 from the denominator so that the final number of
# quantiles (which will include both a minimum and maximum value)
# will match the 'quantile quantity' specified within
# tick_count.

# Using quantile_increment to specify which quantiles to retrieve:

# quantile_range will increase from 0 to 1 by the amount
# specified in quantile_increment. `quantile_increment/2`
# is added to 1 in the function call in order to ensure that
# the output will include, but also stop at, 1.

quantile_range = np.arange(0, 1+quantile_increment/2,
                          quantile_increment)

# The 'lower' interpretation method will help ensure that only
# actual percentile ranks present in the DataFrame get retrieved.
# This is a necessary prerequisite for using df.query() to
# locate the scores that match these percentile ranks,
# which we'll do shortly.

# Finding the actual percentile ranks within our DataFrame
# that correspond to these quantiles:

# (A quantile of 0 refers to the smallest percentile rank in the
# dataset, while a quantile of 1 refers to the highest percentile
# rank; a quantile of 0.5, if specified, refers to the median
# percentile rank.)

# Note: in an earlier version of this code, I created revised
# colorbars using the following steps:

# 1. I used quantile() to figure out which data column values to
# pass to the colorbar's 'ticktext' property.
# 2. I then multiplied quantile_range() by 100 to create a range
# of percentile ranks (called 'percentile_range') that could be
# passed to the colorbar's 'tickvals' property.

# However, this approach had a significant flaw: because datasets
# often wouldn't have an exact copy of one of the percentile ranks
# in percentile_range, the scores that replaced those
# percentile ranks often corresponded to a slightly different
# percentile rank. (In one actual example, Pandas calculated
```

(continues on next page)

(continued from previous page)

```

# the quantile of 0.5 (e.g. the 50.000th percentile) as 6.289%,
# but this percentage actually represented the 50.98th percentile.
# It would therefore be inaccurate to replace the 50th-percentile
# marker in the colorbar with this value.)

# In order to avoid this issue, I revised the code so that it
# would find the quantiles of the actual *percentile ranks* in
# our dataset. This way, all of the percentile ranks passed to
# the colorbar's tickvals property would have actual corresponding
# values that could be passed to colorbar_ticktext.

# This sort_values() call is necessary for the quantile
# ranges to line up with the score_list values that will
# get created shortly.
percentile_quantiles = gdf[percentile_col].quantile(
    quantile_range, interpolation = 'lower').sort_values(
    ascending = False)

percentile_quantile_list = percentile_quantiles.to_list()

# Determining the actual data_col values within the dataset
# that correspond to these percentile ranks:

# Note that only one row will be retained for each percentile rank;
# this will ensure that the lengths of the percentile rank and
# percentile score lists match. (If these lengths differed, we
# could encounter issues when trying to replace the former with
# the latter within our colorbar.)

percentile_rank_score_pairs = gdf.query(
    f"`{percentile_col}` in @percentile_quantile_list"
).drop_duplicates(percentile_col).copy()

score_series = percentile_rank_score_pairs[data_col]
if tick_round_value is not None:
    score_series = score_series.round(tick_round_value)
score_list = score_series.to_list()

color = percentile_col # Setting percentile_col as the color
# argument will allow for a wider diversity of colors in the
# event that outliers exist within the dataset.
else:
    color = data_col
if show_tiles == False: # px.choropleth() will be called
    # to produce a tileless choropleth map.
    if colorbar_len == -1:
        colorbar_len = 1
    fig = px.choropleth(gdf,
        geojson=gdf[geojson_col],
        locations=gdf.index,
        color=color,
        hover_data=[data_col] + extra_hover_cols,
        color_continuous_scale=color_continuous_scale,
        scope=scope,
        title=title,
        basemap_visible = basemap_visible)

```

(continues on next page)

(continued from previous page)

```
else: # px.choropleth_map() will be called
    # to allow a map with tiles to be produced.
    if colorbar_len == -1:
        colorbar_len = 0.8
    fig = px.choropleth_map(gdf,
                           geojson = gdf[geojson_col],
                           locations = gdf.index,
                           color = color,
                           hover_data = [data_col] + extra_hover_cols,
                           color_continuous_scale = color_continuous_scale,
                           title = title,
                           map_style=tile_source, zoom = zoom,
                           center = {'lat':starting_loc[0], 'lon':starting_loc[1]},
                           opacity=opacity)

if margin_list is None:
    if show_tiles == True:
        # Setting each margin to 0 will allow the map to take up more
        # of the window--which can be particularly useful for tiled maps.
        # (This code is based on a snippet from
        # https://plotly.com/python/map-configuration/)
        fig.update_layout(margin = {
            "r":0,"t":0,"l":0,"b":0})
        # No changes will be made in this case if show_tiles is set to
        # False.

    else: # Updating the margins (if requested by the user)
        fig.update_layout(margin = {
            "r":margin_list[0],"t":margin_list[1],
            "l":margin_list[2],"b":margin_list[3]})

    # This code will also shorten the colorbar a little so that
    # the reduced bottom margin does not cut off our title.
    # (This code was derived from
    # https://plotly.com/python/reference/layout/coloraxis/)

fig.update_layout(coloraxis_colorbar_len = colorbar_len)

if colormap_type == 'percentile':
    # The following code updates the figure's colorbar to show the values
    # corresponding to each percentile rather than the percentiles
    # themselves. It does so by (1)
    # selecting the percentile quantiles calculated earlier
    # as the colorbar tick locations; (2) selecting these quantiles'
    # corresponding percentile scores as the colorbar values;
    # and (3) changing the title of the colorbar to reflect
    # the data being displayed within the tick text entries.

    if custom_colorbar_title is not None:
        colorbar_title = custom_colorbar_title
    else:
```

(continues on next page)

(continued from previous page)

```

colorbar_title = data_col

# The documentation at
# https://plotly.com/python/reference/layout/coloraxis/
# proved indispensable in drafting this code.

fig.update_coloraxes(
    colorbar_tickvals = percentile_quantile_list,
    colorbar_tickmode = 'array',
    colorbar_ticktext = score_list,
    colorbar_title = colorbar_title,
    colorbar_title_side = 'bottom')

# I chose to set colorbar_title_side as 'bottom' because it ended up
# pretty close to the topmost tick when the default setting
# ('top') was used.)

if add_labels == True:
    label_col = data_col + '_for_labels'
    if label_col in gdf.columns:
        raise ValueError(f"The name of the column that will be used \
to store text labels ({label_col}) is already present \
within gdf. Rename this column before calling the function to prevent \
any conflicts.")
    # The function assumes that the caller wishes to plot data_col
    # values as text labels; however, it could be revised
    # to allow for an alternative set of labels.
    gdf[label_col] = gdf[data_col].copy()
    if label_round_value is not None:
        gdf[label_col] = gdf[label_col].round(label_round_value)

    # Determining points within each region that can serve as
    # text label locations:
    # (See https://geopandas.org/en/stable/docs/reference/
    # api/geopandas.GeoSeries.representative_point.html
    # for more information.)
    for column in ['label_loc', 'label_lat', 'label_lon']:
        if column in gdf.columns:
            raise ValueError(f"Rename the column {column} in order \
to prevent a conflict with gen_choropleth.")
    gdf['label_loc'] = gdf['geometry'].representative_point()
    # Adding the x and y coordinates stored within label_loc to
    # standalone fields for use within Plotly's
    # add_scattergeo() function:
    gdf['label_lat'] = [coord.y for coord in gdf['label_loc']]
    gdf['label_lon'] = [coord.x for coord in gdf['label_loc']]

    if revise_state_label_points == True:
        # Shifting data labels to make them easier to locate
        # and read:
        # (These points were determined using Openstreetmap
        # coordinates as a reference.)
        gdf.at['Maryland', 'label_lat'] = 39.4
        gdf.at['Maryland', 'label_lon'] = -77.24

        gdf.at['Michigan', 'label_lat'] = 43.63

```

(continues on next page)

(continued from previous page)

```
gdf.at['Michigan', 'label_lon'] = -84.97

gdf.at['Louisiana', 'label_lat'] = 30.5
gdf.at['Louisiana', 'label_lon'] = -92.54

# This code was based mostly on
# https://plotly.com/python/scatter-plots-on-
# maps/#simple-us-airports-map .
fig.add_scattergeo(
    text = gdf[label_col],
    mode = 'text',
    lat = gdf['label_lat'],
    lon = gdf['label_lon'])

# Disabling hover functionality for these text labels (as they
# can interfere with the pre-existing labels):
fig.update_traces(hoverinfo = 'skip', selector = dict(
    type='scattergeo'))
# This code was based on
# https://plotly.com/python/reference/scattergeo/

# Saving this map to HTML and static files
# (if requested by the caller):
if (save_html == True) or (save_static == True):
    update_and_save_plotly_map(
        fig = fig, save_html = save_html, save_static = save_static,
        static_file_folder = static_file_folder,
        html_file_folder = html_file_folder, filename = filename,
        image_extension = image_extension,
        include_plotlyjs = include_plotlyjs,
        screenshot_label_font_size = screenshot_label_font_size,
        screenshot_width = screenshot_width,
        screenshot_height = screenshot_height,
        screenshot_scale = screenshot_scale,
        screenshot_title_y = screenshot_title_y,
        screenshot_title_font_size = screenshot_title_font_size,
        screenshot_colorbar_thickness = screenshot_colorbar_thickness,
        screenshot_colorbar_len = screenshot_colorbar_len,
        screenshot_colorbar_tickfont_size = \
screenshot_colorbar_tickfont_size,
        screenshot_colorbar_title_font_size = \
screenshot_colorbar_title_font_size)

    if debug == True:
        return fig, percentile_quantiles, score_list, gdf
    else:
        return fig
```

CHAPTER
TWELVE

CREATING CHOROPLETH MAPS WITH FOLIUM

By Kenneth Burchfiel

Released under the MIT license

Note: If the Plotly library meets your mapping needs, feel free to skip this section and focus on choropleth_maps.ipynb instead. If you're curious about an alternative mapping tool, though, read on!

I had originally planned to base PFN's mapping code on Folium; however, I then realized that Plotly had some significant advantages for my own mapping work, so I went ahead and created a Plotly-based mapping notebook also. Both libraries are fantastic tools, and you may very well find that Folium is a better fit for your projects.

Also, you can find Google Sites-hosted copies of two interactive maps shown within this notebook via the following links:

[Link to Net Migration by County map](#)

[Link to Net Migration by State map](#)

This code shows how to use Python's Folium library to generate choropleth maps of county- and state-level net domestic migration rates (which I'll often refer to as 'domestic migration' or 'net migration' to save space). The first part of the code demonstrates how to use Folium's built-in `folium.choropleth()` function; the second part demonstrates how to create a more efficient version of this same choropleth map using a custom function.

A quick overview of domestic net migration

Domestic net migration data comprises movements from one part of a country to another. For instance, if someone moves from Fairfax County, VA to Harris County, TX, Fairfax County's net migration totals for the year would decrease by 1 whereas Harris County's would increase by 1. Note that international migration, births, and deaths do *not* factor into net migration totals. By focusing only on movements within the US, this data offers an intriguing look into which parts of a country are attracting pre-existing residents—and which are failing to attract (or retain) them.

Let's say that NVCU's seniors want to know which parts of the country are particularly popular areas to which to move. In order to help answer this question, you've decided to create county-level net domestic migration maps. (The seniors will need to interpret this analysis with caution: for instance, they might not necessarily want to move to a place whose high net domestic migration rates are driven by retirees. Then again, that might be the *perfect* destination for a college grad who loves playing bridge and pickleball!)

Our analyses within this notebook will use net migration *rates* (i.e. net migration totals divided by population totals) rather than nominal net migration counts because the former measure better reflects individuals' likelihood of moving to or from a given region. For instance, a county with 1 million residents and a positive net migration total of 2,000 would be a less 'hot' destination than a one with 100,000 residents and a net migration total of 1,000. The latter county's total net migration is 50% that of the former's, but its rate ($1000 / 100,000$, or 1%) is five times that of the larger county's ($2,000 / 1,000,000$, or 0.2%), thus indicating a greater interest in moving there relative to its size.

Determining whether to display HTML and PNG maps within the notebook's output: (These maps can also be accessed within the 'maps' and 'map_screenshots' folders.)

```
import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, wadi
display_type = config_notebook(display_max_columns = 5)
```

Importing the libraries we'll need for this script:

```
import time
program_start_time = time.time()
import pandas as pd

import folium
import geopandas
import numpy as np
from IPython.display import Image # Source: StackOverflow user 'zach' at
# https://stackoverflow.com/a/11855133/13097194

from branca.utilities import color_brewer
# color_brewer source code:
# https://github.com/python-visualization/branca/blob/main/branca/
# utilities.py

from selenium import webdriver # selenium will be used to generate
# screenshots of our HTML-based maps. I found that selenium didn't work
# correctly within JupyterLab Desktop right after the library was
# installed; however, when I restarted my kernel and then restarted
# JupyterLab Desktop, it ended up working fine. (Restarting JupyterLab
# Desktop alone probably would have resolved this issue.)

import time

import os

import branca.colormap as cm
# From https://python-visualization.github.io/folium/latest/
# advanced_guide/colormaps.html#StepColormap
from branca.colormap import StepColormap # From Folium's features.py
# source code:
# https://github.com/python-visualization/folium/blob/main/
# folium/features.py
```

12.1 Gathering our net migration data

The ‘co-est2023-alldata.csv’ file, located within the same folder as this code, contains net migration data for US states and counties during the 2020-2023 time period. I found this data on the Census site’s [County Population Totals and Components of Change: 2020-2023](#) page. The dataset is listed under a relatively long title:

Annual Resident Population Estimates, Estimated Components of Resident Population Change, and Rates of the Components of Resident Population Change for States and Counties: April 1, 2020 to July 1, 2023 (CO-EST2023-alldata)

Definitions of each column within this file can be found within CO-EST2023-ALLDATA.pdf, which I also downloaded from the aforementioned website to this section’s folder.

You may want to check the Census website to see whether a later version of the file exists. As long as that newer file

matches the format of ‘co-est2023-alldata.csv’, the rest of this code should still work correctly (provided that you update the following cell with the new filename).

```
df_nm = pd.read_csv(
    'co-est2023-alldata.csv',
    encoding = 'latin_1') # nm = 'net migration'.
# This dataset contains other population data as well, but we'll mostly
# use its net migration totals.
# I needed to add in 'encoding = latin_1' because
# the default encoding setting produced an error message.
df_nm.head()
```

	SUMLEV	REGION	...	RNETMIG2022	RNETMIG2023
0	40	3	...	6.486978	7.096186
1	50	3	...	8.896064	8.745044
2	50	3	...	29.985390	28.377843
3	50	3	...	9.708935	-1.298570
4	50	3	...	-13.620476	-3.101199

[5 rows x 67 columns]

12.1.1 Calculating total net migration across all years within the dataset

As specified within the ALLDATA.pdf file, ‘DOMESTICMIG’ columns show net domestic migration values. Note that there’s a column for each year, and that all but one of these years begins on July 1 and ends on June 30. (The one exception is DOMESTICMIG2020, which uses a date range of 2020-04-01 to 2020-06-30).

For our choropleth map, we’ll look into net migration across the entire date range in the dataset (e.g. 2020-04-01 to the latest date available). As of 2024-05-24, we had data up to June 30, 2023, but the Census will probably provide data for additional years in the future.

Therefore, rather than explicitly specifying the years available within our code, we’ll instead add in Python code that determines the first and last years with net migration data. This *should* make it easier to update the code with a later dataset. (I say ‘should’ because, if the Census Bureau decides to change the column names within this dataset, we’ll need to rewrite the code anyway to handle those new columns.)

```
# Determining all columns with nominal net migration data:
# (These columns all begin with DOMESTICMIG. There are also
# RDOMESTICMIG columns that show migration rates, but checking
# to see whether the *first* 11 characters contain 'DOMESTICMIG'
# will exclude them.
nm_cols = [column for column in df_nm.columns
           if column[0:11] == 'DOMESTICMIG']
nm_cols.sort() # This ensures that the first and last columns in our
# list will show the earliest and latest years with net migration data,
# respectively.
nm_cols
```

[‘DOMESTICMIG2020’, ‘DOMESTICMIG2021’, ‘DOMESTICMIG2022’, ‘DOMESTICMIG2023’]

Using our list of net migration columns to identify the earliest and latest years with net migration data:

```
first_nm_year = nm_cols[0][-4:] # -4: retrieves the final 4 characters
# within the column names (i.e. the year).
```

(continues on next page)

(continued from previous page)

```
last_nm_year = nm_cols[-1][-4:]
first_nm_year, last_nm_year
```

```
('2020', '2023')
```

We can now use our column list and starting/ending years to create a column that sums up all domestic net migration values for all years present in the dataset. In addition, we'll create a column that divides this sum by the population values in the first year of the dataset, thus allowing us to determine total net migration rates.

```
# Adding all of the net migration values within nm_cols together:
df_nm[f'{first_nm_year}-{last_nm_year} Total Domestic \
Net Migration'] = df_nm[nm_cols].sum(axis = 1)

# Creating our domestic migration rate column:
# Saving this column name to a variable will make it easier to incorporate
# into other lines of code that use it.
total_nm_rate_col = f'{first_nm_year}-{last_nm_year} Total Domestic Net \
Migration as % of {first_nm_year} Population'

df_nm[total_nm_rate_col] = (
    100 * df_nm[f'{first_nm_year}-{last_nm_year} \
Total Domestic Net Migration'] /
    df_nm[f'ESTIMATESBASE{first_nm_year}']) # Multiplying by 100 converts
    # the proportions into percentages
# As of 2024-05-24, the starting population selected by this code will be
# ESTIMATEBASE2020, which shows population estimates on 2020-04-01.
# There's also a POPESTIMATE2020 column, but its reference date is
# 2020-07-01. Since we're incorporating net migration data from 2020-04-01
# to 2020-07-01 into our analysis, the ESTIMATEBASE2020 column is the best
# set of population data to use.

df_nm.head()
```

SUMLEV	REGION	...	2020-2023 Total Domestic Net Migration	\
0	40	3	...	96538
1	50	3	...	1340
2	50	3	...	22425
3	50	3	...	-307
4	50	3	...	-219

2020-2023 Total Domestic Net Migration as % of 2020 Population	
0	1.921424
1	2.278563
2	9.675624
3	-1.216854
4	-0.982019

[5 rows x 69 columns]

This table contains both county-specific and statewide data. Rows with COUNTY values of 0 represent statewide totals; therefore, we can create county- and state-level versions of this dataset by selecting rows that have non-zero and zero COUNTY values, respectively.

(We could also have used SUMLEV column values (50 for counties and 40 for states) as a basis for this split.)

```
df_nm_county = df_nm.query("COUNTY != 0").copy()
df_nm_county.head()
```

SUMLEV	REGION	...	2020-2023 Total Domestic Net Migration \
1	50	3	...
2	50	3	...
3	50	3	...
4	50	3	...
5	50	3	...

	2020-2023 Total Domestic Net Migration as % of 2020 Population
1	2.278563
2	9.675624
3	-1.216854
4	-0.982019
5	1.875528

[5 rows x 69 columns]

```
df_nm_state = df_nm.query("COUNTY == 0").copy()
df_nm_state.head()
```

SUMLEV	REGION	...	2020-2023 Total Domestic Net Migration \
0	40	3	...
68	40	4	...
99	40	4	...
115	40	3	...
191	40	4	...

	2020-2023 Total Domestic Net Migration as % of 2020 Population
0	1.921424
68	-2.366596
99	3.049036
115	1.902381
191	-3.029854

[5 rows x 69 columns]

12.1.2 Adding percentile data to each dataset

In order to more easily compare net migration rates, we'll now create a column that calculates the percentile corresponding to each rate.

(Applying this code to each dataset separately prevents state values from influencing county percentiles and vice versa.)

```
percentile_col = f'{first_nm_year}-{last_nm_year} \
Domestic Net Migration Percentile'
percentile_col

for df in [df_nm_state, df_nm_county]: # Using a for loop allows us to
    # apply the same code to our state and county DataFrames.
    df[percentile_col] = 100 * df[total_nm_rate_col].rank(pct = True)
    # See https://pandas.pydata.org/pandas-docs/stable/reference/api/
```

(continues on next page)

(continued from previous page)

```
# pandas.DataFrame.rank.html  
  
df_nm_county.head()
```

```
SUMLEV REGION ... \  
1 50 3 ...  
2 50 3 ...  
3 50 3 ...  
4 50 3 ...  
5 50 3 ...  
  
2020–2023 Total Domestic Net Migration as % of 2020 Population \  
1 2.278563  
2 9.675624  
3 -1.216854  
4 -0.982019  
5 1.875528  
  
2020–2023 Domestic Net Migration Percentile  
1 68.225191  
2 96.660305  
3 22.073791  
4 24.809160  
5 64.408397  
  
[5 rows x 70 columns]
```

Let's now take a look at the states with the highest and lowest net migration rates in our dataset. (You'll need to scroll to the right to see the actual rates and percentiles.)

Highest rates:

```
df_nm_state.sort_values(  
    percentile_col, ascending = False).head()
```

```
SUMLEV REGION ... \  
565 40 4 ...  
2358 40 3 ...  
1626 40 4 ...  
325 40 3 ...  
331 40 3 ...  
  
2020–2023 Total Domestic Net Migration as % of 2020 Population \  
565 5.671907  
2358 4.846318  
1626 4.467998  
325 3.885869  
331 3.801438  
  
2020–2023 Domestic Net Migration Percentile  
565 100.000000  
2358 98.039216  
1626 96.078431  
325 94.117647  
331 92.156863
```

(continues on next page)

(continued from previous page)

```
[5 rows x 70 columns]
```

Lowest rates:

Note that, while the percentile for the highest state is 100, the lowest state's percentile is 1.96 (100 / 51). I believe this is because Pandas calculated each entry's percentile as the percentage of entries that are equal to or lower than all entries within the dataset. Therefore, since the lowest entry's percentile is at least equal to itself, its percentile won't be 0. This isn't the only way to define percentiles, but it works well enough for our purposes.

```
df_nm_state.sort_values(percentile_col, ascending = True).head()
```

	SUMLEV	REGION	...	\
1862	40	1	...	
329	40	3	...	
191	40	4	...	
559	40	4	...	
610	40	2	...	

	2020–2023 Total Domestic Net Migration as % of 2020 Population \			
1862	-4.369181			
329	-4.126181			
191	-3.029854			
559	-2.863378			
610	-2.844218			

	2020–2023 Domestic Net Migration Percentile			
1862	1.960784			
329	3.921569			
191	5.882353			
559	7.843137			
610	9.803922			

```
[5 rows x 70 columns]
```

We'll now perform the same analysis for counties. However, in order to prevent unusual migration patterns within smaller counties from skewing our results, we'll limit our results to counties that had at least 100,000 residents in 2020.

```
print(f"{len(df_nm_county.query('POPESTIMATE2020 >= 100000'))} counties out of {len(df_nm_county)} \
had over 100,000 residents in 2020.")
```

```
605 counties out of 3144 had over 100,000 residents in 2020.
```

```
df_nm_county.query("POPESTIMATE2020 >= 100000").sort_values(
    percentile_col, ascending = False).head()
```

	SUMLEV	REGION	...	\
2697	50	3	...	
391	50	3	...	
2767	50	3	...	
2614	50	3	...	

(continues on next page)

(continued from previous page)

```
1935      50      3 ...  
  
2020–2023 Total Domestic Net Migration as % of 2020 Population \  
2697              25.596840  
391               21.892944  
2767               20.253329  
2614               20.129665  
1935               19.072387  
  
2020–2023 Domestic Net Migration Percentile  
2697            100.000000  
391             99.968193  
2767             99.936387  
2614             99.904580  
1935             99.840967  
  
[5 rows x 70 columns]
```

```
df_nm_county.query("POPESTIMATE2020 >= 100000").sort_values(  
    percentile_col, ascending = True).head()
```

```
SUMLEV REGION ... \  
1865      50      1 ...  
1886      50      1 ...  
229       50      4 ...  
1903      50      1 ...  
1253      50      1 ...  
  
2020–2023 Total Domestic Net Migration as % of 2020 Population \  
1865                  -10.586540  
1886                  -8.905899  
229                   -8.816523  
1903                  -8.672521  
1253                  -7.983719  
  
2020–2023 Domestic Net Migration Percentile  
1865            0.445293  
1886            0.604326  
229             0.636132  
1903            0.763359  
1253            0.954198  
  
[5 rows x 70 columns]
```

12.2 Importing county and state boundaries

(See choropleth_maps.ipynb for more information on this step.)

```
# Reading in our shapefile data:
gdf_counties = geopandas.read_file(
    'shapefiles/cb_2023_us_county_500k/cb_2023_us_county_500k.shp')
# This abbreviation of 'GeoDataFrame' as 'gdf' derives from
# https://geopandas.org/en/stable/docs/reference/api/geopandas.
# GeoDataFrame.html

# The current versions of the county outlines would result in a very large
# output file (e.g. around 42 MB).
# We can reduce this file size (and the time required to generate and
# save the map) by simplifying the county outlines.
# I found a tolerance argument of 0.005 to work well for this project,
# but feel free to experiment with larger or smaller values.
gdf_counties['geometry'] = gdf_counties['geometry'].simplify(
    tolerance = 0.005)
# Source: https://geopandas.org/en/stable/docs/reference/api/geopandas.
# GeoSeries.simplify.html

# Adding a column that combines County and State names: (This will help
# prevent the code from confusing counties in different states with
# the same name.)
gdf_counties.insert(0, 'County/State', gdf_counties['NAMELSAD']
                    + ', ' + gdf_counties['STUSPS'])
gdf_counties.head()
```

	County/State	STATEFP	...	AWATER	\
0	Baldwin County, AL	01	...	1132887203	
1	Houston County, AL	01	...	4795415	
2	Barbour County, AL	01	...	50523213	
3	Sumter County, AL	01	...	24634880	
4	Miller County, AR	05	...	36848741	

	geometry
0	POLYGON ((-88.02858 30.22676, -87.96685 30.235...
1	POLYGON ((-85.71209 31.19727, -85.69231 31.210...
2	POLYGON ((-85.74803 31.61918, -85.73117 31.629...
3	POLYGON ((-88.42145 32.30868, -88.34043 32.991...
4	POLYGON ((-94.04343 33.55158, -94.00037 33.564...

[5 rows x 14 columns]

Ensuring that no states have two or more counties with the same name (which would likely cause all sorts of headaches):

```
gdf_counties['County/State'].duplicated(keep=False).sum()
```

```
np.int64(0)
```

Since there are far fewer states than there are counties, our state dataset won't require as much processing time as our county dataset. In addition, maps that show only data at the state level will generally take up less storage space. One disadvantage of these maps, though, is that the state-level data will hide interesting variations in net migration data at the county level.

```
gdf_states = geopandas.read_file(
    'shapefiles/cb_2023_us_state_500k/cb_2023_us_state_500k.shp')
gdf_states['geometry'] = gdf_states['geometry'].simplify(
    tolerance = 0.005)
gdf_states.head()
```

	STATEFP	STATENS	...	AWATER	\
0	35	00897535	...	726463919	
1	46	01785534	...	3387709166	
2	06	01779778	...	20291770234	
3	21	01779786	...	2384223544	
4	01	01779775	...	4582326383	

geometry

	POLYGON	((-109.05004 31.3325, -109.04522 36.99...)
0	POLYGON	((-109.05004 31.3325, -109.04522 36.99...))
1	POLYGON	((-104.05788 44.9976, -104.03969 44.99...))
2	MULTIPOLYGON	(((-118.60442 33.47855, -118.5386...)))
3	MULTIPOLYGON	(((-89.41728 36.49901, -89.37637 ...)))
4	MULTIPOLYGON	(((-88.05338 30.50699, -88.03867 ...)))

[5 rows x 10 columns]

12.2.1 Merging shape and demographic data tables together

Creating a table that stores both demographic and state data may make graphing tasks easier. Therefore, we'll now merge gdf_counties and df_nm_county together using their state and county ID columns as keys. In order to make this process easier, we'll rename gdf_counties' copies of these ID fields and change their values to integers in the following cell.

```
# Renaming the ID fields within gdf_counties so that they match the
# names of their corresponding fields within df_nm_county:
gdf_counties.rename(
    columns = {'STATEFP':'STATE',
               'COUNTYFP':'COUNTY'}, inplace = True)
# df_nm_county's ID fields are stored as integers, so in order to
# prevent the merge from failing to match integer- and string-formatted
# numbers we'll convert these to integers as well:
# (The fact that the IDs have leading 0s is a dead giveaway
# that they're currently
# stored as strings, though you could also check the output of
# gdf_counties.dtypes to confirm this.)
for column in ['STATE', 'COUNTY']:
    gdf_counties[column] = gdf_counties[column].astype('int')

gdf_counties.head()
```

	County/State	STATE	...	AWATER	\
0	Baldwin County, AL	1	...	1132887203	
1	Houston County, AL	1	...	4795415	
2	Barbour County, AL	1	...	50523213	
3	Sumter County, AL	1	...	24634880	
4	Miller County, AR	5	...	36848741	

(continues on next page)

(continued from previous page)

```

geometry
0 POLYGON ((-88.02858 30.22676, -87.96685 30.235...
1 POLYGON ((-85.71209 31.19727, -85.69231 31.210...
2 POLYGON ((-85.74803 31.61918, -85.73117 31.629...
3 POLYGON ((-88.42145 32.30868, -88.34043 32.991...
4 POLYGON ((-94.04343 33.55158, -94.00037 33.564...

[5 rows x 14 columns]
```

We're now ready to merge our shapefile and net migration data together:

Note: merging the DataFrame onto the GeoDataFrame ensures that the resulting file is also a GeoDataFrame—which will prove useful in our mapping code. (If we instead merged the GeoDataFrame onto our DataFrame, we'd end up with a DataFrame.)

```

gdf_counties_and_stats = gdf_counties.merge(
    df_nm_county,
    on = ['STATE', 'COUNTY'])
# Because we did not specify an argument for the 'how' parameter,
# the default 'inner' option will be used. This option will
# exclude any counties that either (1) aren't present
# in the net migration stats table or (2) don't have a shape defined.
# This will prevent our mapping code from (1) showing counties with
# missing net migration data and (2) attempting
# to map a county that lacks a corresponding outline.
gdf_counties_and_stats.head()
```

	County/State	STATE	...	\
0	Baldwin County, AL	1	...	
1	Houston County, AL	1	...	
2	Barbour County, AL	1	...	
3	Sumter County, AL	1	...	
4	Miller County, AR	5	...	

	2020-2023 Total Domestic Net Migration as % of 2020 Population \
0	9.675624
1	1.472920
2	-1.216854
3	-3.880428
4	0.523486

	2020-2023 Domestic Net Migration Percentile
0	96.660305
1	58.842239
2	22.073791
3	6.424936
4	46.024173

```
[5 rows x 82 columns]
```

Performing the same merge process for our state-level boundary and net migration DataFrames:

```

# I'd like to use 'State' as the field name for state names,
# so I'll rename this field within both gdf_states and df_nm_state.
# I'll also rename a pre-existing 'STATE' column within df_nm_state
```

(continues on next page)

(continued from previous page)

```
# so that it won't get confused with the new 'State' field.  
gdf_states.rename(columns = {'NAME':'State'}, inplace = True)  
df_nm_state.rename(columns = {'STNAME':'State',  
                           'STATE':'State_Code'}, inplace = True)
```

```
gdf_states_and_stats = gdf_states.merge(  
    df_nm_state, on = 'State')  
gdf_states_and_stats.head()
```

```
STATEFP STATENS ... \n  
0      35  00897535 ...  
1      46  01785534 ...  
2      06  01779778 ...  
3      21  01779786 ...  
4      01  01779775 ...  
  
2020-2023 Total Domestic Net Migration as % of 2020 Population \n  
0                      -0.287505  
1                      2.200485  
2                     -3.029854  
3                      0.501321  
4                      1.921424  
  
2020-2023 Domestic Net Migration Percentile  
0                      47.058824  
1                      80.392157  
2                      5.882353  
3                     58.823529  
4                     74.509804  
  
[5 rows x 79 columns]
```

```
df_nm_state.head()
```

```
SUMLEV REGION ... \n  
0      40     3 ...  
68     40     4 ...  
99     40     4 ...  
115    40     3 ...  
191    40     4 ...  
  
2020-2023 Total Domestic Net Migration as % of 2020 Population \n  
0                      1.921424  
68                     -2.366596  
99                     3.049036  
115                    1.902381  
191                     -3.029854  
  
2020-2023 Domestic Net Migration Percentile  
0                      74.509804  
68                     13.725490  
99                     88.235294  
115                    72.549020
```

(continues on next page)

(continued from previous page)

```
191          5.882353
[5 rows x 70 columns]
```

12.3 Creating a choropleth map with interactive tooltips

We can now create a choropleth map that shows net migration data by county. However, if we feed our entire gdf_counties_and_stats DataFrame into the mapping code, the resulting HTML file will be much larger than necessary, as it will include a number of columns that aren't actually necessary in creating our map. Therefore, we'll use a condensed version of gdf_counties_and_stats as a basis for this map instead.

```
# Storing columns that we'll need to retain within
# our condensed state- and county-level DataFrames as variables:
shape_col = 'geometry'
county_boundary_name_col = 'County/State'
state_boundary_name_col = 'State'
data_col = total_nm_rate_col

gdf_counties_and_stats_condensed = gdf_counties_and_stats[
[shape_col, county_boundary_name_col, data_col, percentile_col]].copy()

gdf_counties_and_stats_condensed.head()
```

	geometry	County/State	\
0	POLYGON ((-88.02858 30.22676, -87.96685 30.235...	Baldwin County, AL	
1	POLYGON ((-85.71209 31.19727, -85.69231 31.210...	Houston County, AL	
2	POLYGON ((-85.74803 31.61918, -85.73117 31.629...	Barbour County, AL	
3	POLYGON ((-88.42145 32.30868, -88.34043 32.991...	Sumter County, AL	
4	POLYGON ((-94.04343 33.55158, -94.00037 33.564...	Miller County, AR	

	2020–2023 Total Domestic Net Migration as % of 2020 Population \
0	9.675624
1	1.472920
2	-1.216854
3	-3.880428
4	0.523486

	2020–2023 Domestic Net Migration Percentile
0	96.660305
1	58.842239
2	22.073791
3	6.424936
4	46.024173

In order to create our choropleth map, we'll need to determine which item to use as the key for creating our data. For our county-level map, we'll want to use the 'County/State' column as the key, since it contains unique county names found in both the demographics and county outlines datasets.

However, if we attempt to pass 'County/State' to the key_on argument within our choropleth function, we'll get the following error:

```
key_on 'County/State' not found in GeoJSON.
```

That may strike you as strange: ‘County/State’ is definitely one of the columns in our GeoDataFrame. However, note that this code is referring to the *GeoJSON* version of this GeoDataFrame. The firstn 500 characters of this file appear as follows:

```
gdf_counties_and_stats_condensed.to_json()[0:500]
# to_json() is listed at https://geopandas.org/en/stable/docs/reference/api/geopandas.GeoDataFrame.html#geopandas.GeoDataFrame
```

```
'{"type": "FeatureCollection", "features": [{"id": "0", "type": "Feature",
  "properties": {"County/State": "Baldwin County, AL", "2020-2023 Total Domestic Net Migration as % of 2020 Population": 9.675623899761831, "2020-2023 Domestic Net Migration Percentile": 96.66030534351145}, "geometry": {"type": "Polygon", "coordinates": [[[[-88.02858, 30.226763], [-87.966847, 30.235618], [-87.936041, 30.261469], [-87.893201, 30.239237], [-87.78775, 30.254244], [-87.768003, 30.262455], [-87.747171, 30.287768],
```

The ‘County/State’ column in this output is preceded by ‘Feature’ and ‘properties.’ Meanwhile, the choropleth example found in the Folium documentation uses ‘feature.id’ as the key_on value. Here’s what the beginning of *that* GeoJson file looks like:

```
{"type": "FeatureCollection", "features": [{"type": "Feature", "id": "AL", "properties": {"name": "Alabama"}, "geometry": {"type": "Polygon", "coordinates": [[[[-87.359296, 35.00118],
```

Note that ‘id’ is the column with state data (as it’s a state-level choropleth map), and that ‘properties’ comes *after* this value (whereas, in our GeoJson file, ‘properties’ comes *before* our county column). Therefore, we can make an educated guess that ‘feature.properties.County/State’ is the right value to pass to the key_on column, and thankfully this guess is correct!

(Folium’s API reference also hints at this solution by listing ‘feature.properties.statename’ as an example of a key_on value.

In my experience, figuring out what value to use for the ‘key_on’ argument is one of the trickier parts of using Folium’s Choropleth feature. If you’re having trouble with this step, my advice would be to always look at the GeoJson version of your GeoDataFrame and see what values precede it, then incorporate those values (‘feature’, ‘properties’, etc.) into your entry.

12.3.1 Creating our bin ranges

By default, folium.Choropleth() will group regions into 6 colored bins; the ranges from these bins will stretch, in equal segments, from the lowest to the highest value. However, because some counties have very high or low net migration rates, this approach will cause our map to look relatively dull, since most counties will be in one of the middle bins.

Therefore, a better setup will be to base our bin ranges on percentiles, as this will result in a relatively equal number of counties appearing in each bin. (The bins will no longer be equally sized, but that’s not an issue in this case.) We can easily determine these bins’ cutoff points using pandas’ quantile() function:

```
bin_thresholds = list(gdf_counties_and_stats_condensed[
    total_nm_rate_col].quantile(
    np.linspace(0, 1, 11)))
# See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.quantile.html
# and https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
bin_thresholds
```

```
[-26.57727350868145,
 -2.8124548451811697,
```

(continues on next page)

(continued from previous page)

```
-1.3842847014410304,
-0.57942086570913,
0.10509734758955473,
0.803350933220706,
1.5648502030833924,
2.497913057064161,
3.8660584451404807,
5.966706571952004,
25.59683979436641]
```

We can now finally create our choropleth map! (Note: I commented out the output of this and other Folium maps so that this notebook would be small enough to display correctly on GitHub. However, if you're running this notebook locally on your computer, you can uncomment the 'm' at the end of the cell see what this map actually looks like.)

```
# The following code is based on:
# https://python-visualization.github.io/folium/latest/user_guide/
# geojson/choropleth.html
# Note: the 'RdYlGn' (red-yellow-green) fill color
# argument derives from:
# https://colorbrewer2.org/#type=diverging&scheme=RdYlGn&n=3 ,
# a helpful interactive site referenced within the Folium API reference
# (https://python-visualization.github.io/folium/latest/
# reference.html#module-folium.features).
# This color palette isn't listed as one of the available palettes
# within the choropleth documentation, but it still works fine.

m = folium.Map([38, -95], zoom_start=6)

folium.Choropleth(
    geo_data=gdf_counties_and_stats_condensed,
    data=gdf_counties_and_stats_condensed,
    bins = bin_thresholds,
    columns=[county_boundary_name_col, total_nm_rate_col],
    key_on=f'feature.properties.{county_boundary_name_col}',
    fill_color = 'RdYlGn',
    fillOpacity = 0.6
).add_to(m)

m.save('maps/basic_choropleth.html') # Useful for file size comparisons;
# an updated copy of this map that also contains tooltips will be shown
# shortly.
```

12.3.2 Adding in tooltips

We have a choropleth map in place, but it's missing something very important: tooltips that will display both the name of a county and its net migration data when the user hovers over it. These tooltips aren't present in the choropleth code by default, but Folium's documentation demonstrates how to easily add them in.

```
# In the following code, 'fields' lists the values that we wish
# to display within our tooltip, and 'aliases' lists the titles for
# those values.
```

(continues on next page)

(continued from previous page)

```
tooltip = folium.GeoJsonTooltip(
    fields=[county_boundary_name_col, total_nm_rate_col,
            percentile_col],
    aliases=[{"County": f"\u00a0{first_nm_year}-{last_nm_year} Net Migration \u00a0\u00a0as % of {first_nm_year} Population:", 'Percentile'],
    localize=True,
    sticky=False,
    labels=True,
    style="""
        background-color: #FFFFFF;
        border: 1px;
        border-radius: 1px;
        box-shadow: 1px;
    """,
    max_width=800
)
# Based on:
# https://python-visualization.github.io/folium/latest/user_guide/
# geojson/geojson_popup_and_tooltip.html

# We'll now add these tooltips to our map by linking them
# to an invisible GeoJson object. (The county outlines were
# already present within the map as a result of the choropleth
# mapping code, so there's no need to add them again here.)

g = folium.GeoJson(
    gdf_counties_and_stats_condensed,
    style_function=lambda x: {
        "fillOpacity": 0,
        "weight":0,
    },
    tooltip=tooltip
).add_to(m)
# Also based on:
# https://python-visualization.github.io/folium/latest/user_guide/
# geojson/geojson_popup_and_tooltip.html

# We'll now go ahead and save the map:

# m.save('maps/choropleth_with_tooltips.html') # Useful for file size
# # comparisons

m.save(
    f'maps/net_migration_rate_county_{first_nm_year}-{last_nm_year}.html')
```

12.3.3 Saving a screenshot of this map

The Plotly charting library makes saving static copies of charts simple: we can just call `write_image()`. Folium doesn't yet have this sort of functionality, but thankfully, it's still possible to create static copies of its map files. We'll just need to run some Selenium code that opens a web browser; navigates to our map; and then saves a screenshot of the map to a PNG file. The following code shows how to accomplish these steps.

(Note: I have found that Selenium may not work correctly when running a script *right after* installing it. Therefore, if you just installed Selenium and the following cell crashes, try exiting out of your Jupyter Notebook application; reopening it; and then rerunning this script.)

If your error message reads, in part, `NoSuchDriverException: Message: Unable to obtain driver for chrome`, these instructions still apply. Selenium *should* be able to obtain the driver on its own—but you may just need to close and reopen your notebook before it can do so.

