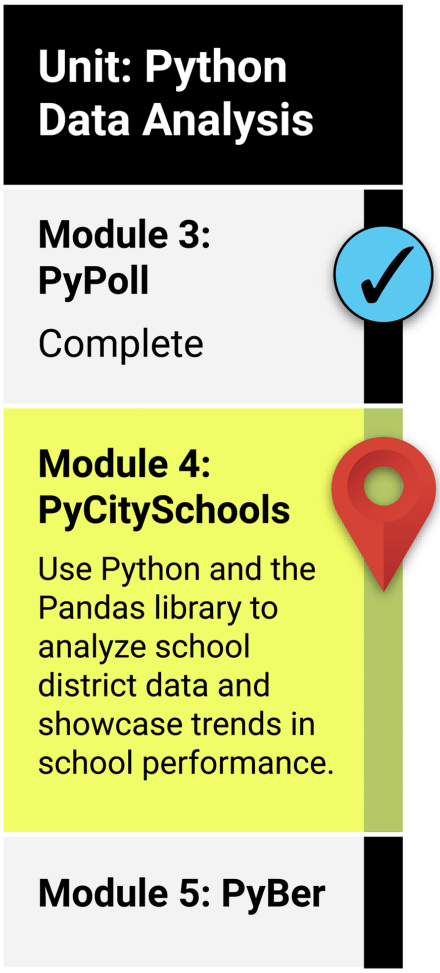
4.0.1: Using Pandas and Jupyter Notebook

4.0.2: Module 4 Roadmap

Looking Ahead

This week you'll be introduced to the open-source distribution software called Anaconda and one of its products, Jupyter Notebook. This software allows you to create documents that contain live code using Python. Jupyter Notebook supports over 40 different programming languages, but in this module, you'll use Python as you learn the basics of the Pandas library. Pandas is an open-source library that provides high-performance data analysis tools. Using Jupyter Notebook and the Pandas library, you'll read raw data from CSV files, inspect and clean data, merge datasets, perform mathematical calculations, and visualize the data with charts and graphs to tell a story.



What You Will Learn

By the end of this module, you will be able to:

- Open Jupyter Notebook files from local directories using a development environment.
- Read an external CSV file into a DataFrame.
- Format a DataFrame column.
- Determine data types of row values in a DataFrame.

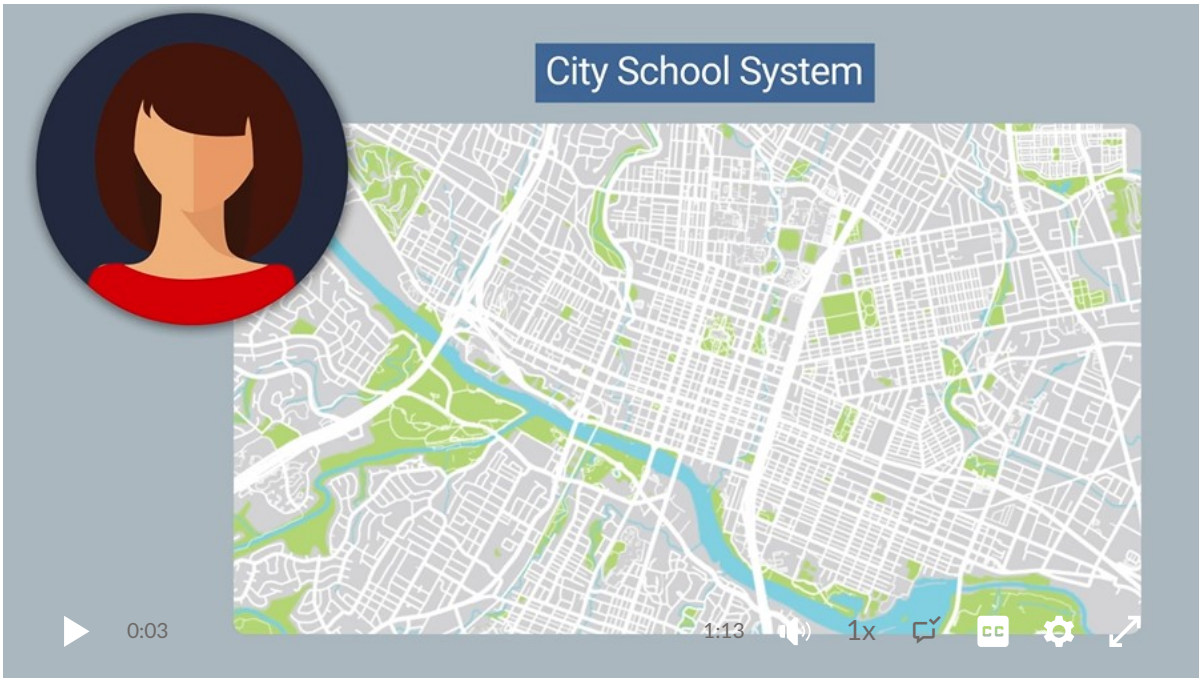
- Retrieve data from specific columns of a DataFrame.
- Merge, filter, slice, and sort a DataFrame.
- Apply the `groupby()` function to a DataFrame.
- Use multiple methods to perform a function on a DataFrame.
- Perform mathematical calculations on columns of a DataFrame or Series.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to Module 4 (30 minutes)
- Anaconda Installation and Jupyter Notebook (1 hour)
- Creating and Activating A Development Environment (30 minutes)
- Working with Jupyter Notebook and Pandas (1 hour 30 minutes)
- Convert CSV Files to a Pandas DataFrame (1 hour)
- Exploring the Data (1 hour 30 minutes)
- Verify the Clean Student Data (15 minutes)
- Generate the School District Summary (1 hour 30 minutes)
- Generate the School Summary (1 hour 30 minutes)
- High and Low Performing Schools (1 hour)
- Average Math and Reading Scores by Grade (1 hour)
- Group Scores by School Spending per Student(1 hour)
- Group Scores by School Size (1 hour)
- Group Scores by School Type (30 minutes)
- Application (5 hours)

4.0.3: PyCitySchools



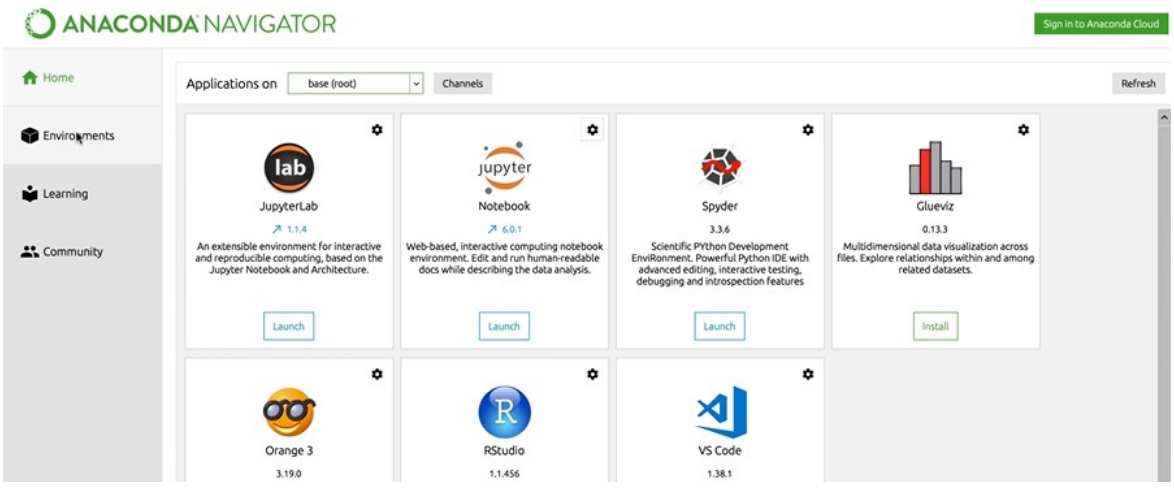
4.1.1: Overview of Anaconda

Since you're new on the job, Maria tells you that you need to download and install some software to perform this analysis. She has provided some instructions as well as video tutorials.

The analysis of school data will require you to use Anaconda, a free, open-source distribution software for over 1,500 packages suitable for Windows and macOS. One of those packages is Jupyter Notebook. When you download and install Anaconda, you will have access to Jupyter Notebook, along with many other packages. In essence, Jupyter Notebook is to Anaconda what Microsoft Word is to Microsoft Office.

Anaconda packages support the Python and R programming languages for data science, machine learning, and data processing, among other data-related tasks. We'll be using Python in this module.