```
options = webdriver.ChromeOptions()
# Source:
# https://www.selenium.dev/documentation/webdriver/browsers/chrome/
options.add_argument('--window-size=3000,1688') # I found that this window
# size, along with a starting zoom of 6 within our mapping code,
# created a relatively detailed map of the contiguous 48 US states.
# If you'd like to create an even more detailed map, consider setting
# your starting zoom to 7 and your window size to 6000,3375.
options.add_argument('--headless') # In my experience, this addition
# (which prevents the Selenium-driven browser from displaying on your
# computer) was necessary for allowing 4K screenshots to get saved
# as 3840x2160-pixel images. Without this line, the screenshots would
# get rendered with a resolution of 3814x1868 pixels.
# Source of the above two lines:
# https://www.selenium.dev/documentation/webdriver/browsers/chrome/
# and
# https://github.com/GoogleChrome/chrome-launcher/blob/main/docs/
# chrome-flags-for-tools.md
# I learned about the necessity of using headless mode *somewhere* on
# StackOverflow. Many answers to this question regarding generating
# screenshots reference it as an important step, for instance:
# https://stackoverflow.com/questions/41721734/take-screenshot-of-
# full-page-with-selenium-python-with-chromedriver/57338909

# Launching our Selenium driver:
driver = webdriver.Chrome(options=options)
# Source: https://www.selenium.dev/documentation/webdriver/
# browsers/chrome/

# Navigating to our map:
# Note: I needed to precede the local path with 'file://' as
# noted by GitHub user lukeis here:
# https://github.com/seleniumhq/selenium-google-code-issue-archive\
# /issues/3997#issuecomment-192014472
driver.get(
    'file://' + os.getcwd() + '/maps/' + f'net_migration_rate_county_\
{first_nm_year}-{last_nm_year}.html')
# Source: https://www.selenium.dev/documentation/
# Adding in os.getcwd() + '/' converts our relative path
# (which, by itself wouldn't be compatible with Selenium)
# into an absolute path. (Note that '/' still works on Windows, at least
# in my experience.)
```

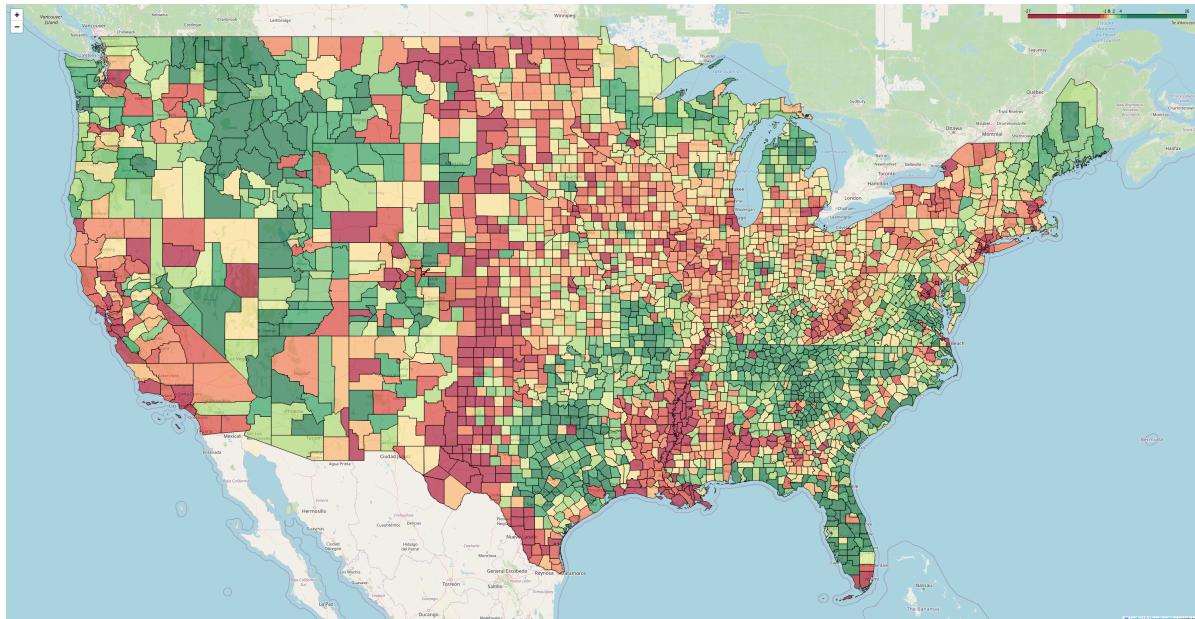
(continues on next page)

(continued from previous page)

```
time.sleep(3) # Helps ensure that the browser has enough time to download  
# map contents from the tile provider  
# Taking our screenshot and then saving it as a PNG image:  
  
driver.get_screenshot_as_file(  
f'map_shots/net_migration_rate_county_\\  
{first_nm_year}-{last_nm_year}.png')  
# Source:  
# https://selenium-python.readthedocs.io/api.html#selenium.webdriver.  
#remote.webdriver.WebDriver.get_screenshot_as_file  
  
# Exiting out of our webdriver:  
driver.quit()  
# Source: https://www.selenium.dev/documentation/
```

Here's a look at the map:

```
wadi(m, f'map_shots/net_migration_rate_county_\\  
{first_nm_year}-{last_nm_year}',  
      generate_image = False, display_type = display_type)
```



12.3.4 The drawback of the choropleth mapping approach shown above

We now have a choropleth map that shows additional information when the user hovers over a given county. That's great! However, although this means of creating our map is relatively simple (and a good introduction to the use of Folium), it does have one significant flaw: it results in unnecessarily large file sizes.

On my computer, the first version of the map (with no tooltips) was around 6.23 MB in size. Once tooltips were added in, this map nearly doubled in size to 12.4 MB. This increase occurred because we added many items to the underlying HTML file twice, including our county boundaries.

Now, 12 MB isn't that large of a file, but the larger your maps get, the harder (and potentially more expensive) it is to host them online, and the longer they'll take to load. (For instance, Google Sites currently limits single embedded HTML files

to 15 MB in size, so our 12-megabyte map is getting pretty close to that limit.) If you’re working with zip code-level data, this doubling in size could become a major issue and might even prevent the HTML version of your map from loading at all.

Therefore, I’ll now demonstrate a method that bypasses Folium’s Choropleth map class and instead allows the same set of borders to be used for tooltips and region colors. The map created with this method will be only 6.23 MB in size, yet it will still have the same interactive tooltips as the map we just created.

12.4 Creating a more efficient version of this map

In order to create a choropleth map without relying on Folium’s Choropleth class, we’ll need to develop a color map that we can use to assign different colors to different net migration values. I borrowed from the folium.Choropleth() source code in order to determine how best to accomplish this.

```
color_range = color_brewer('RdYlBu', n = 10)
# Based on folium.Choropleth() definition within
# https://github.com/python-visualization/folium/blob/main/folium/
# features.py

# Note: to reverse one of these color schemes, add '_r'
# to the end of the first argument (e.g. 'RdYlBu_r').
# Source: https://github.com/python-visualization/branca/blob/
# main/branca/utilities.py

# The following output shows the 10 colors (in hexadecimal format)
# that comprise this color range:
color_range
```

```
[ '#a50026',
  '#d73027',
  '#f46d43',
  '#fdbe2e',
  '#fee090',
  '#e0f3f8',
  '#abd9e9',
  '#74add1',
  '#4575b4',
  '#313695']
```

12.4.1 Using this color range to initialize our colormap

Note that passing our quantile-based bins to the index argument allows the colormap to reference those bins when determining which colors to assign to which values.

```
stepped_cm = StepColormap(
    colors = color_range,
    vmin = bin_thresholds[0], vmax = bin_thresholds[-1],
    index = bin_thresholds)
# Based on the self.color_scale initialization within
# Folium's Choropleth() source code (available at
# https://github.com/python-visualization/folium/blob/main/folium/
# features.py)
```

(continues on next page)

(continued from previous page)

```
stepped_cm
```

```
<branca.colormap.StepColormap at 0x7b260ce77af0>
```

We can apply this colormap to determine the colors corresponding to a given net migration value as follows:

```
stepped_cm(gdf_counties_and_stats_condensed.iloc[0][total_nm_rate_col])  
  
'#313695ff'
```

12.4.2 Creating our tooltips and choropleth map simultaneously via folium.GeoJson

```
# Much of this cell is based on the sample code found at  
# https://python-visualization.github.io/folium/latest/user_guide/  
# geojson/geojson_popup_and_tooltip.html#  
# and https://python-visualization.github.io/folium/latest/  
# user_guide/geojson/geojson.html .  
  
m = folium.Map([40, -95], zoom_start=5)  
  
tooltip = folium.GeoJsonTooltip(  
    fields=[county_boundary_name_col, total_nm_rate_col,  
            percentile_col],  
    aliases=["County:",  
            f"{{first_nm_year}}-{{last_nm_year}} Net Migration \\"  
as % of {{first_nm_year}} Population:", 'Percentile'],  
    localize=True,  
    sticky=False,  
    labels=True,  
    style=""",  
        background-color: #FFFFFF;  
        border: 1px;  
        border-radius: 1px;  
        box-shadow: 1px;  
    """,  
    max_width=800  
)  
  
# The following code will both assign colors from our StepColorMap  
# to each county *and* add in our tooltips. This approach allows both  
# our colors and tooltips to reference the same set of county outlines,  
# thus making for a more efficient map.  
  
g = folium.GeoJson(  
    gdf_counties_and_stats_condensed,  
    style_function=lambda x: {  
        "fillColor": stepped_cm(  
            x["properties"][total_nm_rate_col]),  
        "fillOpacity": 0.6,  
        "weight":1,  
        "color":"black"
```

(continues on next page)

(continued from previous page)

```

    },
    tooltip=tooltip
).add_to(m)
# The Folium.GeoJSON overview at
# https://python-visualization.github.io/folium/latest/user_guide/
# geojson/geojson.html
# contributed to this code as well.
# Note that we need to add ["properties"] in between x and
# [total_nm_rate_col], likely because gdf_counties_and_stats_condensed
# is being interpreted as a GeoJSON object. I based this off of the
# "if "e" in feature["properties"]["name"].lower()" line within
# the above link.

# Adding our StepColorMap to the output as well:
stepped_cm.add_to(m)

m.save(
f'maps/net_migration_rate_county_{first_nm_year}-{last_nm_year}.html')

# A view of this map will be shown later within this notebook.

```

12.4.3 Converting this second mapping approach into a function

In order to make this code easier to apply to other datasets, I converted it into a function (`cptt()`) by replacing hard-coded values with variables. (This is often how I create functions: I start with a working example that uses built-in values, then make that example more flexible by substituting variable names for those values.) **This function is located within `folium_choropleth_map_functions.py`; I placed it there so that it could be more easily called by different notebooks.**

I also added code to this function that allows a selected column's data to be displayed directly atop each boundary. This code works much better for state-level data (which we'll graph later) than county-level data, since the latter shapes are too small to accommodate these labels at a nationwide zoom level.

12.4.4 Calling this function separately for HTML and PNG maps

I've found that, if I try to use the same zoom level for both an HTML file and a screenshot, one of the maps will end up showing too much or too little zoom. Therefore, it can be useful to call `cptt()` twice—once to create a PNG version of a map, and again to create an HTML version. `create_map_and_screenshot()`, also available within `folium_choropleth_map_functions.py`, makes it easier to perform these two function calls.

Ideally, we could eliminate the need to call `cptt()` twice by updating the zoom level of the map within our code; this would allow us to save an HTML copy of the map for interactive viewing, then update its zoom to better accommodate a screenshot. Even better, we could try using Selenium to adjust the zoom, which would eliminate the need to create two separate HTML maps.

I tried out some code for both of these options, but neither worked successfully. Therefore, I'm sticking for now to the slower, yet reliable approach shown within `create_map_and_screenshot()`.

```

# Importing the two functions described above:
from folium_choropleth_map_functions import cptt, create_map_and_screenshot

```

12.4.5 Applying these functions to create HTML and PNG versions of our net migration by county map with optimized zoom levels

```
# Defining data_col_alias outside of the function call to make the latter
# a bit more readable:

data_col_alias = f'{first_nm_year}-{last_nm_year} Net Migration \
as % of {first_nm_year} Population:'

map_filename = f'net_migration_rate_county_{first_nm_year}-{last_nm_year}'

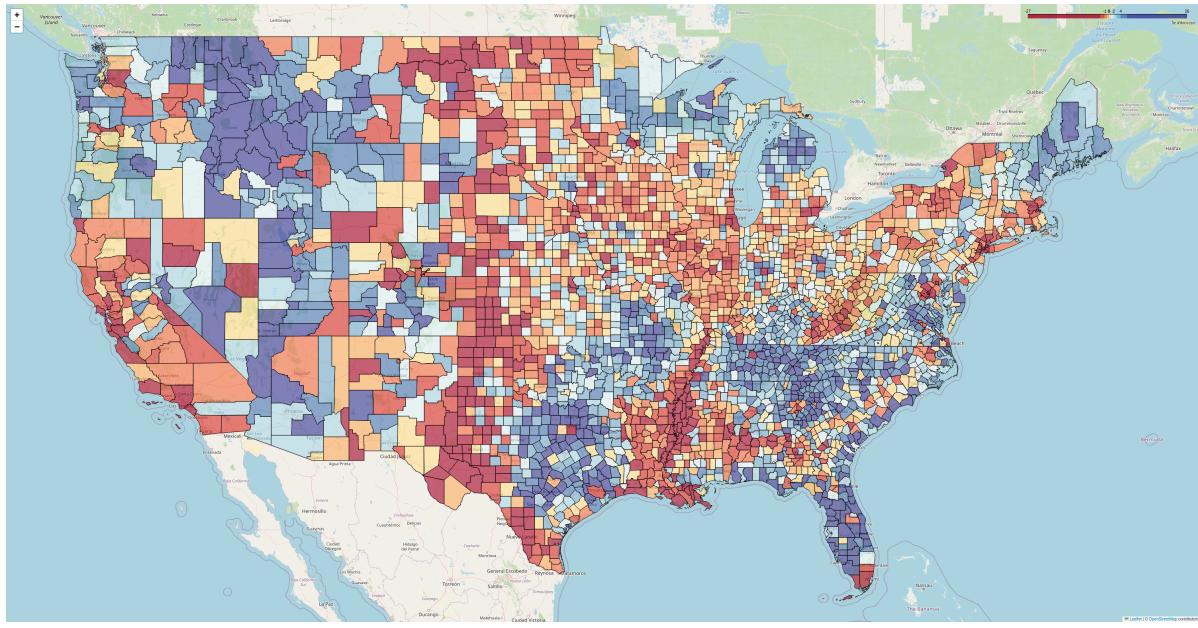
# Because cptt() automatically condenses the GeoDataFrame to include
# only those columns necessary for creating the map and tooltips,
# we can pass the original gdf_counties_and_stats GeoDataFrame as our
# gdf argument.

m = create_map_and_screenshot(
    starting_lat = 38, starting_lon = -95,
    html_zoom_start = 5,
    screenshot_zoom_start = 6,
    gdf = gdf_counties_and_stats,
    data_col = total_nm_rate_col,
    boundary_name_col = county_boundary_name_col,
    data_col_alias = data_col_alias, boundary_name_alias = 'County:',
    tooltip_variable_list = [percentile_col],
    tooltip_alias_list = ['Percentile:'],
    bin_type = 'percentile', bin_count = 10, color_scheme = 'RdYlBu',
    map_filename = map_filename,
    html_map_folder = os.getcwd() + '/maps',
    png_map_folder = os.getcwd() + '/map/screenshots')
# Note the use of os.getcwd() to convert the relative paths to the 'maps'
# and 'map/screenshots' folders into full paths.

wadi(m, f'map/screenshots/net_migration_rate_county_\
{first_nm_year}-{last_nm_year}', generate_image = False, display_type = display_type)
```

Generating screenshot.

Removed HTML copy of map.



12.4.6 Calling `create_map_and_screenshot()` again to create state-level net migration maps in HTML and PNG form

For demonstration purposes, I set the ‘tiles’ argument to `None` in order to create a clean and simple map without any underlying tile data. Since there won’t be anything to show under the choropleth boundaries, we can set `choropleth_opacity` to 1, resulting in a more vivid map.

I also chose to add net migration rates as boundary labels. In order to make the labels a bit easier to read, I switched the color scheme from `RdYlBu` to `RdYlGn`.

```
# We'll use the same data_col_alias that we created for our county-level
# map

map_filename = f'net_migration_rate_state_{first_nm_year}-{last_nm_year}'

m = create_map_and_screenshot(
    starting_lat = 38, starting_lon = -95,
    html_zoom_start = 5,
    screenshot_zoom_start= 6,
    gdf = gdf_states_and_stats,
    data_col = total_nm_rate_col,
    boundary_name_col = state_boundary_name_col,
    data_col_alias = data_col_alias, boundary_name_alias = 'State:',
    tooltip_variable_list = [percentile_col],
    tooltip_alias_list = ['Percentile:'],
    bin_type = 'percentile', bin_count = 10, color_scheme = 'RdYlGn',
    map_filename = map_filename, tiles = None,
    choropleth_opacity = 1,
    add_boundary_labels = True,
    boundary_label_col = total_nm_rate_col,
    round_boundary_labels = True,
    boundary_label_round_val = 2,
    html_map_folder = os.getcwd() + '/maps',
    png_map_folder = os.getcwd() + '/map_screenshots')
```

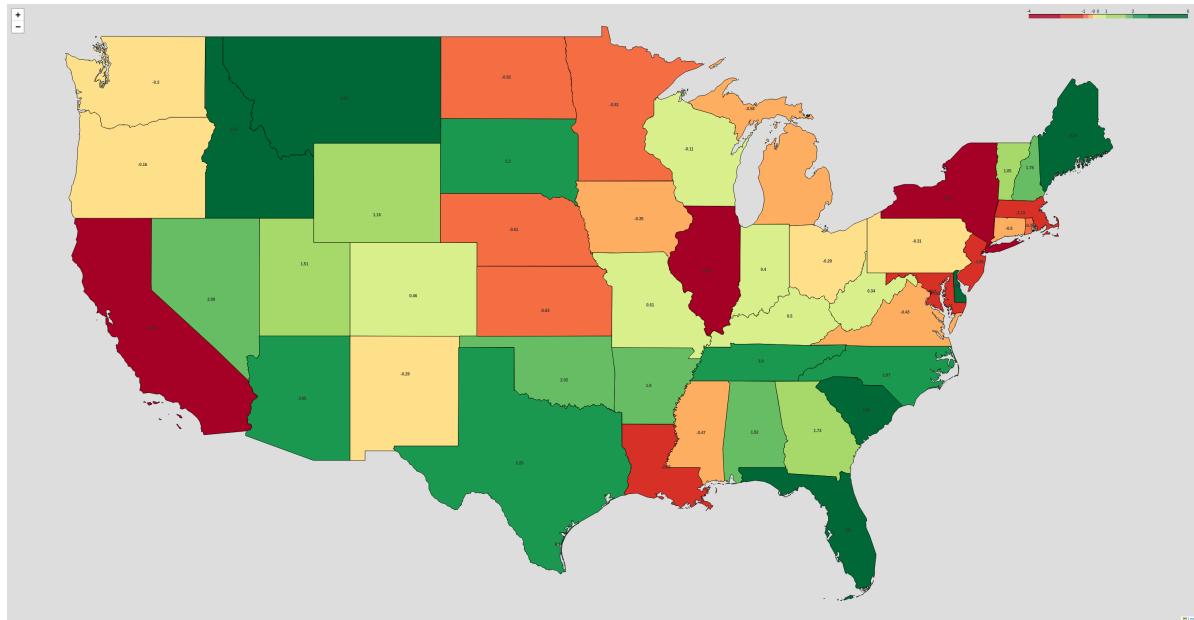
(continues on next page)

(continued from previous page)

```
wadi(m, f'map_screenshots/net_migration_rate_state_\\
{first_nm_year}-{last_nm_year}',
      generate_image = False, display_type = display_type)
```

Generating screenshot.

Removed HTML copy of map.



That's it for this notebook! I would have liked to keep this section simpler, but building choropleth maps can get a little complex, especially if you wish to bypass folium.Choropleth(). I hope you'll find both this notebook and the functions shown within folium_choropleth_map_functions.py useful for your own mapping projects.

```
program_end_time = time.time()
run_time = program_end_time - program_start_time
print(f"Finished running program in {round(run_time, 3)} seconds.")
```

Finished running program in 21.175 seconds.

12.5 folium_choropleth_map_functions.py

```
# Folium-based choropleth map functions
# By Kenneth Burchfiel
# Released under the MIT License

import folium
import geopandas
import numpy as np
```

(continues on next page)

(continued from previous page)

```

import time
import os
from branca.utilities import color_brewer
# color_brewer source code:
# https://github.com/python-visualization/branca/blob/main/branca/
# utilities.py

from selenium import webdriver

import branca.colormap as cm
# From https://python-visualization.github.io/folium/latest/
# advanced_guide/colormaps.html#StepColormap
from branca.colormap import StepColormap # From Folium's features.py
# source code:
# https://github.com/python-visualization/folium/blob/main/
# folium/features.py


def cptt(
    starting_lat, starting_lon, gdf,
    data_col, boundary_name_col,
    data_col_alias, boundary_name_alias,
    zoom_start = 6,
    bin_type = 'linear',
    bin_count = 6, custom_threshold_list = [],
    color_scheme = 'RdYlBu',
    tooltip_variable_list = [], tooltip_alias_list = [],
    save_html = True, save_screenshot = True,
    driver_window_width = 3000,
    driver_window_height = 1688,
    map_filename = 'map', html_map_folder = '',
    png_map_folder = '',
    geometry_col = 'geometry',
    tiles = 'OpenStreetMap', choropleth_opacity = 0.6,
    add_boundary_labels = False, boundary_label_lon_shift = 10,
    boundary_label_lat_shift = 10, boundary_label_col = '',
    round_boundary_labels = False,
    boundary_label_round_val = 0,
    delete_html_file = False):

    '''This function creates a choropleth map with a set of tooltips
    via folium.GeoJson(). (cptt stands for 'choropleth and tooltip.'
    Creating these two items together saves
    storage space versus building them individually.
    The function also adds in boundary labels upon request.

    Note: Much of this function is based on the sample code found at
    https://python-visualization.github.io/folium/latest/user_guide/
    geojson/geojson_popup_and_tooltip.html
    and https://python-visualization.github.io/folium/latest/user_guide/
    geojson/geojson.html .

    starting_lat, starting_lon, and zoom_start: the initial latitude,
    longitude, and zoom to pass to folium.Map(), respectively. For maps

```

(continues on next page)

(continued from previous page)

of the contiguous 48 US states, consider using a starting latitude of 38 and a starting longitude of -95.

gdf: A GeoDataFrame containing both boundary outlines and statistical data to visualize. Note that, in order to reduce the size of the resulting map, gdf will be condensed to include only those columns necessary for creating the map and adding in tooltips.

data_col: the column within gdf containing data to visualize.

boundary_name_col: the column within gdf that displays names of the boundaries being visualized.

data_col_alias and boundary_name_alias: The labels to use for data_col and county_boundary_name_col, respectively, within the map's tooltips.

tooltip_variable_list: A list of variables (other than boundary_name_col and data_col, which will get added in automatically) to display within the tooltip.

bin_type: Set to 'linear' (the default argument) to create equally spaced bins; 'percentile' to base choropleth colors on percentiles (resulting in roughly equal numbers of results per bin); or 'custom' to pass in a list of custom bins. (The custom option can be particularly useful when you wish to use the same set of bins for multiple maps.)

bin_count: The number of separate colors to show within the map. This parameter will be ignored when bin_type is set to 'custom.'

custom_threshold_list: A list of custom bin ranges to use for the map. Will only get applied when bin_type is set to 'custom.'

color_scheme: The color scheme to use within the map (e.g. 'RdYlBu'). Options can be found on <https://colorbrewer2.org/>. In order to reverse a scheme, add '_r' to the end (e.g. 'RdYlBu_r'). Source: <https://github.com/python-visualization/branca/blob/main/branca/utilities.py>

save_html: set to True to save this map as an HTML file.

save_screenshot: set to True in order to generate a screenshot of the map, then save it as a .PNG file. In order for this screenshot to get created, save_html must also be set to True.

driver_window_width and driver_window_height: the default width and height, respectively, of the Selenium driver window that will be called to generate a screenshot. The default settings work well for maps of the contiguous United States.

map_filename: The name to use when saving the map. The script will add the correct extension (e.g. 'html') to this map, so leave that portion out of the argument.

(continues on next page)

(continued from previous page)

`html_map_folder` and `png_map_folder`: The folders in which to store HTML and PNG versions of maps, respectively. Note that, if you're generating screenshots, `html_map_folder` needs to be an absolute path (so that it can get interpreted correctly by the Selenium-driven browser). For instance, if your `html_map_folder` is titled 'maps' and stored within your directory, pass `os.getcwd() + '/maps'` as your `html_map_folder` argument.

`geometry_col`: The column in `gdf` that stores shape boundary data.

`Tiles`: the tile provider to use for your map. Note that different tile services use different licenses for their tiles. If you don't want to use any tiles, enter `None` (no quotes) as your argument for this parameter.

`choropleth_opacity`: a float, ranging from 0 to 1, that determines how opaque to make the colors within the choropleth map. If `tiles` is set to `None`, consider setting `choropleth_opacity` to 1.

`add_boundary_labels`: Set to `True` to label each boundary within the map.

`boundary_label_lon_shift` and `boundary_label_lat_shift`: Integers that specify how far west and north to shift boundary labels so that they appear more centered. (These values will get passed to the `icon_anchor` parameter of the `DivIcons`.)

`boundary_label_col`: The column to use as a source for boundary labels. You may choose to set this to be the same as `data_col` or `boundary_name_col`, but you're not limited to those options.

`round_boundary_labels`: set to `True` to round boundary labels by `boundary_label_round_val` (see below).

`boundary_label_round_val`: An integer specifying the amount by which to round the boundary labels. Set to 0 to round to whole numbers, to 1 to round to tenths, and so forth.

`delete_html_file`: set to `True` in order to delete the HTML file saved by this function. (This can be useful when you are only calling this function only to create a screenshot of a map. Once the screenshot has been saved, the original HTML file no longer needs to be retained.) Note that, if `save_html` is set to `False`, the program will *not* attempt to delete the HTML file, as it's likely that one doesn't exist to begin with.

'''

You can uncomment the following two print statements to compare
the sizes of `gdf` and `gdf_condensed`.

```
# print(gdf.memory_usage(deep=True).sum() / 1000000)
# See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.
# memory_usage.html
```

(continues on next page)

(continued from previous page)

```
# Creating a condensed version of gdf that can serve as the basis for
# the map: (Condensing the DataFrame prevents unnecessary columns
# from getting included in the HTML output and thus increasing
# the map's file size.)

gdf_condensed = gdf.copy() [
    boundary_name_col, data_col, geometry_col] + tooltip_variable_list

# Removing any NaN data_col entries from our dataset so that they
# won't interfere with our mapping code:
# {data_col} is surrounded by ` characters to make this code
# compatible with column names that contain spaces.
gdf_condensed.query(f"`{data_col}`.isna() == False", inplace = True)

# print(gdf_condensed.memory_usage(deep=True).sum() / 10000000)

# Creating a blank map as the starting point for our choropleth:
m = folium.Map([starting_lat, starting_lon],
               zoom_start = zoom_start, tiles = tiles)

# Adding labels to each boundary:
# (Note: this code will work better for larger shapes, such as US
# states, and likely less well for smaller boundaries like counties
# or zip codes.)
# Adding these labels prior to creating our tooltips ensures that
# the former won't block the latter when the user is interacting with
# the map.

# Calculating reference points for adding labels to each boundary:
# The following code applies GeoPandas' representative_point() method
# to calculate central points for each boundary that always lie within
# that boundary. It then uses a list comprehension to create lists of
# x and y coordinates that can be fed into folium mapping code.
# (The POINT items returned by representative_point() don't appear
# to work as well with Folium.)

if add_boundary_labels == True:
    gdf_condensed['boundary_label_reference_points'] = [
        [coord.y, coord.x] for coord in
        gdf_condensed['geometry'].representative_point()]
    # Documentation for representative_point():
    # https://geopandas.org/en/stable/docs/reference/api/geopandas.
    # GeoSeries.representative_point.html#geopandas.GeoSeries.
    # representative_point
    # x and y are attributes of Geoseries objects:
    # https://geopandas.org/en/stable/docs/reference/geoseries.html

    # Using these reference points to add the values stored
    # in boundary_label_col as boundary labels:
    for i in range(len(gdf_condensed)):
        boundary_label = gdf_condensed.iloc[i][
            boundary_label_col].copy()
        if round_boundary_labels == True:
            boundary_label = boundary_label.round(
```

(continues on next page)

(continued from previous page)

```

        boundary_label_round_val)
folium.Marker(location = gdf_condensed.iloc[i][
    'boundary_label_reference_points'],
    icon = folium.features.DivIcon(
        f"<b>{boundary_label}</b>",
        icon_anchor = (boundary_label_lon_shift,
                       boundary_label_lat_shift)))
    ).add_to(m)
# Part of the folium.Marker() call above is based on an example
# by StackOverflow user 'r-beginners'
# (who pointed out that you could pass a DivIcon to the 'icon'
# parameter
# within a Marker in order to add text labels to shapes).
# Source: https://stackoverflow.com/a/72588910/13097194
# The use of icon_anchor to adjust the labels' locations
# comes from the DivIcon documentation at:
# https://python-visualization.github.io/folium/latest/
# reference.html#folium.features.DivIcon

# Creating the tooltips:

tooltip_field_list = [
    boundary_name_col, data_col] + tooltip_variable_list

alias_list = [boundary_name_alias,
              data_col_alias] + tooltip_alias_list

# print(tooltip_field_list, alias_list)

tooltip = folium.GeoJsonTooltip(
    fields= tooltip_field_list,
    aliases=alias_list,
    localize=True,
    sticky=False,
    labels=True,
    style="""
        background-color: #FFFFFF;
        border: 1px;
        border-radius: 1px;
        box-shadow: 1px;
    """,
    max_width=800
)

# Creating our set of colors to use for the choropleth map:
if bin_type == 'custom': # In this case, bin_count will be overwritten
    # by the length of custom_threshold_list minus 1. (The number of
    # bins will always be one less than the number of thresholds, as
    # two thresholds are needed to establish the boundaries for
    # one bin.
    # For instance, if you have three thresholds (0, 1, and 2),
    # two bins can be created from this list: 0-1 and 1-2.)
    bin_count = len(custom_threshold_list) - 1
color_range = color_brewer(color_scheme, n = bin_count)

```

(continues on next page)

(continued from previous page)

```
# Based on Choropleth() definition within
# https://github.com/python-visualization/folium/blob/main/folium/
# features.py

# To reverse the set of colors passed to color_scheme, add '_r'
# to the end of the string (e.g. 'RdYlBu_r'). Source:
# https://github.com/python-visualization/branca/blob/main/branca
# /utilities.py

# Determining which colors to apply to each result:

if bin_type == 'linear': # Equally-spaced bins will be used.
    # The number of bins will be derived from the number of
    # colors in color_range.
    stepped_cm = StepColormap(
        colors = color_range,
        vmin = gdf_condensed[data_col].min(),
        vmax = gdf_condensed[data_col].max())
    # Based on:
    # https://python-visualization.github.io/branca/colormap.html#
    # branca.colormap.StepColormap

else: # In this case, a different approach to creating the
    # StepColorMap
    # will be used that better accommodates non-equally-spaced bins.

    if bin_type == 'percentile': # In this case,
        # percentile-based bins will
        # be used.
        bin_thresholds = list(gdf_condensed[data_col].quantile(
            np.linspace(0, 1, bin_count+1)))
        # For np.linspace() documentation, see:
        # https://numpy.org/doc/stable/reference/generated/
        # numpy.linspace.html

    elif bin_type == 'custom': # This condition allows for a set of
        # custom bins to be passed in.
        bin_thresholds = custom_threshold_list.copy()

    else:
        raise ValueError("bin_type must be set to 'linear', \
'percentile', or 'custom.'")

    # The following approach works for both 'percentile' and
    # 'custom' bin_type conditions.
    stepped_cm = StepColormap(
        colors = color_range,
        vmin = bin_thresholds[0], vmax = bin_thresholds[-1],
        index = bin_thresholds)
    # Based on the self.color_scale initialization within Folium's
    # Choropleth() source code (available at
    # https://github.com/python-visualization/folium/blob/main/
    # folium/features.py )
```

(continues on next page)

(continued from previous page)

```

# The following code will both assign colors from StepColorMap
# to each region *and* add in tooltips. This approach allows the
# colors and tooltips to reference the same set of outlines,
# thus allowing for a smaller file size.

g = folium.GeoJson(
    gdf_condensed,
    style_function=lambda x: {
        "fillColor": stepped_cm(
            x["properties"][data_col]),
        "fillOpacity": choropleth_opacity,
        "weight":1,
        "color":"black"
    },
    tooltip=tooltip
).add_to(m)
# The Folium.GeoJSON overview at
# https://python-visualization.github.io/folium/latest/
# user_guide/geojson/geojson.html
# contributed to this code as well.
# Note that we need to add ["properties"] in between x and
# [data_col], likely because gdf_condensed
# is being interpreted as a GeoJSON object. I based this off of the
# "if "e" in feature["properties"]["name"].lower()" line within
# the above link.

# Adding the color legend for the choropleth to the map:
stepped_cm.add_to(m)
# Based on:
# https://python-visualization.github.io/folium/latest/user_guide/
# geojson/geojson.html

if save_html == True:
    m.save(f"{html_map_folder}/{map_filename}.html")

# Generating a screenshot of the map:
if save_screenshot == True:
    print("Generating screenshot.")
    options = webdriver.ChromeOptions()
    # Source: https://www.selenium.dev/documentation/webdriver/
    # browsers/chrome/
    options.add_argument(f'--window-size={driver_window_width},\
{driver_window_height}') # I found that this window
    # size, along with a starting zoom of 6 within our mapping code,
    # created a relatively detailed map of the contiguous 48 US
    # states.
    # If you'd like to create an even more detailed map,
    # consider setting your starting zoom to 7 and your window size
    # to 6000,3375.
    options.add_argument('--headless') # In my experience, this
    # addition (which prevents the Selenium-driven browser from
    # displaying on your computer) was necessary for allowing 4K
    # screenshots to get saved
    # as 3840x2160-pixel images. Without this line, the
    # screenshots would get rendered with a resolution of
    # 3814x1868 pixels.

```

(continues on next page)

(continued from previous page)

```
# Source of the above two lines:  
# https://www.selenium.dev/documentation/webdriver/  
# browsers/chrome/  
# and  
# https://github.com/GoogleChrome/chrome-launcher/blob/  
# main/docs/chrome-flags-for-tools.md  
# I learned about the necessity of using headless mode  
# *somewhere* on StackOverflow. Many answers to the question  
# linked below regarding generating screenshots reference it  
# as an important step, for instance.  
# https://stackoverflow.com/questions/41721734/take-screenshot  
# -of-full-page-with-selenium-python-with-chromedriver/57338909  
  
# Launching the Selenium driver:  
driver = webdriver.Chrome(options=options)  
# Source: https://www.selenium.dev/documentation/webdriver/  
# browsers/chrome/  
  
# Navigating to our map:  
# Note: In order to get the following code to work within  
# Linux, I needed to precede the local path with 'file://' as  
# noted by GitHub user lukeis here:  
# https://github.com/seleniumhq/selenium-google-code-issue-  
# archive/issues/3997#issuecomment-192014472  
driver.get(f"file:///{html_map_folder}/{map_filename}.html")  
# Source: https://www.selenium.dev/documentation/  
time.sleep(3) # Helps ensure that the browser has enough  
# time to download  
# map contents from the tile provider. This time might need to be  
# increased if a slow internet connection is in use. Conversely,  
# if no tiles are being incorporated into the map,  
# there may not be any need to call  
# time.sleep().  
# Taking our screenshot and then saving it as a PNG image:  
driver.get_screenshot_as_file(  
    f"{png_map_folder}/{map_filename}.png")  
# Source:  
# https://selenium-python.readthedocs.io/api.html#selenium.  
# webdriver.remote.webdriver.WebDriver.get_screenshot_as_file  
  
# Exiting out of the webdriver:  
driver.quit()  
# Source: https://www.selenium.dev/documentation/  
  
if (delete_html_file == True) & (save_html == True):  
    os.remove(f"{html_map_folder}/{map_filename}.html")  
    print("Removed HTML copy of map.")  
  
return m  
  
def create_map_and_screenshot(  
    starting_lat, starting_lon, gdf,  
    data_col, boundary_name_col,  
    data_col_alias, boundary_name_alias,
```

(continues on next page)

(continued from previous page)

```

html_zoom_start = 5,
screenshot_zoom_start = 6,
bin_type = 'linear', bin_count = 6,
custom_threshold_list = [],
color_scheme = 'RdYlBu',
tooltip_variable_list = [], tooltip_alias_list = [],
map_filename = 'map', html_map_folder = '',
png_map_folder = '',
geometry_col = 'geometry',
tiles = 'OpenStreetMap', choropleth_opacity = 0.6,
add_boundary_labels = False, boundary_label_lon_shift = 10,
boundary_label_lat_shift = 10, boundary_label_col = '',
round_boundary_labels = False,
boundary_label_round_val = 0):
    '''This function calls cptt() twice in order to create separate PNG
    and HTML versions of a map. This approach allows separate zoom levels
    to be passed to each map, which can prevent one or both maps from
    displaying a non-ideal zoom level.

```

html_zoom_start and screenshot_zoom_start: the zoom settings to use for the HTML and PNG maps, respectively.

For information on other variables, consult the documentation within cptt().'''

```

# Creating an HTML map optimized for generating a screenshot;
# creating the screenshot; and then deleting the HTML copy of the map
# (as we only needed it in order to create the screenshot):
cptt(
    starting_lat = starting_lat,
    starting_lon = starting_lon, gdf = gdf,
    data_col = data_col, boundary_name_col = boundary_name_col,
    data_col_alias = data_col_alias,
    boundary_name_alias = boundary_name_alias,
    zoom_start = screenshot_zoom_start,
    bin_type = bin_type,
    bin_count = bin_count,
    custom_threshold_list = custom_threshold_list,
    color_scheme = color_scheme,
    tooltip_variable_list = tooltip_variable_list,
    tooltip_alias_list = tooltip_alias_list,
    save_html = True, save_screenshot = True,
    map_filename = map_filename, html_map_folder = html_map_folder,
    png_map_folder = png_map_folder,
    geometry_col = geometry_col,
    tiles = tiles, choropleth_opacity = choropleth_opacity,
    add_boundary_labels = add_boundary_labels,
    boundary_label_lon_shift = boundary_label_lon_shift,
    boundary_label_lat_shift = boundary_label_lat_shift,
    boundary_label_col = boundary_label_col,
    round_boundary_labels = round_boundary_labels,
    boundary_label_round_val = boundary_label_round_val,
    delete_html_file = True)

```

Creating a copy of the map optimized for interactive viewing:

(continues on next page)

(continued from previous page)

```
# (This HTML file will get retained, whereas that created in order to
# produce the screenshot in the earlier cptt() call got deleted.)
# Note that this code was called *after* the screenshot generation
# code so that the latter's HTML map doesn't overwrite this one.
m = cptt(
    starting_lat = starting_lat,
    starting_lon = starting_lon, gdf = gdf,
    data_col = data_col, boundary_name_col = boundary_name_col,
    data_col_alias = data_col_alias,
    boundary_name_alias = boundary_name_alias,
    zoom_start = html_zoom_start,
    bin_type = bin_type,
    bin_count = bin_count,
    custom_threshold_list = custom_threshold_list,
    color_scheme = color_scheme,
    tooltip_variable_list = tooltip_variable_list,
    tooltip_alias_list = tooltip_alias_list,
    save_html = True, save_screenshot = False,
    map_filename = map_filename, html_map_folder = html_map_folder,
    png_map_folder = png_map_folder,
    geometry_col = geometry_col,
    tiles = tiles, choropleth_opacity = choropleth_opacity,
    add_boundary_labels = add_boundary_labels,
    boundary_label_lon_shift = boundary_label_lon_shift,
    boundary_label_lat_shift = boundary_label_lat_shift,
    boundary_label_col = boundary_label_col,
    round_boundary_labels = round_boundary_labels,
    boundary_label_round_val = boundary_label_round_val,
    delete_html_file = False)

# Returning the map:
return m
```

Part V

Regressions

CHAPTER
THIRTEEN

REGRESSION ANALYSES

Descriptive statistics, which Python for Nonprofits has focused on so far, are immensely valuable. Averages, means, medians, and totals are well-known, easy-to-interpret measures that provide crucial information about how organizations are performing.

However, depending on your work, you may need to venture into the world of *inferential* statistics from time to time. In this world, you aren't simply reporting on differences in mean values between two or more groups: you're also making an *inference* about the nature of the distributions of those groups' values. (For more on statistical inference, consult https://en.wikipedia.org/wiki/Statistical_inference).

An example should make this clearer. Suppose NVCU's bookstore management is looking into differences in average fall semester sales among colleges. The bookstore's leaders notice that the highest-ranking college by fall sales was STB, with an average of 80.28 in spending per student; meanwhile, the lowest-ranking college was STL, whose students averaged 79.69 in bookstore spending.

It's accurate to report, from a descriptive statistics perspective, that "STL students spent less at the bookstore, on average, than did STB students last fall." However, does this mean that NVCU should make a concerted marketing effort to raise sales at STL? Not necessarily—STL's average spending was only 0.73% less than was STB's. But what if the difference was 2%? Or 5%? At what point would we want to address apparent flagging bookstore sales at one of our colleges?

This is where inferential statistics—such as the regression models I'll focus on in this section—can prove very helpful. From an inferential stats standpoint, we can look beyond mere differences in means and instead ask: "What is the likelihood that, if STB and STL students actually have the same spending patterns, we would have seen a difference of this magnitude in average sales?" If the likelihood is small enough—say, less than 5%—we might choose to conclude that there is a significant difference between STB and STL sales. From a business perspective, we might then decide to take some concrete action to address this difference, such as advertising the bookstore more heavily within STL's dorms. (Note: for more information on what p values do (and do not) mean, see <https://ethanweed.github.io/pythonbook/04.04-hypothesis-testing.html#the-p-value-of-a-test> .

13.1 An introduction to regressions

Regression models allow us to examine how one or more independent variables relates to a particular dependent variable. (See https://en.wikipedia.org/wiki/Linear_regression for more details.) They also allow us to 'control' for various factors in order to better assess whether another variable had a significant relationship with a given outcome. Their flexibility, intuitiveness, and practicality make them an excellent tool to have in your statistics arsenal.

In this section, we'll first use regression models to determine how college, gender, level, and fall bookstore spending can help predict spring spending. Next, we'll create additional models that use American Community Survey data to evaluate the relationship between different education levels—and the percentage of regions that have a bachelor's degree—on median incomes.

There will be plenty of visualizations along the way, as plotting data can help prevent us from misinterpreting it—or from applying regression models incorrectly. (For a classic example of the importance of visualization, see

https://en.wikipedia.org/wiki/Anscombe's_quartet. For an even better example, see https://en.wikipedia.org/wiki/Datasaurus_dozen

13.2 Important caveats

This notebook is meant to serve as a *simplified* introduction to regression analyses in Python. Its aim is merely to show you how to use the Statsmodels library to run regressions and interpret key parts of the output; it's certainly *not* meant to be the definitive resource on regression analyses or inferential statistics. If you're new to the concept of regressions, or need a refresher on the theory underlying them, you'll also want to review a more comprehensive guide.

13.2.1 1. Regression assumptions

This notebook does *not* provide detailed coverage of the assumptions that regression models make. Thus, before you begin using regressions for real-world analyses, you would do well to review those assumptions (as your model may not be valid if these assumptions do not hold). In addition, you should not assume that the models shown within this chapter satisfy all of these assumptions.

For more information about these assumptions, I recommend reading the 'Model checking' section within *Learning Statistics with Python*'s 'Linear Regression' chapter, available at <https://ethanweed.github.io/pythonbook/05.04-regression.html#model-checking>. The rest of this chapter is a valuable resource in its own right. (Note: this chapter does contain some mild language.) The 'Regression diagnostics' section of Introduction to Regression Models (<https://kirenz.github.io/regression/docs/diagnostics.html>) is another helpful resource.

Statsmodels offers plenty of tools for testing regression assumptions. You can learn more about these tools at https://www.statsmodels.org/dev/examples/notebooks/generated/regression_diagnostics.html and <https://www.statsmodels.org/stable/diagnostic.html>.

13.2.2 2. Criticisms of the 'frequentist' model

In this section, we'll consider regression results to be statistically significant if they have a p-value below 0.05. However, you should take care not to over-rely on p-values when making decisions. (For more information about the limitations of p-values, see https://en.wikipedia.org/wiki/Misuse_of_p-values and <https://theconversation.com/the-problem-with-p-values-how-significant-are-they-really-20029>).

You may ultimately decide to forego frequentist statistics altogether for a given project and try a Bayesian approach instead; a free overview of Bayesian statistics in Python is available at <https://allendowney.github.io/ThinkBayes2/>.

13.2.3 Unweighted data

This section does not demonstrate how to perform regressions on data that incorporate sampling weights (which is a common feature of survey data). Unfortunately, coverage for such regressions is very limited in Python. The Samplics library (<https://samplics-org.github.io/samplics/>) may offer support for weighted regressions in the near future, but as of January 2025, its regression code wasn't yet ready for use. (In the meantime, you may still find its chi-squared and t-tests to be useful tools.)

You can, however, do all sorts of weighted regression analyses in R. (For an excellent overview of such analyses, check out Exploring Complex Survey Data Analysis Using R-available at <https://tidy-survey-r.github.io/tidy-survey-book/index.html>). If you'd like to incorporate R code for weighted survey analyses—or any other domain where R has more comprehensive tools—into your Python scripts, make sure to check out the Rpy2 library, which allows you to run R code directly within Python and to pass variables from Python to R (and back). For more details on applying Rpy2 within Jupyter notebooks, see <https://rpy2.github.io/doc/latest/html/interactive.html>.

I don't list these caveats in order to scare you: I merely want to characterize this chapter (like all of the other chapters of Python for Nonprofits, to be honest) as a *starting point* for further exploration, rather than a comprehensive guide.

Importing various libraries:

```
import pandas as pd
import math
import numpy as np
from sqlalchemy import create_engine
e = create_engine('sqlite:///../Appendix/nvcu_db.db')
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.stats.api as sms
import plotly.express as px
from statsmodels.stats.weightstats import ttest_ind
import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, wadi
display_type = config_notebook(display_max_columns = 7,
                                display_max_rows = 8)
from IPython.display import Image
regression_display_width = 400
```

13.3 Part 1: Analyzing bookstore sales

The management of NVCU's bookstore is analyzing fall and spring sales from the previous year. They were excited to learn that per-student spending increased by around 7 dollars from the fall semester to the spring semester; however, they would also like you to help them analyze these changes in spending by various demographic criteria (specifically, gender, college, and level).

A few notable changes took place during the spring semester that the bookstore would like you to investigate further. First, due to pressure from a nutrition-focused professor, unhealthy snacks were removed from the checkout aisles. These snacks were especially popular among freshmen and sophomores, so the bookstore would like to determine whether their removal may have affected sales for younger students.

Second, an intensive marketing campaign was carried out at STM and STL; if it ended up being successful, the bookstore will implement it at the other colleges also. (The bookstore's decision to launch this campaign at just two randomly-selected colleges not only saved money; it also gave its leaders a chance to assess, by comparing sales at the targeted schools with sales at the other schools, whether it had a positive impact. This is an example of an A/B test; for more details on this strategy, see https://en.wikipedia.org/wiki/A/B_testing.)

We'll begin our analyses here with some tried-and-true descriptive statistics and visualizations; these will show us how changes in sales differed for various groups of interest. Next, we'll implement a linear regression model so that we can make *inferences* about the meaning of these changes. By setting gender, college, level, and fall spending as our independent variables and spring spending as our dependent variable, we'll be able to assess whether these factors had a significant effect on students' spring purchasing behavior.

To prepare for these analyses, we'll first import our sales data from the NVCU database:

```
df_sales = pd.read_sql('select * from bookstore_sales', con = e)
df_sales
```

	student_id	gender	college	level	Fall	Spring	Fall_Spring_Change
0	2020-1	F	STC	Se	66.80	58.24	-8.56

(continues on next page)

(continued from previous page)

1	2020-2	F	STM	Se	104.67	151.90	47.23
2	2020-3	F	STC	Se	46.17	16.56	-29.61
3	2020-4	F	STC	Se	58.68	73.77	15.09
...
16380	2023-5440	M	STM	Fr	59.95	50.37	-9.58
16381	2023-5441	M	STL	Fr	71.57	43.60	-27.97
16382	2023-5442	M	STB	Fr	100.52	38.29	-62.23
16383	2023-5443	M	STM	Fr	57.45	29.43	-28.02

[16384 rows x 7 columns]

13.3.1 Descriptive statistics and visualizations

Before we set up our regression, we'll first use descriptive analyses and visualizations to get a better sense of our underlying data.

As noted above, the bookstore saw a significant increase in per-student sales from the fall to the spring:

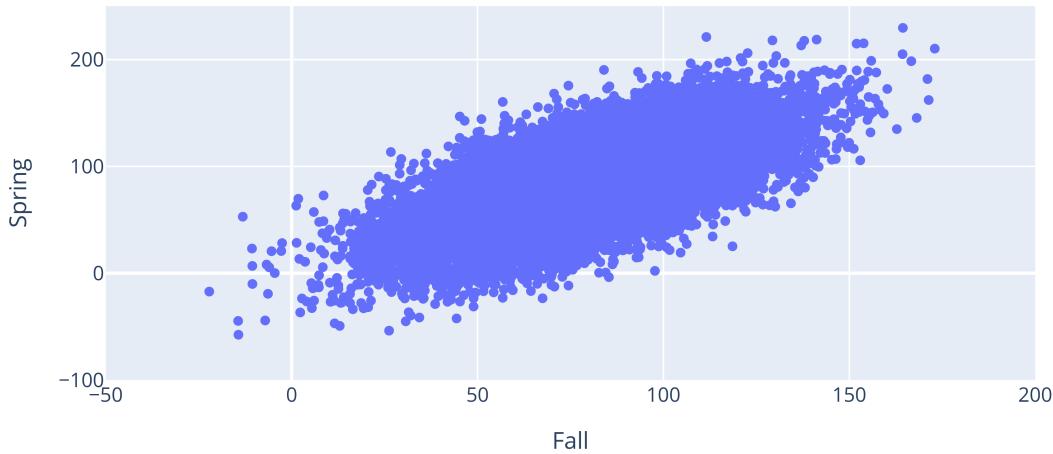
```
df_sales[['Fall', 'Spring']].mean()
```

```
Fall      80.091662
Spring    83.644286
dtype: float64
```

We can gain a better sense of the relationship between fall and spring sales by plotting each set of sales within a scatter chart. Each dot in the following chart reflects a single student's fall and spring spending (shown within the x and y axes, respectively).

```
fig_fall_spring_sales_scatter = px.scatter(
    df_sales, x = 'Fall', y = 'Spring',
    title = 'Fall Sales vs. Spring Sales').update_layout(
    xaxis_range = [-50, 200], yaxis_range = [-100, 250])
wadi(fig_fall_spring_sales_scatter,
    'Charts/fall_spring_sales_scatter', display_type = display_type)
```

Fall Sales vs. Spring Sales



The ‘blob’ of results slopes upward, indicating that students who spent more in the fall also tended to spend more in the spring. However, we can clearly observe a great deal of variation in spring spending among students with similar fall spending patterns. Some students who spent around 100 in the fall, for instance, went on to spend roughly 200 in the spring; other students with this fall spending total hardly spent anything at the bookstore during the spring semester.

Did these spending patterns vary by college and level, however? To investigate that question, we can first create a pivot table that determines mean changes in spending by these groups.

```
df_sales_pivot = df_sales.pivot_table(
    index = ['college', 'level'], values = 'Fall_Spring_Change',
    aggfunc = 'mean').reset_index()
df_sales_pivot['level_for_sorting'] = df_sales_pivot['level'].map(
    {'Fr':0, 'So':1, 'Ju':2, 'Se':3})
df_sales_pivot.sort_values('level_for_sorting', inplace = True)
df_sales_pivot.head()
```

	college	level	Fall_Spring_Change	level_for_sorting
0	STB	Fr	-10.410894	0
4	STC	Fr	-12.115880	0
12	STM	Fr	-1.225103	0
8	STL	Fr	0.107170	0
11	STL	So	-0.292982	1

We can then use Plotly to create a grouped bar chart of these results:

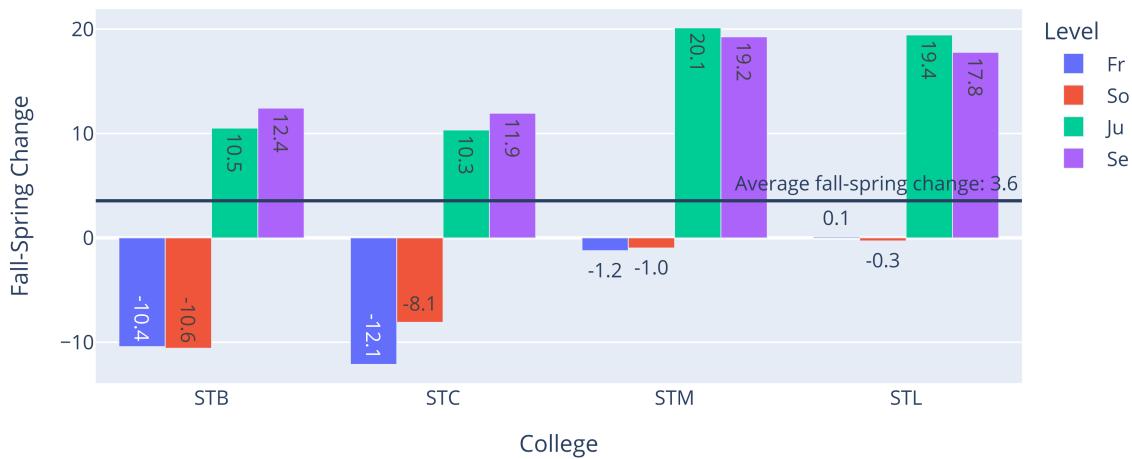
```
fig_fall_spring_sales_grouped_bar = px.bar(
    df_sales_pivot, x = 'college', y = 'Fall_Spring_Change',
    color = 'level', barmode = 'group', text_auto = '.1f',
    title = 'Average Fall-Spring Sales Growth \
by College and Level').update_layout(
    xaxis_title = 'College', yaxis_title = 'Fall-Spring Change',
    legend_title = 'Level')
```

(continues on next page)

(continued from previous page)

```
# Adding a reference line that displays the average fall-spring change:
avg_fall_spring_change = df_sales['Fall_Spring_Change'].mean()
fig_fall_spring_sales_grouped_bar.add_hline(y =
    avg_fall_spring_change,
    annotation_text = f'Average fall-spring change: \
{round(avg_fall_spring_change, 1)}')
# For more on add_hline,
# see https://plotly.com/python/horizontal-vertical-shapes/ .
wadi(fig_fall_spring_sales_grouped_bar,
    'Charts/fall_spring_sales_grouped_bar',
    display_type=display_type)
```

Average Fall-Spring Sales Growth by College and Level



There are some interesting patterns here. First, STL and STM appeared to have higher fall-spring spending growth than did STB and STC. This suggests that the bookstore's marketing campaign had a positive impact, but it doesn't prove it. (Some unrelated factor, unbeknownst to us, might have driven this growth—but it's still a positive sign for the marketing team.)

Second, average spending *dropped* for freshmen and sophomores but rose for juniors and seniors. Removing unhealthy snacks from the checkout aisles may truly have depressed sales for younger students!

(In the Graphing section of Python for Nonprofits, I suggested that you compare this chart to the histogram matrix titled 'Fall/Spring Sales Distributions by College and Level'. Both show similar sets of data, but I find this grouped bar chart to be far more intuitive.)

13.3.2 Creating a linear regression using a formula approach:

We can now use a regression in order to examine the relationship between college, gender, level, and fall sales on spring spending. Thanks to Python's powerful statsmodels library, we can set up this regression with only a few lines of code.

I'll use Statsmodels' *formula* API for this section, as it easily accommodates categorical variables. (College and gender are examples of categorical variables, as their values are specific points rather than a range; a student can't be 'between' STB and STL. Fall spending, on the other hand, is an example of a continuous variable).

The formula argument in the following code instructs Statsmodels to create a regression that evaluates the impact of our four independent variables (fall spending, college, level, and gender) on our dependent variable (spring spending).

(The documentation at <https://www.statsmodels.org/stable/examples/notebooks/generated/formulas.html> proved very helpful in writing this section.)

```
sales_lr_1 = smf.ols(formula = "Spring ~ Fall + college + level + gender",
                      data = df_sales)
sales_lr_results_1 = sales_lr_1.fit()
# sales_lr_results_1.summary()
```

Note: the following output could be displayed via sales_lr_results_1.summary(); however, this output was not compatible with my current Jupyter Book export setup. (See <https://github.com/jupyter-book/jupyter-book/issues/2279> for more details on this error.) Therefore, I'm instead displaying screenshots of the output of my summary() function call. (I'll do the same with subsequent regression examples within this section.) (Also note that not all of these screenshots contain all Statsmodels-provided warnings that normally precede regression statistics.)

```
Image('Regression_Screenshots/sales_lr_results_1_summary.png', width = 600)
```

OLS Regression Results								
Dep. Variable:	Spring		R-squared:	0.539				
Model:	OLS		Adj. R-squared:	0.539				
Method:	Least Squares		F-statistic:	2397.				
Date:	Mon, 24 Feb 2025		Prob (F-statistic):	0.00				
Time:	21:09:35		Log-Likelihood:	-76189.				
No. Observations:	16384		AIC:	1.524e+05				
Df Residuals:	16375		BIC:	1.525e+05				
Df Model:	8							
Covariance Type:	nonrobust							
	coef	std err	t	P> t	[0.025	0.975]		
Intercept	-10.4393	0.821	-12.711	0.000	-12.049	-8.830		
college[T.STC]	0.1104	0.575	0.192	0.848	-1.017	1.238		
college[T.STL]	9.3680	0.573	16.361	0.000	8.246	10.490		
college[T.STM]	8.9259	0.536	16.649	0.000	7.875	9.977		
level[T.Ju]	20.8468	0.544	38.328	0.000	19.781	21.913		
level[T.Se]	21.3143	0.565	37.733	0.000	20.207	22.421		
level[T.So]	0.6421	0.528	1.216	0.224	-0.393	1.677		
gender[T.M]	-0.0920	0.396	-0.233	0.816	-0.868	0.684		
Fall	1.0021	0.008	126.745	0.000	0.987	1.018		
Omnibus:	0.966	Durbin-Watson: 1.992						
Prob(Omnibus):	0.617	Jarque-Bera (JB): 0.941						
Skew:	-0.016	Prob(JB): 0.625						
Kurtosis:	3.019	Cond. No. 422.						
Notes:								
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.								

13.3.3 Analyzing our first regression summary

There's a lot of output here, so I'll just focus on a few key elements.

The R-squared value at the top right suggests that 53.9% of all variance in spring spending can be explained by students' fall spending, college, gender, and level. That's a sizeable amount, though it still leaves almost half of students' spring spending patterns unexplained.

The midpoint of the regression summary contains a list of coefficients; these show the effect of each variable on spending after adjusting for all other variables. The `coeff` column displays the actual coefficients; the `[0.025 and 0.975]` columns show the 95% confidence interval for these coefficients; and the `P>|t|` column shows their p-values—i.e. the likelihood that we would have seen a coefficient of this magnitude if the actual coefficient was *not* 0.

Note that the regression treats each college, gender, and level option as a boolean (binary) value: each respondent either belongs to that value or does not. (Meanwhile, fall spending is viewed as a continuous variable.) Thus, the `gender[T.M]` coefficient refers to male students; the `college[T.STL]` coefficient refers to STC students; and so on.

You'll also note that, for each categorical variable, one value (F for gender, Fr for level, and STB for college) is missing from the results. This is the baseline value against which other values are compared. For example, the `college[T.STL]` coefficient of 9.368 means that, after adjusting for the other variables in our model, STL students spent around 9.37 more than STB students. Meanwhile, the `level[T.Ju]` coefficient of 20.8468 means that juniors can be expected to spend around 20.85 more than freshmen (after accounting for our other variables).

The 'Fall' variable of 1.0021 means that every dollar of fall spending is associated with slightly more than one dollar in spring spending (again, after other variables are accounted for).

The intercept predicts what a given student's spending would be if all input values equaled 0. In this case, a student with input values of 0 would be a female freshman at STB who spent 0 dollars in the fall; this is because 'female', 'Fr' (for 'freshman'), and 'STB' are all missing from our coefficient lists, indicating that they are being treated as our baseline.

Our p-values can help us determine which variables have a significant relationship with spending and which do not. The 0.816 p-value for the `gender[T.M]` variable, for instance, suggests that male students' spring spending didn't differ significantly from female students', even though women did spend slightly more than men. Our list of p-values also reveals that STL and STM students did spend significantly more than STB students, but STC students did not. (This suggests that the bookstore's marketing campaign had a significant positive impact on STL and STM spending—and that the management should consider launching this same campaign at STB and STC also.) Finally, we can see that juniors and seniors, but not sophomores, had significantly higher spending than freshmen. This gives us further evidence (but not proof) that the unhealthy snacks the bookstore phased out *had* indeed drawn in traffic from younger students!

13.3.4 Using these regression results to make predictions

Now that we have our regression coefficients, we can use them to make predictions about spring spending patterns. (There are two issues with the simplified approach I'm showing here. First, we don't really *need* to make predictions because we have actual spring spending data for everyone at NVCU. And second, a more robust approach would be to create a 'training' and 'test' dataset; build our model only off the former; and then test it on the latter. However, to keep this section simple, I'm using the same dataset for both my training and testing procedures.)

Let's try predicting the spring sales of the student whose ID is 2020-1:

```
df_sales.query("student_id == '2020-1'").drop('Spring', axis = 1)
```

	student_id	gender	college	level	Fall	Fall_Spring_Change
0	2020-1	F	STC	Se	66.8	-8.56

This is a female senior at STC who spent 66.80 at the bookstore in the fall. According to our model, how much could we have expected her to spend in the spring?

We can predict her spring spending total by starting with the intercept (which, as mentioned earlier, represents a female STB freshman who spent 0 dollars in the fall); adding the product of our Fall coefficient and 66.8; and then adding the value of the STC and Senior (Se) coefficients. Here's what that total amounts to:

```
# Note that you can retrieve each regression coefficient via
# the regression results' params attribute.
(sales_lr_results_1.params['Intercept']
 + sales_lr_results_1.params['Fall'] * 66.8
 + sales_lr_results_1.params['college[T.STC]']
 + sales_lr_results_1.params['level[T.Se]'])
```

```
np.float64(77.92859960593265)
```

As shown below, however, this student's actual spring spending was only 58.24, so the model overshot here by nearly 20 dollars.

```
df_sales.query("student_id == '2020-1'").iloc[0]
```

```
student_id      2020-1
gender          F
college         STC
level           Se
Fall            66.8
Spring          58.24
Fall_Spring_Change -8.56
Name: 0, dtype: object
```

We can easily generate predictions for all rows within df_sales by using the results' predict method. (These predictions would be particularly useful for a new set of data for which we don't yet have our actual results.) Note that the value of our first prediction (for the student with an ID of 2020-1) matches the one calculated above.

```
df_sales['Predictions'] = sales_lr_results_1.predict(
    df_sales[['Fall', 'gender', 'college', 'level']])
df_sales
```

```
student_id  gender  college  ...  Spring  Fall_Spring_Change  Predictions
0          2020-1     F       STC  ...    58.24             -8.56    77.928600
1          2020-2     F       STM  ...   151.90              47.23   124.695205
2          2020-3     F       STC  ...    16.56             -29.61    57.254379
3          2020-4     F       STC  ...    73.77              15.09   69.791194
...
16380    2023-5440    M       STM  ...    50.37             -9.58    58.473027
16381    2023-5441    M       STL  ...    43.60             -27.97   70.560045
16382    2023-5442    M       STB  ...    38.29             -62.23   90.204115
16383    2023-5443    M       STM  ...    29.43             -28.02   55.967668
```

```
[16384 rows x 8 columns]
```

Now that we've added all of our predictions, we can compare them to our actual results via a scatter plot:

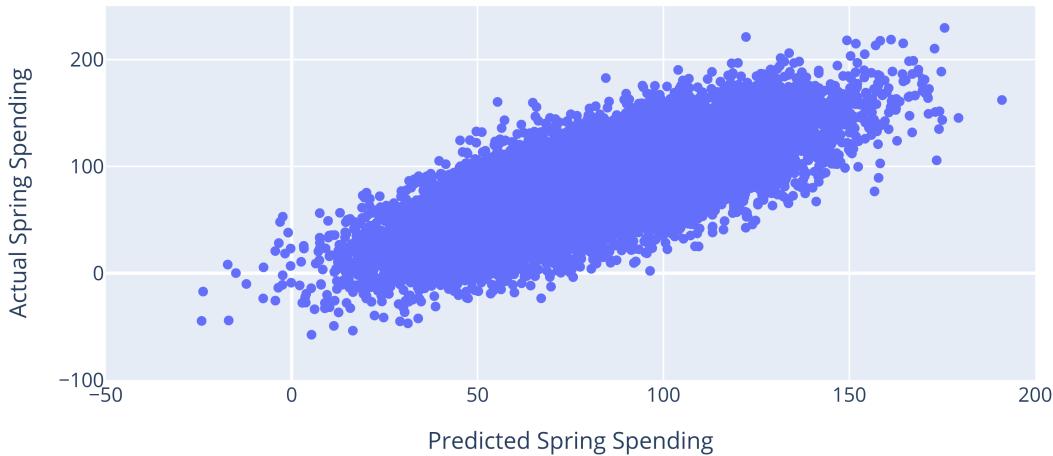
```
fig_spring_predictions_vs_actual = px.scatter(
    x = df_sales['Predictions'], y = df_sales['Spring'],
    title = 'Predicted vs. Actual Spring Spending').update_layout(
        xaxis_title = 'Predicted Spring Spending',
```

(continues on next page)

(continued from previous page)

```
yaxis_title = 'Actual Spring Spending',
xaxis_range = [-50, 200], yaxis_range = [-100, 250])
wadi(fig_spring_predictions_vs_actual,
'Charts/spring_predictions_vs_actual', display_type = display_type)
```

Predicted vs. Actual Spring Spending



If you compare this graph to our ‘Spring Sales as Function of Fall Sales’ scatter plot, you’ll find (perhaps after squinting for a bit) that our new chart’s y axis values are clustered a bit more closely together. (I made the two easier to compare by setting the same x and y axis ranges for each.) This makes sense, as our first chart took only fall sales into account; this new chart also considers the impact of college, level, and gender, and thus can make more accurate predictions of spring spending.)

13.3.5 Checking the normality of our residuals

For a linear regression to be valid, certain criteria need to be met. Helpful overviews of these assumptions can be found at <https://ethanweed.github.io/pythonbook/05.04-regression.html#assumptions-of-regression> and at <https://www.kirenz.com/blog/posts/2021-11-14-linear-regression-diagnostics-in-python/>. In addition, a list of Statsmodels functions for testing regression assumptions is available at <https://www.statsmodels.org/stable/diagnostic.html>.

As noted in the introduction, this chapter won’t provide a detailed overview of all of these assumptions; however, if you plan to implement regressions in a professional setting, I strongly recommend that you look into these criteria and their corresponding tests. In the following cells, I’ll check just one assumption: that the residuals of our regression are normally distributed.

We can visualize the distribution of our residuals via a histogram: (setting histnorm to ‘probability density’ will make this chart more compatible with a normal distribution curve that we’ll add on top of it.)

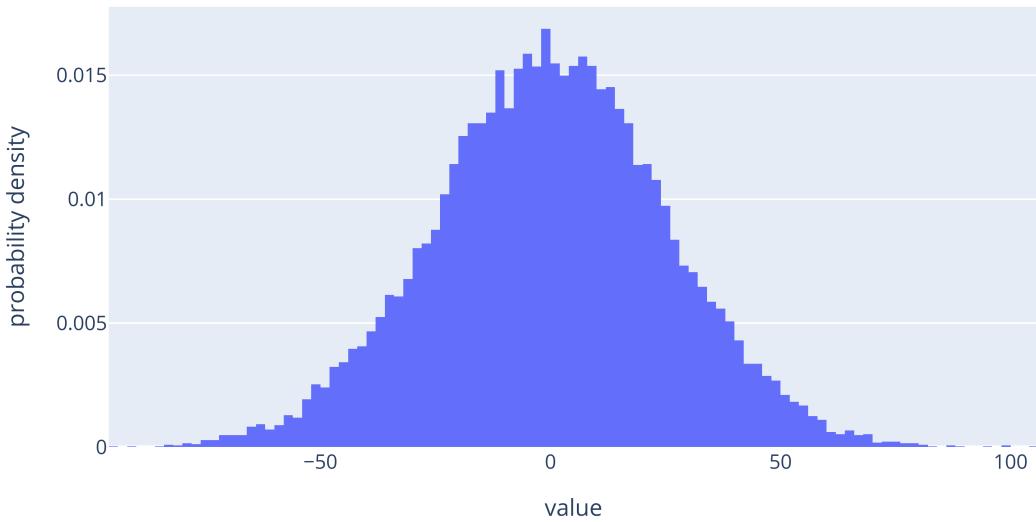
```
fig_sales_residual_hist = px.histogram(
    sales_lr_results_1.resid,
    histnorm = 'probability density').update_layout(
    showlegend = False, title = 'Histogram of regression residuals')
```

(continues on next page)

(continued from previous page)

```
wadi(fig_sales_residual_hist, 'Charts/sales_residual_hist',
     display_type = display_type)
```

Histogram of regression residuals



Let's compare this distribution to a plot of a standard normal distribution. To visualize this normal distribution, we'll first create a DataFrame that has a similar range of x values (-100 to 100) as the histogram above. This DataFrame's y values will be based on the normal distribution's probability density function; the standard deviation used within this function will match that of our residuals.

I converted the code for creating our normal distribution DataFrame into a function so that I could reuse it later in this section.

```
def gen_normdist(min, max, rows = 1000, mean = 0, stdev = 1):
    '''This function creates a DataFrame that
    can be used to plot a normal distribution. The 'y' field within
    this DataFrame will show the normal distribution function's
    output for each 'x' field.

    min and max: the minimum and maximum x values to include within
    the DataFrame.

    rows: the number of rows to include within the DataFrame. Higher
    numbers will result in smoother plots.

    mean and stdev: the mean and standard deviation to use within the
    normal distribution function.
    '''
    df_normdist = pd.DataFrame(
        data={'x':np.linspace(min, max, rows+1)})
    # The code above uses np.linspace to list 'rows' values of x that range
    # from 'min' to 'max' (inclusive). For more on this function, see:
    # https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
```

(continues on next page)

(continued from previous page)

```
df_normdist['y'] = (
    1/math.sqrt(2*math.pi*(stdev**2)))*math.e**(
        (-1*(df_normdist['x'] - mean)**2) / (2*stdev**2))
# This code is based on the function found at
# https://en.wikipedia.org/wiki/Normal_distribution
return df_normdist
```

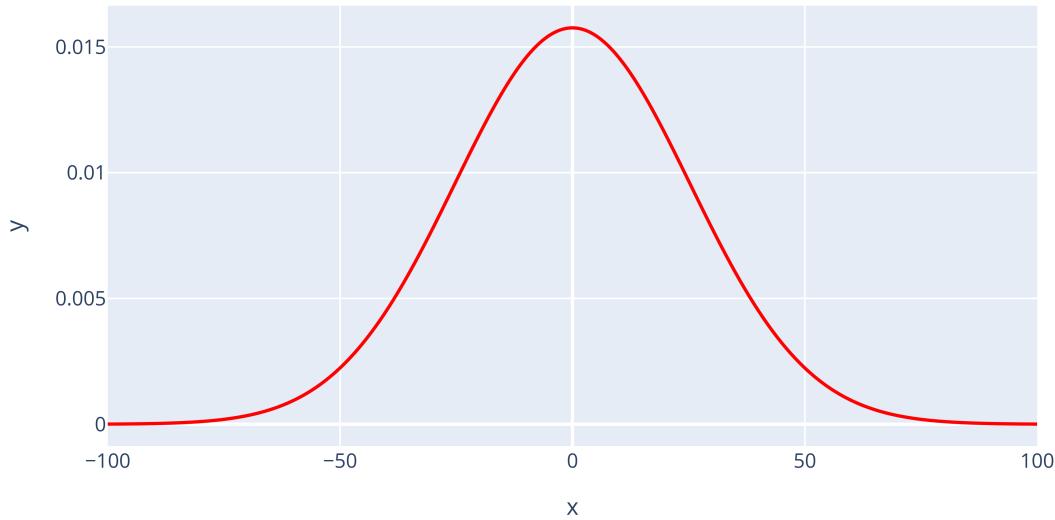
```
df_normdist = gen_normdist(
    min = -100, max = 100,
    mean = 0, stdev = np.std(sales_lr_results_1.resid))
df_normdist
```

	x	y
0	-100.0	0.000006
1	-99.8	0.000007
2	-99.6	0.000007
3	-99.4	0.000007
...
997	99.4	0.000007
998	99.6	0.000007
999	99.8	0.000007
1000	100.0	0.000006

[1001 rows x 2 columns]

We'll graph these x and y values using px.line():

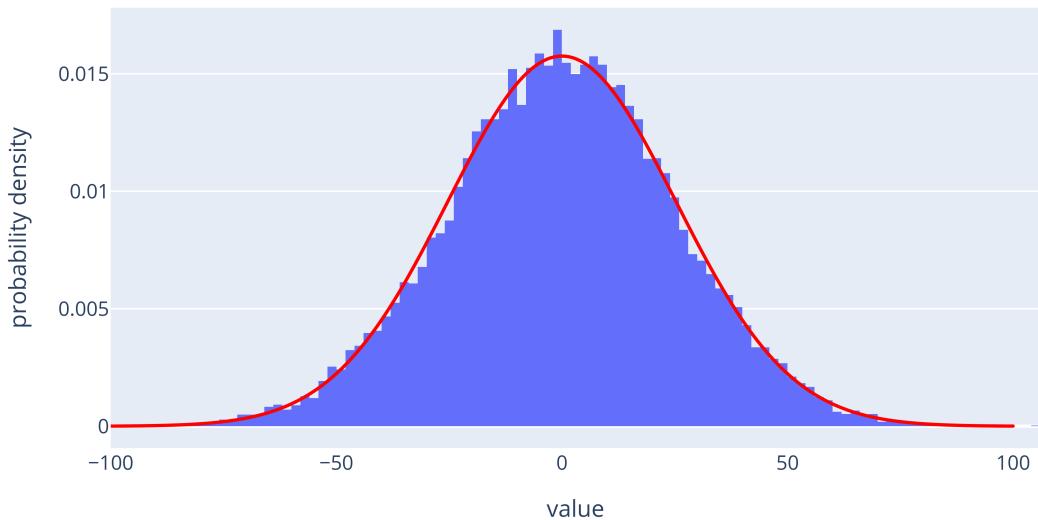
```
fig_normdist = px.line(df_normdist, x = 'x', y = 'y',
                       color_discrete_sequence = ['red'])
wadi(fig_normdist, 'Charts/normal_distribution_plot',
     display_type = display_type)
```



We can now add this curve directly onto our regression residuals plot via `add_trace()`. (This code was based on Shaunak Sen's StackOverflow response at <https://stackoverflow.com/a/66891500/13097194>.) We'll also update our title to reflect our addition.

```
fig_sales_residual_hist.add_trace(  
    fig_normdist['data'][0]).update_layout(  
    title= 'Histogram of regression residuals compared to normal \\  
distribution')  
wadi(fig_sales_residual_hist, 'Charts/sales_residual_hist_vs_normdist',  
    display_type = display_type)
```

Histogram of regression residuals compared to normal distribution



Our residuals graph aligns very closely to this normal distribution plot. For a more precise test, though, we can incorporate Statsmodels' `omni_normtest` function:

```
sms.omni_normtest(sales_lr_results_1.resid)
```

```
NormaltestResult(statistic=np.float64(0.9660949405011305), pvalue=np.float64(0.  
˓→6169005382553423))
```

This p-value is above 0.05, which suggests that our residuals are indeed normally distributed.

If you glance back at our regression output, you'll see that these two numbers (0.966 and 0.617) match the 'Omnibus:' and 'Prob(Omnibus):' rows near the bottom. In other words, Statsmodels performs this test for you when you run your regression—thus saving you the trouble of running it yourself.

What sort of results would we get if our residuals did not match the normal distribution so well? Let's find out! The following cell creates a 'skewed' set of these residuals by multiplying all positive values by 2, then visualizes them using a histogram:

```
skewed_resids = [resid * 2 if resid > 0 else resid \  
for resid in sales_lr_results_1.resid]  
  
fig_skewed_resids = px.histogram(
```

(continues on next page)

(continued from previous page)

```
skewed_resids,
histnorm = 'probability density',
title = 'Example of non-normally-distributed regression residuals')
fig_skewed_resids
wadi(fig_skewed_resids, 'Charts/skewed_residuals')
```

We'll now rerun our omni_normtest:

```
sms.omni_normtest(skewed_resids)
```

```
NormaltestResult(statistic=np.float64(1347.574679202082), pvalue=np.float64(2.
˓→3871320020229674e-293))
```

This p value is incredibly small, indicating that our residuals are *not* normally distributed.

13.3.6 Comparing fall sales between colleges

In the introduction, we discussed the whether the difference in fall STB and STL sales was statistically significant. We can now use a linear regression to find out.

First, we'll evaluate fall sales for each college, then determine how much higher STB's sales were than STL's.

```
df_fall_sales = df_sales.pivot_table(index = 'college',
                                       values = 'Fall',
                                       aggfunc = 'mean').reset_index()
df_fall_sales
```

	college	Fall
0	STB	80.281666
1	STC	80.201572
2	STL	79.691695
3	STM	80.153226

Calculating and reporting the nominal and % differences in fall sales between STL and STB:

```
fall_stb_sales = df_fall_sales.query("college == 'STB'").iloc[0]['Fall']
fall_stl_sales = df_fall_sales.query("college == 'STL'").iloc[0]['Fall']
stl_stb_nominal_diff = round(fall_stl_sales - fall_stb_sales, 2)
stl_stb_pct_diff = (round(100*(fall_stl_sales/fall_stb_sales-1), 2))
print(f"STB's average fall sales were {stl_stb_pct_diff}% lower (and \
{stl_stb_nominal_diff} lower in nominal terms) than were STL's.")
```

```
STB's average fall sales were -0.73% lower (and -0.59 lower in nominal terms) than
˓→were STL's.
```

```
sales_lr_2 = smf.ols(formula = "Fall ~ college",
                     data = df_sales.query("college in ['STL', 'STB']"))
sales_lr_results_2 = sales_lr_2.fit()
# sales_lr_results_2.summary()
Image('Regression_Screenshots/sales_lr_results_2_summary.png', width = 600)
```

OLS Regression Results						
Dep. Variable:	Fall		R-squared:	0.000		
Model:	OLS		Adj. R-squared:	0.000		
Method:	Least Squares		F-statistic:	1.083		
Date:	Mon, 24 Feb 2025		Prob (F-statistic):	0.298		
Time:	21:35:17		Log-Likelihood:	-36655.		
No. Observations:	7894		AIC:	7.331e+04		
Df Residuals:	7892		BIC:	7.333e+04		
Df Model:	1					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
Intercept	80.2817	0.390	206.062	0.000	79.518	81.045
college[T.STL]	-0.5900	0.567	-1.041	0.298	-1.701	0.521
Omnibus:	0.914	Durbin-Watson: 1.989				
Prob(Omnibus):	0.633	Jarque-Bera (JB): 0.878				
Skew:	0.001	Prob(JB): 0.645				
Kurtosis:	3.052	Cond. No. 2.56				

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Note that the intercept matches the fall STB spending that we calculated earlier; that's because it indeed reflects fall STB spending. In addition, the STL coefficient matches the nominal difference in spending between STL and STB that we calculated earlier.

Our p-value is 0.298, which is well above the 0.05 significance threshold that we're using within this project. Thus, we'll conclude here that there was not a significant difference in fall spending between these two colleges.

13.3.7 Comparing these results to those of a t-test

If you have a background in statistics, you may be familiar with t-tests (https://en.wikipedia.org/wiki/Student's_t-test). Because our comparison of STL's and STB's fall spending includes two distinct sets of means, we could also have used an independent-samples t-test to determine whether they were statistically significant. That's exactly what the following cell does:

```
tstat, pvalue, dof = ttest_ind(
    x1 = df_sales.query("college == 'STL'")['Fall'],
    x2 = df_sales.query("college == 'STB'")['Fall'])
# This code was based on the example shown at
# https://www.statsmodels.org/stable/generated/statsmodels.stats.
# weightstats.ttest_ind.html#statsmodels.stats.weightstats.ttest_ind
print("T statistic:",tstat, "\np-value:",pvalue)
```

```
T statistic: -1.0407829313964725
p-value: 0.2980081862689233
```

Note that the T statistic and p-value reported here match those for the STL coefficient in our regression model.

13.4 Part 2: Evaluating the relationship between bachelor's degree prevalence and median incomes

Now that we've tried out regression models on simulated data, let's have a look at some real-world data. As explained back in the Census Data Imports section, NVCU's admissions department wishes to demonstrate that getting a bachelor's degree is, on average, a financial benefit to college graduates.

Therefore, we retrieved American Community Survey data within that section that showed (1) median earnings for all residents aged 25 or older *and* (2) the percentage of residents who had a bachelor's degree. If these two metrics are positively correlated, that will boost the admissions department's case that going to college can help boost individuals' future earnings.* We'll now create regression models that evaluate whether there's a significant relationship between these two variables.

**This is a flawed approach to demonstrating the financial benefits of a bachelor's degree: after all, the mere fact that median incomes are positively correlated with the percentage of residents who have a 4-year degree does not conclusively prove that bachelor's-degree holders earn more than high-school graduates. The latter group might earn just as much, or even more, than the former! However, I included this analysis so that I could show a real-world example of a regression model with continuous independent and dependent variables. Later in this section, I'll provide an additional model that directly compares bachelor's and high-school-degree earnings.*

13.4.1 Importing our county-level education and income dataset

```
df_cmib = pd.read_csv(
    '../Census_Data_Imports/Datasets/education_and_earnings_county.csv')
# 'cmib' is short for 'county median incomes and bachelor's degrees'.
# Simplifying our dataset by removing extraneous fields:

df_cmib.drop(['Year', 'state', 'county'], axis = 1, inplace = True)
# Limiting our output to counties in one of the 50 US states (plus DC):
df_cmib.query("State_Abbrev != 'PR'", inplace = True)
df_cmib[['NAME', 'Median_Earnings_for_Total_25plus_Population',
        'Pct_With_Bachelors_Degree']]
```

```

NAME  Median_Earnings_for_Total_25plus_Population \
0    Autauga County, Alabama          47719
1    Baldwin County, Alabama         47748
2    Barbour County, Alabama        34557
3    Bibb County, Alabama           34001
...
3140   Teton County, Wyoming       ...
3141   Uinta County, Wyoming      55463
3142   Washakie County, Wyoming   43883
3143   Weston County, Wyoming    42474
                                         ...
                                         ...
Pct_With_Bachelors_Degree
0            28.282680
1            32.797637
2            11.464715
3            11.468207
...
3140          ...
3141          61.219716
3142          21.319190
3143          22.032358
                                         ...
                                         ...
[3144 rows x 3 columns]

```

13.4.2 Cleaning our dataset

Our dataset includes not only median earnings for all county residents, but earnings for specific education levels also. Let's take a look at these averages (the ‘mean of the medians’, if you will):

```

median_earnings_cols = ['Median_Earnings_for_Total_25plus_Population',
                       'Median_Earnings_Less_Than_HS',
                       'Median_Earnings_HS',
                       'Median_Earnings_Some_College',
                       "Median_Earnings_Bachelors_Degree",
                       "Median_Earnings_Postgraduate"]
df_cmib[median_earnings_cols].mean()

```

Median_Earnings_for_Total_25plus_Population	-3.794012e+05
Median_Earnings_Less_Than_HS	-4.089420e+07
Median_Earnings_HS	-3.779011e+06
Median_Earnings_Some_College	-1.653628e+06
Median_Earnings_Bachelors_Degree	-4.396728e+06
Median_Earnings_Postgraduate	-1.795533e+07
<i>dtype:</i>	float64

Well, these numbers don't seem right—they're all negative! What's going on?

The issue here is that the number -666,666,666 is getting used as a code for missing income statistics. The presence of this giant negative number within our dataset is dramatically skewing our average calculations.

You can identify this number by calling `.min()` on the dataset's median income columns:

```
df_cmib[median_earnings_cols].min()
```

```
Median_Earnings_for_Total_25plus_Population    -6666666666
Median_Earnings_Less_Than_HS                  -6666666666
Median_Earnings_HS                          -6666666666
Median_Earnings_Some_College                -6666666666
Median_Earnings_Bachelors_Degree             -6666666666
Median_Earnings_Postgraduate                 -6666666666
dtype: int64
```

I also looked for other unusually small values that might represent additional codes for invalid data, but I didn't find any:

```
for col in median_earnings_cols:
    print(df_cmib.sort_values(col)[col].unique())
```

```
[-6666666666  19366   23388 ...  96611   97482  106932]
[-6666666666  2499    3417  ...  87929  105764  128554]
[-6666666666  13961   15238 ...  75434  84750   88493]
[-6666666666  16806   17040 ...  90183  93458   100365]
[-6666666666  2499    18963 ...  110960  115313  158333]
[-6666666666  2499    13750 ...  154872  164519  250001]
```

I'll now remove invalid 'Median_Earnings_for_Total_25+Population' entries from our dataset by filtering out rows with *negative* values. (I could have only filtered out rows with entries of -666666666, but that code would fail to work for future versions of this dataset if the Census Bureau happened to replace that number with a different one.)

Since the only earnings row that we'll be using in our dataset is the 'Median_Earnings_for_Total_25+Population' one, I won't filter out invalid numbers in other columns just yet, as that would cause many rows with valid 'Median_Earnings_for_Total_25+Population' entries to get removed.

```
df_cmib.query("`Median_Earnings_for_Total_25plus_Population` >= 0",
              inplace = True)
# Enclosing this field in backticks helped query() correctly parse it.

# Confirming that we removed all -666666666 values from our overall
# median earnings column:
df_cmib[['NAME', 'Median_Earnings_for_Total_25plus_Population',
        'Pct_With_Bachelors_Degree']].sort_values(
        'Median_Earnings_for_Total_25plus_Population').head(5)
```

	NAME	Median_Earnings_for_Total_25plus_Population
2593	Edwards County, Texas	19366
1113	Wolfe County, Kentucky	23388
571	Custer County, Idaho	23782
2548	Brooks County, Texas	24585
285	Mineral County, Colorado	24783
	Pct_With_Bachelors_Degree	
2593	25.921909	
1113	7.393428	
571	25.196424	
2548	14.440783	
285	54.532164	

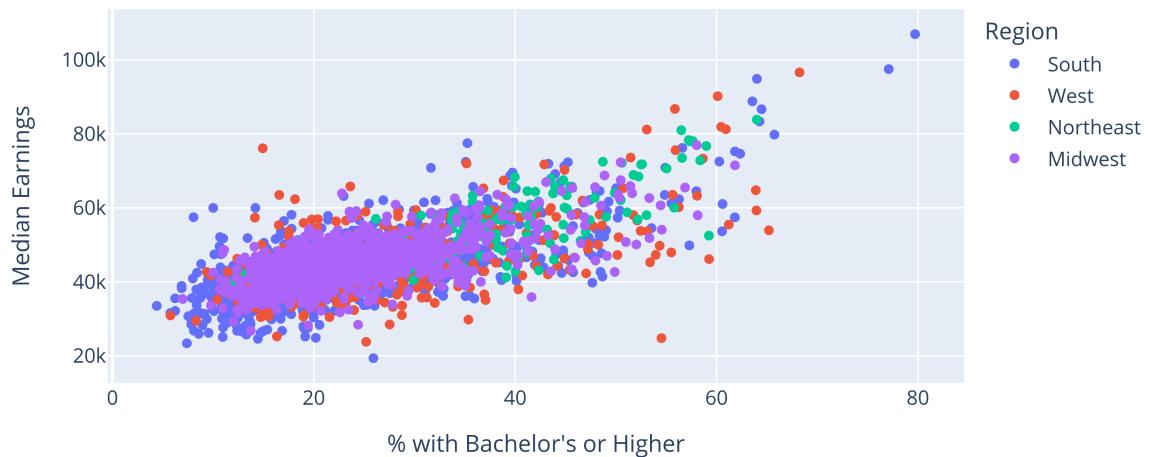
As in the first section, we'll visualize our data before incorporating it into a model. The following scatter plot shows how the prevalence of bachelor's degrees correlates with median earnings. The dots are also color-coded by region to help us identify any regional patterns that might be present.