Install Anaconda on macOS



Check Your Anaconda Installation

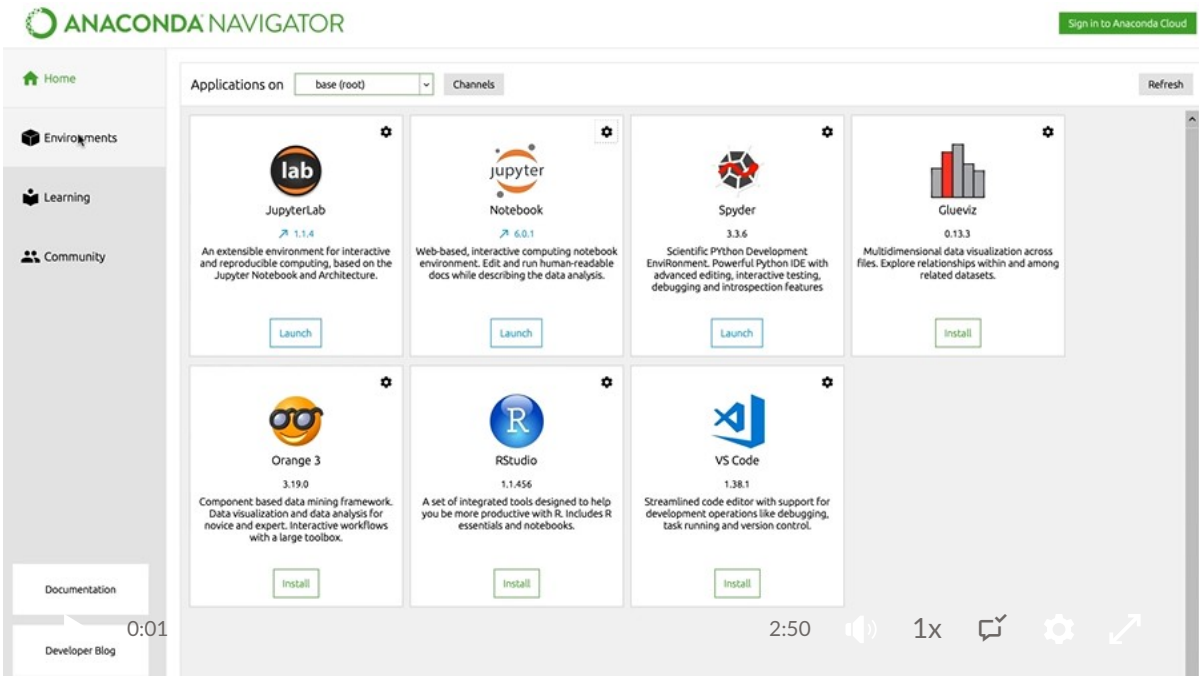
As a best practice, you should check the version of the software you installed and confirm that it's installed correctly. Let's do this now for Anaconda.

Open the command line and run the command `conda --version`. If the command line returns conda 4.7.11 or later, congratulations! The software is installed correctly.

You need to type "conda" and not "anaconda" because Anaconda packages are managed by the package management system "conda." Anaconda is the distribution management system.

Note that package versions change often, so you may need to update the packages managed by conda. To find out what version of conda you have on your computer, in the command line, type and run the following: `conda --version`. If the command line returns `conda 4.7.11` or later, you should have the latest version of conda.

Install Anaconda on Windows

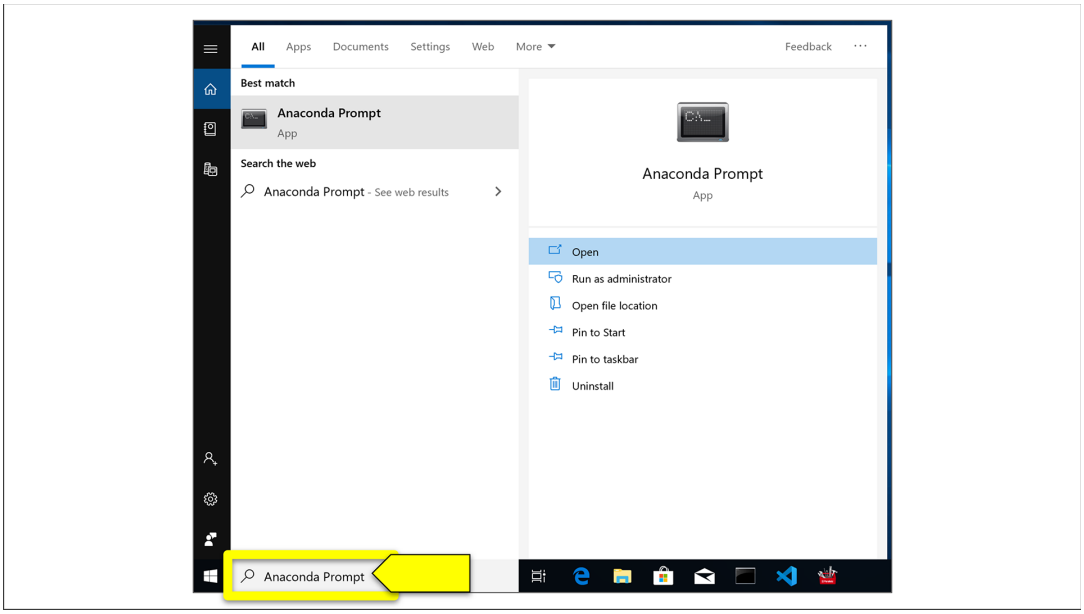


Check Your Anaconda Installation

As a best practice, you should check the version of the software you installed and confirm that it's installed correctly. On Windows, we can do this using the Anaconda Prompt (just like how we checked the version of Python with the Python Prompt).

> **Rewind**
> On Windows 10, find the application you want to open by typing the name in the search bar.

In the search bar, type the "Anaconda Prompt."



The Anaconda Prompt should look like this:

```
(base) C:\Users\your_home_directory>
```

After the prompt, `>`, type and run `conda --version`. If the prompt returns `conda 4.7.11` or later, congratulations! The software is installed correctly.

You need to type “conda” and not “anaconda” because Anaconda packages are managed by the package management system "conda." Anaconda is the distribution management system.

Note that package versions change often, so you may need to update the packages managed by conda. To find out what version of conda you have on your computer, in the command line, type and run the following: `conda --version`. If the command line returns `conda 4.7.11` or later, you should have the latest version of conda.

If you need to get the latest version of conda and other packages, perform these steps. (These steps are the same for both macOS and Windows.)

1. Type and run the `conda update conda` to update the packages distributed by Anaconda.
2. Run `conda --version` to check if conda was updated.

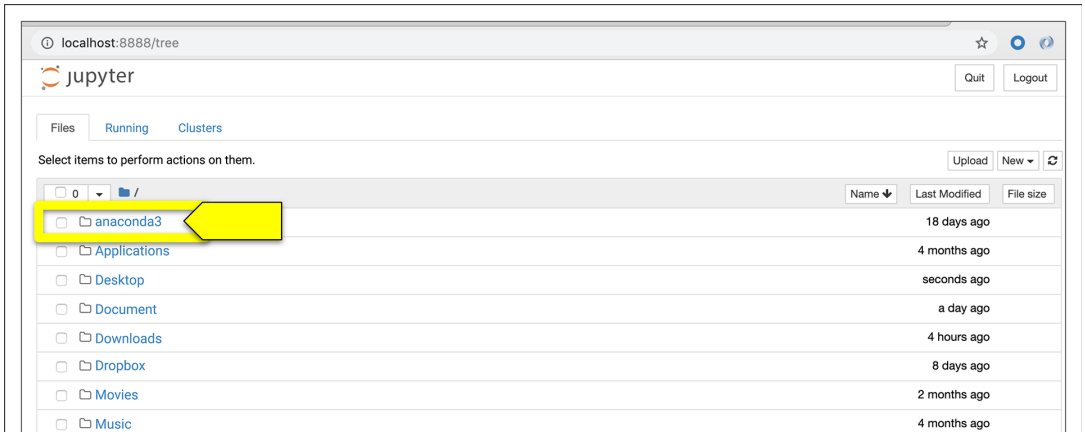
4.1.2: Introduction to Jupyter Notebook

Maria wants you to take some time to become familiar with using Jupyter Notebook and has provided a tutorial for you. Go through the tutorial now before you get started on the project.