```

fig_ba_pct_earnings_scatter = px.scatter(
    df_cmib, x = "Pct_With_Bachelors_Degree",
    y = 'Median_Earnings_for_Total_25plus_Population',
    color = 'Region', hover_data = 'NAME',
    title = "Median County Earnings as Function of % of County \
Residents<br>With a Bachelor's Degree or Higher").update_layout(
    xaxis_title = "% with Bachelor's \
or Higher", yaxis_title = "Median Earnings")
# Note the use of <br>, a piece of HTML code, in the title to add in
# a line break. For more documentation on applying HTML code within
# Plotly text elements, visit
# https://plotly.com/chart-studio-help/adding-HTML-and-links-to-charts/ .
# (This page is meant for Plotly's paid Chart Studio offering, but
# many or all of the HTML items will also work within free Plotly
# graphs.)
wadi(fig_ba_pct_earnings_scatter, 'Charts/county_ba_pct_earnings_scatter',
    display_type = display_type)

```

Median County Earnings as Function of % of County Residents
With a Bachelor's Degree or Higher



I'm seeing a hint of a curve in the above scatter plot. Therefore, in order to allow our model to better account for this curve, I'll add in a squared version of the 'Pct_With_Bachelors_Degree' column.

```

df_cmib['Pct_With_Bachelors_Degree_Squared'] = df_cmib[
    'Pct_With_Bachelors_Degree']**2

```

13.4.3 Building our first regression model

This model will use both the original and squared versions of our bachelor's degree prevalence field in order to predict median earnings. (This is because I found both of these elements to have a statistically significant relationship with our earnings values.)

```
cmib_lr_1 = smf.ols(formula = "Median_Earnings_for_Total_\n25plus_Population ~ Pct_With_Bachelors_Degree + \
Pct_With_Bachelors_Degree_Squared",
                     data = df_cmib)
cmib_lr_results_1 = cmib_lr_1.fit()
# cmib_lr_results_1.summary()
Image('Regression_Screenshots/cmib_lr_results_1_summary.png', width = 550)
```

OLS Regression Results													
Dep. Variable: Median_Earnings_for_Total_25plus_Population				R-squared: 0.498									
Model:		OLS	Adj. R-squared: 0.497										
Method:		Least Squares		F-statistic: 1555.									
Date:		Mon, 24 Feb 2025		Prob (F-statistic): 0.00									
Time:		21:40:53		Log-Likelihood: -31696.									
No. Observations:		3142		AIC: 6.340e+04									
Df Residuals:		3139		BIC: 6.342e+04									
Df Model:		2											
Covariance Type:		nonrobust											
		coef	std err	t	P> t	[0.025	0.975]						
		Intercept	3.477e+04	588.871	59.038	0.000	3.36e+04	3.59e+04					
		Pct_With_Bachelors_Degree	283.5300	41.660	6.806	0.000	201.847	365.213					
		Pct_With_Bachelors_Degree_Squared	4.5607	0.655	6.961	0.000	3.276	5.845					
Omnibus: 175.709		Durbin-Watson: 1.591											
Prob(Omnibus): 0.000		Jarque-Bera (JB): 664.486											
Skew: 0.119		Prob(JB): 5.11e-145											
Kurtosis: 5.240		Cond. No. 5.35e+03											
Notes:													
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.													
[2] The condition number is large, 5.35e+03. This might indicate that there are strong multicollinearity or other numerical problems.													

Our R-squared value is close to 0.5, indicating that the prevalence of bachelor's degrees explains roughly half of all variation in county-level incomes.

Next, let's create a revised version of this regression model that also takes counties' regional settings into account.

```
cmib_lr_2 = smf.ols(formula = "Median_Earnings_for_Total_\\
25plus_Population ~ Pct_With_Bachelors_Degree + \\
Pct_With_Bachelors_Degree_Squared + Region",
                     data = df_cmib)
cmib_lr_results_2 = cmib_lr_2.fit()
# cmib_lr_results_2.summary()
Image('Regression_Screenshots/cmib_lr_results_2_summary.png', width = 600)
```

OLS Regression Results												
Dep. Variable:	Median_Earnings_for_Total_25plus_Population				R-squared:	0.514						
Model:	OLS				Adj. R-squared:	0.513						
Method:	Least Squares				F-statistic:	663.0						
Date:	Fri, 21 Feb 2025				Prob (F-statistic):	0.00						
Time:	00:47:20				Log-Likelihood:	-31645.						
No. Observations:	3142				AIC:	6.330e+04						
Df Residuals:	3136				BIC:	6.334e+04						
Df Model:	5											
Covariance Type:	nonrobust											
		coef	std err	t	P> t	[0.025	0.975]					
	Intercept	3.709e+04	635.611	58.356	0.000	3.58e+04	3.83e+04					
	Region[T.Northeast]	1232.3922	436.812	2.821	0.005	375.926	2088.858					
	Region[T.South]	-1944.9595	237.961	-8.173	0.000	-2411.535	-1478.384					
	Region[T.West]	-1682.2203	326.246	-5.156	0.000	-2321.897	-1042.543					
	Pct_With_Bachelors_Degree	199.2065	42.178	4.723	0.000	116.507	281.906					
	Pct_With_Bachelors_Degree_Squared	5.6411	0.657	8.584	0.000	4.353	6.930					
Omnibus:	225.565	Durbin-Watson:	1.636									
Prob(Omnibus):	0.000	Jarque-Bera (JB):	902.064									
Skew:	0.245	Prob(JB):	1.32e-196									
Kurtosis:	5.579	Cond. No.	5.98e+03									
Notes:												
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.												
[2] The condition number is large, 5.98e+03. This might indicate that there are strong multicollinearity or other numerical problems.												

Each region-level predictor is significant as well. (Note that, because ‘Region’ is a categorical variable, each region (aside from the baseline midwest value) is listed as a separate coefficient; these coefficients show how much more (or less) each region’s median earnings are after adjusting for our bachelor’s-degree fields.)

Next, we’ll examine the difference between our predicted and actual county-level median incomes. (Instead of using `predict()` to generate our predicted values, we’ll simply retrieve these from the `fittedvalues` attribute of our regression results. Similarly, we’ll source our regression residuals from our results’ `resid` attribute.)

```

df_cmib_pred_vs_actual = df_cmib.copy()[['NAME',
'Median_Earnings_for_Total_25plus_Population',
'Pct_With_Bachelors_Degree', 'Pct_With_Bachelors_Degree_Squared',
'Region']].copy()

# Displaying the model's predictions for each county:
df_cmib_pred_vs_actual['Pred_Val'] = cmib_lr_results_2.fittedvalues
# Source of this attribute:
# https://www.statsmodels.org/dev/generated/statsmodels.regression.
# linear_model.OLSResults.html
# Note: we could also have generated these values using the following
# code:
# df_cmib_pred_vs_actual['Pred_Val'] = cmib_lr_results_2.predict(
#     df_cmib_pred_vs_actual[
#         ['Pct_With_Bachelors_Degree',
#             'Pct_With_Bachelors_Degree_Squared', 'Region']]

# Displaying our model's errors for each county:
df_cmib_pred_vs_actual['Actual-Pred'] = cmib_lr_results_2.resid
# We could also have calculated these using the following code:
# df_cmib_pred_vs_actual['Actual-Pred'] = (
#     df_cmib_pred_vs_actual[
#         'Median_Earnings_for_Total_25plus_Population'] -
#     df_cmib_pred_vs_actual['Pred_Val'])

df_cmib_pred_vs_actual.sort_values(
    'Actual-Pred', ascending = False).head(5)

```

	NAME \	Median_Earnings_for_Total_25plus_Population	Pct_With_Bachelors_Degree \	Pct_With_Bachelors_Degree_Squared	Region	Pred_Val	Actual-Pred
87	North Slope Borough, Alaska	76090	14.927417	222.827783	West	39640.377288	36449.622712
1198	Calvert County, Maryland	77471	35.262776	1243.463362	South	49186.122657	28284.877343
2874	Loudoun County, Virginia	94853	64.013746	4097.759728	South	71014.991163	23838.008837
1202	Charles County, Maryland	70792	31.637943	1000.959444	South	47096.033650	23695.966350
1212	St. Mary's County, Maryland	72426	35.100217	1232.025268	South	49089.216077	23336.783923

The table above shows the five counties whose median incomes our model underestimated the most. The first county equivalent within this list (North Slope Borough, AK) has a median income of 76,090-around 36,449 higher than would be predicted using its Region (West) and bachelor's degree-prevalence (14.92%) only. My guess is that many workers in this county without college degrees are highly-paid oil and natural gas workers, which would explain why its median income is higher than expected.

Next, we'll visualize our predicted and actual earnings. If our R-squared were 100, all of these points would exist on the

same line. However, because our R-squared was only 0.514, we can see plenty of variation in actual median earnings for most predicted medians.

```
fig_cmib_pred_vs_actual = px.scatter(
    df_cmib_pred_vs_actual, x = 'Pred_Val',
    y = 'Median_Earnings_for_Total_25plus_Population',
    color = 'Region',
    hover_data = 'NAME', title = "Predicted vs. Actual Median \
County Earnings").update_layout(
    xaxis_title = 'Predicted Median Earnings',
    yaxis_title = 'Actual Median Earnings')
wadi(fig_cmib_pred_vs_actual, 'Charts/cmib_pred_vs_actual',
    display_type=display_type)
```

Predicted vs. Actual Median County Earnings



Finally, we'll compare a histogram of our residuals to a representative normal distribution:

```
fig_cmib_hist = px.histogram(
    x = cmib_lr_results_2.resid,
    title = 'Histogram of regression residuals compared to \
normal distribution',
    histnorm = 'probability density')

# Adding a normal distribution line:
# (Unlike in the earlier section, we'll generate the DataFrame for our
# normal distribution *and* its corresponding line chart in the same
# line that adds this chart to our histogram. This saves a few lines
# of code but does make the output a bit less readable.)

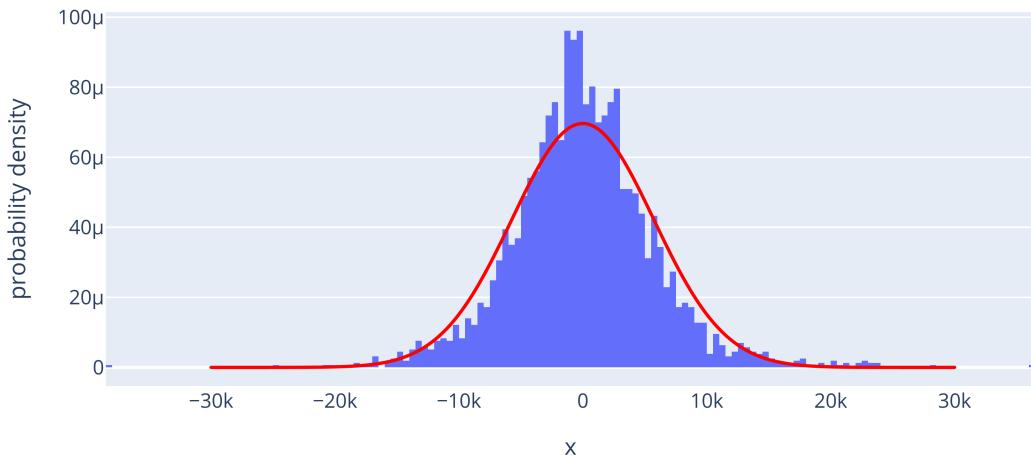
fig_cmib_hist.add_trace(
    px.line(gen_normdist(
        min = -30000, max = 30000, rows = 1000,
        mean = 0, stdev = np.std(cmib_lr_results_2.resid)),
        x = 'x', y = 'y', color_discrete_sequence = ['red'])['data'][0])
```

(continues on next page)

(continued from previous page)

```
wadi(fig_cmib_hist, 'Charts/cmib_residual',
      display_type=display_type)
```

Histogram of regression residuals compared to normal distribution



The central section of our histogram is notably higher than the corresponding normal distribution line, which suggests that our residuals are not normally distributed. The following `omni_normtest` agrees with this conclusion:

```
sms.omni_normtest(cmib_lr_results_2.resid)
```

```
NormaltestResult(statistic=np.float64(225.5648729751166), pvalue=np.float64(1.0452259364221645e-49))
```

13.4.4 Performing state-level regressions

We'll now perform a similar set of analyses at the state, rather than county, level.

```
df_smib = pd.read_csv(
    '../Census_Data_Imports/Datasets/education_and_earnings_state.csv')
# Limiting our output to counties in one of the 50 US states (plus DC):
df_smib.query("State_Abbrev not in ['PR']", inplace = True)
df_smib.drop(['Year', 'state'], axis = 1, inplace = True)

df_smib[['NAME', 'Median_Earnings_for_Total_25plus_Population',
          'Pct_With_Bachelors_Degree']].head()
```

	NAME	Median_Earnings_for_Total_25plus_Population
0	Alabama	45197
1	Alaska	56492
2	Arizona	49333

(continues on next page)

(continued from previous page)

3	Arkansas	42450
4	California	54437
0	Pct_With_Bachelors_Degree	27.755139
1		31.242276
2		32.610332
3		25.111375
4		36.503058

Confirming that no median earnings columns within our state-level dataset contain the invalid entries that we saw in our county-level file:

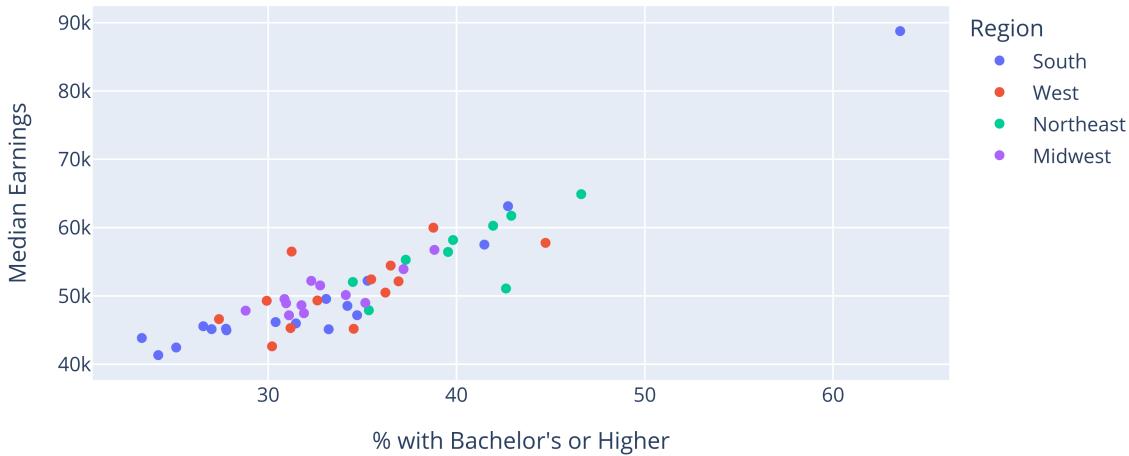
```
df_smib[median_earnings_cols].min()
```

Median_Earnings_for_Total_25plus_Population	41331
Median_Earnings_Less_Than_HS	26749
Median_Earnings_HS	33805
Median_Earnings_Some_College	38500
Median_Earnings_Bachelors_Degree	52657
Median_Earnings_Postgraduate	64563
dtype: int64	

Visualizing the relationship between bachelor's-degree prevalence and median earnings at the state level:

```
fig_smib_scatter = px.scatter(
    df_smib, x = "Pct_With_Bachelors_Degree",
    y = 'Median_Earnings_for_Total_25plus_Population',
    color = 'Region', hover_data = 'NAME',
    title = "Median County Earnings as Function of % of State \
Residents<br>With a Bachelor's Degree or Higher").update_layout(
    yaxis_title = 'Median Earnings', xaxis_title = "% with Bachelor's \
or Higher")
wadi(fig_smib_scatter, 'Charts/state_ba_pct_earnings_scatter',
    display_type = display_type)
```

Median County Earnings as Function of % of State Residents With a Bachelor's Degree or Higher



DC has a far higher median income *and* percentage of residents with bachelor's degrees than any state. Therefore, as it constitutes an outlier, I'll go ahead and remove it from our dataset (though one could also make a solid argument for keeping it in).

```
df_smib.query("NAME != 'District of Columbia'", inplace = True)
```

Here's an updated scatter plot without DC. This plot also shows a best fit line that helps us determine which states have higher-than-expected earnings given their bachelor's-degree stats—and which have lower earnings.

```
fig_smib_scatter = px.scatter(
    df_smib, x = "Pct_With_Bachelors_Degree",
    y = 'Median_Earnings_for_Total_25plus_Population',
    color = 'Region', hover_data = 'NAME',
    text = 'State_Abbrev', trendline = 'ols', trendline_scope = 'overall',
    title = "Median County Earnings as Function of % of State \
Residents<br>With a Bachelor's Degree or Higher").update_layout(
    xaxis_title = 'Median Earnings', yaxis_title = "% with Bachelor's \
or Higher").update_traces(
    textposition = 'bottom center')

# trendline_scope = 'overall' instructs Plotly to plot a single trendline
# rather than one for each region. This code was based on the example
# found at https://plotly.com/python/linear-fits/ .

# 'textposition = bottom center' argument allows us to move our dot labels
# just below their corresponding dots, thus making them a bit more
# readable.
# For more on scatter plot updates, see
# https://plotly.com/python/line-and-scatter/

wadi(fig_smib_scatter, 'Charts/state_ba_pct_earnings_scatter',
     display_type = display_type)
```



To me, the relationship between our independent and dependent variables looks quite linear; however, I ultimately found that a squared version of the 'Pct_With_Bachelors_Degree' field ended up having higher predictive power than the non-squared version. The following cell creates that squared copy.

```
df_smib['Pct_With_Bachelors_Degree_Squared'] = (df_smib['Pct_With_Bachelors_Degree']**2)
```

13.4.5 Performing a linear regression

First, we'll try building a regression model that includes bachelor's degree percentages; a squared version of this percentage; and our Region field.

```
smib_lr_3 = smf.ols(formula = "Median_Earnings_for_Total_\\
25plus_Population ~ Pct_With_Bachelors_Degree_Squared + \
Pct_With_Bachelors_Degree + Region",
                     data = df_smib)
smib_lr_results_3 = smib_lr_3.fit()
# smib_lr_results_1.summary()
Image('Regression_Screenshots/smib_lr_results_3_summary.png', width = 600)
```

OLS Regression Results							
Dep. Variable:	Median_Earnings_for_Total_25plus_Population				R-squared:	0.764	
Model:	OLS				Adj. R-squared:	0.737	
Method:	Least Squares				F-statistic:	28.47	
Date:	Fri, 21 Feb 2025				Prob (F-statistic):	9.27e-13	
Time:	00:47:28				Log-Likelihood:	-466.59	
No. Observations:	50				AIC:	945.2	
Df Residuals:	44				BIC:	956.7	
Df Model:	5						
Covariance Type:	nonrobust						
		coef	std err	t	P> t	[0.025	0.975]
	Intercept	4.782e+04	1.53e+04	3.134	0.003	1.71e+04	7.86e+04
	Region[T.Northeast]	-824.2885	1508.486	-0.546	0.588	-3864.442	2215.865
	Region[T.South]	-1632.7421	1198.674	-1.362	0.180	-4048.511	783.027
	Region[T.West]	-689.3360	1178.346	-0.585	0.562	-3064.136	1685.464
	Pct_With_Bachelors_Degree_Squared	22.4137	12.595	1.780	0.082	-2.969	47.796
	Pct_With_Bachelors_Degree	-670.3622	875.607	-0.766	0.448	-2435.032	1094.308
Omnibus:	4.800	Durbin-Watson:		2.124			
Prob(Omnibus):	0.091	Jarque-Bera (JB):		5.691			
Skew:	-0.110	Prob(JB):		0.0581			
Kurtosis:	4.638	Cond. No.		4.62e+04			

Notes:

- [1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
- [2] The condition number is large, 4.62e+04. This might indicate that there are strong multicollinearity or other numerical problems.

Only the intercept is statistically significant! Is our model doomed? Not at all—we just need to reduce the number of predictors. Let's see what happens when we remove the 'Pct_With_Bachelors_Degree' predictor, as its p-value is far higher than its squared version:

```
smib_lr_2 = smf.ols(formula = "Median_Earnings_for_Total_\\
25plus_Population ~ Pct_With_Bachelors_Degree_Squared + Region",
                     data = df_smib)
```

(continues on next page)

(continued from previous page)

```
smib_lr_results_2 = smib_lr_2.fit()
# smib_lr_results_2.summary()
Image('Regression_Screenshots/smib_lr_results_2_summary.png', width = 600)
```

OLS Regression Results									
Dep. Variable: Median_Earnings_for_Total_25plus_Population				R-squared:		0.761			
Model:				OLS		Adj. R-squared:			
Method:				Least Squares		F-statistic:			
Date:				Fri, 21 Feb 2025		Prob (F-statistic):			
Time:				00:47:28		Log-Likelihood:			
No. Observations:				50		AIC:			
Df Residuals:				45		BIC:			
Df Model:				4					
Covariance Type:				nonrobust					
		coef	std err	t	P> t 	[0.025	0.975]		
		Intercept	3.621e+04	1693.707	21.381	0.000	3.28e+04 3.96e+04		
		Region[T.Northeast]	-554.2037	1459.895	-0.380	0.706	-3494.584 2386.177		
		Region[T.South]	-1304.8976	1114.414	-1.171	0.248	-3549.443 939.648		
		Region[T.West]	-609.9461	1168.364	-0.522	0.604	-2963.153 1743.261		
Pct_With_Bachelors_Degree_Squared		12.8269	1.345	9.538	0.000	10.118	15.535		
Omnibus:		4.113	Durbin-Watson:		2.050				
Prob(Omnibus):		0.128	Jarque-Bera (JB):		4.268				
Skew:		-0.110	Prob(JB):		0.118				
Kurtosis:		4.414	Cond. No.		5.73e+03				

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 5.73e+03. This might indicate that there are strong multicollinearity or other numerical problems.

The Pct_With_Bachelors_Degree_Squared field is now a statistically significant predictor. However, all of the 'Region' categories are still insignificant, so I'll go ahead and remove them.

Keeping only the Pct_With_Bachelors_Degree_Squared predictor ended up increasing the adjusted R-squared of my regression. This metric is very useful when refining models, as it accounts for the number of independent variables that

that model uses. If you find that adding a certain independent variable causes your adjusted R-squared to decrease, you *may* want to remove it.

```
smib_lr_1 = smf.ols(formula = "Median_Earnings_for_Total_\\
25plus_Population ~ Pct_With_Bachelors_Degree_Squared",
    data = df_smib)
smib_lr_results_1 = smib_lr_1.fit()
# smib_lr_results_1.summary()
Image('Regression_Screenshots/smib_lr_results_1_summary.png', width = 600)
```

Next, we'll compare our model's predictions to actual median earnings.

```
df_smib_pred_vs_actual = df_smib.copy()[
    ['NAME', 'Median_Earnings_for_Total_25plus_Population',
```

(continues on next page)

(continued from previous page)

```
'Pct_With_Bachelors_Degree_Squared', 'Region', 'State_Abbrev']]  
df_smib_pred_vs_actual.insert(1, 'Pred_Val', smib_lr_results_1.fittedvalues)  
df_smib_pred_vs_actual.insert(1, 'Actual-Pred', smib_lr_results_1.resid)
```

Here are the five states that had the highest earnings relative to our model's predictions: (Alaska and North Dakota may appear here due to their oil and gas industry, which could provide higher-paying jobs for non-college grads.)

```
df_smib_pred_vs_actual.sort_values(  
    'Actual-Pred', ascending = False).head(5)
```

	NAME	Actual-Pred	Pred_Val	\
1	Alaska	8483.325552	48008.674448	
47	Washington	5096.231011	54890.768989	
20	Maryland	4021.565354	59111.434646	
34	North Dakota	3334.522903	48873.477097	
30	New Jersey	2429.788162	59306.211838	
				Median_Earnings_for_Total_25plus_Population \
1			56492	
47			59987	
20			63133	
34			52208	
30			61736	
				Pct_With_Bachelors_Degree_Squared Region State_Abbrev
1		976.079786	West	AK
47		1503.152620	West	WA
20		1826.396975	South	MD
34		1042.311655	Midwest	ND
30		1841.314202	Northeast	NJ

And here are the 5 states with the *lowest* earnings relative to our predictions. (The first three 3 states here are relatively rural, so it's possible that they have fewer high-paying jobs for bachelor's degree holders. Again, though, this is just a guess.)

```
df_smib_pred_vs_actual.sort_values(  
    'Actual-Pred').head(5)
```

	NAME	Actual-Pred	Pred_Val	\
45	Vermont	-7906.384818	58991.384818	
26	Montana	-5654.293537	50840.293537	
31	New Mexico	-4552.923887	47173.923887	
9	Florida	-4547.900210	49664.900210	
33	North Carolina	-3828.634015	51004.634015	
				Median_Earnings_for_Total_25plus_Population \
45			51085	
26			45186	
31			42621	
9			45117	
33			47176	
				Pct_With_Bachelors_Degree_Squared Region State_Abbrev
45		1817.202826	Northeast	VT

(continues on next page)

(continued from previous page)

26	1192.942470	West	MT
31	912.149489	West	NM
9	1102.923670	South	FL
33	1205.528668	South	NC

Finally, we'll compare each state's predicted earnings to its actual earnings; create a histogram of our residuals; and then apply statsmodels' `omni_normtest()` function to gauge the normality of those residuals.

```
fig_smib_pred_vs_actual = px.scatter(
    df_smib_pred_vs_actual, x = 'Pred_Val',
    y = 'Median_Earnings_for_Total_25plus_Population',
    color = 'Region', hover_data = 'NAME', text = 'State_Abbrev',
    title = "Predicted vs. Actual Median State Earnings",
    trendline = 'ols', trendline_scope = 'overall').update_layout(
    xaxis_title = 'Predicted Median Earnings',
    yaxis_title = 'Actual Median Earnings').update_traces(
    textposition = 'bottom center')

wadi(fig_smib_pred_vs_actual, 'Charts/smib_pred_vs_actual',
    display_type=display_type)
```

Predicted vs. Actual Median State Earnings



```
fig_smib_hist = px.histogram(
    x = smib_lr_results_1.resid,
    title = 'Histogram of regression residuals compared to \
normal distribution',
    histnorm = 'probability density')

fig_smib_hist.add_trace(
    px.line(gen_normdist(
        min = -10000, max = 10000, rows = 1000,
        mean = 0, stdev = np.std(smib_lr_results_1.resid))),
```

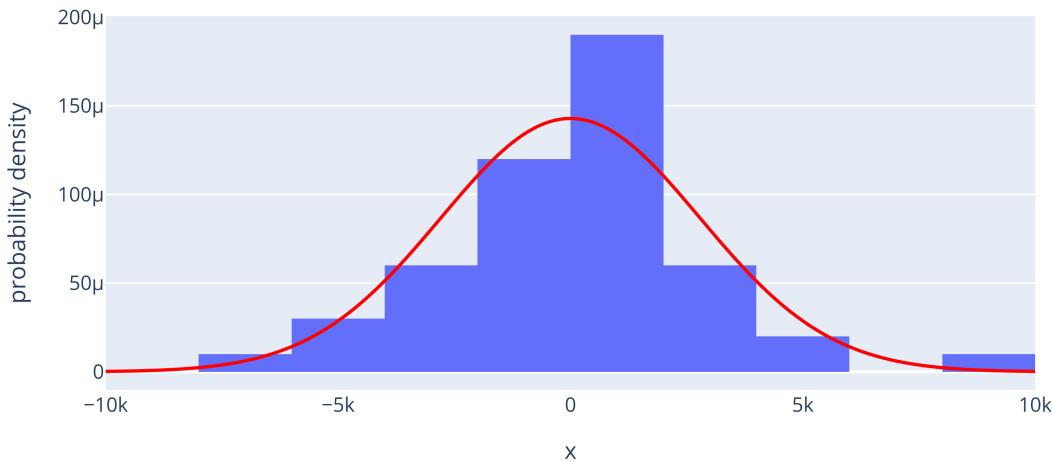
(continues on next page)

(continued from previous page)

```
x = 'x', y = 'y', color_discrete_sequence = ['red'])['data'][0])

wadi(fig_smib_hist, 'Charts/smib_residual',
    display_type=display_type)
```

Histogram of regression residuals compared to normal distribution



Although this histogram appears to diverge from our normal distribution also, our p-value from the following `omni_normtest()` is above 0.05. My guess is that this p-value is higher than that for our county-level `omni_normtest` simply because our sample size is smaller.

```
sms.omni_normtest(
    smib_lr_results_1.resid)
```

```
NormaltestResult(statistic=np.float64(3.6759444784110493), pvalue=np.float64(0.
˓→15913979659171434))
```

13.4.6 Comparing median earnings for different education levels

We have pretty strong evidence that, as the percentage of a county or state's population that has a bachelor's degree increases, the median income for that county also increases. However, that doesn't prove that bachelor's degree holders actually earn more than high-school-degree holders. Therefore, we'll now use descriptive statistics, visualizations, and linear regressions to compare earnings for different education levels directly.

First, we'll import national-level median earnings data for different education levels. (This data, like the state- and county-level data, was retrieved via Census Data Imports.)

```
df_nmib = pd.read_csv(
    '../Census_Data_Imports/Datasets/education_and_earnings_national.csv')
df_nmib[median_earnings_cols]
```

	Median_Earnings_for_Total_25plus_Population	Median_Earnings_Less_Than_HS	\
0		51184	31492
	Median_Earnings_HS	Median_Earnings_Some_College	\
0	38789	45840	
	Median_Earnings_Bachelors_Degree	Median_Earnings_Postgraduate	
0	67849	88719	

Using this national-level dataset, we can see that the median income for college graduates was nearly 30,000 higher than that for high school graduates:

```
df_nmib['Median_Earnings_Bachelors_Degree'] - df_nmib['Median_Earnings_HS']
```

```
0      29060
dtype: int64
```

We could stop here, but I think it's still worth investigating the differences between bachelor's and high-school-diploma earnings (and those for other education categories) at the county level. For instance, we might find that the earnings advantage that comes with a bachelor's degree is particularly high (or low) in certain parts of the country.

In order to compare these stats by county, we'll first create a filtered copy of our county-level earnings data that does not contain any rows with invalid median income values. (These rows could interfere with our regression analyses.)

```
df_cmib_ec = df_cmib.copy()
# ec = 'earnings_comparison'

# Removing all rows with negative median earnings values, which we'll
# assume are invalid (though negative earnings are indeed possible
# at the personal level)

for col in median_earnings_cols:
    df_cmib_ec.query(f'{col} >= 0', inplace = True)

# The following code creates a new set of columns that determines
# the percentage change in median earnings between certain educational
# attainment categories.

for df in [df_nmib, df_cmib_ec]:

    df['Pct_Median_Earnings_Increase_non_HS_to_HS'] = (
        100*(df['Median_Earnings_HS'] /
              df['Median_Earnings_Less_Than_HS'] - 1))

    df['Pct_Median_Earnings_Increase_HS_to_Bachelors'] = (
        100*(df['Median_Earnings_Bachelors_Degree'] /
              df['Median_Earnings_HS'] - 1))

    df['Pct_Median_Earnings_Increase_Bachelors_to_Postgraduate'] = (
        100*(df['Median_Earnings_Postgraduate'] /
              df['Median_Earnings_Bachelors_Degree'] - 1))

    df['Pct_Median_Earnings_Increase_HS_to_Postgraduate'] = (
        100*(df['Median_Earnings_Postgraduate'] /
              df['Median_Earnings_HS'] - 1))
```

We're now ready to compare earnings by education level at both the national and county level.

```
df_national_earnings_by_ed_level = df_nmib[median_earnings_cols].transpose()
).reset_index().rename(
    columns = {'index':'Category', 0:'Median Earnings'})

# The following code uses both str.replace() and replace() to
# convert the education levels in the Category field to a more concise
# and readable format.
df_national_earnings_by_ed_level['Education Level'] = (
    df_national_earnings_by_ed_level['Category'].str.replace(
        'Median_Earnings_', '').str.replace('_', ' ')).replace(
            {'for Total 25plus Population':'Total 25+ Population',
             'Bachelors Degree':"Bachelor's"})
df_national_earnings_by_ed_level
```

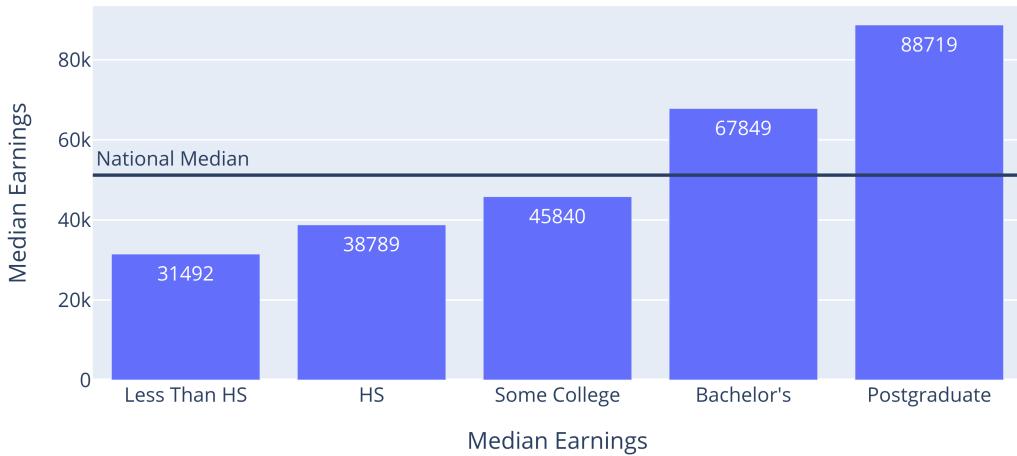
	Category	Median Earnings \
0	Median_Earnings_for_Total_25plus_Population	51184
1	Median_Earnings_LessThan_HS	31492
2	Median_Earnings_HS	38789
3	Median_Earnings_SomeCollege	45840
4	Median_Earnings_BachelorsDegree	67849
5	Median_Earnings_Postgraduate	88719
	Education Level	
0	Total 25+ Population	
1	Less Than HS	
2	HS	
3	Some College	
4	Bachelor's	
5	Postgraduate	

From this table alone, we can see stark differences in median earnings by education level. We can also compare these earnings using a bar chart:

```
fig_national_median_earnings = px.bar(
    df_national_earnings_by_ed_level.query(
        "Category != 'Median_Earnings_for_Total_25plus_Population'"),
    x = 'Education Level', y = 'Median Earnings', text_auto = '.0f',
    title = 'Median Earnings for 25+ Population by Education Level \
(According to<br> National ACS5 2023 Data)').update_layout(
    xaxis_title = 'Median Earnings')

# Adding a horizontal line to represent the national median income--along
# with an annotation to clarify to the reader that their monitor is not
# glitching out!
# https://plotly.com/python/horizontal-vertical-shapes/
fig_national_median_earnings.add_hline(
    y = df_national_earnings_by_ed_level.query("Category == \
'Median_Earnings_for_Total_25plus_Population").iloc[0][
    'Median Earnings'], annotation_text = 'National Median',
    annotation_position = 'top left')
wadi(fig_national_median_earnings, 'Charts/national_median_earnings',
    display_type=display_type)
```

Median Earnings for 25+ Population by Education Level (According to National ACS5 2023 Data)



Each of these educational categories is separated by thousands of dollars in median earnings. The largest gaps are between (1) the ‘Some College’ and ‘Bachelor’s’ categories and (2) the ‘Bachelor’s’ and ‘Postgraduate’ categories.

Even this data doesn’t *prove* that a bachelor’s degree will increase one’s earnings power. For instance, it’s possible that the people who choose to enroll in bachelor’s and postgraduate programs would still have earned more than other high school graduates if they never pursued higher education.

In order to make a particularly strong case for the financial benefits of a college degree, you would want to set up a randomized trial that helps some participants, but not others, graduate from college. (For instance, you could randomly assign college scholarships to half of a group of high-school graduates who weren’t otherwise planning to get a bachelor’s degree, then see if their earnings ultimately outpaced the other students in that group.) However, such a study is beyond the scope of this section!

Interestingly, when we evaluate median county-level earnings for each of our educational categories, we find *lower* values for higher education levels:

```
# Finding the median income for each education level within our county
# medians dataset (the medians of our medians, if you will):

df_county_earnings_by_ed_level = df_cmib_ec[median_earnings_cols].median(
).reset_index().rename(
    columns = {'index':'Category', 0:'Median Earnings'})

# I could have used the same replace()/str.replace() function shown
# within the national median earnings dataset to create our
# 'Education Level' column, but for comparison purposes, I'll instead
# use Series.map():

df_county_earnings_by_ed_level['Education Level'] = (
    df_county_earnings_by_ed_level['Category'].map(
        {'Median_Earnings_for_Total_25plus_Population':
            'Total 25+ Population',
        'Median_Earnings_Less_Than_HS':'Less Than HS',
        'Median_Earnings_HS':'HS',
        'Median_Earnings_Some_College':'Some College',
        'Median_Earnings_Bachelors':'Bachelor\'s',
        'Median_Earnings_Postgraduate':'Postgraduate'}))
```

(continues on next page)

(continued from previous page)

```
'Median_Earnings_Bachelors_Degree': "Bachelor's",
'Median_Earnings_Postgraduate': 'Postgraduate'
))

df_county_earnings_by_ed_level
```

	Category	Median Earnings \
0	Median_Earnings_for_Total_25plus_Population	43831.0
1	Median_Earnings_Less_Than_HS	31706.0
2	Median_Earnings_HS	37525.0
3	Median_Earnings_Some_College	42494.0
4	Median_Earnings_Bachelors_Degree	55859.0
5	Median_Earnings_Postgraduate	69258.0

	Education Level
0	Total 25+ Population
1	Less Than HS
2	HS
3	Some College
4	Bachelor's
5	Postgraduate

In order to compare national and county-level medians side by side, we can add both datasets to the same table (with a ‘Type’ column to help distinguish between them), then plot them within a grouped bar chart:

(I tried plotting two horizontal lines to reflect the county-level and national median earnings, but this caused the chart to appear cluttered. Therefore, I instead chose to represent these median earnings in bar form.)

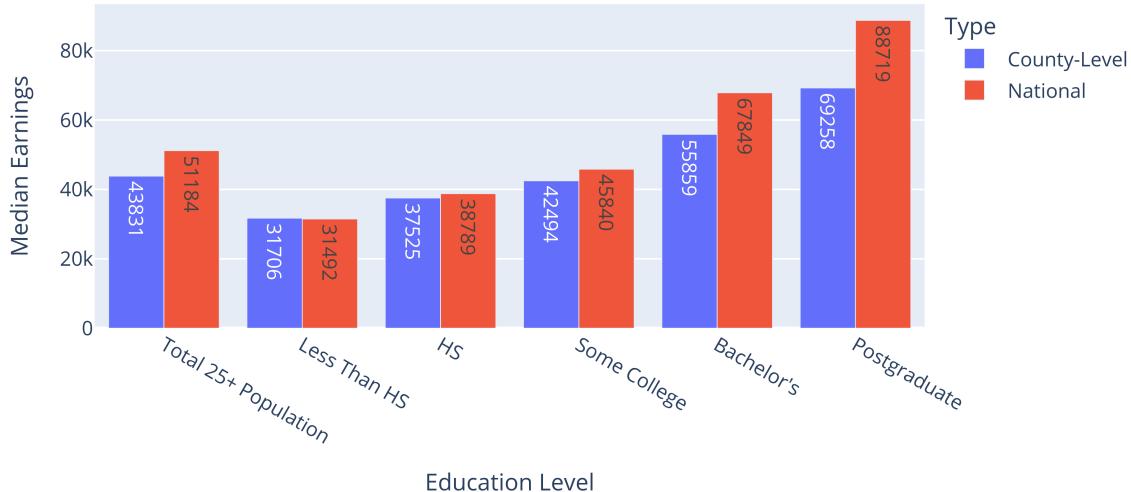
```
df_county_earnings_by_ed_level['Type'] = 'County-Level'
df_national_earnings_by_ed_level['Type'] = 'National'
df_county_vs_national_earnings_by_ed_level = pd.concat(
    [df_county_earnings_by_ed_level, df_national_earnings_by_ed_level])

# Note the use of <sub> in the following chart to create a subtitle.
# (sub actually stands for 'subscript,' not 'subtitle,' but it still
# works well for creating subtitles. You can also try <sup> as an
# alternative.)
# I may have first learned about this technique from this response
# by InaMelloMood on StackOverflow:
# https://stackoverflow.com/a/68641333/13097194
fig_county_vs_national_median_earnings = px.bar(
    df_county_vs_national_earnings_by_ed_level,
    x = 'Education Level', y = 'Median Earnings', text_auto = '.0f',
    color = 'Type',
    title = 'Median National and County-Level Earnings by Education \
Level<br><sub>(Source: ACS5 2023 Data)</sub>',
    barmode = 'group')

wadi(fig_county_vs_national_median_earnings,
      'Charts/county_vs_national_median_earnings',
      display_type=display_type)
```

Median National and County-Level Earnings by Education Level

(Source: ACS5 2023 Data)



For lower education levels, national median earnings come very close to the median county-level median earnings in our dataset. However, for higher education levels, the national median is considerably higher than the equivalent county-level median. This means that, while median bachelor's-degree earnings are 74.9% higher than high-school-diploma earnings at the national level, this difference goes down to 48.9% when we instead compare median county-level medians for these two groups. (See the following two cells for my percentage calculations.)