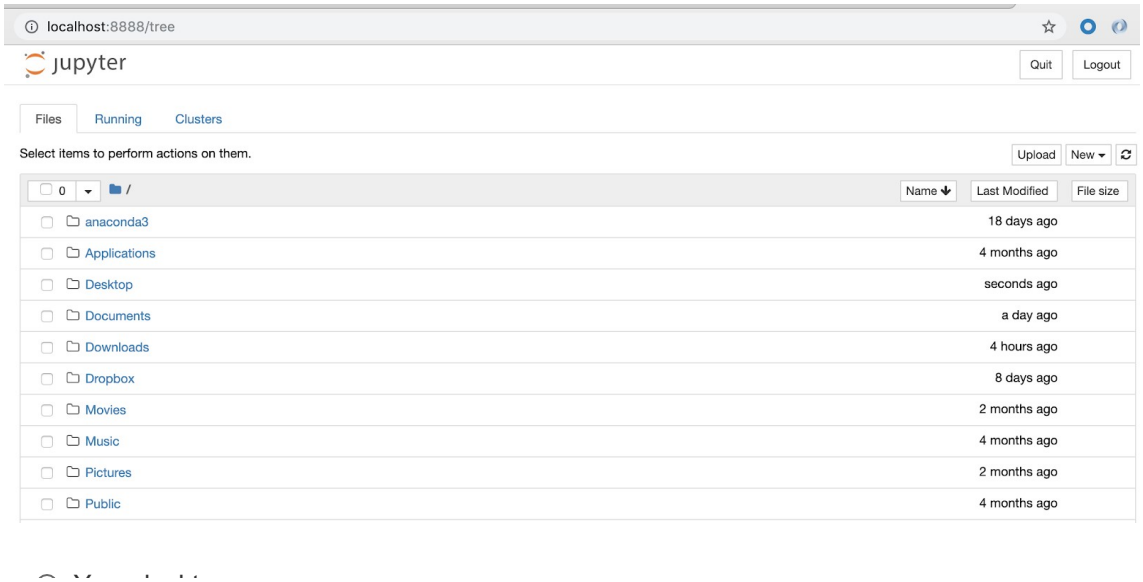
Jupyter Notebook is an open-source, web-based application that allows its users to create and share documents containing live code, equations, visualizations, and explanatory text. It combines a console, text editor, and file browser for your folders into one web-based application.

Jupyter Notebook is a popular tool for data analysts and scientists. With Jupyter Notebook, we can import data from a variety of formats, clean data, merge similar datasets, filter data based on conditionals, slice data on specific ranges, sort data based on values, and group data into similar categories. It can also be used to visualize data, write SQL queries, perform statistical analysis, and build and train machine learning models.

The following screenshot shows a folder structure similar to the one you'll see when you launch Jupyter Notebook. (We'll launch the program together later.)



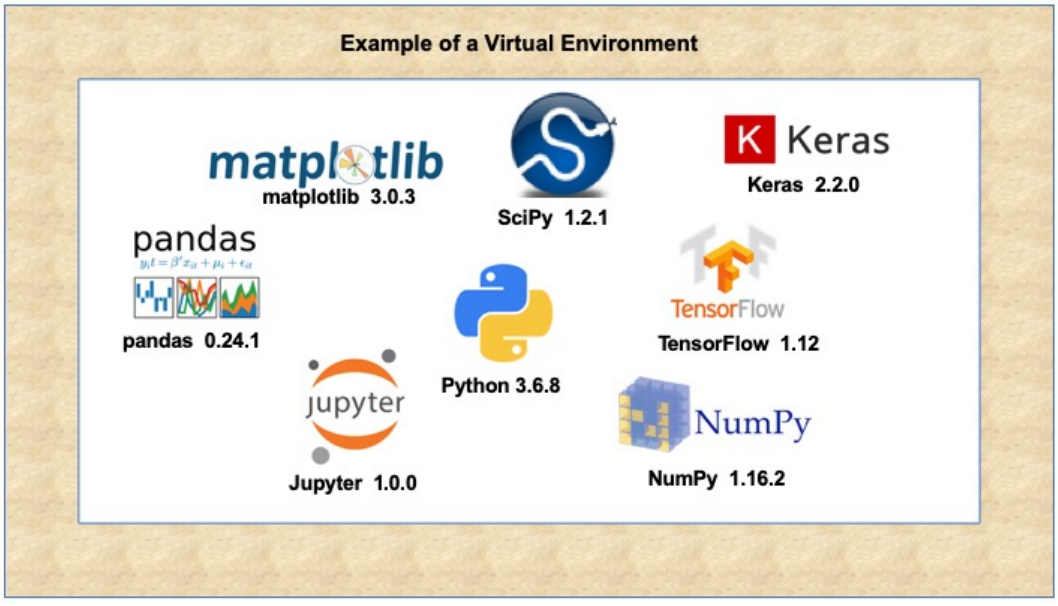
Based on the following image, when Jupyter Notebook is launched, what will you see on the start page?



4.2.1: Create Your Development Environment

Maria has asked you to create the same development environment that other analysts on the team have on their computers. This will ensure that the whole team is using the same software version and that there will not be any problems running different versions of Python or other software packages.

Creating a development environment is a common practice for programmers. A **development environment,** or **virtual environment,** is an isolated, working copy of the coding environment that programmers use to install different versions of software packages for specific projects. Some projects may require the latest versions of software, and some may require older versions.



A virtual environment is like a sandbox that contains the toys you want to play with for specific activities. We'll create a virtual environment that includes the modules and packages we need for our Python projects. Unlike a sandbox at a playground, you can leave this virtual sandbox anytime and all the toys will still be there when you come back.

In the virtual environment, we can switch or move between environments, which is called **activating the environment**. When we activate the virtual environment, we know that we have the right packages for our projects.

We're going to create a virtual environment that runs Python 3.7 and contains all the Anaconda packages that work with the latest version of Python 3.7. Note that some of these packages may be older versions than the ones you installed with Anaconda. This is because for this project, Maria's team needs to use the Anaconda packages that work with the latest version of Python 3.7.

The following videos walk you through the process of creating a virtual environment. Watch the one that corresponds to your operating system. You can also follow along with the written directions provided.

Create a Development Environment on macOS

To see all of your conda environments, type `conda info --envs` in the command line and press Enter. You should have at least two conda environments: “base” and “PythonData.”

```
# conda environments:
#
base                  /Users/<computer_name>/anaconda3
PythonData            /Users/<computer_name>/anaconda3/envs/PythonData
```

> **Note**
> For more information, read the [documentation for creating a development environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands)](https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands).

Create a Development Environment on Windows

To see all of your conda environments, type `conda info --envs` in the Anaconda Prompt and press Enter. You should have at least two conda environments: “base” and “PythonData.”

```
# conda environments:
#
base                  /Users/<computer_name>/anaconda3
PythonData            /Users/<computer_name>/anaconda3/envs/PythonData
```

> **Note**
>
> For more information, read the [documentation for creating a development environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands)](https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands) .

4.2.2: Activate Your Development Environment

Maria tells you to activate your development environment that you will use in Jupyter Notebook so that you can analyze the school data. She has provided you with the steps to activate your development environment.

Now that you have created the PythonData environment, you need to activate it. Think of this process like entering the sandbox to check out all of your toys, or tools. Or, to use a computer-based analogy, it's like clicking on the Microsoft Word icon to launch the application. Clicking the icon commands the computer to open the program so that it can be used; if you don't click the icon, the program won't open.

The process for activating the development environment is a bit different depending on whether you're using macOS or Windows. Follow the instructions that correspond to your operating system.

Activate a Development Environment on macOS

In the command line, type and run `conda activate PythonData`. This tells the computer to activate the PythonData environment.

What does the command line look like after activating your PythonData environment?

- ○ `<your computer name>:~ $`
- ○ `(PythonData) <your computer name>:~ $`
- ○ `(base) <your computer name>:~ $`

Check Answer

If you have set up your environment correctly and it's activated, you will see something like this in the command line:

```
(PythonData) your_computer_name:~ your_home_directory$
```

`(PythonData)` indicates the environment you are using.

Now let's make sure that we're using the correct version of Python in this environment. In the command line, type `python --version` and press Enter.

What version of Python do you have?

- ○ `Python 3.7.6 :: Anaconda, Inc.`
- ○ `Python 3.6.8 :: Anaconda, Inc.`
- ○ `Python 2.7.10`

Check Answer

Finish ▸

Delete a Development Environment on macOS

If you didn't create your environment correctly, you will need to remove that environment and create the correct PythonData environment.

To remove an environment from your computer, follow these steps:

1. In the command line, type `conda deactivate`.
2. Next, type and run `conda env remove --name <env_name>` to delete the environment.

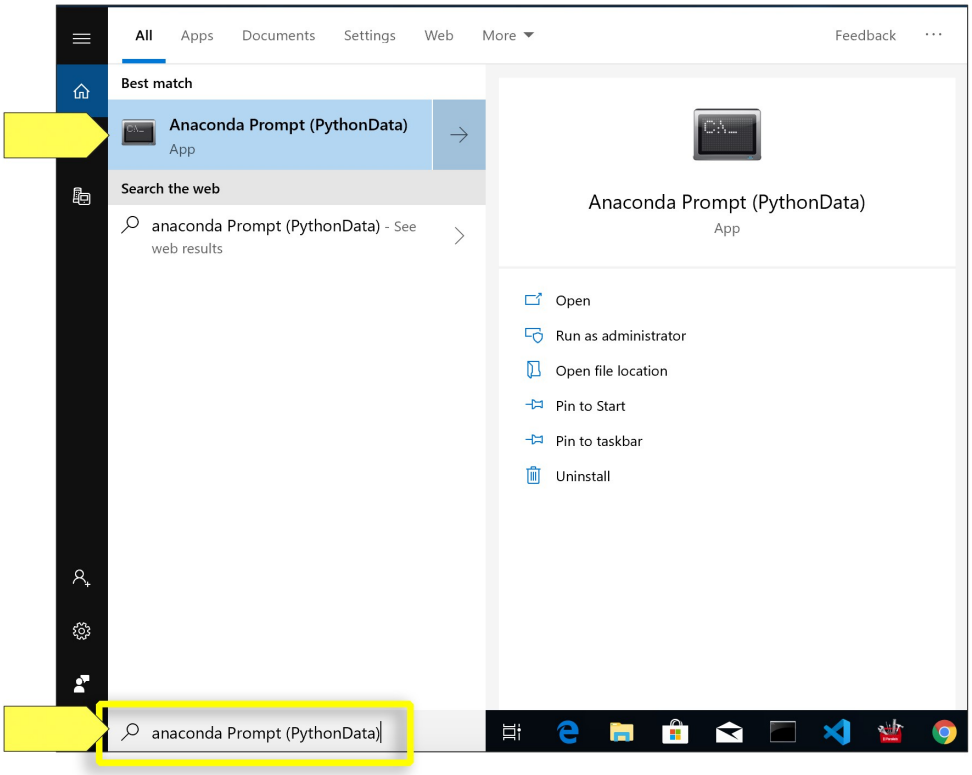
If you need to recreate the development environment, follow the steps in the previous section or watch the video.

Deactivate a Development Environment on macOS

When you want to update Anaconda on your computer, it's a best practice to deactivate your development environment before doing so. To deactivate the environment, in the command line type `conda deactivate` and press Enter.

Activate a Development Environment on Windows

When you created the PythonData environment, a new Anaconda Prompt application was created for that environment. Check your applications by typing “Anaconda Prompt” in the search menu. You will also see “Anaconda Prompt (PythonData)." Click on the app to start the PythonData environment. You can add this app to the Start menu or taskbar for quicker access.



What does the prompt look like when you open the Anaconda Prompt for your PythonData environment? (Choose the best response.)

If you have set up your environment correctly and it is activated, you will see something like this in the command line:

```
(PythonData) your_computer_name:~ your_home_directory$
```

`(PythonData)` indicates the environment you are using.

Now let's make sure that we're using the correct version of Python in this environment. In the command line, type `python --version` and press Enter.

What version of Python do you have?

- ○ `Python 3.7.6 :: Anaconda, Inc.`
- ○ `Python 3.5.7 :: Anaconda, Inc.`
- ○ `Python 3.6.8 :: Anaconda, Inc.`

Check Answer

Finish ▸

Delete a Development Environment on Windows

If you didn't create your environment correctly, you will need to remove that environment and create the correct PythonData environment.

To remove an environment from your computer, follow these steps:

1. In the Anaconda Prompt for the environment you want to delete, type `conda deactivate`.
2. Type and run `conda env remove --name <env_name>` to delete the environment.

If you need to recreate the development environment, follow the steps in the previous section or watch the video.

Deactivate a Development Environment on Windows

When you want to update Anaconda on your computer, it's a best practice to deactivate your development environment before doing so. To deactivate your environment, in the command line type `conda deactivate` and press Enter.

> **Note**
>
> For more information about the topics covered so far in this module, see the following documentation:
>
> - **Activating a development environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#activating-an-environment)**
> - **Deactivating a development environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#deactivating-an-environment)**
> - **Deleting a development environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#removing-an-environment)**

4.2.3: Create and Clone a New GitHub Repository

Maria thinks it would be a good idea to create a GitHub repository for the school analysis. This will allow her and the rest of the team to review your most recent work and provide assistance if you need it.

Before we can get started with our analysis, we need to create a place to store our important project files. We're going to create a new GitHub repository for this project and clone it on our computer. Follow these steps:

Log in to GitHub and create a new repository entitled "School_District_Analysis."

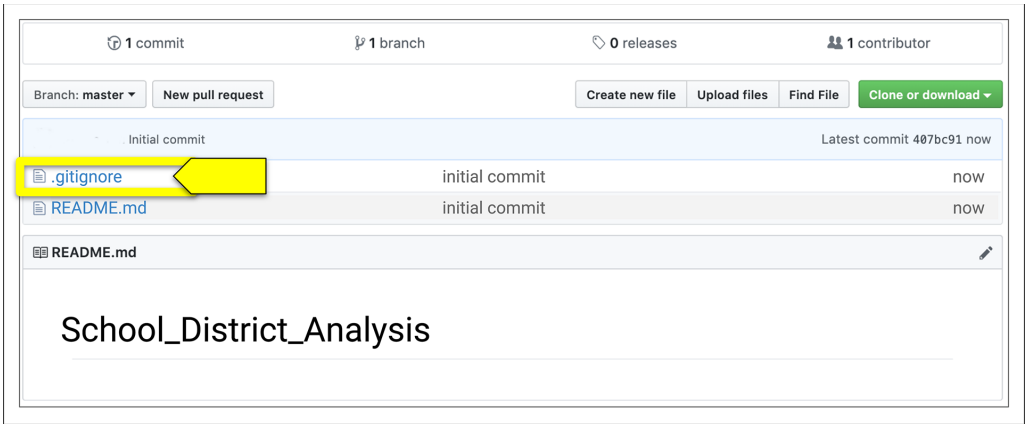
> **Rewind**
>
> To create a new repository on GitHub:
>
> 1. Add a repository name.
> 2. (Optional) Add a brief description of the repository and what type of programming software you are using for this project.
> 3. Make the repository public.
> 4. Click "Initialize this repository with a README." Each GitHub repo has a README file that serves as a kind of homepage for the repository. This is where you can add a description of your project. We will add a description at the end of this module.
> 5. Click "Add .gitignore" and type "Python."
> 6. Click the green "Create repository" box.
>
> After clicking "Create repository," you'll be on the homepage of your repository.

Before we download the repository to our computer, we need to edit the “gitignore” file by adding a file extension, `.DS_Store`.

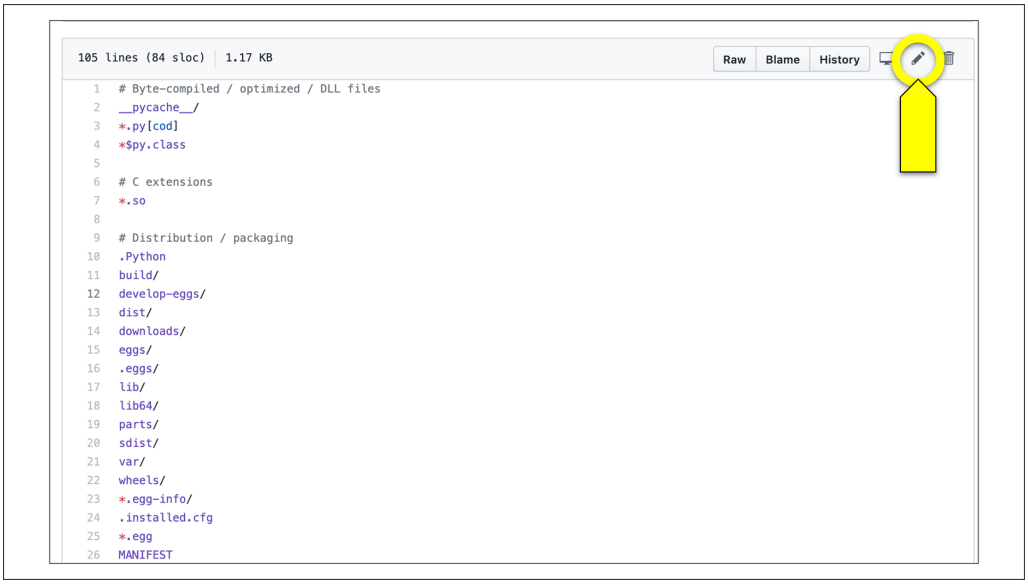
On macOS, “.DS_Store” (Desktop Services Store) is created in any directory and accessed by the Finder application. This contains information about your system configuration and settings, such as icon size and other directory metadata. If you upload these “.DS_Store” files along with other files, the files may release information about your computer.

To edit the `.gitignore` file, navigate to your School_District_Analysis repository on GitHub.