```
100 * (df_county_earnings_by_ed_level.query(
    "Category.str.contains('Bachelor's')").iloc[0][
    'Median Earnings'] / df_county_earnings_by_ed_level.query(
        ``Education Level`` == 'HS').iloc[0]['Median Earnings'] - 1)
```

```
np.float64(48.8580946035976)
```

```
100 * (df_national_earnings_by_ed_level.query(
    "Category.str.contains('Bachelor's')").iloc[0][
    'Median Earnings'] / df_national_earnings_by_ed_level.query(
        ``Education Level`` == 'HS').iloc[0]['Median Earnings'] - 1)
```

```
np.float64(74.91814689731623)
```

I find this discrepancy quite fascinating. One possible explanation is that people with bachelor's and postgraduate degrees tend to move to higher-earning counties after they graduate. These shifts would cause greater increases in our national-level medians, but they would have little or no impact within the median county-level medians shown in this graph.

13.4.7 Exploring county-level differences between median high-school-diploma and bachelor's degree earnings

First, let's find the counties (and county equivalents) with 50,000 or more residents in which bachelor's-degree holders have the highest earnings advantage, in percentage terms, relative to high-school graduates:

```
df_cmib_ec.query("Total_Population >= 50000").sort_values(
    'Pct_Median_Earnings_Increase_HS_to_Bachelors',
    ascending = False)[
    ['NAME', 'Pct_Median_Earnings_Increase_HS_to_Bachelors']].head(8)
```

	NAME \
2828	Arlington County, Virginia
224	San Francisco County, California
1860	New York County, New York
2917	Alexandria city, Virginia
321	District of Columbia, District of Columbia
229	Santa Clara County, California
2850	Fairfax County, Virginia
2874	Loudoun County, Virginia
	Pct_Median_Earnings_Increase_HS_to_Bachelors
2828	184.147394
224	169.921759
1860	166.556683
2917	161.109609
321	142.831086
229	135.489613
2850	132.574208
2874	130.743637

5 of these 8 regions are in the Northern Virginia/DC area. Thus, the NVCU admissions team can point out to prospective applicants that a college degree *may* prove particularly lucrative in the NoVA area, where NVCU is located.

Meanwhile, here are the regions with 50K+ residents that had the *lowest* estimated increase in median earnings from a high-school diploma to a bachelor's degree:

```
df_cmib_ec.query("Total_Population >= 50000").sort_values(
    'Pct_Median_Earnings_Increase_HS_to_Bachelors')[
    ['NAME', 'Pct_Median_Earnings_Increase_HS_to_Bachelors']].head(5)
```

	NAME \
1767	Carroll County, New Hampshire
1224	Franklin County, Massachusetts
1115	Acadia Parish, Louisiana
2592	Ector County, Texas
2049	Athens County, Ohio
	Pct_Median_Earnings_Increase_HS_to_Bachelors
1767	9.674226
1224	13.896043
1115	14.493175
2592	15.193989
2049	16.409321

Those who choose not to go to college may be particularly interested in the following table, which shows shows counties

with particularly strong median high-school-diploma earnings. My guess, as before, is that lucrative oil and/or natural gas extraction jobs explain many of these medians.

```
df_cmib_ec.sort_values('Median_Earnings_HS', ascending = False) [
    ['NAME', 'Total_Population', 'Median_Earnings_HS']].head(8)
```

	NAME	Total_Population	\
87	North Slope Borough, Alaska	10891	
2966	Garfield County, Washington	2326	
2772	Winkler County, Texas	7540	
265	Elbert County, Colorado	27152	
2018	McKenzie County, North Dakota	14280	
2001	Cavalier County, North Dakota	3663	
949	Meade County, Kansas	3949	
93	Southeast Fairbanks Census Area, Alaska	6936	
	Median_Earnings_HS		
87	88493		
2966	68646		
2772	67924		
265	62404		
2018	61058		
2001	60083		
949	59145		
93	59050		

Meanwhile, the counties with the highest median college-graduate earnings tend to be located near DC or in California:

```
df_cmib_ec.sort_values(
    'Median_Earnings_Bachelors_Degree', ascending = False) [
    ['NAME', 'Total_Population', 'Median_Earnings_Bachelors_Degree']].head(8)
```

	NAME	Total_Population	\
2874	Loudoun County, Virginia	427082	
224	San Francisco County, California	836321	
229	Santa Clara County, California	1903297	
75	Dillingham Census Area, Alaska	4780	
207	Marin County, California	258765	
227	San Mateo County, California	745100	
2828	Arlington County, Virginia	235463	
1465	Sharkey County, Mississippi	3848	
	Median_Earnings_Bachelors_Degree		
2874	110960		
224	109016		
229	108483		
75	105104		
207	101238		
227	100853		
2828	98781		
1465	97500		

Comparing median HS and bachelor's degree earnings: (counties above the regression line have higher-than-expected median bachelor's degree earnings relative to HS ones; for those below the line, the relationship is the opposite.)

You might also be wondering whether some correlation exists between median earnings for different education levels. For

instance, do high-school-degree holder in counties with higher median earnings for college grads *also* have higher-than-usual earnings?

We can explore this relationship via both a scatter plot and a regression. In order to do so, we'll first create a variant of df_cmib that includes only counties with 50K+ residents and excludes all rows with invalid median high-school or college-degree earnings:

```
df_cmib_hs_bachelors_earnings = df_cmib.query(
    "Median_Earnings_HS != -666666666 &
    Median_Earnings_Bachelors_Degree != -666666666 & \
    Total_Population >= 50000").copy()
```

As the following scatter plot shows, county-level high-school median earnings increase in a relatively linear fashion with college-degree earnings.

```
fig_hs_vs_bachelors_earnings = px.scatter(
    df_cmib_hs_bachelors_earnings,
    x = 'Median_Earnings_Bachelors_Degree',
    y = 'Median_Earnings_HS',
    trendline = 'ols', hover_data = 'NAME',
    title = "Median High-School-Degree Earnings by County as a Function \
    of<br>Median Bachelor's-Degree Earnings").update_layout(
    xaxis_title = "Median Bachelor's Earnings",
    yaxis_title = "Median HS Degree Earnings")

wadi(fig_hs_vs_bachelors_earnings, 'Charts/hs_bachelors_earnings_scatter',
    display_type=display_type)
```

Median High-School-Degree Earnings by County as a Function of Median Bachelor's-Degree Earnings



As the following regression shows, variation in bachelor's-degree earnings accounts for around 41% of the variation in high-school degree earnings among counties with 50,000 or more residents. (It's worth noting, though, that when I ran this regression on *all* counties with valid data, regardless of population, the R-squared decreased to 15.7%—and when I modified it to include *only* counties with fewer than 50,000 residents, it sank to just 8.4%).

This model's intercept predicts that a hypothetical county in which *no one*'s highest education level is a bachelor's degree

will have median high-school-diploma earnings of 22,070. Meanwhile, the bachelor's-degree coefficient reports that each dollar increase in earnings for college grads is associated with an increase in median income of around 27 cents for high-school-degree holders.

```
hs_vs_bachelors_lr_results = smf.ols(
    formula = "Median_Earnings_HS ~ Median_Earnings_Bachelors_Degree",
    data = df_cmib_hs_bachelors_earnings)
hs_vs_bachelors_lr_results = hs_vs_bachelors_lr_results.fit()
# hs_vs_bachelors_lr_results.summary()
Image('Regression_Screenshots/hs_vs_bachelors_lr_results_summary.png',
      width = 600)
```

OLS Regression Results											
Dep. Variable:	Median_Earnings_HS	R-squared:	0.412								
Model:	OLS	Adj. R-squared:	0.411 <th data-cs="4" data-kind="parent"></th> <th data-kind="ghost"></th> <th data-kind="ghost"></th> <th data-kind="ghost"></th>								
Method:	Least Squares	F-statistic:	697.4								
Date:	Fri, 21 Feb 2025	Prob (F-statistic):	5.94e-117								
Time:	00:47:32	Log-Likelihood:	-9560.2								
No. Observations:	999	AIC:	1.912e+04								
Df Residuals:	997	BIC:	1.913e+04								
Df Model:	1										
Covariance Type:	nonrobust										
		coef	std err	t	P> t	[0.025	0.975]				
	Intercept	2.207e+04	658.838	33.497	0.000	2.08e+04	2.34e+04				
Median_Earnings_Bachelors_Degree		0.2734	0.010	26.409	0.000	0.253	0.294				
Omnibus:	10.774	Durbin-Watson:	1.650								
Prob(Omnibus):	0.005	Jarque-Bera (JB):	16.259								
Skew:	0.035	Prob(JB):	0.000295								
Kurtosis:	3.621	Cond. No.	3.82e+05								

Notes:

- [1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
- [2] The condition number is large, 3.82e+05. This might indicate that there are strong multicollinearity or other numerical problems.

13.4.8 Exploring differences in median county-level earnings by education level

Earlier in this section, we saw that, on average, median county-level earnings for higher education levels outpace those for lower education levels. In order to figure out whether these differences are significant, we can implement—you guessed it!—a regression analysis.

In order to prepare for this regression, we'll use Pandas' powerful `melt()` function (<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.melt.html>) to create a modified version of our county-earnings DataFrame in which all education levels are stored within a single column. We can then treat that column as a single independent variable.

In the following `melt()` function call, `id_vars` specifies the identifying variable(s) that we wish to associate with each new row; in this case, we'll choose our county names column (along with our population column, which will allow us to more easily perform regressions on counties with particular population sizes). Passing our median earnings columns to `value_vars` will allow us to ‘melt’ them into a single column. `var_name` will store the new name of this melted column, and `value_name` will store the name of this column's corresponding values.

```
df_county_earnings_melt = df_cmib.melt(
    id_vars= ['NAME', 'Total_Population'], value_vars=[
        'Median_Earnings_Less_Than_HS',
        'Median_Earnings_HS',
        'Median_Earnings_Some_College',
        'Median_Earnings_Bachelors_Degree',
        'Median_Earnings_Postgraduate'], var_name = 'Education Level',
        value_name = 'Median Earnings').sort_values(
    'NAME').query("`Median Earnings` != -666666666).reset_index(drop=True)
df_county_earnings_melt['Education Level'] = (df_county_earnings_melt[
    'Education Level'].str.replace('Median_Earnings_', '')).str.replace(
    '_', ' ').str.replace('Bachelors',"Bachelor's").copy()

# Creating an integer-based sort column that can be used to sort
# graph entries more logically:
df_county_earnings_melt['Ed_Sort_Map'] = df_county_earnings_melt[
    'Education Level'].map({'Less Than HS':0,'HS':1,
                           'Some College':2,
                           "Bachelor's Degree":3,
                           'Postgraduate':4})
df_county_earnings_melt.sort_values('Ed_Sort_Map', inplace = True)
```

Here's a look at this ‘melted’ DataFrame for a single county:

```
df_county_earnings_melt.query(
    "NAME == 'Albemarle County, Virginia'")
```

	NAME	Total_Population	Education Level	\
148	Albemarle County, Virginia	113683	Less Than HS	
146	Albemarle County, Virginia	113683	HS	
144	Albemarle County, Virginia	113683	Some College	
147	Albemarle County, Virginia	113683	Bachelor's Degree	
145	Albemarle County, Virginia	113683	Postgraduate	
	Median Earnings	Ed_Sort_Map		
148	28087	0		
146	41402	1		
144	48340	2		
147	65503	3		
145	91670	4		

Compare this output to our original median earnings data within df_cmib. All median earnings for specific education levels have now been ‘melted’ into a single column; previously, they existed within separate columns.

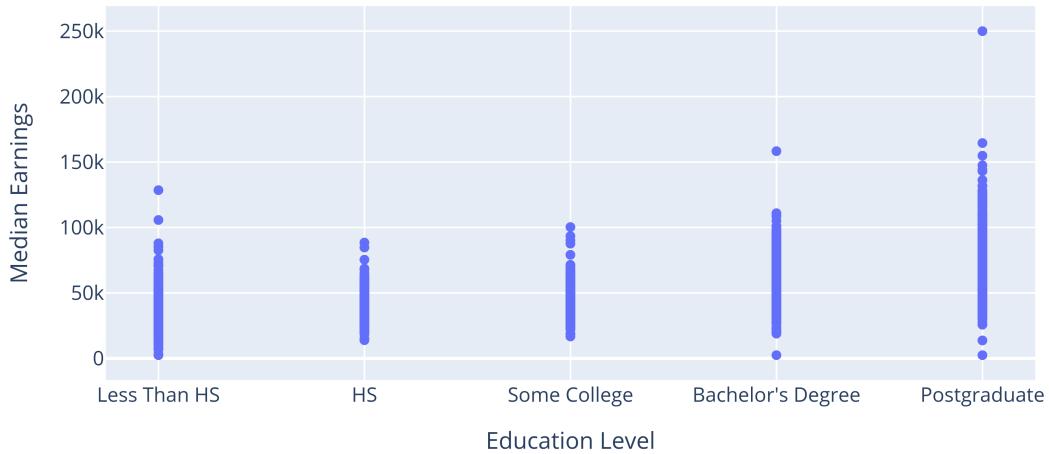
df_cmib.query("NAME == 'Albemarle County, Virginia'") [median_earnings_cols]
	Median_Earnings_for_Total_25plus_Population \
2823	61032
	Median_Earnings_Less_Than_HS Median_Earnings_HS \
2823	28087 41402
	Median_Earnings_Some_College Median_Earnings_Bachelors_Degree \
2823	48340 65503
	Median_Earnings_Postgraduate
2823	91670

13.4.9 Visualizing county medians by education level

We can try visualizing the effect of degree type on median earnings via a scatter plot, but since our categorical independent variable only has a few levels, this chart will be challenging to interpret:

```
fig_county_earnings_scatter = px.scatter(  
    df_county_earnings_melt,  
    x = 'Education Level', y = 'Median Earnings',  
    hover_data = 'NAME', title = "County-Level Median Earnings by \  
Education Level")  
wadi(fig_county_earnings_scatter,  
    'Charts/county_earnings_scatter',  
    display_type = display_type)
```

County-Level Median Earnings by Education Level

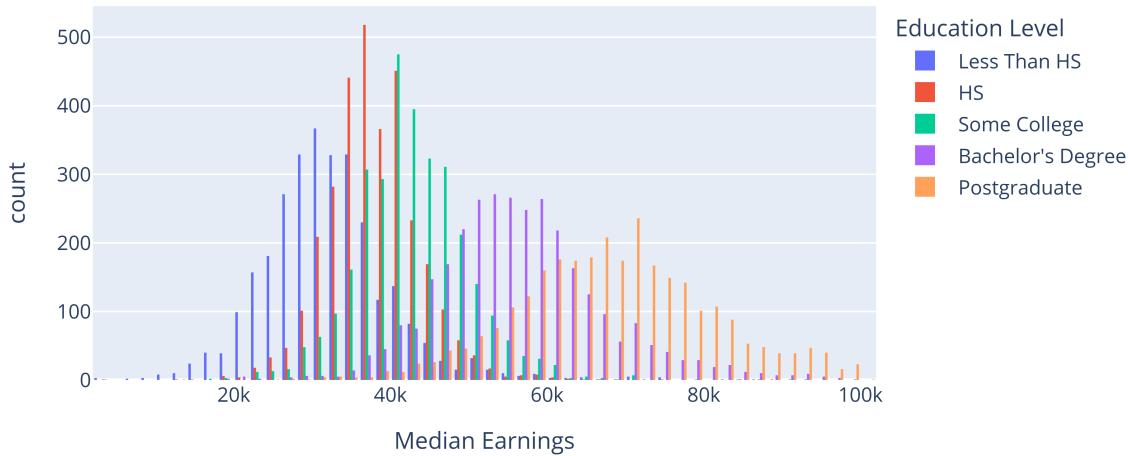


A better option will be to create a grouped histogram: (Note: to make the histogram a bit clearer, I'll filter out bars that represent median earnings above 100,000.)

```
fig_county_earnings_hist = px.histogram(
    df_county_earnings_melt.query("`Median Earnings` <= 100000"),
    x = 'Median Earnings', color = 'Education Level', barmode = 'group',
    title = "Distribution of County-Level Median Earnings by Education Level",
    nbins = 80)
# nbins helps control the level of detail featured in our histogram.
# For more on this variable, see
# https://plotly.com/python/histograms/

wadi(fig_county_earnings_hist,
     'Charts/hs_bachelors_earnings_hist', display_type = display_type)
```

Distribution of County-Level Median Earnings by Education Level



I'll admit that this isn't the clearest chart either; however, it does reveal notable differences in the shapes and locations of each education level's distribution. For instance, though there's some overlap between the two, the 'HS' category is far to the left of the 'Bachelor's Degree' category, which in turn is located to the left of the 'Postgraduate' category. In addition, the 'HS' bars appear more concentrated than do the 'Postgraduate' bars, which are quite spread out. (I imagine that this is because there are fewer postgraduate-degree holders in the US; as a result, their median county-level earnings may be more varied due to their smaller sample size.)

13.4.10 Evaluating median earnings differences via a regression model

We'll now use a linear regression to explore whether the differences in county-level median earnings between bachelor's degree holders and other education levels are statistically significant.

(Because the 'Median Earnings' field has a space in it, we need to use `Q()`, a function of the `patsy` library that helps power `statsmodels`' formula functionality, to correctly parse it. For more on this function, see <https://patsy.readthedocs.io/en/latest/builtins-reference.html#patsy.builtins.Q>)

```
county_ed_level_earnings_lr_1 = smf.ols(
    formula = "Q('Median Earnings') ~ Q('Education Level')",
    data = df_county_earnings_melt)
county_ed_level_earnings_lr_1_results = county_ed_level_earnings_lr_1.fit()
# county_ed_level_earnings_lr_1_results.summary()
Image('Regression_Screenshots/county_ed_level_earnings_lr_1_results_summary.png', width = 600)
```

OLS Regression Results													
Dep. Variable: Q('Median Earnings')		R-squared:		0.642									
Model:		OLS		Adj. R-squared:		0.642							
Method:		Least Squares		F-statistic:		6895.							
Date:		Fri, 21 Feb 2025		Prob (F-statistic):		0.00							
Time:		00:47:37		Log-Likelihood:		-1.6394e+05							
No. Observations:		15393		AIC:		3.279e+05							
Df Residuals:		15388		BIC:		3.279e+05							
Df Model:		4											
Covariance Type:		nonrobust											
		coef	std err	t	P> t	[0.025	0.975]						
	Intercept	5.656e+04	182.779	309.428	0.000	5.62e+04	5.69e+04						
	Q('Education Level')[T.HS]	-1.856e+04	258.405	-71.811	0.000	-1.91e+04	-1.8e+04						
	Q('Education Level')[T.Less Than HS]	-2.426e+04	262.206	-92.538	0.000	-2.48e+04	-2.37e+04						
	Q('Education Level')[T.Postgraduate]	1.376e+04	259.815	52.966	0.000	1.33e+04	1.43e+04						
	Q('Education Level')[T.Some College]	-1.372e+04	258.220	-53.133	0.000	-1.42e+04	-1.32e+04						
Omnibus:		5856.049		Durbin-Watson:		1.962							
Prob(Omnibus):		0.000		Jarque-Bera (JB):		100042.074							
Skew:		1.379		Prob(JB):		0.00							
Kurtosis:		15.181		Cond. No.		5.79							

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Note that this model compares each education level to one specific baseline, which in this case happens to be the median income for bachelor's-degree holders. Thus, each coefficient specifies how much higher, or lower, we can expect a given median income to be relative to that for college graduates. All of these coefficients are statistically significant (with p-values below 0.05); in addition, our R-squared value shows that roughly two thirds of the variation in median incomes can be explained by these education categories.

If you wanted to compare two other medians (such as those for non-high-school graduates and high school graduates), you could do so by creating a separate regression model whose data source includes only those two categories:

```
# Specifying which categories to compare:
levels_to_compare = ['Less Than HS', 'HS']
```

Running a regression that evaluates differences in median county-level incomes for non-HS grads and HS grads:

```

county_ed_level_earnings_lr_2 = smf.ols(
    formula = "Q('Median Earnings') ~ Q('Education Level')",
    data = df_county_earnings_melt.query(
        "`Education Level` in @levels_to_compare"))
county_ed_level_earnings_lr_2_results = county_ed_level_earnings_lr_2.fit()
# county_ed_level_earnings_lr_2_results.summary()
Image('Regression_Screenshots/county_ed_level_earnings_lr_2_\nresults_summary.png', width = 600)

```

OLS Regression Results											
Dep. Variable:	Q('Median Earnings')	R-squared:	0.131								
Model:	OLS	Adj. R-squared:	0.131								
Method:	Least Squares	F-statistic:	913.3								
Date:	Fri, 21 Feb 2025	Prob (F-statistic):	4.93e-187								
Time:	00:47:37	Log-Likelihood:	-62729.								
No. Observations:	6077	AIC:	1.255e+05								
Df Residuals:	6075	BIC:	1.255e+05								
Df Model:	1										
Covariance Type:	nonrobust										
		coef	std err	t	P> t	[0.025	0.975]				
	Intercept	3.8e+04	131.609	288.738	0.000	3.77e+04	3.83e+04				
Q('Education Level')[T.Less Than HS]	-5707.6208	188.862	-30.221	0.000	-6077.858	-5337.383					
Omnibus:	2296.657	Durbin-Watson:	1.976								
Prob(Omnibus):	0.000	Jarque-Bera (JB):	31482.606								
Skew:	1.422	Prob(JB):	0.00								
Kurtosis:	13.782	Cond. No.	2.59								

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

This model shows that median county-level earnings for high school graduates are around 5,700 higher than those who didn't graduate high school; this difference is statistically significant.

13.5 Part 3: A word of caution regarding significance testing

We've done quite a few regression models in this section; similarly, in your own research, you might end up throwing lots of different regressions, or other significance tests, into the same notebook. However, it's worth keeping in mind that, the more tests you perform, *the greater your chance of committing a Type I error*. (A 'Type I error' is one in which you mistakenly conclude a result to be statistically significant.)

The following cell provides a rather extreme example of the risk of performing too many regression analyses. It performs 1,000 linear regressions of 1,000 different datasets, each of which has randomly-generated values for its independent and dependent variables.

Because we're generating these variables at random, there shouldn't be any significant relationship between them. However, if we use a significance level of 0.05, there will, on average, be a 5% chance that we deem one of these randomly-generated patterns to be statistically significant—thus committing a Type I error.

```

rng = np.random.default_rng(seed = 214442)

pvals_list = [] # This list will store all of our p-values from our
# regressions.
df_list = []

for i in range(1000):
    # Creating a DataFrame with randomly-generated and
    # normally-distributed independent- and dependent-variable values:
    df_random = pd.DataFrame({'iv':rng.normal(
        loc = 50, scale = 5, size = 1000),
        'dv':rng.normal(loc = 50, scale = 5, size = 1000)})
    # Running a regression to evaluate the relationship between our
    # independent and dependent variables, then adding the p-value
    # to our list of results:
    pvals_list.append(
        smf.ols(formula = "dv ~ iv", data = df_random).fit().pvalues['iv'])
    df_list.append(df_random)

# Converting this list into a Series: (we can then use the Series's
# index values to find the DataFrame corresponding to a particular
# p-value.)
s_pvals = pd.Series(pvals_list)

```

Here's our list of p-values, sorted from lowest to highest:

```
s_pvals.sort_values()
```

878	0.000239
788	0.003854
786	0.003956
217	0.004219
	...
200	0.996843
327	0.996902
49	0.998139
74	0.999707

Length: 1000, dtype: float64

Our four smallest p-values are all below 0.005, or less than one tenth our normal significance level. These results would strike most researchers as 'significant,' but in this case, they were purely due to chance.

It turns out that we found a ‘significant’ result within around 4.5% of our datasets (a percentage that comes pretty close to our 5% significance level):

```
print(f"100* len([pval for pval in pvals_list if pval < 0.05])\n/\nlen(pvals_list)}% of our simulated regressions produced 'significant' \\\nresults, even though the data on which they were based was randomly \\ngenerated.")
```

4.6% of our simulated regressions produced ‘significant’ results, even though the data on which they were based was randomly generated.

In your own work, of course, you probably won’t be performing 1,000 regressions all at once. However, even if you’re including a more modest number of significance tests in your research, it won’t take long before you risk committing a Type I error.

The following code plots the likelihood that, in the course of performing 1 to 50 regressions, you will commit a Type I error:

```
test_count = list(range(1, 51))
t1_error_odds = [1 - (1-0.05)**test_num for test_num in test_count]

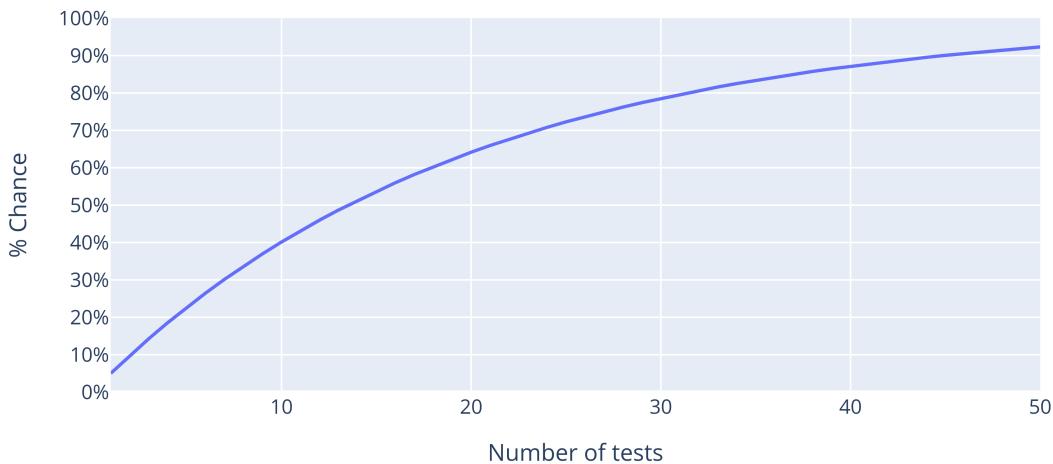
fig_t1_error = px.line(x = test_count, y = t1_error_odds,
title = 'Likelihood of \
finding a significant result (even with random data)<br><sub>\n(This graph assumes a 95% significance level)</sub>').update_layout(
    yaxis_tickformat = '.0%', xaxis_title = 'Number of tests',
    yaxis_title = '% Chance', yaxis_dtick = 0.1, yaxis_range = [0, 1])

# yaxis_dtick allows us to specify the spacing in between ticks;
# yaxis_range lets us determine the lower and upper boundaries of our chart.
# For more on these two options,
# see https://plotly.com/python/reference/layout/yaxis/ .
```

wadi(fig_t1_error, 'Charts/t1_error_odds', display_type=display_type)

Likelihood of finding a significant result (even with random data)

(This graph assumes a 95% significance level)



As the chart demonstrates, you won't need to complete too many tests before the risk of a Type I error becomes substantial. If you perform five significance tests, your odds of a Type I error will be around 23%; and should you complete 14 tests, your odds of making a Type I error will have risen above 50%:

```
# Odds of committing a Type I error after 5 and 14 tests:
100*(1-0.95**5), 100*(1-0.95**14)
```

(22.621906250000023, 51.23250208844705)

What can be done about this issue of multiple comparisons (described in more detail at https://en.wikipedia.org/wiki/Multiple_comparisons_problem)? One option is to adjust your definition of a significant p-value by dividing it by the number of tests you're performing. For instance, if you perform 5 tests, you might choose to use a p value of 0.01 (0.05 / 5) rather than 0.05; if you perform 10 tests, you might choose to use a p value of just 0.005. This is an example of the Bonferroni correction (https://en.wikipedia.org/wiki/Bonferroni_correction).

However, although this method is quite simple, I recommend looking into the Holm-Bonferroni method (https://en.wikipedia.org/wiki/Holm–Bonferroni_method) as an alternative, as this method helps reduce your likelihood of committing a Type II error (e.g. overlooking a result that actually *was* significant).

The following cell shows the likelihood of committing a Type I error if a Bonferroni correction has been applied.

```
test_count = list(range(1, 51))

# The following calculation divides 0.05 (our threshold for a significant
# p-value) by the number of tests being carried out, thus
# helping reduce the risk of a Type I error.

t1_bc_error_odds = [1 - ((1-0.05/test_num)**test_num)
                     for test_num in test_count]

fig_t1_error = px.line(x = test_count, y = t1_bc_error_odds,
                       title = 'Likelihood of \
finding a significant result (even with random data)<br><sub>\
```

(continues on next page)

(continued from previous page)

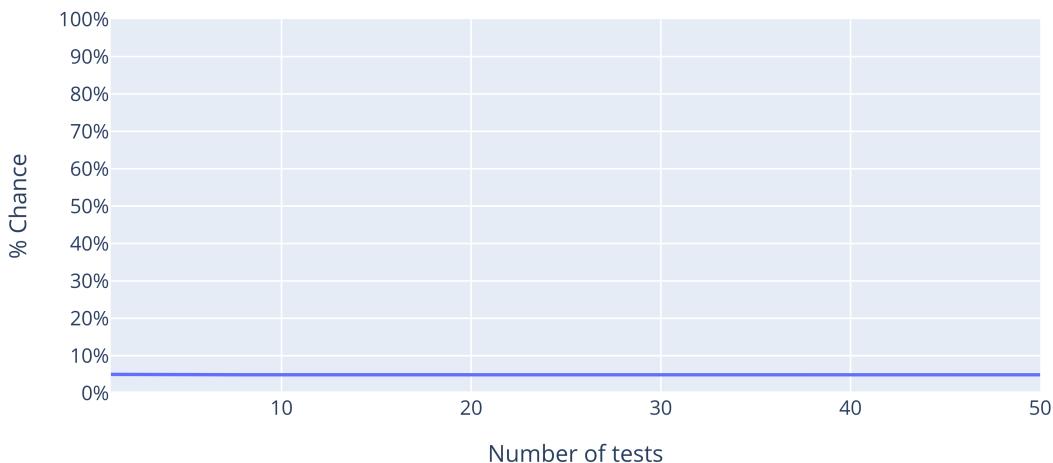
```
(This graph assumes a 95% significance level, but also applies \
a Bonferroni correction)').update_layout(
    yaxis_tickformat = '.0%', xaxis_title = 'Number of tests',
    yaxis_title = '% Chance', yaxis_dtick = 0.1, yaxis_range = [0, 1])

# yaxis_dtick allows us to specify the spacing in between ticks;
# yaxis_range lets us determine the lower and upper boundaries of our chart.
# For more on these two options,
# see https://plotly.com/python/reference/layout/yaxis/ .

wadi(fig_t1_error, 'Charts/t1_error_odds_bonferroni_correction',
    display_type=display_type)
```

Likelihood of finding a significant result (even with random data)

(This graph assumes a 95% significance level, but also applies a Bonferroni correction)



I do not mean to say here that you should always use a correction method to adjust your p-values within a given body of research. However, you might find that these tools help boost your confidence that a given p-value—or set of p-values—is statistically significant.

13.6 Conclusion

In this section, we explored linear regression models for both simulated and real-world data. These models incorporated both categorical variables (like college, region, and education level) and continuous variables (like fall bookstore sales and the percentage of county residents with a bachelor's degree). We also discussed the concept of statistical significance along the way.

There is much, much more that we could have covered within this section. For instance, logistic regressions (<https://www.statsmodels.org/stable/generated/statsmodels.formula.api.logit.html>) are a useful tool for evaluating datasets whose dependent variable is boolean (e.g. 1 or 0 or True or False). In addition, we didn't get into the subject of refining a given model to make it as accurate as possible—or the risk of 'overfitting' a model to your data. My hope, however, is that this section will inspire you to continue studying—and applying—regressions in your own work.

Part VI

Publishing Analyses Online

CREATING A SIMPLE STATIC WEBPAGE

One way to share visualizations and other content online is through a static website. The content in such websites is relatively fixed, unlike dynamic websites (such as the Dash apps we'll create within the Online Visualizations Section of Python for Nonprofits). If you have a set of content that doesn't need to be updated very regularly, if at all, a static site might be your best option.

For instance, we created some colorful net migration and population growth maps within the Mapping section of Python for Nonprofits. To share those maps with others, we could send them directly as HTML files; the users could then open those maps within a web browser. However, a simpler approach (at least for the end user) would be to share them online as a static website. We'll learn how to do so within this section.

Note: the static website created by this notebook can be found at https://kburchfiel.github.io/simple_static_site/. The repository that contains this website's contents is available at https://github.com/kburchfiel/simple_static_site.

14.1 Creating Static Sites with GitHub Pages

One method (among others) for hosting static content online for free is GitHub Pages (<https://pages.github.com/>). The GitHub pages documentation is quite robust, so I'll merely summarize the steps that I took. (Selecting the 'Project site' and 'Start from scratch' options will let you view the steps that most closely resembled my workflow,—though I did certain steps a bit differently.) However, my static website may serve as a helpful reference if you get stuck along the way.

(I recommend reviewing the GitHub Pages terms of service at <https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages#prohibited-uses> in order to make sure that this site is a good option for your static hosting needs, as certain uses are prohibited.)

14.1.1 Preparing local copies of the site's materials

First, I created a new repository (https://github.com/kburchfiel/simple_static_site) in which I could store my maps. Next, I created a corresponding 'simple_static_site' folder on my computer.

I then ran the following code in order to copy some (but not all) of the maps created within PFN's mapping section into this folder. (If you wish to run this code as well, you'll need to create this folder on your computer also, then add a 'maps' folder within it. It should be located within the same folder as your Python for Nonprofits folder.)

For more on the `shutil.copyfile()` function used below, see <https://docs.python.org/3/library/shutil.html>.

```
import shutil
# Defining the original location of these maps--and the location
# to which they'll be copied:
source_map_folder = '../Mapping/maps/' # This folder exists within
# the main Python for Nonprofits folder.
```

(continues on next page)

(continued from previous page)

```
dest_map_folder = '../../../../../simple_static_site/maps/' # This folder  
# is in a separate location from Python for Nonprofits  
# Specifying which maps within the source folder to copy:  
maps_to_copy = ['net_migration_rate_state_2020-2023.html',  
                 'net_migration_rate_county_2020-2023.html',  
                 'state_25-29_pop_pct_growth_2011-2021.html',  
                 'state_pop_pct_growth_2011-2021.html',  
                 'county_25-29_pop_pct_growth_2011-2021_tiled.html',  
                 'county_pop_pct_growth_2011-2021_tiled.html']  
for map_for_site in maps_to_copy: # Note that 'map' is a Python keyword,  
# so it's best not to use it as a variable name here.  
    shutil.copyfile(source_map_folder + map_for_site,  
                    dest_map_folder + map_for_site)
```

I could also have just copied and pasted these maps into my new folder, but this approach should save time in the long run if I happened to have lots of maps or made frequent updates to them. (Of course, if the updates were very frequent, a dynamic site could potentially be a better option than a static one.)

The maps will form the main content of our static site. However, we'll also need a landing page that can introduce the content and direct visitors to each page.

The following cell shows the Markdown-formatted text that I wrote for this landing page. (You can also use HTML for this page, but Markdown is a bit simpler. If you're not familiar with Markdown, check out Matt Cone's handy introduction at <https://www.markdownguide.org/basic-syntax/>.)

Note that the links to each map are relative, thus making them compatible within various repository names. (The links to Python for Nonprofits, in contrast, are absolute. Because the maps are located within a 'maps/' subfolder, I needed to reference that folder when creating these relative links.)

```
# Selected output from the mapping section of Python for Nonprofits
```

By Kenneth Burchfiel

Released under the MIT License

This simple webpage provides links to six maps created within the [Mapping] (<https://github.com/kburchfiel/pfn/tree/main/Mapping>) section of [Python for Nonprofits] (<https://github.com/kburchfiel/pfn>). It **is** meant to demonstrate how visualizations can be hosted within a static GitHub Pages website.

Python **for** Nonprofits, like this repository, **is** released under the MIT License.

```
## Population growth maps
```

These maps visualize population growth rates at the state **and** county level **from 2011 to 2021**. They were created using the Plotly graphing library within PFN's `choropleth_maps.ipynb` notebook, available at https://github.com/kburchfiel/pfn/blob/main/Mapping/choropleth_maps.ipynb.

Note: Certain maps will be easier to view on a laptop **or monitor than on your phone.**

(continues on next page)

(continued from previous page)

```

### Total population growth

[State-level total population growth] (maps/state_pop_pct_growth_2011-2021.html)

[County-level total population growth] (maps/county_pop_pct_growth_2011-2021_tiled.
↪html)

### Population growth for adults aged 25 to 29

[State-level 25-29 population growth] (maps/state_25-29_pop_pct_growth_2011-2021.html)

[County-level 25-29 population growth] (maps/county_25-29_pop_pct_growth_2011-2021_
↪tiled.html)

## Net domestic migration maps

These maps show net domestic migration rates at the state and
county level **from 2020 to 2023**. They are based on the Folium mapping
library and were created within PFN's
choropleth_maps_with_folium.ipynb notebook, available at
https://github.com/kburchfiel/pfn/blob/main/Mapping/choropleth\_maps\_with\_folium.ipynb
↪.

[State-level net migration] (maps/net_migration_rate_state_2020-2023.html)

[County-level net migration] (maps/net_migration_rate_county_2020-2023.html)

```

I enclosed the above text within a code block so that you could see the actual markdown code being used. To see how this text will actually appear on the website, simply visit https://kburchfiel.github.io/simple_static_site/.

I then saved this text into a new file within my simple_static_site folder called `index.md`. GitHub Pages will use this file to construct my site's landing page.

14.1.2 Publishing this static site to GitHub

I now have all the material I need for my site! My next step will be to export it to the GitHub repository that I created earlier. The steps for doing so via the command line are documented well on GitHub's website, so review that documentation (available at <https://docs.github.com/en/migrations/importing-source-code/using-the-command-line-to-import-source-code/adding-locally-hosted-code-to-github>) if you 'git' stuck.

(You can also upload files directly to your GitHub page if you prefer. That approach should work fine for a simple static page like this one, but for more complex projects, you might want to become accustomed to using the command line for Git and GitHub operations.)

After uploading the content to GitHub, I went into my repository's Settings page; navigated to Pages; and instructed GitHub to deploy my site from the root tab of my main branch. (The GitHub Pages documentation linked to earlier provides more details.)

Your site most likely will not appear right away; similarly, whenever you make updates to your online repository's contents, it may take a little while for those updates to manifest. However, after a minute or two, the GitHub Pages tab within the Settings page informed me that "Your site is live at https://kburchfiel.github.io/simple_static_site/." Great—my site is now live!

As you'll see when navigating to the above link, this site is *very* simple indeed. The landing page has essentially no colors and no images. The HTML-based map pages to which it links are colorful and interactive, but they don't offer any built-in

navigation menus; to revisit the landing page, you'll need to use your back button.

14.2 More advanced sites

The method for deploying a static site that this notebook features should work fine when you just need to make static content (like maps—or HTML versions of Jupyter Notebooks) available online. However, I thought I would also list some more robust alternatives to this approach in case it doesn't meet your own needs.

First, for more complex static sites, look into Jekyll (<https://jekyllrb.com/docs/>), the static site generator that GitHub Pages uses. (GitHub Pages-specific Jekyll instructions can be found at <https://docs.github.com/en/pages/setting-up-a-github-pages-site-with-jekyll>, but you'll likely find the Jekyll site to be helpful also.)

Jupyter Book (<https://jupyterbook.org/en/stable/intro.html>) is also worth looking into. Python for Nonprofits makes use of this tool to convert its many notebooks and markdown files into a PDF file that can serve as the basis for a printed book. This tool can *also* be used to generate static webpages, so consider using it if you too have a manual or book that you'd like to host online. (For more information on this functionality, see <https://jupyterbook.org/en/stable/web/index.html>.)

As of March 2025, GitHub Pages sites also can't be made private unless you have an Enterprise License. Therefore, if you need to limit whom can see your site, consider using Google Sites (<https://sites.google.com/>), a free tool that allows you to share your content with only specific users (or just yourself). You can add full-screen interactive maps (and other visualizations) to Google Sites Pages via the 'Full page embed' option.

Finally, you may determine that a purely static site doesn't meet your needs. In that case, stay tuned for the Online Visualizations section of Python for Nonprofits, where you'll learn how to create a dynamic site that responds to user input.

UPDATING ONLINE SPREADSHEETS

This script will demonstrate how to upload the contents of a DataFrame into a Google Sheets file using Python's `gspread` and `gspread-dataframe` libraries. This is a convenient option for sharing your output with others, especially if you need to update that output on a regular basis.

The Google Sheets worksheet that this script will update can be found at <https://docs.google.com/spreadsheets/d/17aDJ3mg49-n0IEnDgN7ZB85pO87fiUpkZPULYDB8dmo/edit?usp=sharing>.

Since the NWS weather data accessed by this script generally gets updated on an hourly basis, it makes sense to have this script run automatically every hour. That way, your Google Sheets data, along with your local .csv copies of historical weather data, will always be relatively up to date. To automate this script, I recommend saving it as a .py file (which is easy to do within JupyterLab Desktop); setting up a script that will run this file; and then having your computer run this script on an hourly basis. (More information on accomplishing these steps within Linux can be found at the end of this notebook.)

15.1 Prerequisites

1. Python's `gspread` library provides a very helpful overview of connecting to Google Sheets workbooks via a service account at <https://docs.gspread.org/en/latest/oauth2.html#for-bots-using-service-account>. (There are multiple ways to connect to workbooks, but I find the service account approach to be pretty straightforward.) Go ahead and complete these steps if you haven't already. (Note: I chose to save my service account key to a custom path rather than the default one specified in the documentation; that way, I could more easily work with multiple account keys on my computer.) Also make sure that, prior to enabling the Google Sheets and Google Drive APIs, the Google Cloud project that you want to use for this script has been selected.
2. If you also specified a custom path, you'll need to create a file called 'service_key_path.txt' within this folder that points to it. That way, the following cell will be able to read in its location for use within `gspread` functions. (Alternatively, you could simply replace the following cell's code with `service_key_path = (path_to_your_key)`. Or, if you're using the default key location, you can comment out this cell altogether.)
3. As noted in the `gspread` service account documentation, you'll need to give the email associated with your service account editor access to the Google Sheets workbook that you'd like your script to update. (This email is *not* the same as your regular Google Email; it will likely look something like `account-name@cloudprojectname.iam.gserviceaccount.com`. You can find it within your service account file.)

The following cell checks whether this script is running on my own laptop. (If yours happens to have the name 'kjb3-lm', which seems very unlikely, it will run on that machine also!) If so, it copies a new set of weather data from a Google Drive folder; if not, this task will be skipped.

For an explanation of why I added in this cell, reference the 'Updating my laptop's copy of the files in `weather_data` with my server's copy' section of this notebook's appendix.

```
import platform
if platform.node() == 'kjb3-lm':
    # The use of platform.node() here to check whether
    # the script is running on my computer came from
    # Eric Palakovich Carr at
    # https://stackoverflow.com/a/799799/13097194 .
    # If you have a program running on multiple
    # computers, but need to have different computers
    # perform different steps at times, consider
    # having your code check the output of
    # platform.node(), then respond to this output
    # as needed.
    print("Copying the latest weather_data contents \
from the server.")
    import shutil
    shutil.copytree('/home/kjb3/kjb3server_drive/weather_data',
                   'weather_data', dirs_exist_ok=True)
```

Copying the latest weather_data contents from the server.

```
import sys
sys.path.insert(1, '../Appendix')
from helper_funcs import config_notebook, wadi
display_type = config_notebook(display_max_columns = 5)
# Specifying which columns to render within the output:
display_cols = ['Station', 'Date/Time', 'Temp',
                 '1-Hour Precip', 'Rolling 24-Hour Precip']

with open('service_key_path.txt') as file:
    service_key_path = file.read()
```

```
import gspread
from gspread_dataframe import set_with_dataframe, get_as_dataframe
# From https://pypi.org/project/gspread-dataframe/
# gspread_dataframe isn't available within conda-forge as of 2025-02-20;
# therefore, you'll need to install it via the following pip command:
# pip install gspread-dataframe
```

(If you're using gspread's default path, you can comment out the first line and then uncomment the following one.)

```
gc = gspread.service_account(filename=service_key_path)

# gc = gspread.service_account()

# (This code comes from
# # From https://docs.gspread.org/en/latest/index.html)
```

Importing additional libraries:

(Note: the weather_import.py file imported below derives from recent_weather_data.ipynb within the Automated_Notebooks section of Python for Nonprofits.)

```
import pandas as pd
from weather_import import weather_import
```

15.2 Importing weather data

The following cells will import weather data for three Virginia weather stations; combine it with pre-existing data; and then save a revised copy of this table to a .csv file.

```
data_folder = 'weather_data'
```

15.2.1 Importing weather data for Charlottesville

```
print("Downloading KCHO data.")
```

Downloading KCHO data.

```
weather_import(  
    station_code = 'KCHO',  
    data_folder = data_folder)
```

Original length of historical data file: 3415
New length of historical data file: 3415

15.2.2 Importing weather data for Dulles International Airport

```
print("Downloading KIAD data.")
```

Downloading KIAD data.

```
weather_import(  
    station_code = 'KIAD',  
    data_folder = data_folder)
```

Original length of historical data file: 3000
New length of historical data file: 3001

15.2.3 Importing weather data for Winchester

(This data appears to be recorded at 20-minute intervals rather than hourly ones.)

```
print("Downloading KOKV data.")
```

Downloading KOKV data.

```
test_df = weather_import(  
    station_code = 'KOKV',  
    data_folder = data_folder)  
test_df
```

```
Original length of historical data file: 2965  
New length of historical data file: 2965
```

15.3 Reading these datasets into DataFrames

```
df_weather_kcho = pd.read_csv(  
    data_folder+'/'+'KCHO'+'_'  
    +'historical_hourly_data_updated.csv')  
df_weather_kcho[display_cols].tail()
```

	Station	Date/Time	Temp	1-Hour Precip	Rolling 24-Hour	Precip
3410	KCHO	2025-03-25 18:53	54.0	0.0		0.0
3411	KCHO	2025-03-25 19:53	52.0	0.0		0.0
3412	KCHO	2025-03-25 20:53	51.1	0.0		0.0
3413	KCHO	2025-03-25 21:53	48.9	0.0		0.0
3414	KCHO	2025-03-25 22:53	48.0	0.0		0.0

```
df_weather_kiad = pd.read_csv(  
    data_folder+'/'+'KIAD'+'_'  
    +'historical_hourly_data_updated.csv')  
df_weather_kiad[display_cols].tail()
```

	Station	Date/Time	Temp	1-Hour Precip	Rolling 24-Hour	Precip
2996	KIAD	2025-03-25 18:52	57.0	0.0		0.0
2997	KIAD	2025-03-25 19:52	55.0	0.0		0.0
2998	KIAD	2025-03-25 20:52	53.1	0.0		0.0
2999	KIAD	2025-03-25 21:52	52.0	0.0		0.0
3000	KIAD	2025-03-25 22:52	51.1	0.0		0.0

```
df_weather_kokv = pd.read_csv(  
    data_folder+'/'+'KOKV'+'_'  
    +'historical_hourly_data_updated.csv')  
df_weather_kokv[display_cols].tail()
```

	Station	Date/Time	Temp	1-Hour Precip	Rolling 24-Hour	Precip
2960	KOKV	2025-03-25 18:55	51.8	0.0		0.0
2961	KOKV	2025-03-25 19:55	50.0	0.0		0.0
2962	KOKV	2025-03-25 20:55	48.2	0.0		0.0
2963	KOKV	2025-03-25 21:55	46.4	0.0		0.0
2964	KOKV	2025-03-25 22:55	44.6	0.0		0.0

15.4 Importing these DataFrames into a Google Sheets workbook

In order to export these datasets to a Google Sheets workbook, we'll first need to open that workbook with gspread. There are a few ways to do this (see <https://docs.gspread.org/en/latest/user-guide.html#opening-a-spreadsheet> for reference), but I like the `open_by_key()` option, which allows you—as the function's name suggests—to open a workbook using its key.

These keys are located within the center of each workbook URL. For instance, the full URL of the workbook I'll be updating is <https://docs.google.com/spreadsheets/d/17aDJ3mg49-n0IEnDgN7ZB85p087fiUpkZPULYDB8dmo/edit?usp=sharing>, so the key—located in between the `/d/` component of that URL and the following `/-is` `17aDJ3mg49-n0IEnDgN7ZB85p087fiUpkZPULYDB8dmo`.

```
wb = gc.open_by_key('17aDJ3mg49-n0IEnDgN7ZB85p087fiUpkZPULYDB8dmo')
# Based on
# https://docs.gspread.org/en/latest/user-guide.html#opening-a-spreadsheet
wb
```

```
<Spreadsheet 'Hourly VA Weather Data' id:17aDJ3mg49-
n0IEnDgN7ZB85p087fiUpkZPULYDB8dmo>
```

Next, I'll select the 'KCHO' worksheet within this workbook, as that's the first one I'd like to update. I'll also clear out the current contents using `ws.clear()`; that way, only the latest DataFrame contents will appear within the spreadsheet after I call `set_with_dataframe` below. (If the most recent DataFrame is smaller than the previous version, parts of the previous one would still appear unless `ws.clear()` is called.)

```
ws = wb.worksheet('KCHO')
# https://docs.gspread.org/en/latest/user-guide.html#opening-a-spreadsheet
```

```
ws.clear()
ws
```

```
<Worksheet 'KCHO' id:0>
```

Finally, I'll call `set_with_dataframe` to export this DataFrame to `df_weather`.

```
# Only the most recent 960 rows (representing 40 days' worth of data if
# no entries were missing) will get exported to Google Sheets. This will
# limit the time (and potentially memory) needed to import this data
# into a Dash app (https://github.com/kburchfiel/pfn/tree/main/Online\_Visualizations/Simple\_App\_Without\_Login)
# that utilizes it.
set_with_dataframe(ws, df_weather_kcho.iloc[-960:])
# Source: https://pypi.org/project/gspread-dataframe/
```

In order to confirm that this upload was successful, we can call `get_as_dataframe` to import the contents of the worksheet into a new DataFrame:

```
df_weather_from_ws = get_as_dataframe(ws)
df_weather_from_ws[display_cols].tail()
```

	Station	Date/Time	Temp	1-Hour Precip	Rolling 24-Hour Precip
955	KCHO	2025-03-25 18:53	54.0	0.0	0.0
956	KCHO	2025-03-25 19:53	52.0	0.0	0.0
957	KCHO	2025-03-25 20:53	51.1	0.0	0.0
958	KCHO	2025-03-25 21:53	48.9	0.0	0.0
959	KCHO	2025-03-25 22:53	48.0	0.0	0.0

15.4.1 Performing the same data export steps for KIAD and KOKV data

```
ws = wb.worksheet('KIAD')
ws.clear()
set_with_dataframe(ws, df_weather_kiad.iloc[-960:])

ws = wb.worksheet('KOKV')
ws.clear()
set_with_dataframe(ws, df_weather_kokv.iloc[-960:])
```

15.5 Appendix

15.5.1 A shell script and crontab entry for running this notebook automatically

(These steps were written for Linux environments, but Windows also supports automated script operation; you'd just need to write a batch script rather than a shell script and use Windows Task Scheduler instead of your cron editor.)

Your computer is more than happy to run a .py equivalent of this script, rain or shine, every hour of the day (as long as it's powered on, of course). How can you tell it to do so? First, you'll need to create a shell script that activates your Python environment; navigates to the folder containing the .py version of this notebook*; and then runs that file.

The following cell shows what this script looks like on my 'server,' which is really just an old laptop that I keep powered on 24/7 in order to run these hourly weather imports. (I gave the script the imaginative name *online_spreadsheet_update.sh*.)

This script also copies the weather_data files that it creates to a Google Drive folder; this makes it easier to transfer them to my main laptop and also keeps the files backed up. (Since this server, like my main laptop, runs Linux, I needed to mount the drive via RClone so that it could be accessed via my local file system; otherwise, the cp command would not have worked.)

Steps for configuring RClone to connect to a Google Drive account are available at <https://rclone.org/drive/>. Also, in case it helps, my shell script for mounting my server's Google Drive account to my main laptop is as follows:

```
rclone mount kjb3server: ~/kjb3server_drive/ --vfs-cache-mode full (This code was
based on https://ostechnix.com/install-rclone-in-linux/ and Andre's comment on it. --vfs-cache-mode full
allows RClone to access files more quickly.)
```

**(To create a .py version of this notebook, open it within JupyterLab Desktop; navigate to File -> Save and Export Notebook As -> Executable Script; and then save it (preferably within the same folder as updating_online_spreadsheets.ipynb) as updating_online_spreadsheets.py. You could also just run the .ipynb file directly if you'd prefer; see the comments within the following script for more details.)*

```
#!/bin/bash

# It appears that the line above needs to be the first entry within this script.
```

(continues on next page)

(continued from previous page)

```

# For a discussion of the above line,
# see: https://stackoverflow.com/questions/8967902/why-do-you-need-to-put-bin-bash-at-
# the-beginning-of-a-script-file

echo "Activating Python environment:"

# Activating my custom Python for Nonprofits environment:
# (You may be able to delete the following two lines if you're planning
# to execute the notebook within your base environment.)
# These lines are based on Lamma's post at:
# https://stackoverflow.com/a/60523131/13097194

source ~/miniforge3/etc/profile.d/conda.sh
conda activate kjb3server

# Navigating to the folder that hosts this script:

cd /home/kjb3lxms/kjb3python/Updating_Online_Spreadsheets

# Executing the Python script:

python updating_online_spreadsheets.py

# Copying weather_data folder over to the KJB3Server Google Drive folder for backup_
# and transfer purposes:

cp -r weather_data /home/kjb3lxms/kjb3server_drive/

echo "Finished running script."
sleep 2

```

In order to instruct your computer to run this script 10 minutes after each hour, you can then run `crontab -e` within your terminal and paste the following line at the bottom of the page: (You'll of course need to replace my path with your own path to this file.)

(If you're new to crontabs, make sure to review the documentation that appears at the top of your crontab file. Also, if you're working within the Nano editor, make sure to hit Ctrl + X to exit out of the editor after your updates have been saved; if you instead close the window, your changes won't be saved.)

```
10 * * * * /home/kjb3lxms/kjb3python/Updating_Online_Spreadsheets/online_spreadsheet_
update.sh
```

15.5.2 Updating my laptop's copy of the files in `weather_data` with my server's copy

As noted earlier in the appendix, my server runs a Python version of this script each hour, thus keeping my Google Sheets copy of recent weather data up to date. However, when I run this notebook on my main laptop, this comprehensive Google Sheets copy will get overwritten by a more spotty copy of weather data on my local computer. (Since the script can only receive the most recent 3 days of weather data from the NWS, my local `recent_weather` folder will likely be missing quite a few hours of data unless I run this script once every 72 hours.)

In order to prevent this issue, I added code at the start of this notebook that would copy the latest contents of the `weather_data` folder within my server to the `weather_data` folder within Python for Nonprofits. That way, this script would have access to the same comprehensive weather data file that the server does—and, as a result, would load a more comprehensive set of data to Google Sheets.

In order for this code to work successfully, I also needed to run a shell script via Linux Mint’s Startup Applications program that would mount my server’s Google Drive folder to my main laptop. Having this folder accessible both on my server and my main laptop made the process of copying weather data from the former to the latter much easier.

15.6 weather_import.py

```
# Weather Data Import
# By Kenneth Burchfiel
# Released under the MIT license

import pandas as pd
import os
import numpy as np
from datetime import datetime, timedelta

def weather_import(station_code, data_folder = ''):
    '''This function retrieves NWS hourly weather data for the last
    3 days for the station specified in station_code; adds it to
    pre-existing data (if any); and then saves this data to
    the folder specified in data_folder.
    (Specifying a data folder is optional; if none is specified, files
    will simply be saved to the current working directory.)'''
    if len(data_folder) > 0:
        post_folder_char = '/'
    else:
        post_folder_char = ''
    today = datetime.today().date()
    retrieval_date = str(today) # The date that the current set of 3-day
    # weather history data is being retrieved. Note that this date won't
    # match the dates for earlier records within this dataset.
    # Importing the latest set of hourly observations from the
    # National Weather Service:

    # read_html returns a list of tables (even though only one table is
    # currently present on this site), so we'll use [0] to access that
    # table.
    # The final 3 rows are simply a repetition of the header, so I added in
    # [-3:] to exclude them from the DataFrame.
    # header = 2 specifies that the third row in the DataFrame should be
    # used as a header. (The first two rows' values are mostly duplicates
    # of this row's data.)

    df_3day_data = pd.read_html(
        f'https://forecast.weather.gov/data/obhistory/{station_code}.html',
        header = 2)[0][-3:]
    df_3day_data.tail()

    # The observation time column within this dataset shows the time zone;
    # this will cause issues when daylight savings time begins or ends
    # (as those events will cause the column name to change). To prevent
    # this from causing any issues when combining our latest data with
    # our historical dataset, we'll add this time zone data to a
    # separate column, then rename the original time column to 'Time.'
```

(continues on next page)

(continued from previous page)

```

original_time_col = [
    column for column in df_3day_data.columns if 'Time' in column][0]

tz = original_time_col.split(' ')[1].split(':')[0].upper()

df_3day_data.rename(columns={original_time_col:'Time', 'Date':'Day'},
                     inplace = True)
df_3day_data.insert(2, 'Time Zone', tz)
df_3day_data['Day'] = df_3day_data['Day'].astype('int')
df_3day_data['Data Retrieval Date'] = pd.to_datetime(
    datetime.today().date())

## Determining the year and month of each observation:

# The original NWS dataset only shows the day of the month for each row
# (which is understandable, since it only contains data for the most
# recent 72 hours; thus, the year and month can easily be inferred).
# However, because we'll be keeping a historical copy of this data,
# calculating each observation's corresponding year and month will
# be crucial for avoiding ambiguous results.

# We'll use the following approach to generate a YYYY-MM-DD-formatted
# 'Date' column for each observation:

# 1. We'll use the year and month found within our Data Retrieval Date
# column as a basis for our observation years and months.

# 2. If the observation day is less than the day within our Data
# Retrieval Date column, we'll assume that that observation took place
# within the current month; otherwise, we'll assume it took place
# during the previous month and thus reduce the observation month
# by 1. (For instance, an observation day of 5 and a data retrieval
# day of 6 would indicate that the observation and data retrieval
# months are the same; meanwhile, an observation day of 31 and a data
# retrieval day of 2 demonstrates that the observation took place
# during the previous month.)

# 3. The previous step will result in an observation month of 0 if
# data for December were retrieved in January. In this case, we'll
# decrease the observation year by 1 and switch the observation
# month to 12. Otherwise, we'll use the data retrieval year
# as our observation year.

# 4. Finally, we'll add these observation month and years to the
# pre-existing observation day column in order to produce a new
# 'Date' column in YYYY-MM-DD format.

df_3day_data['Data Retrieval Date'] = pd.to_datetime(
    df_3day_data['Data Retrieval Date'])

df_3day_data['Obs_Month'] = np.where(
    df_3day_data['Day'] <= df_3day_data['Data Retrieval Date'].dt.day,
    df_3day_data['Data Retrieval Date'].dt.month,
    df_3day_data['Data Retrieval Date'].dt.month -1)

df_3day_data['Obs_Year'] = np.where(

```

(continues on next page)

(continued from previous page)

```
df_3day_data['Obs_Month'] != 0,
df_3day_data['Data Retrieval Date'].dt.year,
df_3day_data['Data Retrieval Date'].dt.year-1)

df_3day_data['Obs_Month'] = df_3day_data['Obs_Month'].replace(
    0, 12).copy()

# Creating our Date column:
# Note the use of str.zfill() to add a leading 0 to single-digit
# months and dates (which will make it easier to sort them in
# chronological order).
df_3day_data.insert(0, 'Date', (
    df_3day_data['Obs_Year'].astype('str') + '-' +
    df_3day_data['Obs_Month'].astype('str').str.zfill(2) + '-' +
    df_3day_data['Day'].astype('str').str.zfill(2)))

# Sorting the dataset chronologically:
df_3day_data.sort_values(['Date', 'Time'], inplace = True)
df_3day_data

# Saving this data to a .csv file:
df_3day_data.to_csv(
    f'{data_folder}{post_folder_char}{station_code}\\
_most_recent_3_day_data.csv',
    index = False)

# Appending new data within this file to our historical dataset:

# Creating a historical copy of the 3-day dataset for the weather
# station being evaluated if one does not exist already:
# Determining which argument to pass to os.listdir(): (This argument
# should be None if no data folder was provided so as not to raise
# an error.)
if len(data_folder) > 0:
    listdir_arg = data_folder
else:
    listdir_arg = None
if f'{station_code}_historical_hourly_data.csv' not in os.listdir(
    listdir_arg):

    print("Historical copy of this dataset doesn't yet exist. \
Initializing it as a copy of the 3-day dataset.")
    df_3day_data.to_csv(
        f'{data_folder}{post_folder_char}{station_code}\\
_historical_hourly_data.csv', index = False)

df_historical_data = pd.read_csv(
    f'{data_folder}{post_folder_char}\\
{station_code}_historical_hourly_data.csv')
print("Original length of historical data file:",
    len(df_historical_data))

# Recreating df_3day_data by importing the .csv copy of the table
# that we just created:
```

(continues on next page)

(continued from previous page)

```

# (This step may appear unnecessary, but it does help ensure that
# both this data and that found in df_historical_data will use
# the same data types.)

df_3day_data = pd.read_csv(f'{data_folder}{post_folder_char}\\
{station_code}_most_recent_3_day_data.csv')

# Combining our previous historical data with our latest dataset
# from the last 3 days:

df_wx = pd.concat([df_historical_data, df_3day_data])
# Storing the station code within the DataFrame:
df_wx['Station'] = station_code
# Removing duplicate Date/Time entries:

# Calculating the 24-digit hour corresponding to each DataFrame:
# (This code assumes that a 24-hour clock is being used to display
# times.)
df_wx['Hour'] = df_wx['Time'].str.split(
    ':').str[0].astype('int')

# Keeping only one result per date and hour:
# (This approach will prove useful when using .rolling() to calculate
# rainfall totals for specific periods, as it will allow us to
# assume that N rows of data represent N hours. However,
# at least one NWS station (KOKV) reports recent weather data
# every 20 minutes; only one in three of such reports will get
# retained after drop_duplicates() gets called.
df_wx.drop_duplicates(
    ['Date', 'Hour'], keep = 'last', inplace = True)
# This isn't a perfect approach, as it will likely cause an
# hour of data to get lost when Daylight Savings Time ends.

print("New length of historical data file:",
      len(df_wx))

# Saving this updated copy of df_wx to a .csv file:
df_wx.to_csv(
    f'{data_folder}{post_folder_char}\\
{station_code}_historical_hourly_data.csv', index = False)

## Making further updates to this combined dataset:

# Removing percentages from Relative Humidity column so that these
# values can be converted to floats:

df_wx['Relative Humidity'] = df_wx[
    'Relative Humidity'].str.replace('%', '')

# Converting numerical data in certain columns to floats:

for column in ['Air', 'Dwpt', 'altimeter (in)', 'sea level (mb)',
               '1 hr', '3 hr', '6 hr', 'Relative Humidity']:
    df_wx[column] = df_wx[column].astype(
        'float')

```

(continues on next page)

(continued from previous page)

```
# Replacing NaN precipitation values with 0s:
for column in ['1 hr', '3 hr', '6 hr']:
    df_wx[column] = df_wx[
        column].fillna(0).copy()

df_wx.tail()

# Adding 'Precip' prefixes to the hourly precipitation rows; making the
# temperature and dew point column names more intuitive; and removing
# time zone data from the Time field:
df_wx.rename(columns = {
    '1 hr':'1-Hour Precip',
    '3 hr':'3-Hour Precip',
    '6 hr':'6-Hour Precip',
    'Air':'Temp',
    'Dwpt':'Dew Point',
    'altimeter (in)':'Altimeter (in.)',
    original_time_col:'Time'},
    inplace = True)

# Sorting the table in chronological order so that our charts will
# be easier to interpret:

df_wx.sort_values(['Date', 'Time'], inplace = True)
df_wx.reset_index(drop=True,inplace=True)

# Adding columns that show cumulative precipitation totals for various
# periods: (These all update every hour unlike the built-in
# NWS hourly precipitation totals. However, they're also
# susceptible to errors caused by missing rows.)
for hourly_interval in [3, 6, 12, 24]:
    df_wx[f'Rolling {hourly_interval}-Hour Precip'] = (
        df_wx['1-Hour Precip'].rolling(
            window=hourly_interval).sum())

# Adding a windspeed column:
# This code assumes that the 'Wind (mph)' column within the original
# dataset uses the following formats:
# 'Calm' (when no wind was reported)
# DIRECTION WINDSPEED (e.g. 'W 8') when wind, but no gusts, were
# reported
# DIRECTION WINDSPEED GUST_DIRECTION GUST_SPEED (e.g. 'W 28 G 40')
# when both wind and gusts were reported
# The following code uses str.split() and str[1] to retrieve the
# second item within these reports, as this item should always be
# the windspeed (rather than the gust speed).

df_wx['Windspeed'] = df_wx['Wind (mph)'].str.split(' ').str[
    1]

# For 'Calm' readings,
# str[1] will be NaN, and thus we can replace those values with 0.

df_wx['Windspeed'] = np.where(df_wx['Wind (mph)'] == 'Calm',
                               0, df_wx['Windspeed'])
```

(continues on next page)

(continued from previous page)

```
# Removing the 6-hour max and min temperature columns in order to
# simplify the table:
df_wx.drop(['Max.', 'Min.'], axis = 1, inplace = True)

# Creating a 'Date/Time' column for graphing purposes:
df_wx.insert(
    2, 'Date/Time', df_wx['Date'].astype('str')
    + ' ' + df_wx['Time'].astype('str'))

df_wx.tail()

# Saving this revised dataset to a .csv file so that it can be
# visualized and shared with others:

df_wx.to_csv(f'{data_folder}{post_folder_char}\
{station_code}_historical_hourly_data_updated.csv', index = False)
```


SIMPLE DASH APP WITHOUT LOGIN

16.1 Introduction to the Online Visualizations section of Python for Nonprofits

PFN's Online Visualizations section will demonstrate how to host visualizations online via Dash, a powerful Python library made by the same team that develops Plotly.

Two separate Dash apps will be featured in this section:

1. A simple Dash app that doesn't require users to log in.
2. PFN Dash App Demo, a more detailed Dash app that shows dashboards of varying complexity and also requires users to log in.

These apps take the form of a series of files, most of which (but not all) are .py scripts. In order to get them to run successfully on your own computer, it's very important to (1) consult their corresponding documentation closely and (2) use the same folder structure that is shown within the Python for Nonprofits GitHub page.

It's *also* important to note that **you may incur some costs when deploying Dash apps online via Google Cloud Run** (the method shown within these notebooks). I have found the costs to be quite small on my end (literally pennies a month), but more complex and widely accessed apps will of course incur higher costs. If you want to avoid these costs, you can simply run these apps locally on your own machine.

16.2 The 'Simple App Without Login' project itself

16.2.1 Readme

(Note: the Cloud Run-hosted version of this app can be found at <https://simpleappwithoutlogin-470317599391.us-central1.run.app/>.)

This folder provides sample code for a Dash app that can be deployed to Cloud Run. This code was based on the app deployment steps found in <https://dash.plotly.com/deployment> and <https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-python-service>. (The Plotly walkthrough shows how to deploy a Dash app to Heroku, and the Cloud Run walkthrough shows how to deploy a Flask App to Cloud Run; by taking bits from each, you can then learn how to deploy a *Dash* app to *Cloud Run*. :)

Development and setup notes

1. The dash-bootstrap-components (<https://dash-bootstrap-components.opensource.faculty.ai/docs/components/layout/>) library is used extensively within many of these dashboards. It's a great option for making your dashboards more aesthetically pleasing *and* more flexible.
2. This app is hosted on Google Cloud Run, though you can also host it locally by cloning this project. See the following section for more guidance on hosting Dash apps within Cloud Run.

Rationale for hosting this Dash app on Google Cloud Run

Heroku, an alternative site for hosting Dash apps, offers maximum monthly spending caps, which could help you control your costs; however, it's possible that the minimum amount you would pay on Heroku would be higher than your minimum on Cloud Run. (This is why I chose to deploy this app to Cloud Run rather than Heroku.) Heroku pricing information can be found at <https://www.heroku.com/pricing>.

Steps for deploying this app to Cloud Run

(Some of these steps will apply to many different Cloud Run-hosted Dash apps; however, in order to replicate this online app on your end, you'll also need to review the following section, which explains how to connect your Dash app to Google Sheets.)

1. Visit <https://console.cloud.google.com/>. (Create a Google Cloud Console account if needed).
2. Create a new project. (Note that you *can* deploy multiple Dash apps within the same project if needed).
3. Follow the steps outlined here (<https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-python-service>)—but note the modifications that I've made to these steps below.
4. If you need to install the Google Cloud Command Line Interface (CLI), you can do so at (<https://cloud.google.com/sdk/docs/install>). I deselected the ‘Bundled Python’ component during the installation process since (not surprisingly) I already had Python present on my computer.

I also used the following contents in place of the recommended requirements.txt values:

```
gspread
gspread-dataframe
dash
plotly
pandas
gunicorn
dash-bootstrap-components
```

(These libraries will be imported from pip. Therefore, in order to determine what format of their name to use, visit the library on Pypi and then enter the text following the ‘pip install’ example. For instance, on the gspread-dataframe PyPI page (<https://pypi.org/project/gspread-dataframe/>), you’ll see `pip install gspread-dataframe` as the recommended installation command; therefore, add `gspread-dataframe` to the requirements.txt page rather than `gspread-dataframe` (its name within conda) or `gspread_dataframe` (its name within app.py).

(Also note that entering `gcloud run deploy --source .` (don’t forget the space and period at the end!) rather than `gcloud run deploy` saves you a step during the deployment process.)

The Google Cloud Region Picker (<https://cloud.withgoogle.com/region-picker/>) site can help you determine where to host your app. (When prompted for my region choice, I entered `us-central1`. Note that the numbers corresponding to regions can change over time, so if you want to enter a number for the region selection instead, just make sure it still corresponds to your desired area!)

4. You'll also want to add a file named Procfile (no extension) with the following text:

```
web: gunicorn app:server
```

(This line came from <https://dash.plotly.com/deployment>.)

I was prompted to enter this line after seeing the following error message within my Google Cloud Console log:

```
Step #1: [builder] failed to build: for Python, provide a main.py file or set  
an entrypoint with "GOOGLE_ENTRYPOINT" env var or by creating a "Procfile" file"
```

5. Hopefully, after Google Cloud finishes processing your deployment request, you'll see a message like the following:

```
Service [simpleappwithoutlogin] revision [simpleappwithoutlogin-00012-55f] has  
been deployed and is serving 100 percent of traffic. Service URL: https://  
simpleappwithoutlogin-470317599391.us-central1.run.app
```

To confirm that your page is operating correctly, visit its service URL (e.g. <https://simpleappwithoutlogin-470317599391.us-central1.run.app/> for this app). Ideally, you'll see the same screen that you would when deploying the app locally.

If you instead see a black screen with ‘Service Unavailable’ at the top left corner, don’t panic! (I got this screen many, many times when learning how to upload Dash apps to the cloud.) This message simply means that something about your deployment that is not quite correct. Perhaps you forgot to include a library within requirements.txt that your script uses, for instance. Make sure to check your logs (available at <https://console.cloud.google.com/logs/>) for clues as to what might be going wrong with your script.

(For example, seeing “ModuleNotFoundError: No module named ‘pandas’” within my log after one failed deployment made me realize that I had forgotten to add pandas to my requirements.txt file.)

If you receive a message like “service account 13371337-compute@developer.gserviceaccount.com does not have access to the bucket,” try entering `gcloud auth login` and redoing the login procedure (as suggested by StackOverflow user Lee here: <https://stackoverflow.com/a/70377891/13097194>).

6. After deploying a new version of your app, you may want to delete the old version (especially if it failed to run to begin with) in order to save costs. You can do so by searching for ‘Artifact Registry’ within the Cloud Console website; navigating to the list of containers for your site; and then keeping only the most recent container. Likewise, you may want to keep only the most recent bucket within the source/ folder in your Cloud Storage bucket. (To delete the other ones, search for ‘Buckets’ within your Cloud Console; click on your bucket name; click on the ‘source/’ folder; and then delete all but the most recent bucket.)

Using a Cloud Run secret to retrieve data from a Google Sheets document

This app retrieves data from a Google Sheets file called ‘Hourly VA Weather Data’; it’s located at <https://docs.google.com/spreadsheets/d/17aDJ3mg49-n0IEnDgN7ZB85pO87fiUpkZPULYDB8dmo/edit?usp=sharing>; this file gets updated on an hourly basis by a laptop running `updatintg_online_spreadsheets.py` within the Updating Online Spreadsheets section of Python for Nonprofits. (Reference that section for more information about using `gspread`.)

In order for the app to connect to that spreadsheet, it retrieves a set of Google service account credentials (which are required for retrieving data from Google Sheets) from a secret stored within a Cloud Run volume. I referenced <https://cloud.google.com/run/docs/configuring/services/secrets> in the process of enabling this functionality. Secrets can be accessed by Cloud Run either through a path to a volume or through an environment variable; although I chose the former, the latter might actually be more straightforward to set up.

In order to get this code to work, you’ll want to follow the same steps shown within the above link to add your own service account to Cloud Run. You’ll also need to make sure that your compute service account has been granted the Secret Manager Secret Accessor role. More information about that step can be found at <https://cloud.google.com/secret-manager/docs/manage-access-to-secrets>.

If you decide to access your secret through a volume, you can find the path to enter within your code by clicking on your Cloud Run service; selecting ‘Edit & Deploy New Revision’; and clicking on ‘VOLUME MOUNTS’ within the Edit Container menu. The path to use within your code (in my case, ‘/svcacctsecret/kjb3server_service_account’ was listed under ‘Mount path.’)

Note: In my case, the service account whose credentials I uploaded into Cloud Run is the same account that my laptop is using to update the Hourly VA Weather Data workbook. (The content of my ‘secret’ is simply the .json file containing the account’s key that I downloaded in the process of building out my Updating Online Spreadsheets code.) However, it looks like you can also simply use the service account built into your Cloud Run instance; see <https://stackoverflow.com/questions/65128196/is-there-a-way-to-authenticate-gspread-with-the-default-service-account> for more information.

Using your built-in Cloud Run service account should allow you to bypass the hassle of storing a service account key as a secret. However, it’s still ideal to learn how to use Cloud Run secrets in case you’ll ever need to connect to a SQL database within a Dash app. (You could store that database’s password as a secret, which should be much more secure than storing it as plain text within your code.)

Folder structure

(Note that Procfile is a file without a specified extension rather than a folder.)

readme.md [this file]

app.py

Procfile

requirements.txt

16.3 Source code for Simple App Without Login

16.3.1 app.py

```
# Simple Dash app for displaying Virginia weather data

# By Kenneth Burchfiel
# Released under the MIT License

# Note: the Cloud Run-hosted copy of this app can be found
# at https://simpleappwithoutlogin-470317599391.us-central1.run.app/ .

# (See readme for more information on the code used to connect
# to a Google Sheets workbook with recent weather data)

# Parts of the following code were based on the Dash app tutorial
# at https://dash.plotly.com/tutorial ; the Dash callbacks documentation at
# https://dash.plotly.com/basic-callbacks; the online Dash deployment
# guide at https://dash.plotly.com/deployment ; the Dash Bootstrap intro at
# https://dash-bootstrap-components.opensource.faculty.ai/docs/quickstart/
# ; and the Dash Bootstrap Layout documentation at
# https://dash-bootstrap-components.opensource
# .faculty.ai/docs/components/layout/ .

import gspread
from gspread_dataframe import get_as_dataframe
from dash import Dash, html, dcc, dash_table, callback, Input, Output
```

(continues on next page)

(continued from previous page)

```

import plotly.express as px
import pandas as pd
import dash_bootstrap_components as dbc
from datetime import datetime, timedelta

# From https://pypi.org/project/gspread-dataframe/
print("Initializing gspread using service account key stored within \
Cloud Run secrets volume:")

local_data_import = False # This allows data to get imported from local
# .csv files, which can help save time when debugging. It must be
# set to False prior to exporting the app to Cloud Run, however.

if local_data_import == False:
    # Guillaume Blaquier's post at
    # https://stackoverflow.com/a/68536068/13097194
    # was helpful in drafting the following line.

    gc = gspread.service_account(
        filename='/svcacctsecret/kjb3server_service_account')
    wb = gc.open_by_key('17aDJ3mg49-n0IEnDgN7ZB85pO87fiUpkZPULYDB8dmo')

    # Note: the code above will only run successfully when the app is
    # deployed to Cloud Run. That's because the file path is actually a volume
    # within my Cloud Run container.
    # (See the readme for further details on successfully running
    # this code on your end.)

    # Importing data from each spreadsheet:
    wx_df_list = []

    for station in ['KCHO', 'KIAD', 'KOKV']:
        if local_data_import == True:
            wx_df_list.append(pd.read_csv(f'.../Updating_Online_\
Spreadsheets/weather_data/{station}_historical_hourly_data_\
updated.csv')[-960:])
        else:
            ws = wb.worksheet(station)
            wx_df_list.append(get_as_dataframe(ws)[-960:]) # Importing up to
            # 40 days of data for each station (in order to keep the charts
            # readable)

    # Combining these station-specific tables into a single DataFrame:

    df_wx = pd.concat([df for df in wx_df_list])

    df_wx['Date/Time'] = pd.to_datetime(df_wx['Date/Time'])
    print(df_wx.tail())

    # Creating a condensed copy of df_wx for
    # incorporation in a DataTable:
    # (This table will also be sorted in descending chronological order
    # in order to display the most recent metrics first.)

    current_date = datetime.today()

```

(continues on next page)

(continued from previous page)

```
app = Dash(external_stylesheets=[dbc.themes.BOOTSTRAP])
server = app.server

# The dcc.Markdown() code below was based on
# https://dash.plotly.com/dash-core-components/markdown

app.layout = dbc.Container(
    [dbc.Row(dcc.Markdown('''
## Recent Weather Data for Three Virginia Airports
This Dash app is part of [Python for Nonprofits]\
(https://github.com/kburchfiel/pfn) and has been
released under the MIT license. The source code for this app can
be viewed at [this page](https://github.com/kburchfiel/pfn/blob/main/\nOnline\_Visualizations/Simple\_App\_Without\_Login/app.py).

The NWS data displayed within this notebook was accessed via
[this Google Sheets file] (https://docs.google.com/spreadsheets/d/\n17aDj3mg49-n0IEndGn7ZB85p087fiUpkZPULYDB8dmo/edit?gid=0#gid=0).
This file gets updated with new NWS data
on an hourly basis via [this script] (https://github.com/kburchfiel/pfn/blob/main/Updating\_Online\_Spreadsheets/updating\_online\_spreadsheets.py), which in turn calls
[this script] (https://github.com/kburchfiel/pfn/blob/main/Updating\_Online\_Spreadsheets/weather\_import.py).

A more complex \
Dash app can be found within [this part of PFN]\
(https://github.com/kburchfiel/pfn/tree/main/\nOnline\_Visualizations/PFN\_Dash\_App\_Demo).

''')),
dbc.Row([
    dbc.Col(
        dcc.Markdown('**Metric:**'), md = 2),
    dbc.Col(
        dcc.Dropdown(
            options = ['Temp', 'Dew Point',
'1-Hour Precip', 'Rolling 3-Hour Precip', 'Rolling 6-Hour Precip',
'Rolling 12-Hour Precip', 'Rolling 24-Hour Precip',
'Altimeter (in.)', 'Windspeed'], value = 'Temp', id = 'metric'),
        md = 3)),
dbc.Row([
    dbc.Col(dcc.Markdown('**Stations:**'), md = 2),
    dbc.Col(dcc.Dropdown(['KCHO', 'KIAD', 'KOKV'],
['KCHO', 'KIAD', 'KOKV'],
multi=True,
id = 'station_list'), md = 4))),
dbc.Row([dbc.Col(dcc.Markdown('**Days to Include:**'),
md = 2),
dbc.Col(dcc.Slider(1, 40, 3, value = 10,
# I had originally used one-day increments for the slider,
# but this resulted in overlapping text on mobile displays.
id = 'days_to_include'))]),
dbc.Row(dcc.Graph(id = 'fig')),
```

(continues on next page)

(continued from previous page)

```

dbc.Row(dcc.Markdown("By Kenneth Burchfiel"))])

# Defining a callback function that, given the inputs specified
# above, can plot and return a graph of weather data:
@callback(
    Output('fig', 'figure'),
    Input('metric', 'value'),
    Input('station_list', 'value'),
    Input('days_to_include', 'value'))

def plot_graph(metric, station_list, days_to_include):
    # Determining the earliest point at which data should be displayed:
    data_cutoff = str(current_date - timedelta(
        days = days_to_include))

    # Modifying the title so that it's grammatically correct when
    # only one day of data is being displayed:
    if days_to_include == 1:
        day_title_component = 'Past Day'
    else:
        day_title_component = f'Last {days_to_include} Days'

    fig = px.line(
        df_wx.query("(Station in @station_list) \\"/>
        & (`Date/Time` >= @data_cutoff)"),
        x = 'Date/Time', y = metric,
        color = 'Station',
        title = f"{metric} Over {day_title_component}")

    # Updating y axis title based on the selected metric:
    fig.update_layout(xaxis_title = 'Date')
    if metric in ['Temp', 'Dew Point']:
        fig.update_layout(yaxis_title = 'Degrees (F)')
    if metric == 'Windspeed':
        fig.update_layout(yaxis_title = 'Windspeed (mph)')
    if 'Precip' in metric:
        fig.update_layout(yaxis_title = 'Precipitation (in.)')
    return fig

if __name__ == '__main__':
    app.run(debug=True)

```

16.3.2 Procfile

```
web: gunicorn app:server
```

16.3.3 requirements.txt

```
gspread
gspread-dataframe
dash
plotly
pandas
gunicorn
dash-bootstrap-components
```

PFN DASH APP DEMO

17.1 Readme

Note: For more background information on this section of Python for Nonprofits, Dash Apps, and Google Cloud deployment steps, visit the corresponding readme within Online_Visualizations/Simple_App_Without_Login. (Items explained within that readme often won't be discussed here.)

Visit <https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/> to view the Google Cloud Run-hosted version of this app. (If no one has accessed the app recently, it will take several seconds to load, as the app is set to run on demand in order to save costs.)

This project demonstrates how to use Dash to create various types of interactive online visualizations; these range from simple charts to more complex interactive setups. Both the Flask-Login and Dash-Pivottable libraries are featured within this project.

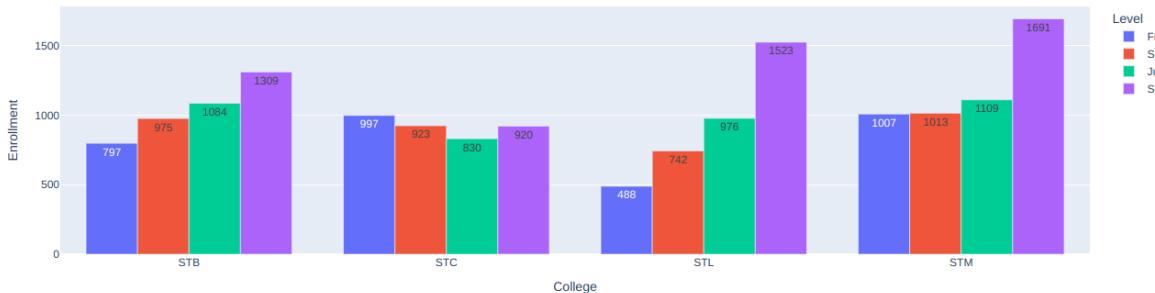
The **Fixed Dashboard** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/fixed_dashboard) page shows a very simple dashboard setup that lacks user-defined filter and comparison settings.

Simple Fixed Dashboard

This dashboard contains three bar charts that display enrollment totals by college and level; college; and level. Plotly's built-in tools make these charts somewhat interactive; however, there aren't any additional comparison or filter options provided.

For an example of a more interactive version of this dashboard, visit the [simple_interactive_dashboard](#) page.

NVCU Enrollment by College and Level



NVCU Enrollment by College



NVCU Enrollment by Level



The **Simple Interactive Dashboard** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/simple_interactive_dashboard) page displays a relatively straightforward interactive enrollment dashboard. This dashboard didn't require much code to write, but its functionality is rather limited.

Simple Interactive Enrollment Dashboard

This dashboard provides a relatively simple overview of NVCU enrollment. There are five preset comparison options and two preset filter options, allowing the user to create a variety of custom views.

This dashboard is not as versatile or flexible as those that apply the Dash-Pivottable library (e.g. the 'Dash pivotable survey results' dashboard), but it can still serve as a helpful introduction to some of Dash's core features.

This dashboard also applies the [Dash Bootstrap Components](#) library in order to create a more condensed layout that accommodates a range of screen sizes.

Comparison Options:

College Filter:

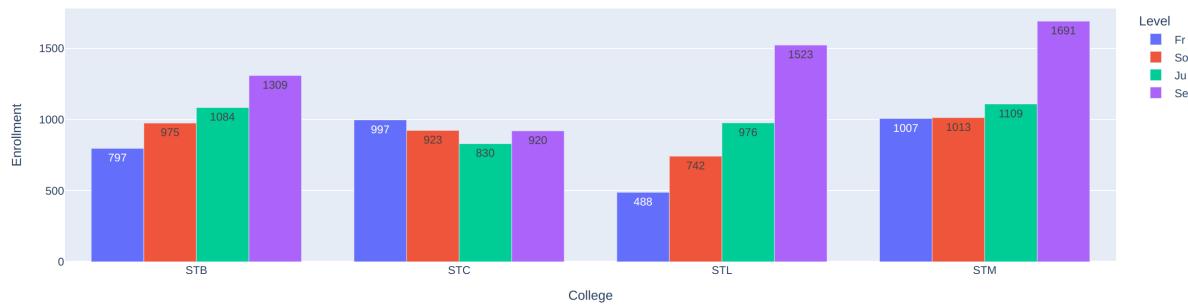
College and Level x ▾

x STC x STM x STL x STB x ▼

x Fr x So x Ju x Se x ▼

Level Filter:

NVCU Enrollment by College and Level



The **Flexible Survey Results** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/flexible_survey_results_dashboard) and **Flexible Enrollment** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/flexible_enrollment_dashboard) dashboard pages allow for a wide range of comparison and color options. These options are made possible by the autopivot() and autobar() functions found within `auto_pivot_and_graph.py` (https://github.com/kburchfiel/pfn/blob/main/Online_Visualizations/PFN_Dash_App_Demo/auto_pivot_and_graph.py). (You may also find these functions useful for developing standalone Plotly charts.)

The Flexible Enrollment Dashboard also makes use of an import_layout() function (stored within `import_layout.py` (https://github.com/kburchfiel/pfn/blob/main/Online_Visualizations/PFN_Dash_App_Demo/import_layout.py)) in order to reduce the amount of code needed to define the page's structure and menu options. In addition, this dashboard applies the autotable() function in `auto_pivot_and_graph.py` to display a tabular view of the data featured in the graph.

Flexible Interactive Enrollment Dashboard

This dashboard provides a flexible overview of NVCU enrollment. It utilizes the autopivot(), autobar(), and autotable() functions found within auto_pivot_and_graph.py to allow for a wide range of display options. It also uses the import_layout() function found in import_layout.py to define a sizeable component of the page's layout.

This dashboard also applies the [Dash Bootstrap Components](#) library in order to create a more condensed layout that accommodates a range of screen sizes.

Comparison Options:

College Filter:

Level Filter:

Gender Filter:

Color Option:



Level For Sorting	Level	College	Enrollment
0	Fr	STB	797
0	Fr	STC	997
0	Fr	STM	1007
1	So	STB	975
1	So	STC	923
1	So	STM	1013
2	Ju	STB	1084
2	Ju	STC	830
2	Ju	STM	1109

The dash-pivottable library makes it very easy to create interactive dashboards. Examples of this library in use can be found within the **Dash Pivottable (Enrollment)** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/dash_pivottable_enrollment) and **Dash Pivottable (Survey Results)** (https://pfndashappdemo-ymc7cs3r5q-uc.a.run.app/dash_pivottable_survey_results) pages.



17.1.1 Development notes:

1. I made use of a standalone Jupyter notebook (notebook_for_testing.ipynb, available at https://github.com/kburchfiel/pfn/blob/main/Online_Visualizations/PFN_Dash_App_Demo/notebook_for_testing.ipynb) to test out code before integrating it into my Dash app files.
2. The source data is imported from GitHub. A more realistic approach would retrieve data from an online database; however, that would cause this project to incur a monthly database hosting expense.

17.1.2 Project structure

The files and folders in this app are arranged as follows:

(Folders have ‘/’ suffixes.)