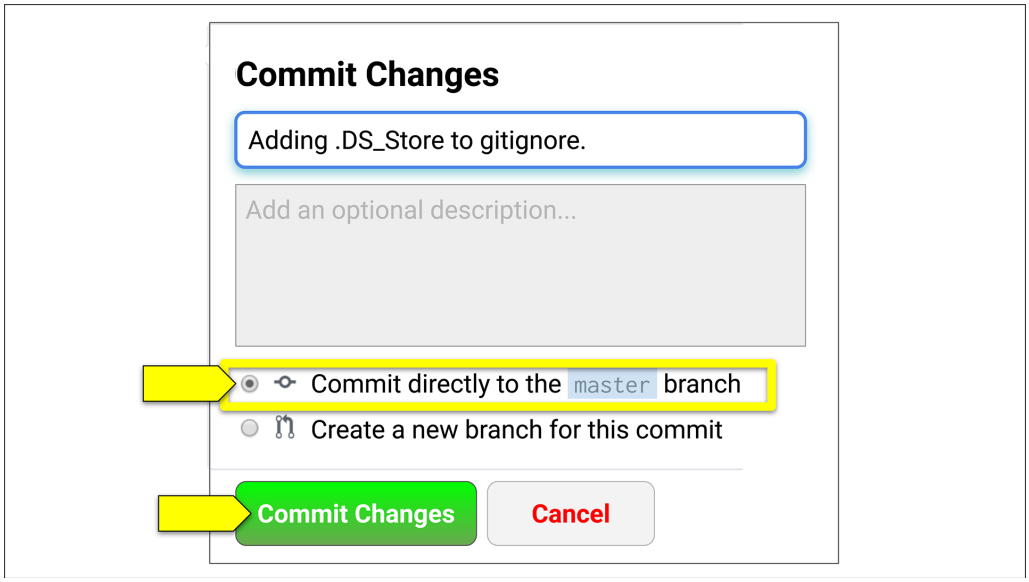
1. Click the `.gitignore` file.



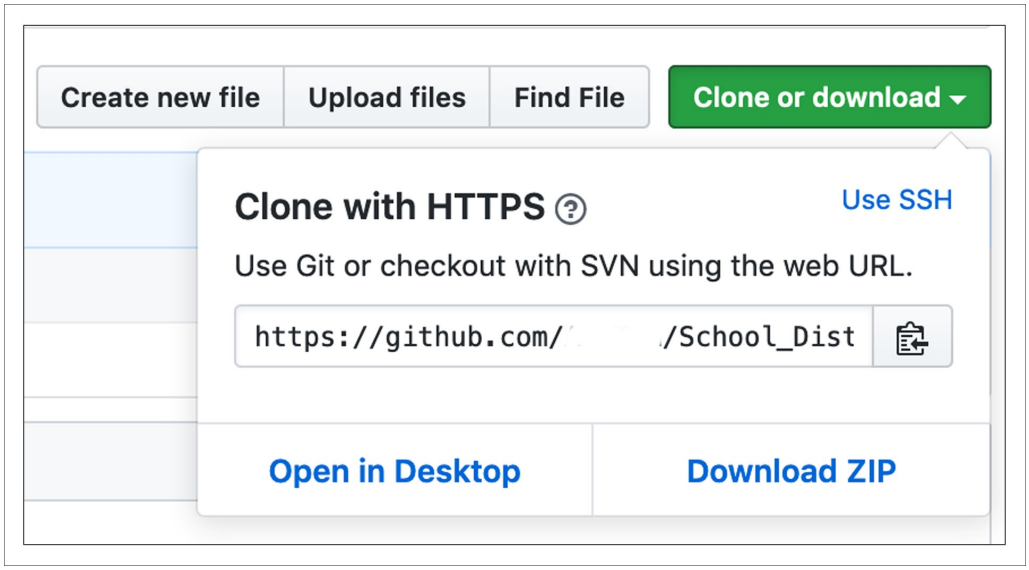
2. Click the pencil icon located in the top right.



3. On line 27, type. `# .DS_Store`.
4. On line 28, type `.DS_Store`.
5. Scroll to the bottom and add a commit message, like you would in the command line.



6. Click “Commit changes."
7. Navigate to your GitHub repository.
8. Click the green "Clone or download" box in the top right.
9. Select HTTPS and copy the URL for the repository.



To finish the cloning process, follow the set of directions that corresponds to your operating system.

Clone a Repository on macOS

1. Launch the command line.
2. Navigate to the `Class` folder.
3. In the `Class` folder, type `git clone` followed by a space, and then paste the URL that you copied previously. Your command line should look like this:

```
<computer_name>:~ Class $ git clone https://github.com/<your_GitHub_acco
```

4. Press Enter to clone the repository.
5. If prompted, enter your username and password for GitHub.

Once the repository has been cloned, you should see a folder on your computer with the same name as the repository.

Clone a Repository on Windows

1. Launch Git Bash.
2. Navigate to the `Class` folder.
3. In the `Class` folder, type `git clone` followed by a space, and then paste the URL that you copied previously. The Git Bash prompt should look like this:

```
tom@TOM MINGW32 ~/Class
$:~ Class tom$ git clone https://github.com/<your_GitHub_account>/School
```

4. Press Enter to clone the repository.
5. If prompted, enter your username and password for GitHub.

Once the repository has been cloned, you should see a folder on your computer with the same name as the repository.

4.3.1: Start Your Jupyter Notebook Server

Familiarity with Jupyter Notebook is essential for this analysis. Maria recommends checking to make sure the PythonData environment has been added to Jupyter Notebook, and then learning how to start Jupyter Notebook.

To start Jupyter Notebook, we'll be using the command line on macOS and the Anaconda Prompt on Windows. But before we can do that, we need to allow Jupyter Notebook to access our PythonData environment. Follow the set of directions below that corresponds to your operating system.

Add a Development Environment to Jupyter Notebook on macOS

In order to use the PythonData environment in Jupyter Notebook, we need to add it to your Jupyter Notebook.

In the command line, type `python -m ipykernel install --user --name PythonData` and press Enter.

If you get an error that states `No module name ipykernel`, you will have to install "ipykernel" by doing the following:

- Run the following in the command line: `python -m pip install ipykernel`.
- When ipykernel is installed, run `python -m ipykernel install --user --name PythonData` again.

If you see `Installed kernelspec PythonData in \Users\<your computer name>\AppData\Roaming\jupyter\kernels\PythonData` in your command line, you have successfully added the PythonData environment to Jupyter Notebook.

The command `python -m ipykernel install --user --name PythonData` tells Python to use the IPython kernel to install the PythonData environment in the Jupyter kernels. A **kernel** is a computer program that runs and examines the Python code. A kernel interfaces between the application and your computer memory.

When we type and run the command `python -m pip install ipykernel`, we are having the software "pip" install the `ipykernel` package, which is needed to install the PythonData environment in Jupyter Notebook.

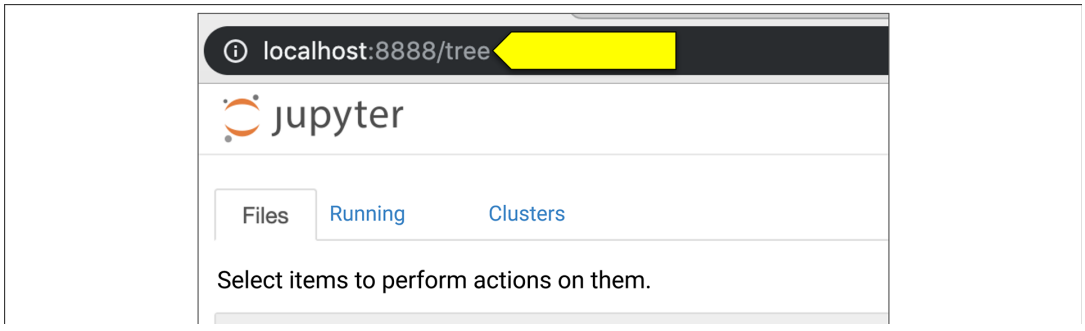
Start Jupyter Notebook on macOS

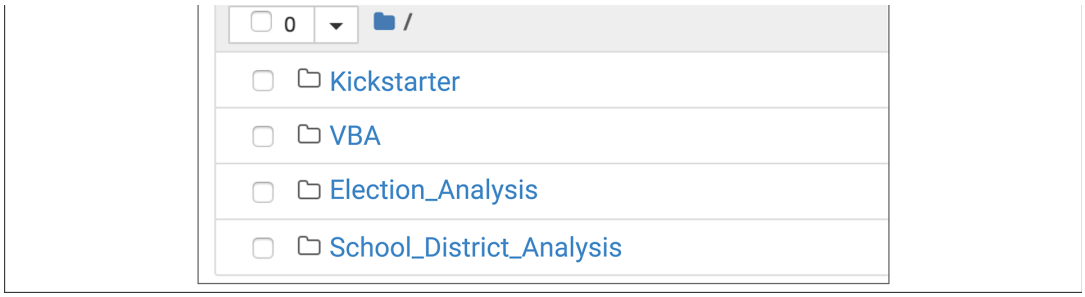
To access folders and files on Jupyter Notebook, you should use the command line to navigate to the working directory containing the files you want to work with.