```

pages/
    --home_page.py
    --fixed_dashboard.py
    --simple_interactive_dashboard.py
    --flexible_survey_results_dashboard.py
    --flexible_enrollment_dashboard.py
    --dash_pivottable_enrollment.py
    --dash_pivottable_survey_results.py
templates/
    --login.html
app.py
auto_pivot_and_graph.py
data_import.py
import_layout.py
Procfile

```

[readme.md](#) [this file]

[requirements.txt](#)

17.1.3 Development and setup notes

Some of the code in this folder was derived from jinnyzor's post at <https://community.plotly.com/t/dash-app-pages-with-flask-login-flow-using-flask/69507/38>. Jinnyzor wrote regarding that post at (<https://community.plotly.com/t/dash-app-pages-with-flask-login-flow-using-flask/69507/55>) that "this is free to use, no license." I am very grateful to jinnyzor (and to Nader Elshehabi (<https://github.com/naderelshehabi/dash-flask-login>)) for allowing us to use their code!

Some code also derives from that found in the Dash Pages documentation (<https://dash.plotly.com/urls>).

Steps for hosting this Dash app (with Flask-Login functionality) on Google Cloud Run

Most of the deployment steps written within the Simple_App_Without_Login readme apply here as well, with the following exceptions:

1. I used the following requirements.txt values (to reflect the libraries used within this app):

```
dash==3.0.0
dash-bootstrap-components==2.0.0
dash-pivottable==0.0.2
Flask==3.0.3
Flask-Login==0.6.3
gunicorn==23.0.0
pandas==2.2.3
plotly==6.0.1
Werkzeug==3.0.6
```

Note: I specified the exact versions of each library within this file because, when updating the app after Dash 3 got released, I found that an earlier version of Dash was being used within my cloud-hosted app. (I think this explains why data within certain graphs was not appearing correctly.) To resolve this issue, I:

1. Created a new Conda environment just for this app
2. Used pip to import each library within my requirements.txt file (since pip would also be used to add libraries to the cloud-hosted app)
3. Confirmed that the local app was working correctly
4. Copied the version numbers for each locally-hosted library into my requirements.txt file
5. Redeployed the app—which, thankfully, now worked fine
6. It's worth highlighting that, whereas the app within the Simple_App_Without_Login folder contained the line `server = app.server` (as indicated by the deployment guide at <https://dash.plotly.com/deployment>), this app will instead use `server = Flask(__name__)` due to the incorporation of Flask-Login.

17.2 Source code for PFN Dash App Demo

17.2.1 dash_pivottable_enrollment.py

```
# This code shows how to use the dash-pivottable library to easily
# create an interactive enrollment dashboard.

# Much of this code derives from
# https://github.com/plotly/dash-pivottable/blob/master/usage.py
# and https://dash.plotly.com/urls .

import dash
from dash import html, callback, Input, Output
import dash_pivottable
import pandas as pd
from data_import import df_curr_enrollment

# Reading in enrollment data:

# df_curr_enrollment = pd.read_csv(
#     'https://raw.githubusercontent.com/kburchfiel/\n'
#     'pfn/main/Appendix/curr_enrollment.csv')

# Note that the 'data' entry below should take the form of a list of lists
# or list of dicts, rather than a DataFrame. (For reference, see
# https://github.com/plotly/react-pivottable/#accepted-formats-for-data)
# Therefore, we'll need to convert our DataFrame into this format
# before we can run the code.
# It's possible to convert a DataFrame into a list of lists,
# but I believe the easiest solution is to use to_dict(orient = 'records')
# to convert the DataFrame into a list of dictionaries.

lod_curr_enrollment = df_curr_enrollment.to_dict(
    orient = 'records')
# lod = 'list of dicts'

dash.register_page(__name__, path = '/dash_pivottable_enrollment') #

layout = html.Div([
    dash_pivottable.PivotTable(
        id='enrollment_table',
        data=lod_curr_enrollment,
        cols=['Level For Sorting', 'Level'],
        colOrder="key_a_to_z",
        rows=['College'],
        rowOrder="key_a_to_z",
        rendererName="Grouped Column Chart",
        aggregatorName="Sum",
        vals=[ "Enrollment"],
        valueFilter={}
    ),
    html.Div(
        id='output'
    )
])
```

(continues on next page)

(continued from previous page)

```
)  
])  
  
@callback(Output('enrollment_output', 'enrollment_children'),  
          [Input('enrollment_table', 'cols'),  
           Input('enrollment_table', 'rows'),  
           Input('enrollment_table', 'rowOrder'),  
           Input('enrollment_table', 'colOrder'),  
           Input('enrollment_table', 'aggregatorName'),  
           Input('enrollment_table', 'rendererName')])  
def display_props(cols, rows, row_order, col_order, aggregator, renderer):  
    return [  
        html.P(str(cols), id='columns'),  
        html.P(str(rows), id='rows'),  
        html.P(str(row_order), id='row_order'),  
        html.P(str(col_order), id='col_order'),  
        html.P(str(aggregator), id='aggregator'),  
        html.P(str(renderer), id='renderer'),  
    ]
```

17.2.2 dash_pivottable_survey_results.py

```
# This code shows how to use the dash-pivottable library to easily create  
# an interactive survey results dashboard.  
  
# For additional documentation, see enrollment_pivot.py.  
  
import dash  
from dash import html, callback, Input, Output  
import dash_pivottable  
import pandas as pd  
  
# Reading in survey data, then merging it with  
# enrollment data:  
# (The benefit of using original student-level data is that  
# our averages will automatically be weighted by student counts, thus  
# producing more accurate averages.)  
  
from data_import import df_survey_results_extra_data  
  
lod_survey_results_extra_data = df_survey_results_extra_data.to_dict(  
    orient = 'records')  
  
dash.register_page(__name__, path = '/dash_pivottable_survey_results')  
  
layout = html.Div([  
    dash_pivottable.PivotTable(  
        id='table',  
        data=lod_survey_results_extra_data,  
        cols=['Season'],
```

(continues on next page)

(continued from previous page)

```

        colOrder="key_a_to_z",
        rows=['College'],
        rowOrder="key_a_to_z",
        rendererName="Grouped Column Chart",
        aggregatorName="Average",
        vals=["Score"],
        valueFilter={}
    ),
    html.Div(
        id='output'
    )
)
])

@callback(Output('output', 'children'),
          [Input('table', 'cols'),
           Input('table', 'rows'),
           Input('table', 'rowOrder'),
           Input('table', 'colOrder'),
           Input('table', 'aggregatorName'),
           Input('table', 'rendererName')])
def display_props(cols, rows, row_order, col_order, aggregator, renderer):
    return [
        html.P(str(cols), id='columns'),
        html.P(str(rows), id='rows'),
        html.P(str(row_order), id='row_order'),
        html.P(str(col_order), id='col_order'),
        html.P(str(aggregator), id='aggregator'),
        html.P(str(renderer), id='renderer'),
    ]
]

```

17.2.3 fixed_dashboard.py

```

# Sample fixed dashboard
# By Kenneth Burchfiel
# Released under the MIT License

import dash
from dash import html, dcc
import dash_bootstrap_components as dbc
from data_import import df_curr_enrollment
import plotly.express as px

# Creating a pivot table that can serve as the basis of our enrollment
# by college and level graph:

df_enrollment_by_college_and_level = df_curr_enrollment.pivot_table(
    index = ['College', 'Level For Sorting', 'Level'],
    values = 'Enrollment', aggfunc = 'sum').reset_index()

# Creating a graph of this pivot table:
fig_enrollment_by_college_and_level = px.bar(
    df_enrollment_by_college_and_level,
    x = 'College', y = 'Enrollment', color = 'Level',

```

(continues on next page)

(continued from previous page)

```
barmode = 'group',
text_auto = '.0f',
title = 'NVCU Enrollment by College and Level')

# Performing the same steps for simpler charts that show enrollment
# by college and by level (but not both):
df_enrollment_by_college = df_curr_enrollment.pivot_table(
    index = ['College'],
    values = 'Enrollment', aggfunc = 'sum').reset_index()

fig_enrollment_by_college = px.bar(df_enrollment_by_college,
    x = 'College', y = 'Enrollment', color = 'College',
    text_auto = '.0f',
    title = 'NVCU Enrollment by College')

df_enrollment_by_level = df_curr_enrollment.pivot_table(
    index = ['Level For Sorting', 'Level'],
    values = 'Enrollment', aggfunc = 'sum').reset_index()

fig_enrollment_by_level = px.bar(df_enrollment_by_level,
    x = 'Level', y = 'Enrollment', color = 'Level',
    text_auto = '.0f',
    title = 'NVCU Enrollment by Level')

dash.register_page(__name__, path='/fixed_dashboard')

layout = dbc.Container([
    dcc.Markdown('''

This dashboard contains three bar charts that display enrollment totals by college and level; college; and level. Plotly's built-in tools make these charts somewhat interactive; however, no additional comparison or filter options are provided. The other dashboards within this app allow for more user interaction.

''''),
    dcc.Graph.figure=fig_enrollment_by_college_and_level),
# This method of hosting a fixed graph within a Dash page (without any
# callbacks) came from https://dash.plotly.com/
# tutorial#visualizing-data .
    dcc.Graph.figure=fig_enrollment_by_college),
    dcc.Graph.figure=fig_enrollment_by_level)
])
```

17.2.4 flexible_enrollment_dashboard.py

```

# Flexible Enrollment Dashboard
# By Kenneth Burchfiel
# Released under the MIT License

# Parts of this code derive from
# and https://dash.plotly.com/urls
# and https://dash.plotly.com/minimal-app .

import dash
from dash import html, dcc, callback, Output, Input, dash_table
import dash_bootstrap_components as dbc

from data_import import df_curr_enrollment
import plotly.express as px
from auto_pivot_and_graph import autopivot_plus_bar
from import_layout import import_layout

dash.register_page(__name__, path='/flexible_enrollment_dashboard')

# Determining which comparison and color options to pass to
# the import_layout() function that will define part of the
# dashboard's layout:

# These lists of options will get initialized as all columns within
# the DataFrame *except* for those present in cols_to_exclude. (If there
# are many columns within the DataFrame, this approach can require
# less typing than would adding in all columns to be included.)

cols_to_exclude = [
    'Date Of Birth', 'First Name', 'Last Name',
    'Student ID', 'Matriculation Number', 'Enrollment']
comparison_list = list(
    set(df_curr_enrollment.columns) - set(cols_to_exclude))
color_list = comparison_list.copy() # These lists can contain
# the same values.

# Setting default comparison and color values:
comparison_default = ['Level For Sorting', 'Level']
color_default = 'College'

filter_cols = ['College', 'Level', 'Gender']
# Note that each of these values must be added to
# the @callback() component of this page along with
# the display_graph inputs.
# (Add '_filter' after each column name within the Callback section.
# For instance, 'College' will map to 'College_filter.')

# Configuring the page's layout:
# (Note the use of + to combine different lists of layout components
# together.)

layout = dbc.Container([
    dbc.Row(dbc.Col(dcc.Markdown('''

# Flexible Interactive Enrollment Dashboard

```

(continues on next page)

(continued from previous page)

This dashboard provides a flexible overview of NVCU enrollment. It utilizes the autopivot(), autobar(), and autotable() functions found within auto_pivot_and_graph.py to allow for a wide range of display options. It also uses the import_layout() function found in import_layout.py to define a sizeable component of the page's layout.

```
'''), lg = 9))] +
import_layout(df = df_curr_enrollment,
              comparison_list=comparison_list,
              comparison_default = comparison_default,
              color_list = color_list,
              color_default = color_default,
              filter_cols = filter_cols) +
# For more information about the multi-dropdown option,
# see https://dash.plotly.com/dash-core-components/dropdown
[dcc.Graph(id='flexible_enrollment_graph')] +
[dcc.Graph(id='flexible_enrollment_table')]
)

# Configuring a callback that can convert the index and filter options
# specified by the user into a custom chart:

@callback(
    Output('flexible_enrollment_graph', 'figure'),
    Output('flexible_enrollment_table', 'figure'),
    Input('comparison_options', 'value'),
    Input('color_option', 'value'),
    Input('College_filter', 'value'),
    Input('Level_filter', 'value'),
    Input('Gender_filter', 'value')
)

# The following function calls autopivot_plus_bar to convert
# the input values specified above into a bar chart:
def display_graph(x_vars, color, college_filter,
                  level_filter, gender_filter):
    print(college_filter,level_filter, gender_filter)

    # Creating a list of tuples that can be used to filter
    # the output:
    filter_tuple_list = [
        ('College',
         college_filter),
        ('Level',level_filter),
        ('Gender',gender_filter)
    ]
    print('x_vars contents and type:',x_vars,type(x_vars))
    print('color contents and type:',color,type(color))

    # '' is passed to custom_aggfunc_name so that
    # the chart title will begin with 'Enrollment'
    # rather than 'Total Enrollment.'
    bar_graph, table = autopivot_plus_bar(
        df = df_curr_enrollment, y = 'Enrollment',
```

(continues on next page)

(continued from previous page)

```

    aggfunc = 'sum', x_vars = x_vars, color = color,
x_vars_to_exclude = ['Level For Sorting'],
overall_data_name = 'All Data',
weight_col = None, filter_tuple_list = filter_tuple_list,
custom_aggfunc_name = '', create_table = True,
text_auto = '.0f')

print(table)

return bar_graph, table

```

17.2.5 flexible_survey_results_dashboard.py

```

# Flexible survey_results Dashboard
# By Kenneth Burchfiel
# Released under the MIT License

# Parts of this code derive from
# and https://dash.plotly.com/urls
# and https://dash.plotly.com/minimal-app .

import dash
from dash import html, dcc, callback, Output, Input
from data import df_survey_results_extra_data
import plotly.express as px
import dash_bootstrap_components as dbc
from auto_pivot_and_graph import autopivot_plus_bar

dash.register_page(__name__, path='/flexible_survey_results_dashboard')

# Configuring the page's layout:
layout = dbc.Container([
    dbc.Row(dbc.Col(dcc.Markdown('''

        # Flexible Interactive Survey Results Dashboard

        This dashboard provides a flexible overview of NVCU student survey
        results. It utilizes the autopivot() and autobar() functions found within
        auto_pivot_and_graph.py to allow for a wide range of display options.

    '''), lg = 9)),
    dbc.Row([
        dbc.Col(html.H5("Comparison Options:"), lg = 3),
        dbc.Col(dcc.Dropdown(
            ['Starting Year', 'Season', 'Gender', 'Matriculation Year',
            'College', 'Class Of', 'Level', 'Level For Sorting'],
            ['College', 'Season'], multi = True,
            id = 'flexible_survey_results_index'), lg = 3),
        dbc.Col(html.H5("Color Option:"), lg = 2),
        dbc.Col(dcc.Dropdown(
            ['Starting Year', 'Season',
            'Score', 'Gender', 'Matriculation Year',
            'College', 'Class Of', 'Level', 'Level For Sorting'],
            [None], multi = True), lg = 2)
    ])
])

```

(continues on next page)

(continued from previous page)

```

'Season', id = 'flexible_survey_results_color'), lg = 2)
        ]),
dbc.Row([
    dbc.Col(html.H5("College Filter:"), lg = 3),
dbc.Col(
    dcc.Dropdown(df_survey_results_extra_data['College'].unique(),
                 df_survey_results_extra_data['College'].unique(),
                 multi = True,
                 id = 'college_filter'), lg = 3),
        ]),
dbc.Row([
    dbc.Col(html.H5("Level Filter:"), lg = 3),
dbc.Col(dcc.Dropdown(df_survey_results_extra_data['Level'].unique(),
                     df_survey_results_extra_data['Level'].unique(),
                     multi = True,
                     id = 'level_filter'), lg = 3)
        ]),
# For more information about the multi-dropdown option,
# see https://dash.plotly.com/dash-core-components/dropdown
dcc.Graph(id='flexible_survey_results_view')))

# Configuring a callback that can convert the index and filter options
# specified by the user into a custom chart:

@callback(
    Output('flexible_survey_results_view', 'figure'),
    Input('flexible_survey_results_index', 'value'),
    Input('flexible_survey_results_color', 'value'),
    Input('college_filter', 'value'),
    Input('level_filter', 'value')
)
# The following function calls autopivot_plus_bar to convert
# the input values specified above into a bar chart:
def display_graph(x_vars, color, college_filter, level_filter):
    print(college_filter, level_filter)

    # Creating a list of tuples that can be used to filter
    # the output:
    filter_tuple_list = [
        ('College',
         college_filter),
        ('Level', level_filter)]

    print('x_vars contents and type:', x_vars, type(x_vars))
    print('color contents and type:', color, type(color))

    return autopivot_plus_bar(
        df = df_survey_results_extra_data, y = 'Score',
        aggfunc = 'mean', x_vars = x_vars, color = color,
        x_vars_to_exclude = ['Level For Sorting'],
        overall_data_name = 'All Data',
        weight_col = None, filter_tuple_list = filter_tuple_list)

```

17.2.6 home_page.py

```
# This code derives from
# and https://dash.plotly.com/urls .

import dash
from dash import html, dcc
import dash_bootstrap_components as dbc

dash.register_page(__name__, path='/')

layout = dbc.Container([
    dcc.Markdown('''
## [Python For Nonprofits](https://github.com/kburchfiel/pfn) \
Main Dash App Demo

This project demonstrates how to use Dash to create
interactive online visualizations. These visualizations range
from simple charts to more complex interactive setups.

The [Fixed Dashboard](/fixed_dashboard) page shows a very simple
dashboard setup that lacks user-defined filter and comparison settings.

The [Simple Interactive Dashboard](/simple_interactive_dashboard) page
displays a relatively straightforward interactive enrollment dashboard.
This dashboard didn't require much code to write, but its functionality
is rather limited.

The [Flexible Survey Results](/flexible_survey_results_dashboard) and
[Flexible Enrollment](/flexible_enrollment_dashboard) dashboard pages
allow for a wide range of comparison and color options. These options are
made possible by the autopivot() and autobar() functions found within
[auto_pivot_and_graph.py](https://github.com/kburchfiel/pfn/blob/main/\n
Online_Visualizations/PFN_Dash_App_Demo/auto_pivot_and_graph.py). (You may
also find these functions useful for developing standalone Plotly charts.)

The Flexible Enrollment Dashboard also makes use of an import_layout()
function (stored within [import_layout.py](https://github.com/\n
kburchfiel/pfn/blob/main/Online_Visualizations/\n
PFN_Dash_App_Demo/import_layout.py))
in order to reduce the amount of code needed to define
the page's structure and menu options. In addition, this dashboard
applies the autotable() function in [auto_pivot_and_graph.py](https://\n
github.com/kburchfiel/pfn/blob/main/Online_Visualizations/\n
PFN_Dash_App_Demo/auto_pivot_and_graph.py)) to display
a tabular view of the data featured in the graph.

The dash-pivottable library makes it very easy to
create interactive dashboards. Examples of this library in use can
be found within the [Dash Pivottable (Enrollment)](\n
/dash_pivottable_enrollment) and [Dash Pivottable (Survey Results)](\n
/dash_pivottable_survey_results) pages.

The source code for these dashboards can be found [at this link](https://\n
github.com/kburchfiel/pfn/tree/main/\n
Online_Visualizations/PFN_Dash_App_Demo).
```

(continues on next page)

(continued from previous page)

```
...  
 ) ])
```

17.2.7 simple_interactive_dashboard.py

```
# Simple Enrollment Dashboard  
# By Kenneth Burchfiel  
# Released under the MIT License  
  
# Parts of this code derive from  
# and https://dash.plotly.com/urls  
# and https://dash.plotly.com/minimal-app .  
  
# The Layout page within the Dash Bootstrap Components documentation  
# (available at https://dash-bootstrap-components.opensource.faculty.ai/  
# docs/components/layout/ )  
# provide very helpful in creating this dashboard (along with a number  
# of other dashboards within this app.)  
  
import dash  
from dash import html, dcc, callback, Output, Input  
from data import df_curr_enrollment  
import plotly.express as px  
import dash_bootstrap_components as dbc  
  
dash.register_page(__name__, path='/simple_interactive_dashboard')  
  
# Configuring the page's layout:  
layout = dbc.Container([  
    dbc.Row(dbc.Col(dcc.Markdown('''  
  
        # Simple Interactive Enrollment Dashboard  
  
        This dashboard provides a relatively simple overview of NVCU  
        enrollment. There are five preset comparison options and two preset  
        filter options, allowing the user to create a variety of custom views.  
  
        This dashboard is not as versatile or flexible as those that apply the  
        Dash-Pivottable library (e.g. the 'Dash pivotable survey results'  
        dashboard), but it can still serve as a helpful introduction  
        to some of Dash's core features.  
  
        This dashboard also applies the Dash Bootstrap Components (  
        https://dash-bootstrap-components.opensource.faculty.ai)  
        library in order  
        to create a more condensed layout that accommodates a range of  
        screen sizes. This library is also used in certain other dashboards  
        within this app.  
  
        '''), lg = 9)),  
    dbc.Row([  
        dbc.Col(html.H5("Comparison Options:"), lg = 3),  
        dbc.Col(dcc.Dropdown(['College', 'Level',  
                            'College and Level', 'Level and College',
```

(continues on next page)

(continued from previous page)

```

        'All Students'], 'College and Level',
        id = 'simple_enrollment_index'), lg = 2)
    ]),
dbc.Row([
    dbc.Col(html.H5("College Filter:"), lg = 3),
dbc.Col(
    dcc.Dropdown(df_curr_enrollment['College'].unique(),
                 df_curr_enrollment['College'].unique(),
                 multi = True,
                 id = 'college_filter'), lg = 3),
]),
dbc.Row([
    dbc.Col(html.H5("Level Filter:"), lg = 3),
dbc.Col(dcc.Dropdown(df_curr_enrollment['Level'].unique(),
                     df_curr_enrollment['Level'].unique(),
                     multi = True,
                     id = 'level_filter'), lg = 3)
]),
# For more information about the multi-dropdown option,
# see https://dash.plotly.com/dash-core-components/dropdown
dcc.Graph(id='simple_enrollment_view'))]

# Configuring a callback that can convert the index and filter options
# specified by the user into a custom chart:

@callback(
    Output('simple_enrollment_view', 'figure'),
    Input('simple_enrollment_index', 'value'),
    Input('college_filter', 'value'),
    Input('level_filter', 'value')
)

# The following function uses the Input values specified above
# to update the chart shown within this page.
def display_graph(pivot_index, college_filter, level_filter):

    # Filtering a copy of df_curr_enrollment to include only the filter
    # values specified by the user:
    # print(college_filter, level_filter)
    df_curr_enrollment_for_chart = df_curr_enrollment.copy().query(
        "College in @college_filter & Level in @level_filter")

    # Using the pivot_index argument to determine which values
    # to pass to the pd.pivot_table() and px.bar() calls within
    # this function:
    if pivot_index == 'College':
        index = 'College'
        x = 'College'
        color = 'College'
        barmode = 'relative'

    elif pivot_index == 'Level':
        index = ['Level For Sorting', 'Level']
        x = 'Level'
        color = 'Level'
        barmode = 'relative'

```

(continues on next page)

(continued from previous page)

```
elif pivot_index == 'College and Level':
    index = ['College', 'Level For Sorting', 'Level']
    x = 'College'
    color = 'Level'
    barmode = 'group'

elif pivot_index == 'Level and College':
    index = ['Level For Sorting', 'Level', 'College']
    x = 'Level'
    color = 'College'
    barmode = 'group'

elif pivot_index == 'All Students': # In this case, all data will
    # be grouped together. We'll create a new column named
    # 'All Students' that can serve as the x axis label for this data.
    df_curr_enrollment_for_chart['All Students'] = 'All Students'
    index = 'All Students'
    color = 'All Students'
    x = 'All Students'
    barmode = 'relative'

# Creating a pivot table that can serve as the basis for our
# enrollment chart:
df_simple_enrollment_pivot = df_curr_enrollment_for_chart.pivot_table(
    index = index,
    values = 'Enrollment', aggfunc = 'sum').reset_index()

# Creating this chart:
fig_simple_enrollment = px.bar(df_simple_enrollment_pivot,
    x = x, y = 'Enrollment', color = color,
    text_auto = '.0f', barmode = barmode,
    title = f'NVCU Enrollment by {pivot_index}')
return fig_simple_enrollment
```

17.2.8 login.html

```
<!DOCTYPE html>
<!-- Source: jinnyzor at https://community.plotly.com/t/dash-app-pages-with-flask-login-flow-using-flask/69507/38. Jinnyzor wrote later in this thread that "this is free to use, no license". -->
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Flask Login Flow</title>
</head>
<body>
<form method="POST" action="/login">
    <div>Please log in to continue:</div>
    <div>Note: this login page is simply meant to demonstrate that Dash apps can be used with the Flask-Login library. No private
```

(continues on next page)

(continued from previous page)

```

information is being
displayed.<br>You can use <strong>hello</strong> as the username
and <strong>world</strong> as the password.</div>
<div><span style="color:red">{message}</span></div>
<input placeholder="Enter your username here" type="text"
id="uname-box" name='username'>
<input placeholder="Enter your password here" type="password"
id="pwd-box" name='password'>
<Button type="Submit" id="login-button">Log in</Button>
</form>
</body>
</html>

```

17.2.9 app.py

```

# Most of the following code derived from:
# https://community.plotly.com/t/dash-app-pages-
# with-flask-login-flow-using-flask/69507/37
# Note that Nader Elshehabi's code (on which this code was based)
# was released under the MIT license:
# https://github.com/naderelshehabi/dash-flask-login
# Ken Burchfiel incorporated some additional code from
# https://dash.plotly.com/urls and https://dash-bootstrap-components.
# opensource.faculty.ai/docs/quickstart/ ;
# he also made minor edits to the display text.

"""
CREDIT: This code was originally adapted for Pages based on Nader
Elshehabi's
article:
https://dev.to/naderelshehabi/securing-plotly-dash-using-flask-login-4ia2
https://github.com/naderelshehabi/dash-flask-login

This version was updated by Dash community member @jinnyzor .
For more info, see:
https://community.plotly.com/t/dash-app-pages-with-
flask-login-flow-using-flask/69507

For other Authentication options, see:
Dash Enterprise:
    https://dash.plotly.com/authentication#dash-enterprise-auth
Dash Basic Auth:
    https://dash.plotly.com/authentication#basic-auth

"""

import os
from flask import Flask, request, redirect, session, jsonify, \
url_for, render_template
from flask_login import login_user, LoginManager, UserMixin, \
logout_user, current_user

import dash

```

(continues on next page)

(continued from previous page)

```
from dash import dcc, html, Input, Output, State, ALL
import dash_bootstrap_components as dbc # See
# https://dash-bootstrap-components.opensource.faculty.ai/docs/quickstart/

# Exposing the Flask Server to enable configuring it for logging in
# Note that this code will be used in place of
# server = app.server , which we used within our Simple_App_Without_Login
# app.
server = Flask(__name__)

@server.before_request
def check_login():
    if request.method == 'GET':
        if request.path in ["/login", "/logout"]:
            return
        if current_user:
            if current_user.is_authenticated:
                return
            else:
                for pg in dash.page_registry:
                    if request.path == dash.page_registry[pg]['path']:
                        session['url'] = request.url
        return redirect(url_for('login'))
    else:
        if current_user:
            if request.path == '/login' or current_user.is_authenticated:
                return
        return jsonify(
            {'status': '401', 'statusText': 'unauthorized access'})

@server.route('/login', methods=['POST', 'GET'])
def login(message=""):
    if request.method == 'POST':
        if request.form:
            username = request.form['username']
            password = request.form['password']
            if VALID_USERNAME_PASSWORD.get(username) is None:
                return """The username and/or password were \
invalid. <a href='/login'>Please try again.</a>"""
            if VALID_USERNAME_PASSWORD.get(username) == password:
                login_user(User(username))
                if 'url' in session:
                    if session['url']:
                        url = session['url']
                        session['url'] = None
                    return redirect(url) ## redirect to target url
                return redirect('/') ## redirect to home
            message = "The username and/or password were invalid."
    else:
        if current_user:
            if current_user.is_authenticated:
                return redirect('/')
    return render_template('login.html', message=message)
```

(continues on next page)

(continued from previous page)

```

@server.route('/logout', methods=['GET'])
def logout():
    if current_user:
        if current_user.is_authenticated:
            logout_user()
    return render_template('login.html',
                           message="You have now been logged out.")

app = dash.Dash(
    __name__, server=server, use_pages=True,
    suppress_callback_exceptions=True,
    external_stylesheets=[dbc.themes.BOOTSTRAP]
)
# See:
# https://dash-bootstrap-components.opensource.faculty.ai/docs/quickstart/

# Keep this out of source code repository - save in a file or a database
# passwords should be encrypted
VALID_USERNAME_PASSWORD = {"test": "test", "hello": "world"}

# Updating the Flask Server configuration with Secret Key to encrypt
# the user session cookie
# server.config.update(SECRET_KEY=os.getenv("SECRET_KEY"))
server.config.update(SECRET_KEY="insecureplaceholder")
# Definitely don't use the above approach in a real-world application!

# Login manager object will be used to login / logout users
login_manager = LoginManager()
login_manager.init_app(server)
login_manager.login_view = "/login"

class User(UserMixin):
    # User data model. It has to have at least self.id as a minimum
    def __init__(self, username):
        self.id = username

@login_manager.user_loader
def load_user(username):
    """This function loads the user by user id. Typically this looks
    up the user from a user database.
    We won't be registering or looking up users in this example,
    since we'll just login using LDAP server.
    So we'll simply return a User object with the passed in username.
    """
    return User(username)

print([
    (page['name'], page["relative_path"])
for page in dash.page_registry.values()])

# print(dash.page_registry.values())

app.layout = html.Div(

```

(continues on next page)

(continued from previous page)

```
[  
    html.A('Log out', href='..../logout'),  
    html.Br(),  
  
    # The following commented-out and active sections of the script  
    # show three different ways of building a navigation menu.  
  
    # html.H3("Page index:"), # Commented out--see notes below  
    # The following html.Div() section came from:  
    # https://dash.plotly.com/urls .  
    # It places all pages on a separate line, which I found  
    # to be unwieldy for apps like this one with larger page  
    # counts.  
    #     html.Div([  
    #         html.Div(  
    #             dcc.Link(f"{page['name']} - {page['path']}",  
    #                     href=page["relative_path"])  
    #         ) for page in dash.page_registry.values()  
    #     ]),  
  
    # This variant of the above commented-out code places all  
    # pages on the same line (though, if the window isn't wide enough,  
    # some pages may get placed on separate lines). It updates  
    # automatically to incorporate page additions and deletions,  
    # but it offers less control over page names and orders than the  
    # Markdown-based headers that follow it.  
    #     html.Div(  
    #         [dcc.Link(f"{page['name']} / ", href=page["relative_path"])]  
    #         for page in dash.page_registry.values()),  
  
    # The following code shows a more manual, Markdown-based  
    # approach to creating a navigation menu. Although the  
    # Markdown will need to be updated manually to incorporate  
    # new pages, this approach allows both page names and orders  
    # to be easily customized.  
    # Note that the relative links shown within this example  
    # suffice for navigation purposes; absolute links aren't  
    # necessary. (If they were, we might have needed to create  
    # separate links for offline and online deployments.)  
  
    # For documentation on dcc.Markdown,  
    # visit: https://dash.plotly.com/dash-core-components#markdown  
  
    dcc.Markdown(''  
        ### Page Index:  
        [Home Page] (/) |  
        [Fixed Dashboard] (/fixed_dashboard) |  
        [Simple Interactive Dashboard] (/simple_interactive_dashboard)  
  
        [Flexible Survey Results Dashboard] (/flexible_survey_results_dashboard) |  
        [Flexible Enrollment Dashboard] (/flexible_enrollment_dashboard) |  
        [Dash PivotTable (Enrollment)] (/dash_pivottable_enrollment) |  
        [Dash PivotTable (Survey Results)] (/dash_pivottable_survey_results)  
    '''),  
    dash.page_container,
```

(continues on next page)

(continued from previous page)

```

# The following explanatory text will appear at the bottom of
# each page.
dcc.Markdown('''
This site is part of Python for Nonprofits
(https://github.com/kburchfiel/pfn), created by Kenneth Burchfiel and
licensed under the MIT License.*

*Blessed Carlo Acutis, pray for us!*
'''),
]
)

if __name__ == "__main__":
    app.run(debug=True)

```

17.2.10 auto_pivot_and_graph.py

```

# Functions for automatically generating pivot tables, graphs,
# and go.Table objects
# By Kenneth Burchfiel

# Released under the MIT license

# These functions were originally developed within the
# pivot_and_graph_functions.ipynb notebook in the Graphing section
# of Python for Nonprofits.

import plotly.express as px
import plotly.graph_objects as go
from dash import dash_table

def autopivot(df, y, aggfunc, x_vars = [],
              color = None, x_vars_to_exclude = [],
              overall_data_name = 'All Data',
              weight_col = None, filter_tuple_list = [],
              convert_x_vars_to_strings = True):
    '''This function will create a pivot table of df that can be used
    within a Plotly graph. It will also return x, y, color, and barmode
    variables that can get incorporated within Plotly charts. (Storing
    the charting code within a separate function makes autopivot more
    versatile, as its output can then be used as the basis for multiple
    chart types.)'''

    This function has been designed to work with varying lengths of x_vars
    (i.e. multiple numbers of x variables), including a length of 0.
    It should therefore prove useful for applications, such as interactive
    dashboards, that allow the user
    to choose an arbitrary number of comparison variables for a graph.

    df: the DataFrame to use as the basis for the pivot table.

    y: the y variable to use within the Plotly graph.

```

(continues on next page)

(continued from previous page)

`aggfunc: the aggregate function to use within the pivot_table() call (e.g. 'mean', 'count', etc.).`

`x_vars: a list of comparison variables to be converted into an x variable. Pass an empty list in order to group all y values together.`

`color: the variable that should serve as the color argument for the Plotly graph.`

`x_vars_to_exclude: variables that will be incorporated into the pivot table (e.g. to ensure it's sorted correctly) but should not be present in the final chart.`

`(For instance, if you've added 'Season' as an x_vars entry or color variable, and its values are 'Fall', 'Winter', and 'Spring,' you may also want to add a 'Season For Sorting' column with values of 0, 1, and 2 for Fall, Winter, and Spring, respectively so that they'll appear in chronological order within your chart. These variables should also be placed in x_vars_to_exclude so that they won't make your final graph more complicated and/or cluttered than it needs to be.)`

`If a variable within x_vars_to_exclude isn't present in your x_vars list or your color argument, it will get ignored by the function and thus shouldn't cause any issues.`

`overall_data_name: When x_vars is empty, autopivot will group all data into a single row. overall_data_name specifies the name that you would like to give to this data point.`

`weight_col: A column containing group sizes that, if included, will be used to calculate weighted averages. (If weight_col is None, weighted averages will *not* be calculated.)`

`filter_tuple_list: A list of tuples that allow the DataFrame to show only a specific set of data. The first item in each tuple should be a field name, and the second item should be a list of values to show within that field name.`

`convert_x_vars_to_strings: if True, the function will convert all x variables not found in x_vars_to_exclude to strings.`

`(This can improve the appearance of any graphs that make use of the pivot table created by this function.`

`'''`

`# Creating a copy of the initial dataset in order to ensure that the # following code does not modify it:`

`df_for_pivot = df.copy()`

`# Filtering the DataFrame based on the contents (if any)`

`# of filter_tuple_list:`

`for pair in filter_tuple_list:`

`df_for_pivot.query(`

`f" `{{pair[0]}}` in {{pair[1]}}", inplace = True)`

`# Determining which x variables will appear in the final chart:`

(continues on next page)

(continued from previous page)

```

x_vars_for_chart = list(set(x_vars) - set(x_vars_to_exclude))

# Converting all x variables wthin chart to string form (if
# requested by the caller):
if convert_x_vars_to_strings == True:
    for x_var in x_vars_for_chart:
        df_for_pivot[x_var] = (
            df_for_pivot[x_var].astype('str').copy())

x_var_count = len(x_vars_for_chart)

if weight_col is not None: # In this case, weighted averages
    # will be calculated.
    # Multiplying each y value by its corresponding weight in order
    # to allow weighted averages to be calculated:
    df_for_pivot[f'{y}_*_{weight_col}'] = (
        df_for_pivot[y] * df_for_pivot[weight_col])
    # These 'y_*_weight' values will get added together during the
    # pivot table call, as will the corresponding weight column
    # values.

if x_var_count == 0: # Because no comparison variables will appear in
    # the final chart, the function will instead group all data
    # together. It will do so by creating a new column that has the
    # same value for each row.
    # This column can then serve as the index for both the pivot
    # function and the color value.
    df_for_pivot[overall_data_name] = overall_data_name

if weight_col is not None: # In this case, a weighted average
    # of all data within the table will be created.
    df_pivot = df_for_pivot.pivot_table(
        index = overall_data_name,
        values = [f'{y}_*_{weight_col}', weight_col],
        aggfunc = 'sum').reset_index()
    # Calculating the weighted average of y by dividing
    # the y_*_weight_col field by its corresponding group size:
    df_pivot[y] = (df_pivot[f'{y}_*_{weight_col}']
                    / df_pivot[weight_col])

else:
    df_pivot = df_for_pivot.pivot_table(
        index = overall_data_name, values = y,
        aggfunc = aggfunc).reset_index()
x_val_name = overall_data_name
color = overall_data_name
barmode = 'relative'
index = [overall_data_name] # This value won't be used
# within autopivot() (defined below), but it will still
# get created here in order to prevent scripts that expect it to
# be returned from crashing.

else:
    # If the color variable is also present in x_vars and more than

```

(continues on next page)

(continued from previous page)

```
# one variable is present within the set of x vars to be charted*,  
# the graph will display redundant data. Therefore,  
# the function will remove this variable from x_vars below.  
# (If the only x variable to be charted is also the color  
# variable, we won't want to remove it from x_vars; otherwise,  
# we'd end up with an empty list of x variables.)  
# *This set (defined above as x_vars_for_chart) excludes any  
# variables also present in x_vars_to_exclude. This will prevent  
# the function from removing a color variable from x_vars that  
# would have been the only variable left in x_vars following the  
# removal of the variable to exclude (which would cause the  
# function to crash).  
  
# (For example: suppose our color variable were 'Season';  
# our x_vars contents were ['Season', 'Season_for_Sorting'];  
# and our x_vars_to_exclude list were ['Season_for_Sorting']. If  
# we chose to remove the color variable from x_vars contents  
# because it contained more than one variable, we'd end up with  
# an empty x_vars list once 'Season_for_Sorting' got removed.  
# Removing this variable to exclude from our list of x vars to  
# pass to len() below will prevent this error.  
  
if (color in x_vars) and (len(x_vars_for_chart) > 1):  
    x_vars.remove(color)  
  
print("x_vars:", x_vars)  
  
# Initializing a list of variables to be passed to the 'index'  
# argument within the pivot_table call:  
index = x_vars.copy()  
  
# We'll want to make sure to include the color variable in the  
# pivot index as well so that it can be accessed by the  
# charting code.  
if color is not None and color not in index:  
    index.append(color)  
  
print("index prior to pivot_table() call:", index)  
  
if weight_col is not None: # In this case, weighted averages  
    # will be calculated.  
    df_pivot = df_for_pivot.pivot_table(  
        index = index, values = [  
            f'{y}_*_{weight_col}', weight_col],  
            aggfunc = 'sum').reset_index()  
    df_pivot[y] = (  
        df_pivot[f'{y}_*_{weight_col}'] / df_pivot[weight_col])  
  
else:  
    df_pivot = df_for_pivot.pivot_table(  
        index = index, values = y,  
        aggfunc = aggfunc).reset_index()  
  
# Now that the pivot table has been created, the variables
```

(continues on next page)

(continued from previous page)

```

# in x_vars_to_exclude can be removed from our x_vars list, our
# index, and our DataFrame, thus preventing our final
# graphs from being too cluttered.
for var_to_exclude in x_vars_to_exclude:
    if var_to_exclude in x_vars:
        x_vars.remove(var_to_exclude)
        index.remove(var_to_exclude)
    # These variables could get removed from df_pivot also,
    # but the user might find them helpful for future sorting
    # needs-- so they'll be retained for now.
    # df_pivot.drop(var_to_exclude, axis = 1, inplace = True)

    # Determining the name of the x value column by
    # joining together all of the values in x_vars that will get
    # incorporated into its own values:
x_val_name = ('/').join(x_vars)

    # Initializing the x_val_name column values (which will serve
    # as the x axis entries within Plotly charts) by
    # converting the first item within x_vars to a string,
    # then adding other string-formatted values to it:
    # (Slashes will separate these various values.)
df_pivot[x_val_name] = df_pivot[x_vars[0]].astype('str')
for i in range(1, len(x_vars)):
    df_pivot[x_val_name] = (
        df_pivot[x_val_name]
        + '/' + df_pivot[x_vars[i]].astype('str'))

    # If there are fewer than two unique variables to be graphed,
    # we'll want to set the barmode argument to 'relative' so that,
    # in the event we create a bar chart, the bars won't be far
    # apart from one another. If there are
    # two or more unique variables, we can use 'group' instead.
    # (If there's a single variable within x_vars that matches
    # the 'color' variable, we will also want to set barmode to
    # 'relative,' since only
    # one variable will actually be displayed on the graph. Therefore,
    # I added in a set() call in the following code; set() keeps only
    # unique instances of a list and will thus prevent the same
    # x_vars variable and color variable from being counted as two
    # variables within this if/else statement.

if color is not None:
    unique_graphed_vars = set(x_vars + [color])
else:
    unique_graphed_vars = set(x_vars)

if len(unique_graphed_vars) < 2:
    barmode = 'relative'
else:
    barmode = 'group'

return df_pivot, x_val_name, y, color, barmode, x_var_count, \
index, aggfunc

```

(continues on next page)

(continued from previous page)

```
def autopobar(df_pivot, x_val_name, y, color, barmode, x_var_count,
              index, aggfunc, custom_aggfunc_name = None,
              text_auto = '.2f'):
    '''This function creates a bar graph of a pivot table (such as one
    created within autopivot()). Most arguments arguments for this function
    correspond to the values returned by autopivot(); more information on
    each can be found within that function.

    custom_aggfunc_name: A string to use in place of the aggregate
    function name in the chart title. (The general chart title format
    created by this code will be (aggfunc + y + 'by' + x_val_name'.
    if custom_aggfunc_name is present, it will take the place of
    aggfunc within the title. Note that '' can be passed to
    this parameter in order to exclude the aggregate function
    from chart titles.)

    text_auto: The value to pass to the text_auto argument of px.bar(),
    which specifies whether (and how) to display data labels on bars.
    Set this variable to False if you wish not to include any labels;
    set to '.2f' to show labels up to 2 decimal points; set to '.0f'
    to show labels as integers; and '.1%' to show labels in percentage
    form with a single decimal point. These are just some of the many
    options you can pass for this argument.

    '''

    # Creating a title for the chart:
    # Suppose our y value is 'Score' and our aggregate function is 'mean.'
    # If our original x variable count was 0, our title can simply be
    # 'Overall Mean Score.' If we had just one graph variable ('College')
    # and no color variable, our title could be 'Mean Score by College.'
    # If we also had a 'Level' color variable, our title
    # could be 'Mean Score by College and Level'. Finally, if we added
    # another x variable ('Season' to our list, our mean title could be
    # 'Mean Score by College, Season, and Level.'
    # The following code includes four title definitions to cover
    # these four scenarios. (Note that 'index' is used rather than
    # 'x_vars' because the former variable includes both our x variables
    # and (if present) our color variable, and both of these should be
    # incorporated into the title.

    # Determining how to represent the aggregate function within
    # titles:
    # Since these function names will immediately precede the chart's
    # y value some names will fit better than others. 'mean' and
    # 'median' can precede y value names without any trouble; however,
    # 'sum' and 'count' can't. (For example, 'Mean Score' works fine,
    # but 'Sum Students' or 'Count Students' isn't grammatically correct.)
    # Therefore, the following code replaces 'count' and 'sum' within
    # titles with 'Total'; other aggregate function names are left
    # in place.

    if custom_aggfunc_name is not None:
        aggfunc_name = custom_aggfunc_name
    else:
        if aggfunc in ['count', 'sum']:
```

(continues on next page)

(continued from previous page)

```

        aggfunc_name = 'Total'
    else:
        aggfunc_name = aggfunc

    if x_var_count == 0:
        # The caller may have set aggfunc_name to '' in order to
        # exclude the aggregate function from display. In this case,
        # aggfunc_name shouldn't be included within the plot_title
        # definition, as doing so would add an extra space
        # to the title.
        if len(aggfunc_name) == 0:
            plot_title = f"Overall {y}"
        else:
            plot_title = f"Overall {aggfunc_name.title()} {y}"
    elif len(index) == 1:
        plot_title = f"{aggfunc_name.title()} {y} by {index[0]}"
    elif len(index) == 2:
        plot_title = f"{aggfunc_name.title()} {y} \
by {index[0]} and {index[1]}"
    else:
        plot_title = f"{aggfunc_name.title()} {y} by {(' , ').join(
            index[0:-1])}, and {index[-1]}"

    if len(df_pivot) > 0:
        # The following code will still work if color is set to None.
        fig = px.bar(df_pivot, x = x_val_name, y = y,
                      color = color, barmode = barmode,
                      text_auto = text_auto, title = plot_title)
    else: # In this case, there's no data to plot,
          # so an empty figure will be returned instead.
        fig = px.bar(title=plot_title)

    if x_var_count == 0: # In this case, the x axis tick, x axis title,
        # and legend entry will all be the same, so we can hide two
        # of those elements.
        fig.update_layout(showlegend = False,
                          xaxis_title = None)
    return fig

def autotable(df_pivot):
    '''This function converts a DataFrame into a Graph Objects-based
    Table.'''
    # For more details about go.Table objects, see:
    # https://plotly.com/python/table/
    # The following code was based on:
    # https://plotly.com/python/table/#use-a-pandas-dataframe
    table = go.Figure(data=[go.Table(
        header=dict(values=list(df_pivot.columns),
                    fill_color='lightgray',
                    align='left'),
        cells=dict(values = [df_pivot[column]
                            for column in df_pivot.columns],
                   fill_color='white',
                   align='left'))
    ])

```

(continues on next page)

(continued from previous page)

```
return table

def autopivot_plus_bar(
    df, y, aggfunc, x_vars = [], color = None,
    x_vars_to_exclude = [], overall_data_name = 'All Data',
    weight_col = None, filter_tuple_list = [],
    custom_aggfunc_name = None, convert_x_vars_to_strings = True,
    create_table = False, text_auto = '.2f'):
    '''This function calls both autopivot() and autopobar(), thus
    simplifying the process of using both functions within a script.
    See autopivot() and autopobar()'s individual function definitions
    for more documentation on each.

    create_table: set to True to return a table along with the bar
    graph.'''
    df_pivot, x_val_name, y, color, barmode, x_var_count, \
    index, aggfunc = autopivot(
        df = df, y = y, aggfunc = aggfunc,
        x_vars = x_vars, color = color,
        x_vars_to_exclude = x_vars_to_exclude,
        overall_data_name = overall_data_name, weight_col = weight_col,
        filter_tuple_list = filter_tuple_list,
        convert_x_vars_to_strings=convert_x_vars_to_strings)

    fig_bar = autopobar(
        df_pivot = df_pivot, x_val_name = x_val_name, y = y,
        color = color, barmode = barmode, x_var_count = x_var_count,
        index = index, aggfunc = aggfunc,
        custom_aggfunc_name=custom_aggfunc_name,
        text_auto = text_auto)

    if create_table == False:
        return fig_bar

    else: # In this case, a tabular view of df_pivot will get
        # created via autotable(), after which the function will
        # return both fig_bar and this tabular view.
        table = autotable(df_pivot)

    return fig_bar, table
```

17.2.11 data_import.py

```

# Data import script:
# This script will import datasets that will be used by multiple pages.
# Taking care of these imports here should make the site more efficient by
# reducing the number of times each dataset needs to be imported.

offline_import = False # Allows for source data to get read in locally,
# which may improve performance. (This should be set to False prior to
# deploying this app online, however.)

import pandas as pd

def improve_col_display(df):
    '''This function replaces underscores in column names with spaces
    and also converts them to title space, thus improving their
    appearance within chart titles.
    Since some values (e.g. 'ID') are best capitalized rather than
    converted to title case, the function also includes a
    df.rename() col. This can be expanded as needed to address any
    issues caused by the title case conversion.
    '''
    df.columns = [column.replace('_', ' ').title()
                  for column in df.columns]
    df.rename(
        columns = {'Student Id':'Student ID'}, inplace = True)

if offline_import == True: # In this case, the source data will get
    # imported from a local file.
    print("Importing source data from local .csv files.")
    df_curr_enrollment = pd.read_csv(
        '../../Appendix/curr_enrollment.csv')
    df_survey_results = pd.read_csv(
        '../../Appendix/survey_results.csv')
else:
    print("Downloading source data from an online source.")
    df_curr_enrollment = pd.read_csv(
        'https://raw.githubusercontent.com/kburchfiel/\n'
        'pfn/main/Appendix/curr_enrollment.csv')
    df_survey_results = pd.read_csv('https://raw.githubusercontent.com/\n'
        'kburchfiel/pfn/main/Appendix/survey_results.csv')

print("Imported current enrollment data.") # Allows us to check how many
# times this data will get imported during the web app's operation
print("Imported survey results.")

# Adding an 'Enrollment' column (which will be useful for pivot tables)
# and chart titles:
improve_col_display(df_curr_enrollment)
df_curr_enrollment['Enrollment'] = 1

improve_col_display(df_survey_results)
df_survey_results['Count'] = 1

```

(continues on next page)

(continued from previous page)

```
# Merging our survey and enrollment data together in order to allow
# survey results to be compared by college, level, etc.:
df_survey_results_extra_data = df_survey_results.merge(
    df_curr_enrollment, on = 'Student ID', how = 'left')[
['Starting Year', 'Season', 'Score', 'Gender', 'Matriculation Year',
 'College', 'Class Of', 'Level', 'Level For Sorting']]
df_survey_results_extra_data
print("Merged enrollment and survey data together to create \
df_survey_results_extra_data.")
```

17.2.12 import_layout.py

```
# import_layout() function definition
# By Kenneth Burchfiel
# Released under the MIT license

import dash
from dash import html, dcc, callback, Output, Input
import dash_bootstrap_components as dbc

def import_layout(df, comparison_list, comparison_default,
                  color_list, color_default, filter_cols = []):
    '''This function generates comparison, color, and filter
    components of an interactive dashboard, thus reducing
    the amount of code needed to create a multi-dashboard Dash app
    and helping ensure consistency across dashboard layouts.

    df: the DataFrame from which filter options should be retrieved.

    comparison_list and color_list: a list of comparison and color
    options to pass to the Comparison Options and Color Option menus,
    respectively.

    comparison_default and color_default: the default comparison
    and color options for comparison_list and color_list, respectively.

    filter_cols: The columns in df for which to create filter dropdowns.
    '''

    df_layout = df.copy() # This step ensures that this function
    # will not make any changes to df.

    # Initializing the layout that will be returned to
    # the Dash app code:
    # This layout will include rows for comparisons and color options.
    layout = [dbc.Row([
        dbc.Col(html.H5("Comparison Options:"), lg = 2),
        dbc.Col(dcc.Dropdown(
            comparison_list,
            comparison_default, multi = True,
            id = 'comparison_options'), lg = 3),
        dbc.Col(html.H5("Color Option:"), lg = 2),
        dbc.Col(dcc.Dropdown(
```

(continues on next page)

(continued from previous page)

```

color_list, color_default,
id = 'color_option'), lg = 2)])]

# Adding filter rows for each column in filter_cols:
# Note that each filter will get added to the layout
# via an addition operation.
for filter_col in filter_cols:
    layout += [dbc.Row([
        dbc.Col(html.H5(f"#{filter_col} Filter:"), lg = 2),
        dbc.Col(
            dcc.Dropdown(df_layout[filter_col].unique(),
                         df_layout[filter_col].unique(),
                         multi = True,
                         id = f'{filter_col}_filter'), lg = 3)
    ])]

# print("Layout:", layout) # May be useful for debugging

return layout

```

17.2.13 Procfile

```
web: gunicorn app:server
```

17.2.14 requirements.txt

```

dash==3.0.0
dash-bootstrap-components==2.0.0
dash-pivottable==0.0.2
Flask==3.0.3
Flask-Login==0.6.3
gunicorn==23.0.0
pandas==2.2.3
plotly==6.0.1
Werkzeug==3.0.6

```


Part VII

Appendix

CHAPTER
EIGHTEEN

NVCU DATABASE GENERATOR

By Kenneth Burchfiel

Released under the MIT License

This script will create a SQLite database for the (fictional) Northern Virginia Catholic University that can be referenced within other sections of Python for Nonprofits. The data within this database will be fictional also.

```
from sqlalchemy import create_engine
import pandas as pd
import numpy as np
# Setting up random number generation capabilities:
rng = np.random.default_rng(2325)
# Based on https://numpy.org/doc/stable/reference/random/generator.html
# The faker library is a great tool for creating fictional
# database records. I set the locale to 'en_US' because NVCU
# is based in the United States.
from faker import Faker; fake = Faker('en_US')
from helper_funcs import config_notebook
display_type = config_notebook(display_max_columns = 7,
                                 display_max_rows = 5)
```

18.1 Connecting to our NVCU database via the SQLAlchemy library

(This code will work even if no database currently exists at the path shown below.)

```
e = create_engine('sqlite:///nvcu_db.db')
# Based on: https://docs.sqlalchemy.org/en/20/dialects/sqlite.html#pysqlite
```

```
student_count = 2**14
student_count
```

```
16384
```

18.2 Creating a current enrollment table

18.2.1 Creating lists of names

We'll use the Faker library to create equal numbers of female and male first names, then concatenate these lists to create a single list of first names.

(Note: The Faker documentation for the 'en_US' (US English) locale was a useful resource in writing this code.)

```
student_count
# It's convenient to also create a variable for the value equal
# to half of the student count. (Dividing the student count by
# 2 produces a float by default, so we'll convert this value to
# an int so that it can be used as a range within list comprehensions.)
half_student_count = int(student_count / 2)
half_student_count
```

8192

```
female_first_names = [
    fake.first_name_female()
    for i in range(half_student_count)]
male_first_names = [
    fake.first_name_male() for i in range(half_student_count)]
first_names = female_first_names + male_first_names
last_names = [
    fake.last_name() for i in range(student_count)]
```

```
# In order to make our genders match our names, we'll
# make the first half of our gender list female and the second
# half male (as the first and second halves of our first names
# list show male and female names, respectively.)
genders = (['F' for i in range(half_student_count)])
+ ['M' for i in range(half_student_count)])
```

18.2.2 Initializing our current enrollment table

```
df_curr_enrollment = pd.DataFrame(
    index = np.arange(0,student_count),
    data = {'first_name':first_names,
            'last_name':last_names,
            'gender':genders})
df_curr_enrollment
```

	first_name	last_name	gender
0	Amanda	Murphy	F
1	Terri	Washington	F
...
16382	Craig	Maxwell	M
16383	Anthony	Kelley	M

(continues on next page)

(continued from previous page)

```
[16384 rows x 3 columns]
```

Creating matriculation years:

In order to simulate increasing enrollment over time, weights were added to the `rng.choice()` call so that recent years would appear more frequently.

```
rng.choice(
    [2020, 2021, 2022, 2023],
    p=[0.20, 0.22, 0.25, 0.33], size = student_count)
```

```
array([2023, 2023, 2020, ..., 2023, 2022, 2023], shape=(16384,))
```

```
df_curr_enrollment['matriculation_year'] = rng.choice(
    [2020, 2021, 2022, 2023], p=[0.20, 0.22, 0.25, 0.33],
    size=student_count)
# https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.choice.html
df_curr_enrollment
```

	first_name	last_name	gender	matriculation_year
0	Amanda	Murphy	F	2020
1	Terri	Washington	F	2020
...
16382	Craig	Maxwell	M	2020
16383	Anthony	Kelley	M	2021

```
[16384 rows x 4 columns]
```

18.2.3 Creating student IDs

Student IDs will use the format `matriculation_year-matriculation_number`. `matriculation_number` represents the order in which students enrolled for a given year; these numbers are unique within each year, but not between years. This number can then be combined with students' matriculation years to form a unique ID.

```
# Calculating matriculation numbers by grouping the DataFrame by
# matriculation year, then assigning each student within each year a unique
# number:
# (This can be achieved via df.groupby() and df.rank(). See
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html
# and https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rank.html)
df_curr_enrollment['matriculation_number'] = df_curr_enrollment.groupby(
    'matriculation_year')['matriculation_year'].rank(
        method = 'first').astype('int')
df_curr_enrollment
```

	first_name	last_name	gender	matriculation_year	matriculation_number
0	Amanda	Murphy	F	2020	1

(continues on next page)

(continued from previous page)

1	Terri	Washington	F	2020	2
...
16382	Craig	Maxwell	M	2020	3289
16383	Anthony	Kelley	M	2021	3653

[16384 rows x 5 columns]

```
# Adding matriculation years, matriculation numbers, and hyphens together
# to create student IDs:
df_curr_enrollment['student_id'] = (
    df_curr_enrollment['matriculation_year'].astype('str')
    + '-'
    + df_curr_enrollment['matriculation_number'].astype('str'))
# Sorting the DataFrame by matriculation year and matriculation number:
# (The DataFrame could also be sorted by student_id, but because these
# numbers would be interpreted as strings, ids like 2020-10 would appear
# in front of ones like 2020-2.)
df_curr_enrollment.sort_values(
    ['matriculation_year', 'matriculation_number'], inplace = True)
df_curr_enrollment.reset_index(drop=True, inplace=True)

df_curr_enrollment
```

	first_name	last_name	gender	matriculation_year	matriculation_number	\
0	Amanda	Murphy	F	2020		1
1	Terri	Washington	F	2020		2
...
16382	Phillip	Benitez	M	2023		5442
16383	Nicholas	Wheeler	M	2023		5443

	student_id
0	2020-1
1	2020-2
...	...
16382	2023-5442
16383	2023-5443

[16384 rows x 6 columns]

We'll also use matriculation years as a basis for dates of birth. (These date of birth calculations aren't the most realistic, but they are good enough for use within our fictional dataset.)

```
df_curr_enrollment['birth_month'] = rng.integers(
    low = 1, high = 13, size = student_count).astype('str')
df_curr_enrollment['birth_day'] = rng.integers(
    low = 1, high = 29, size = student_count).astype('str')

# Adding leading zeros to month and date values:
for column in ['birth_month', 'birth_day']:
    df_curr_enrollment[column] = df_curr_enrollment[column].str.zfill(2)

# In order to prevent a non-existent date (e.g. February 31) from getting
# stored in our results, we'll limit our birthday integers to the
# range [1, 28].
```

(continues on next page)

(continued from previous page)

```

df_curr_enrollment['birth_year'] = (df_curr_enrollment[
    'matriculation_year'] - 18).astype('str')

# Combining these columns together to create date of birth values in
# YYYY-MM-DD format:
df_curr_enrollment['date_of_birth'] = (
    df_curr_enrollment['birth_year'] + '-'
    + df_curr_enrollment['birth_month'] + '-'
    + df_curr_enrollment['birth_day'])

# Now that we have our date_of_birth column in place, we no longer need
# the columns that stored individual parts of those birthdays:
df_curr_enrollment.drop(
    ['birth_month', 'birth_day',
     'birth_year'], axis = 1, inplace = True)

df_curr_enrollment

```

	first_name	last_name	gender	matriculation_year	matriculation_number	\
0	Amanda	Murphy	F	2020		1
1	Terri	Washington	F	2020		2
...
16382	Phillip	Benitez	M	2023		5442
16383	Nicholas	Wheeler	M	2023		5443
	student_id	date_of_birth				
0	2020-1	2002-12-16				
1	2020-2	2002-09-26				
...				
16382	2023-5442	2005-11-03				
16383	2023-5443	2005-04-05				

[16384 rows x 7 columns]

18.2.4 Assigning students to different colleges

(I used Wikipedia's 'List of patron saints by occupation and activity' page to determine the saint after which each college would be named.)

NVCU has four different colleges:

1. St. Luke's, a humanities college. (St. Luke is one of the patron saints of artists.) Abbreviation: STL
2. St. Benedict's, a STEM college. (St. Benedict is one of the patron saints of engineers.) Abbreviation: STB
3. St. Matthew's, a business college. (St. Matthew is the patron saint of accountants.) Abbreviation: STM
4. St. Catherine's, a health college. (St. Catherine of Alexandria is one of the patron saints of nurses.) Abbreviation: STC

We can use np.choice to assign students to different colleges. However, to make the data more interesting, we'll have one college (STL) increase in popularity over time; another (STC) decrease in popularity; and the two remaining colleges remain roughly constant in popularity. We can simulate these changes by (1) creating filtered versions of the DataFrame for each year; (2) calling np.choice() with different probability sets for each year in order to create the 'college' column; and (3) recreating df_curr_enrollment by adding these year-specific DataFrames back together.

```

df_list = []
for year in df_curr_enrollment['matriculation_year'].unique():
    print(f"Now adding in college enrollments for {year}.")
    df = df_curr_enrollment.query("matriculation_year == @year").copy()
    if year == 2020:
        probabilities = [0.15, 0.25, 0.3, 0.3]
    elif year == 2021:
        probabilities = [0.19, 0.26, 0.29, 0.26]
    elif year == 2022:
        probabilities = [0.25, 0.27, 0.27, 0.21]
    elif year == 2023:
        probabilities = [0.27, 0.25, 0.31, 0.17]
    else:
        raise ValueError(
            f"A probability list needs to be added in for {year}.")
    df['college'] = rng.choice(
        ['STL', 'STB', 'STM', 'STC'],
        p = probabilities, size = len(df))
    df_list.append(df)
df_curr_enrollment = pd.concat([df for df in df_list])

```

```

Now adding in college enrollments for 2020.
Now adding in college enrollments for 2021.
Now adding in college enrollments for 2022.
Now adding in college enrollments for 2023.

```

The commented-out cell below shows an alternative approach to assigning colleges to each student. Because it iterates through each row in the DataFrame, it took 2.25 seconds to run on my laptop versus 0.016 seconds for the method shown above; in other words, the method in the previous cell was 133 times faster.

```

# df_curr_enrollment['college'] = ''
# college_col = df_curr_enrollment.columns.get_loc('college')
# for i in range(len(df_curr_enrollment)):
#     year = df_curr_enrollment.iloc[i]['matriculation_year']
#     if year == 2020:
#         probabilities = [0.15, 0.25, 0.3, 0.3]
#     elif year == 2021:
#         probabilities = [0.19, 0.26, 0.29, 0.26]
#     elif year == 2022:
#         probabilities = [0.25, 0.27, 0.27, 0.21]
#     elif year == 2022:
#         probabilities = [0.27, 0.25, 0.31, 0.17]
#     df_curr_enrollment.iloc[i, college_col] = rng.choice(
#         ['STL', 'STB', 'STM', 'STC'], p = probabilities)

```

```
df_curr_enrollment
```

	first_name	last_name	gender	...	student_id	date_of_birth	college
0	Amanda	Murphy	F	...	2020-1	2002-12-16	STC
1	Terri	Washington	F	...	2020-2	2002-09-26	STM
...
16382	Phillip	Benitez	M	...	2023-5442	2005-11-03	STB
16383	Nicholas	Wheeler	M	...	2023-5443	2005-04-05	STM

[16384 rows x 8 columns]

```
## Assigning additional year-related values:
```

```
df_curr_enrollment['class_of'] = df_curr_enrollment[
    'matriculation_year'] + 4
# The earlier the matriculation year, the higher the student's current
# level.
df_curr_enrollment['level'] = df_curr_enrollment[
    'matriculation_year'].map(
    {2020:'Se',2021:'Ju',
     2022:'So',2023:'Fr'}) # Fr, So, Ju, and Se stand for
# Freshman, Sophomore, Junior, and Senior, respectively.
# Creating an integer-based equivalent to 'level':
df_curr_enrollment['level_for_sorting'] = (
    df_curr_enrollment['matriculation_year'] - 2023) * -1
df_curr_enrollment
```

	first_name	last_name	gender	...	class_of	level	level_for_sorting
0	Amanda	Murphy	F	...	2024	Se	3
1	Terri	Washington	F	...	2024	Se	3
...
16382	Phillip	Benitez	M	...	2027	Fr	0
16383	Nicholas	Wheeler	M	...	2027	Fr	0

[16384 rows x 11 columns]

18.2.5 Saving this table to our NVCU database

(This operation will also create our database file (e.g. nvcu_db.db, the path specified when we first created our database engine) if it did not exist already.)

```
df_curr_enrollment.to_sql(
    'curr_enrollment',
    con=e, if_exists='replace', index=False)
```

16384

To demonstrate that the above operation was successful, we can read in a copy of this table via pd.read_sql():

```
pd.read_sql('curr_enrollment', con = e)
```

	first_name	last_name	gender	...	class_of	level	level_for_sorting
0	Amanda	Murphy	F	...	2024	Se	3
1	Terri	Washington	F	...	2024	Se	3
...
16382	Phillip	Benitez	M	...	2027	Fr	0
16383	Nicholas	Wheeler	M	...	2027	Fr	0

[16384 rows x 11 columns]

This same table can also be saved as a standalone .csv file (thus making it easier to examine via a spreadsheet editor):

```
df_curr_enrollment.to_csv('curr_enrollment.csv', index = False)
df_curr_enrollment.to_csv(
    '../Data_Retrieval/curr_enrollment_tab_separated.csv',
    index = False, sep = '\t')
```

You may also save it as an .xlsx file if you'd prefer: (Note that the openpyxl library is required for the following line to run successfully.)

```
df_curr_enrollment.to_excel(
    '../Data_Retrieval/curr_enrollment.xlsx', index = False)
```

18.3 Creating a survey results table

This table will store fall and spring student survey results for the most recent school year. We'll configure our results so that certain groups (e.g. freshmen, seniors, and students in the STB and STL colleges) report higher results over time than do others.

18.3.1 Creating fall survey results

```
def limit_results(df, column, min, max):
    '''This function restricts the values within a particular DataFrame
    column to a range provided by the user.

    df: the DataFrame to modify

    column: the column within df whose values will be restricted.

    min and max: the minimum and max values, respectively, to allow within
    this column.
    '''
    df[column] = np.where(
        df[column] < min, min, df[column])
    df[column] = np.where(
        df[column] > max, max, df[column])
```

```
# Initializing the table as a copy of selected columns from
# df_curr_enrollment:
df_fall_survey = df_curr_enrollment.copy()[
    'matriculation_year', 'matriculation_number',
    'student_id', 'college', 'level']
df_fall_survey['starting_year'] = 2023 # Other years could be stored
# within this table as well.
df_fall_survey['season'] = 'Fall'
# We'll use rng.normal(), which produces random numbers that follow a
# normal distribution, to initialize our fall results.
# See https://numpy.org/doc/stable/reference/random/generated/numpy.
# random.Generator.normal.html
df_fall_survey['score'] = rng.normal(
    loc = 70, scale = 10, size = student_count).astype('int')
# calling limit_results to restrict the scores to the range [0, 100]:
```

(continues on next page)

(continued from previous page)

```
limit_results(df_fall_survey, 'score', 0, 100)
df_fall_survey
```

	matriculation_year	matriculation_number	student_id	...	starting_year	\
0	2020		1	2020-1	...	2023
1	2020		2	2020-2	...	2023
...
16382	2023		5442	2023-5442	...	2023
16383	2023		5443	2023-5443	...	2023

	season	score
0	Fall	88
1	Fall	37
...
16382	Fall	67
16383	Fall	68

[16384 rows x 8 columns]

Confirming that the minimum and maximum ‘score’ values within df_fall_survey lie within the range [0, 100]:

```
df_fall_survey['score'].min(), df_fall_survey['score'].max()

(np.int64(30), np.int64(100))
```

18.3.2 Creating spring results

In order to create these results, we’ll first create a copy of our fall results table. Next, we’ll calculate our spring results by creating a score_change variable that defaults to 10; adjusting this variable up or down based on students’ college membership and level; and then adding the product of this variable and rng.random()* to a student’s fall score.

We’ll use .iloc[] to make these updates. An alternative would be df.at[], but the approach shown in this cell allows for different index labels to be used.

*rng.random() produces a float in the range [0, 1). See [the numpy documentation](#) for more details. The use of rng.random() allows for more variation in individual fall-to-spring changes.

```
df_spring_survey = df_fall_survey.copy().rename(
    columns={'score':'fall_score'}).replace('Fall', 'Spring')
df_spring_survey['score'] = np.nan
score_col = df_spring_survey.columns.get_loc('score')
for i in range(len(df_spring_survey)):
    score_change = 10
    college = df_spring_survey.iloc[i]['college']
    level = df_spring_survey.iloc[i]['level']
    if college in ['STL', 'STM']:
        score_change += 20
    if level in ['So', 'Ju']:
        score_change -= 10
    # Using .iloc to make our updates:
    # This approach prevents 'setting with copy' warnings from being
    # displayed. Note the use of get_loc to determine which column index
    # value to incorporate into .iloc.
```

(continues on next page)

(continued from previous page)

```
df_spring_survey.iloc[i, score_col] = (df_spring_survey.iloc[i]['fall_score'] - 5 + rng.random() * score_change)

limit_results(df_spring_survey, 'score', 0, 100)
df_spring_survey['score'] = df_spring_survey['score'].astype('int')
df_spring_survey.head()
```

	matriculation_year	matriculation_number	student_id	...	season	\
0	2020		1	2020-1	...	Spring
1	2020		2	2020-2	...	Spring
2	2020		3	2020-3	...	Spring
3	2020		4	2020-4	...	Spring
4	2020		5	2020-5	...	Spring

	fall_score	score
0	88	86
1	37	35
2	54	54
3	56	57
4	77	74

[5 rows x 9 columns]

A pivot table of results by college and level reveals the variation in mean scores created by the above for loop.

```
df_spring_survey.pivot_table(
    index = ['college', 'level'],
    values = 'score', aggfunc = 'mean').reset_index()
```

	college	level	score
0	STB	Fr	69.177235
1	STB	Ju	64.950769
..
14	STM	Se	79.309831
15	STM	So	73.415690

[16 rows x 3 columns]

18.3.3 Combining fall and spring results into the same table

We'll use a 'long' format for this table. In other words, rather than show fall and spring results side by side for each student (an example of a 'wide'-formatted table), we'll list these results as separate rows. I consider this setup to be more realistic: in real life, this table would likely be created by stacking individual sets of survey results on top of one another.

The only identifier that we'll preserve is the student ID, as this will be sufficient for linking this dataset with any other table (such as our current enrollment dataset) that also contains this key.

```
df_survey = pd.concat([
    df_fall_survey, df_spring_survey]).reset_index()[
    ['student_id', 'starting_year', 'season', 'score']]
df_survey
```

```

        student_id  starting_year  season  score
0            2020-1       2023    Fall     88
1            2020-2       2023    Fall     37
...
32766      2023-5442      2023  Spring     64
32767      2023-5443      2023  Spring     77

[32768 rows x 4 columns]

```

```

df_survey.to_sql('survey_results', if_exists = 'replace',
                 index = False, con = e)
df_survey.to_csv('survey_results.csv', index = False)

```

18.3.4 Creating a set of winter results that can be incorporated into a data cleaning/reformatting script

In order to demonstrate how the Pandas library can be used to clean and reformat data, I'll now create a winter survey results table that contains several issues, including:

1. Column names that differ from those in the fall/spring results
2. Different data formats
3. A missing column
4. Duplicate values
5. Missing values for certain students
6. Results spread over 16 separate files (one for each school/level pair)

These issues will be addressed within the `data_cleaning_and_reformatting.ipynb` script within PFN's Data Prep section.

```

df_winter_survey = df_fall_survey.copy()
df_winter_survey['SEASON'] = 'W'
# Initializing winter survey results as lower copies of fall ones,
# then converting them to a different data type:
# (Converting the results to integers, then strings prevents the original
# decimal values from getting added to the final output.
df_winter_survey['SURVEY_SCORE'] = (
    df_winter_survey['score']
    + (rng.random(size = student_count) * -10)).astype(
        'int').astype('str')+'.0%'
df_winter_survey['STARTINGYR'] = df_winter_survey['starting_year'] - 2000
df_winter_survey['MATRICYR'] = (
    df_winter_survey['matriculation_year'] - 2000).astype('str')
df_winter_survey.rename(columns = {'matriculation_number':'MATRIC#'},
                        inplace = True)
df_winter_survey.drop(
    ['matriculation_year', 'student_id', 'starting_year',
     'season', 'score'], axis = 1, inplace = True)
# Removing 15% of students from the results via df.sample:
df_winter_survey = df_winter_survey.sample(frac=0.85).copy()
# See https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.
# DataFrame.sample.html
# using df.sample() to add duplicate records for 5% of the students present

```

(continues on next page)

(continued from previous page)

```
# within the sample:
df_winter_survey = pd.concat(
    [df_winter_survey, df_winter_survey.sample(frac=0.05)])
df_winter_survey.reset_index(drop=True, inplace=True)
df_winter_survey
```

	MATRIC#	college	level	SEASON	SURVEY_SCORE	STARTINGYR	MATRICYR
0	387	STC	Fr	W	64.0%	23	23
1	2927	STM	Se	W	42.0%	23	20
...
14620	3787	STM	So	W	50.0%	23	22
14621	62	STC	So	W	45.0%	23	22

[14622 rows x 7 columns]

Dividing these results into 16 individual datasets (one for each college/level pair), then saving those results into the Data Prep section of PFN:

(The ‘college’ and ‘level’ fields within df_winter_survey were only necessary for filtering data and creating school and level-based filenames, so they don’t need to be included in the .csv copies of the data.)

```
for college in df_winter_survey['college'].unique():
    for level in df_winter_survey['level'].unique():
        df_results = df_winter_survey.query(
            "college == @college & level == @level").copy().drop(
            ['college', 'level'], axis = 1)
        df_results.to_csv(
            f'../Data_Prep/winter_results/{college}_{level}_results.csv',
            index = False)
```

18.4 Creating a dining transactions table

I created this table in order to illustrate the ‘average of averages’ problem within descriptive_stats_part_2.ipynb. It currently shows only four items for each transaction: (1) the amount of money spent; (2) the abbreviated level (Fr for freshman, So for sophomore, etc.) of the diner; (3) the starting school year in which the transaction was made; and (4) the weekday of the transaction.

The table is designed so that younger levels will have more entries than older levels, but older levels will spend higher amounts, on average, per transaction.

```
dining_transaction_count = 22453
# Initializing the table:
# Note the use of rng.choice() to make younger-level transactions
# more common than older-level ones and mid-week transactions
# more common than weekend-ones.
# This set of code uses np.random() to initialize each transaction as a
# number between $0 and $20; however, these numbers will get adjusted
# later in this cell.
df_dining_transactions = pd.DataFrame(
    index = np.arange(dining_transaction_count),
    data = {'starting_year':2023,
            'weekday':rng.choice(
```

(continues on next page)

(continued from previous page)

```

['Su', 'Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa'],
p = [0.05, 0.16, 0.23, 0.24, 0.18, 0.11, 0.03],
    size = dining_transaction_count),
'level':rng.choice(
['Fr', 'So', 'Ju', 'Se'],
p = [0.55, 0.21, 0.15, 0.09], size = dining_transaction_count),
'base_charge':[
    rng.random() * 20 for i in range(
        dining_transaction_count)]})

# Creating a 'multiplier' column that will cause average transaction
# amounts for older levels to exceed those of younger levels:
df_dining_transactions['multiplier'] = df_dining_transactions[
'level'].map(
    {'Fr':1, 'So':1.3, 'Ju': 1.7, 'Se':2.1})
df_dining_transactions['amount'] = np.round(
    df_dining_transactions['base_charge']
    * df_dining_transactions['multiplier'], 2)

# Removing columns that are no longer needed:
df_dining_transactions.drop(
    ['base_charge', 'multiplier'], axis = 1, inplace = True)
df_dining_transactions

```

	starting_year	weekday	level	amount
0	2023	We	Fr	13.36
1	2023	Tu	Se	12.22
...
22451	2023	Sa	Fr	13.25
22452	2023	Mo	Fr	1.95

[22453 rows x 4 columns]

```

df_dining_transactions.to_sql('dining_transactions', if_exists = 'replace',
                             index = False, con = e)
df_dining_transactions.to_csv('dining_transactions.csv', index = False)

```

18.5 Creating a table of fall and spring bookstore sales by student for a given year:

(This table will play a key role in the Regressions section of Python for Nonprofits.)

We'll initialize this table as a subset of df_curr_enrollment, as certain demographic items will play a role in students' spring purchase totals.

```

df_sales = df_curr_enrollment[
['student_id', 'gender', 'college', 'level']].copy()

# Specifying an RNG seed: (I sometimes use the time of day, in HHMMSS
# format, as a basis for a seed value.)
rng = np.random.default_rng(seed = 225403)

```

(continues on next page)

(continued from previous page)

```

# Fall sales will be normally distributed for all students:
df_sales['Fall'] = rng.normal(
    loc = 80, scale = 25, size = len(df_sales))

# Spring sales will be higher than fall sales by default. The following
# code specifies these changes using a normal distribution, as I found that
# doing so helped make my regression residuals more normally distributed
# also.)
spring_change = rng.normal(loc = 11, scale = 25, size = len(df_sales))
df_sales['Spring'] = df_sales['Fall'] + spring_change
# Modifying Spring totals based on demographic components:
spring_col = df_sales.columns.get_loc('Spring')
for i in range(len(df_sales)):
    # Unhealthy snacks were removed from the checkout aisle;
    # this ended up reducing revenue among freshmen and sophomore
    # (who particularly liked these snacks.)
    if df_sales.iloc[i]['level'] in ['Fr', 'So']:
        df_sales.iloc[i, spring_col] =
            df_sales.iloc[i, spring_col] + rng.normal(
                loc = -21, scale = 3))
    # An intensive marketing campaign was carried out at STM and STL;
    # if it ended up being successful, it would then be implemented
    # at the other colleges also.
    if df_sales.iloc[i]['college'] in ['STM', 'STL']:
        df_sales.iloc[i, spring_col] =
            df_sales.iloc[i, spring_col] + rng.normal(
                loc = 9, scale = 3))

# I'll leave in negative Fall and Spring values, as they could be
# explained by refunds.

df_sales['Fall'] = df_sales['Fall'].round(2)
df_sales['Spring'] = df_sales['Spring'].round(2)
df_sales['Fall_Spring_Change'] = df_sales['Spring'] - df_sales['Fall']

df_sales.to_sql('bookstore_sales', if_exists = 'replace',
                 index = False, con = e)
df_sales.to_csv('bookstore_sales.csv', index = False)

df_sales

```

	student_id	gender	college	level	Fall	Spring	Fall_Spring_Change
0	2020-1	F	STC	Se	66.80	58.24	-8.56
1	2020-2	F	STM	Se	104.67	151.90	47.23
...
16382	2023-5442	M	STB	Fr	100.52	38.29	-62.23
16383	2023-5443	M	STM	Fr	57.45	29.43	-28.02

[16384 rows x 7 columns]

That's the end of this script for now. I may add additional tables to this database in the future, including:

1. A table that stores starting incomes for graduating seniors over the last 2+ years
2. A table that compares students' college admissions test percentiles to graduate admissions test percentiles.

HELPER_FUNCS.PY

By Kenneth Burchfiel

Released under the MIT license

This notebook contains ‘helper’ functions that will prove useful for multiple sections of Python for Nonprofits.

```
# Helper functions that will prove useful for multiple sections of
# Python for Nonprofits

# By Kenneth Burchfiel

# Released under the MIT license

from IPython.display import Image # Based on
# a StackOverflow answer from 'zach' at
# https://stackoverflow.com/a/11855133/13097194 .
import pandas as pd

render_for_pdf = True # Setting this variable here, and passing it to the
# config_notebook() function defined below by default, makes it easier
# to update the output of multiple notebooks (i.e. when running
# jupyter-book code to produce either PDF or HTML versions of
# Python for Nonprofits).

def config_notebook(render_for_pdf=render_for_pdf,
                     display_max_columns = 6,
                     display_max_rows = 5,
                     debug = False):
    ...

This function helps modify several settings in order to prepare
a Jupyter notebook for either print or HTML display.

render_for_pdf: set to True to optimize notebook outputs for PDF
display. (This setting can also improve the appearance of notebooks
on GitHub. Set to False to optimize notebooks outputs for interactive
viewing. Note that, unless this value is specified by the caller,
the render_for_pdf value within helper_funcs.py will be passed
to this function.

display_max_columns and display_max_rows: the maximum number of
DataFrame columns and rows, respectively, to display when
render_for_pdf is set to True. Limiting these values can prevent
DataFrames from taking up too much space within PDF copies of
notebooks.
```

(continues on next page)

(continued from previous page)

```
debug: set to True in order to print out additional information
about the values set by this function.
'''

if render_for_pdf == True:
    pd.set_option('display.max_columns', display_max_columns)
    pd.set_option('display.max_rows', display_max_rows)
    display_type = 'png' # Will instruct a chart display function
    # to return .png versions of charts (which may show up better
    # within PDF versions of this script than would HTML copies)
else:
    display_type = 'html' # In this case, HTML versions of the charts
    # will be featured instead.
if debug == True:
    print(f"render_for_pdf: {render_for_pdf}\nmax df columns: \
{display_max_columns}\nmax df rows: {display_max_rows}\ndisplay_type: \
{display_type}")
return display_type # Returning this variable allows it to get
# passed to wadi().

def wadi(fig, file_path, height=405, aspect_ratio=16/9,
        width=None, scale=None, include_plotlyjs='cdn',
        display_type='html', display_width=720, html_path_prefix='',
        static_path_prefix='', debug=False,
        generate_image=True, display_image = True):
    '''This function saves the Plotly figure passed to 'fig' to the
    destination represented by 'file_path', then displays it as either
    an .html or .png file depending on the value of display_type. It can
    also display graphics that weren't created within Plotly (such as
    Folium maps).

    'height', 'width', and 'scale' will get passed to the parameters of
    the same name within a px.write_image() call. Don't add .png or
    .html to the end of file_path; these will get added in automatically
    by the function.

    By default, the image width will get initialized as the product
    of the image's height and aspect ratio. This makes it easier to
    increase or decrease image sizes without having to manually
    recalculate the width each time.

    Similarly, if the scale parameter is not manually set by the
    caller, it will get initialized as 2160 divided by the height.
    This will produce an image with a height of 2160 pixels by
    default (as (2160 / height) * height = 2160).

    include_plotlyjs: The argument to pass to the respective
    include_plotlyjs parameter within write_html(). The default setting
    allows for smaller HTML file sizes; however, if it's important for
    your maps to render offline, select True as your argument instead.
    For more details on these and other options, consult
    https://plotly.com/python-api-reference/generated/
```

(continues on next page)

(continued from previous page)

```

plotly.io.write_html.html .

display_type: set to .png to use IPython's Image() function to
return the static image created within the function. (The width of
this image will equal display_width.) If set to 'html', on the other
hand, an HTML rendition of the figure will be returned.
(If a different Plotly default image renderer was specified within
the notebook, the image will be displayed using that format instead.)

display_width: The width, in pixels, to display a screenshot. This
setting will only get applied if display_type is set to 'png.'

If you would like to store .html and .png images within different
folders, pass those folder names to html_path_prefix and
static_path_prefix, respectively. These names will then get added
to the beginning of copies of file_path. If you keep these values
as empty strings, file_path will get used as the save destination
for both HTML and PNG files.

debug: Set to True to print various details about the .png image
that this function creates.

generate_image: set to False in order to skip the process of
creating HTML and PNG files for the figure (e.g. because they have
already been rendered).

display_image: set to False if you do not wish to display your image
after creating it.

('wadi' stands for 'write and display image.')
'''

# Setting the folder paths for both the .html and .png copies
# of each file:
static_file_path = static_path_prefix + file_path
html_file_path = html_path_prefix + file_path
if width == None:
    width = height * aspect_ratio

if scale == None:
    scale = 2160 / height

if debug == True:
    print(f"Height: {height}, width: {width}, aspect ratio: \
{aspect_ratio}, scale: {scale}, HTML file path: {html_file_path}.html, \
static file path: {static_file_path}.png, display type: {display_type}, \
generate_image: {generate_image}")

if generate_image == True:
    # Saving .html and .png copies of the file:
    fig.write_html(html_file_path+'.html',
                   include_plotlyjs=include_plotlyjs)
    fig.write_image(static_file_path+'.png', height=height,
                   width=width, scale=scale)
if display_image == True:
    if display_type == 'png':

```

(continues on next page)

(continued from previous page)

```
    return Image(static_file_path+'.png', width=display_width)
    # Source: StackOverflow user 'zach' at
    # https://stackoverflow.com/a/11855133/13097194
else:
    return fig # The function could instead load and display the
    # .html image stored at html_file_path, but this approach is
    # simpler and should still work fine in most cases.
```