For instance, to create new files in the "School_District_Analysis" folder, use the command line to navigate to the `Class` folder. Activate the PythonData environment (if it isn't already activated), and at the command prompt `$` type and run `jupyter notebook`. The command line should look something like this before you press Enter:

```
(PythonData) computer_name:Class tom$ jupyter notebook
```

When you run this command, your computer's operating system starts a Python server to "serve" up the Jupyter Notebook application and the folders in the directory where you launched Jupyter Notebook. In your default web browser, a new window will open where you'll see this URL in the address bar: http://localhost:8888/tree. (More on **localhost** later.)



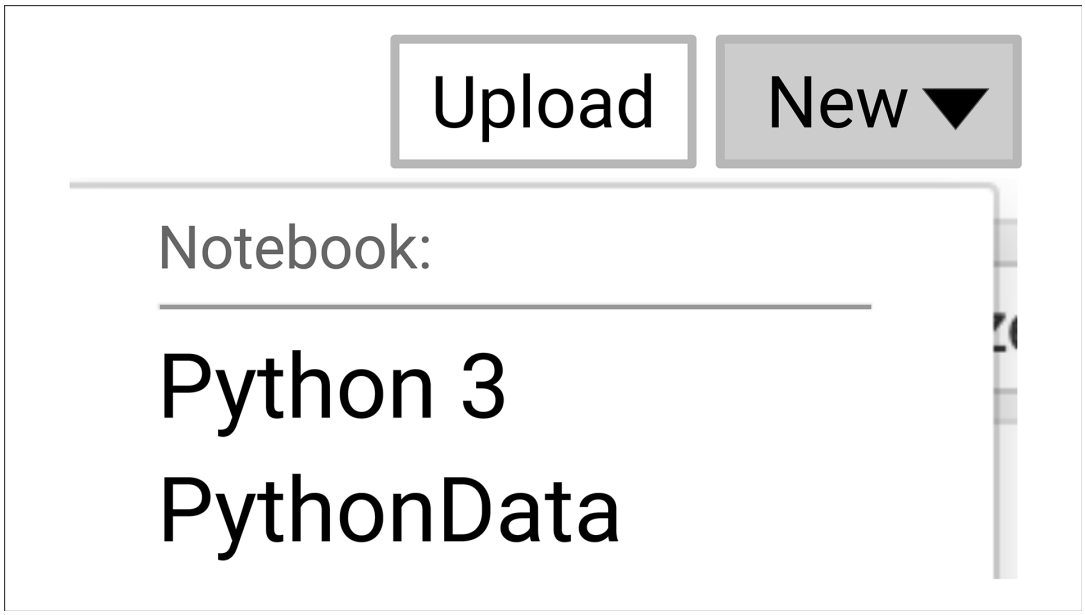


Check the PythonData Environment in Jupyter Notebook on macOS

Now that you have added your PythonData environment in the Jupyter kernels, let's determine if the PythonData environment is loaded.

In the command line, activate the PythonData environment (if it isn't already) and type `jupyter notebook`. Press Enter.

When the Jupyter Notebook webpage opens in your browser, click the "New" dropdown link, which is located on the right side of the screen. If you see the PythonData environment listed, you have successfully added it to your kernels. Note that you may also have an environment that says "Python 3," which is the default environment added during installation.



Add a Development Environment to Jupyter Notebook on Windows

In order to use the PythonData environment in Jupyter Notebook, we need to add it to your Jupyter Notebook.

Open the Anaconda Prompt associated with the environment, i.e., PythonData.

In the prompt, type `python -m ipykernel install --user --name PythonData` and press Enter. If you get an error that states `No module name ipykernel`, you will have to install "ipykernel" by doing the following:

- Run the following in the Anaconda Prompt: `python -m pip install ipykernel`.
- When ipykernel is installed, run `python -m ipykernel install --user --name PythonData` again.

If you see something like `Installed kernelspec PythonData in \Users\<your computer name>\AppData\Roaming\jupyter\kernels\PythonData` in the Anaconda Prompt, you have successfully added the PythonData environment to Jupyter Notebook.

The command `python -m ipykernel install --user --name PythonData` tells Python to use the IPython kernel to install the PythonData environment in the Jupyter kernels. A **kernel** is a computer program that runs and examines the Python code. A kernel interfaces between the application and your computer memory.

When we typed and ran the command `python -m pip install ipykernel`, we are having the software "pip" install the `ipykernel` package, which is needed to install the PythonData environment in Jupyter Notebook.

> **Note**
> "pip" is a package-management system that is used to install and manage software packages written in Python. To install a package, use the command `pip install`. Packages can be found on the [Python Package Index (https://pypi.org/)](https://pypi.org/).

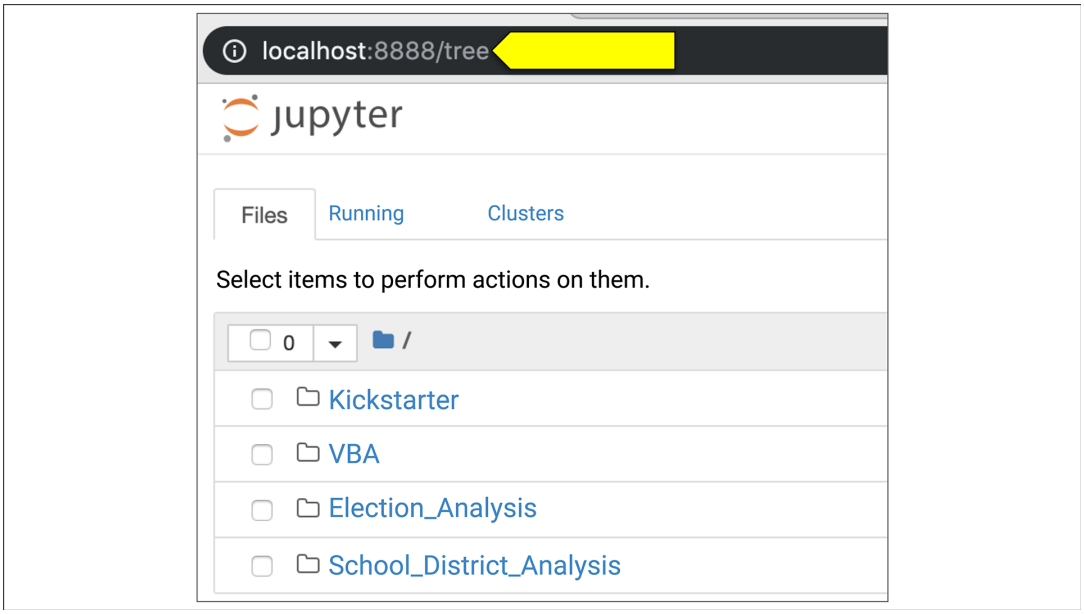
Start Jupyter Notebook on Windows

To access folders and files on Jupyter Notebook, you should use the Anaconda Prompt to navigate to the working directory containing the files you want to work with.

In the Anaconda Prompt, navigate to the Class folder. At the Anaconda prompt, `>`, type `jupyter notebook`. The prompt should look something like this before you press Enter:

```
C:\Users\computer_name\Class> jupyter notebook
```

When you run this command, your computer's operating system starts a Python server to "serve" up the Jupyter Notebook application and the folders in the directory where you launched Jupyter Notebook. In your default web browser, a new window will open where you'll see this URL in the address bar: http://localhost:8888/tree. (More on **localhost** later.)

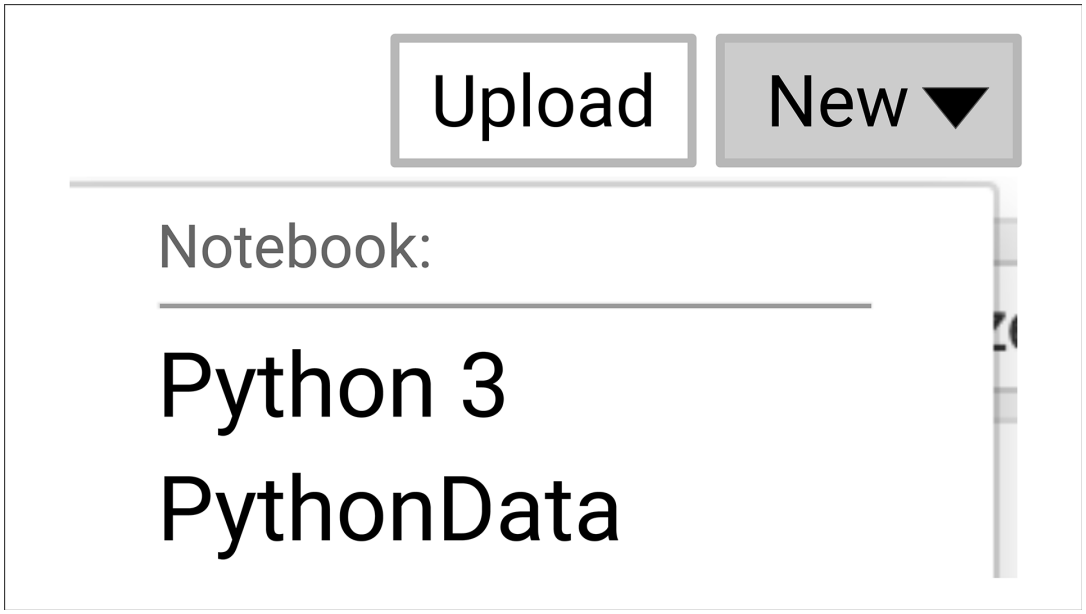


Check the PythonData Environment on Jupyter Notebook on Windows

Now that you have added your PythonData environment in the Jupyter kernels, let's determine if the PythonData environment is loaded.

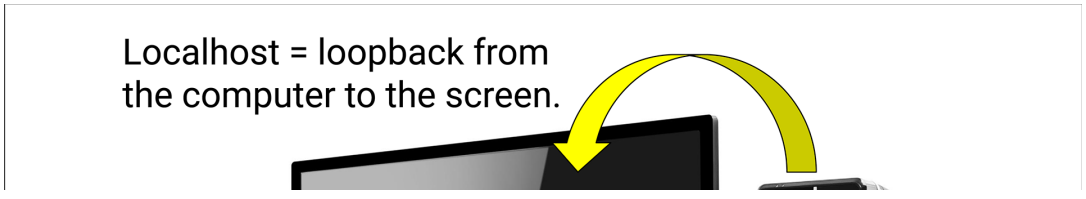
Open the Anaconda Prompt for the PythonData environment and type `jupyter notebook`. Press Enter.

When the Jupyter Notebook webpage opens in your browser, click the "New" dropdown link, which is located on the right side of the screen. If you see the PythonData environment listed, you have successfully added it to your kernels. Note that you may also have an environment that says "Python 3," which is the default environment added during installation.



What Is Localhost?

The **localhost** is your computer running a web browser on your computer, like Jupyter Notebook.



The localhost has the **internet protocol (IP)** address 127.0.0.1. Think of this as your home or street address where you live. In technical terms, this is considered a **loopback** address because the information sent to this IP address is routed back to your computer. (This is kind of like copying, or "CC-ing," yourself on an email to make sure your email was sent.)

The number 8888 is the port number on your computer where the Jupyter Notebook accesses the files and folders in your Class folder. There are 49,151 ports on your computer. The first 1,023 are system ports that have specific uses. Ports 1,024–49,151 are registered ports and are assigned to a service, such as using Jupyter Notebook.

When you are on this webpage, you can navigate to any folders in the Class folder. If you want to access a folder that is not in this tree structure, you must quit Jupyter Notebook, navigate to the folder you want on the command line, and start your Jupyter Notebook server.

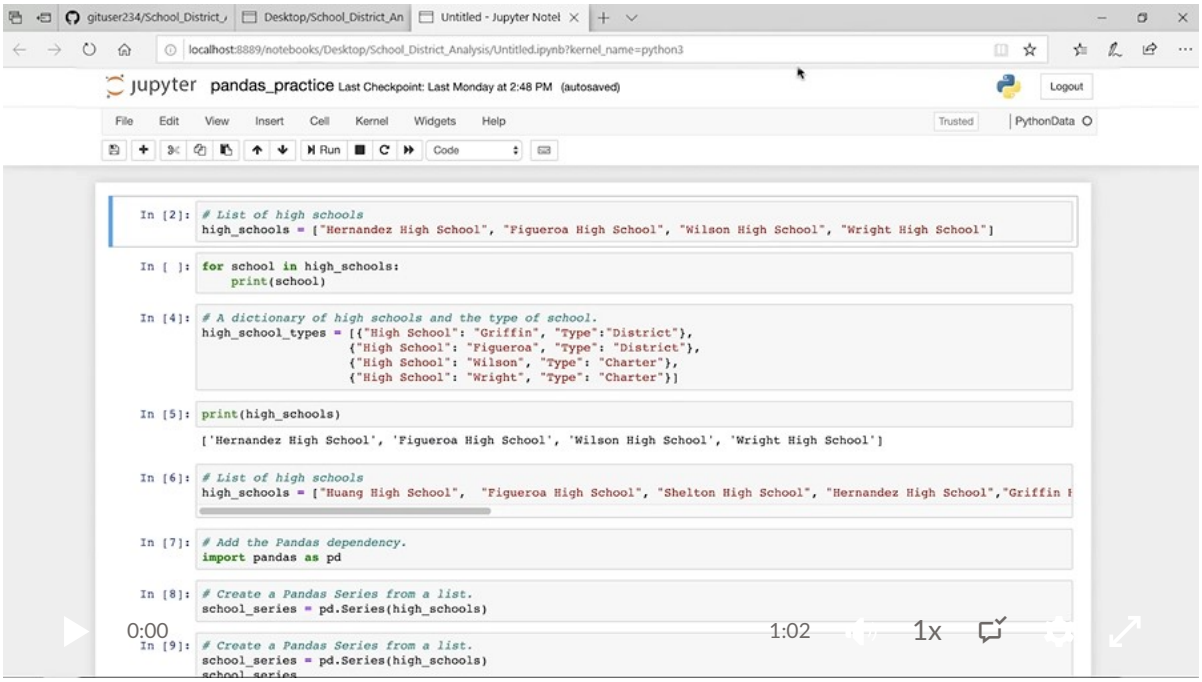
> **Note**
>
> For more information on the topics covered in this lesson, see the following documentation:
>
> - **Adding your development environment to your Jupyter Notebook (https://ipython.readthedocs.io/en/stable/install/kernel_install.html)**
> - **What is localhost? (https://whatismyipaddress.com/localhost)**
> - **What are my computer port numbers? (https://www.expressvpn.com/what-is-my-ip/port-number)**

4.3.2: Create a Jupyter Notebook File

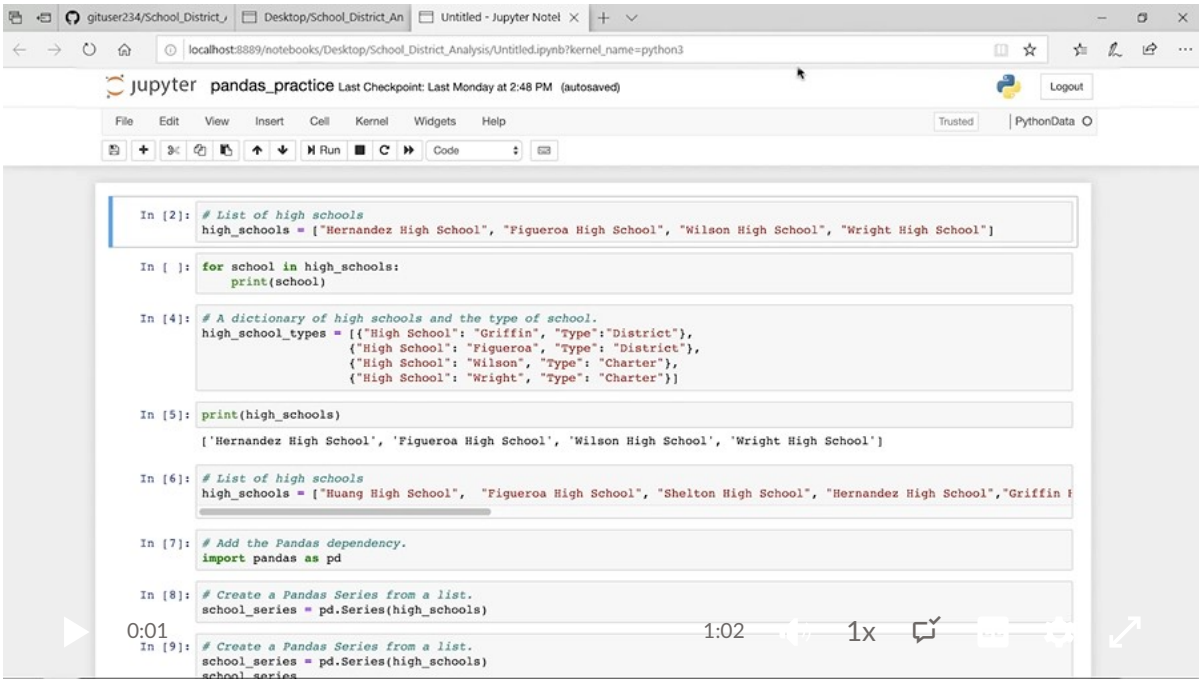
Now that you have installed all the software and have your development environment up and running in Jupyter Notebook, Maria would like you to get up to speed using Jupyter Notebook. She's provided a short manual for getting started with Jupyter Notebook.

The first thing you need to do before any code is written is to create a new file. To do this, launch Jupyter Notebook. Create a new file in the School_District_Analysis folder that uses the PythonData environment.

macOS



Windows



What file extension does the Jupyter Notebook add to a file?

- ○ `.iypnb`
- ○ `.ipy`
- ○ `.py`
- ○ `.ipynb`

Check Answer

Finish ▸

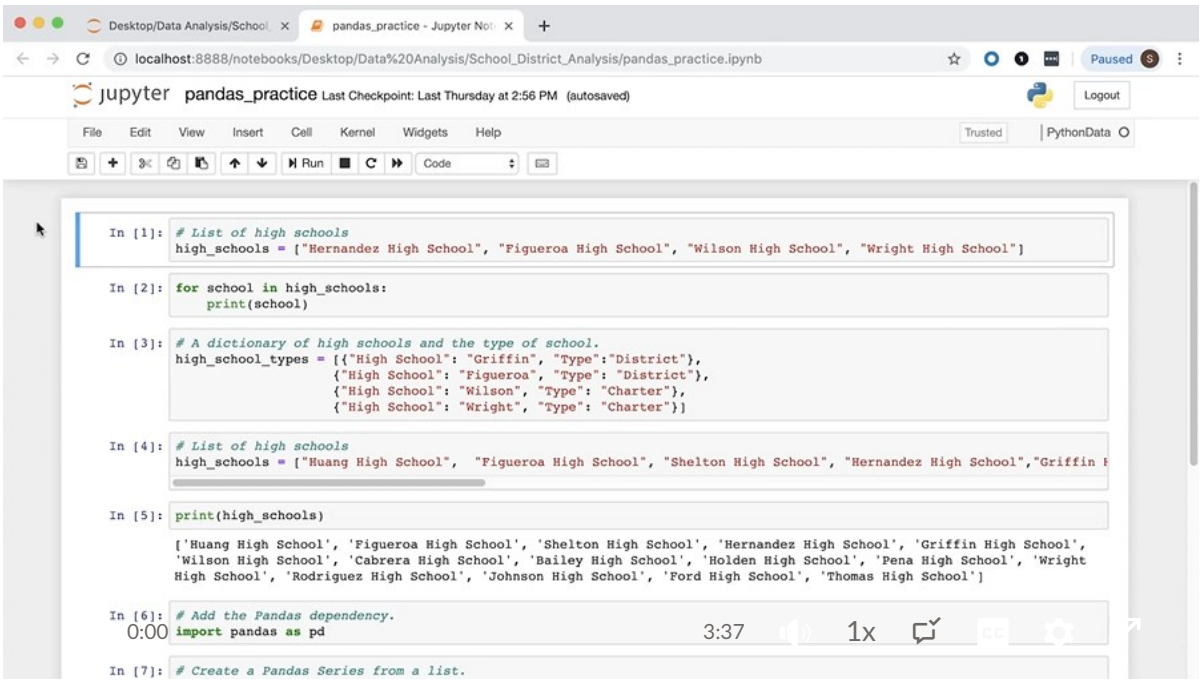
IMPORTANT!

Python files created with Jupyter Notebook are given the `.ipynb` extension rather than `.py`, so they cannot be easily read or altered within a typical text editor like VS Code.

4.3.3: Overview of the Jupyter Notebook Environment

With Jupyter Notebook open, Maria would like you to review the video tutorials on using some key features of Jupyter Notebook.

Let's get more acquainted with Jupyter Notebook by learning how to use the menu and toolbar. The following videos provide an overview of some of the features of Jupyter Notebook. You can read the **official documentation (https://jupyter-notebook.readthedocs.io/en/stable/index.html)** for more in-depth information.



IMPORTANT!

Setting the kernel for Jupyter projects is important because kernels let the program know which libraries will be available for use. Only those libraries loaded into the selected development environment can be used in a Jupyter Notebook project.

By changing the kernel, you can do which of the following?

- ○ Restart and clear the output
- ○ Interrupt the kernel
- ○ Shut down the kernel
- ○ Changes the development environment.

Check Answer

Finish ▸

4.3.4: Practice Using Jupyter Notebook

Now that you're familiar with Jupyter Notebook, Maria would like you to get hands-on experience writing Pythonic code in the Jupyter environment. She has provided a short tutorial as well as sample data so that you can practice. This way, you'll be up-and-running when she gives you the real dataset.

In order to write code, you'll need to create a new Jupyter Notebook file. If you're on a Mac computer, navigate to the School_District_Analysis folder from the command line and activate the PythonData environment. If you're on a Windows computer, launch the Anaconda Prompt for the PythonData environment and navigate to the School_District_Analysis folder.

From there, launch your Jupyter Notebook and create a new file named `pandas_practice.ipynb`, using your PythonData environment.

In the first cell, add the following list of high schools that Maria shared with us over email and run the cell.

```
# List of high schools
high_schools = ["Hernandez High School", "Figueroa High School",
                "Wilson High School","Wright High School"]
```

> **Skill Drill**
> Iterate through the list of high schools and print out each high school, as shown in the following image.

```
Hernandez High School
Figueroa High School
Wilson High School
Wright High School
```

To print out each high school using a `for` loop, use the following code:

```
for school in high_schools:
    print(school)
```

Maria has shared a list of dictionaries containing a high school and the type of high school: district or charter. Copy the code, enter it in a new cell, and then run the cell.

```
# A dictionary of high schools and the type of school.
high_school_types = [{"High School": "Griffin", "Type":"District"},
                     {"High School": "Figueroa", "Type": "District"},
                     {"High School": "Wilson", "Type": "Charter"},
                     {"High School": "Wright", "Type": "Charter"}]
```

Skill Drill

Iterate through the list of dictionaries and print out each high school and their types, as shown in the following image.

```
{'High School': 'Griffin', 'Type': 'District'}
{'High School': 'Figueroa', 'Type': 'District'}
{'High School': 'Wilson', 'Type': 'Charter'}
{'High School': 'Wright', 'Type': 'Charter'}
```

4.3.5: Overview of the Pandas Library

As you are learning this new software, Maria throws a new tool into the mix: the Pandas library. She would like you to use this library to assist you in your analysis. With the Pandas library, you can read raw Excel files, which will help you perform the analysis. Pandas has so much more to offer! So Maria wants you to learn about the Pandas library and what it can do.

Pandas is an open-source Berkeley Software Distribution (BSD)-licensed library that provides high-performance data analysis tools for the Python programming language. The Pandas library is one of the most widely preferred tools for data analysis and carries tremendous power in handling large datasets, which can slow down Excel.

It takes a lot of coding to modify and display datasets using only Python. In Jupyter Notebook, you can use the Pandas library, where raw data can be extracted from a variety of sources, cleaned, transformed, manipulated, analyzed, and visualized, all the while leaving the original dataset intact.

Additionally, with Python, you're limited when it comes to using lists, tuples, and dictionaries to manipulate the data. However, Pandas allows Python programmers the ability to work with two data types, Series and DataFrames, which are structured lists with many built-in convenience methods that allow for quick and easy manipulation of data.

Pandas Series

A Pandas **Series** is a one-dimensional, labeled array capable of holding any data type. This means the data is linear and has an index that acts as a key in a dictionary.

> **Rewind**
> A one-dimensional array is a list of objects.

An Excel file containing a list of high schools is an example of a Series. This Series behaves like a Python dictionary in that it has an index and values for each row, which is the high school name.

	A
1	Huang High School
2	Figueroa High School
3	Shelton High School
4	Hernandez High School
5	Griffin High School
6	Wilson High School
7	Cabrera High School
8	Bailey High School
9	Holden High School
10	Pena High School
11	Wright High School
12	Rodriguez High School
13	Johnson High School
14	Ford High School
15	Thomas High School

We can convert the list of high schools in Jupyter Notebook to a Pandas Series, which will allow us to get information from this list.

In the `pandas_practice.ipynb` file, create a new cell, add the list of high schools, and run the cell.

```
# List of high schools
high_schools = ["Huang High School",  "Figueroa High School", "Shelton High
```

To create a Pandas Series, we'll use the `pandas.Series()` method from the Pandas library. But in order to use this method, we first need to import the Pandas library as a dependency.

> **Rewind**

> To import a dependency, we use `import`.

In a new cell, type `import pandas as pd`. This code tells the program to import the Pandas library as the alias `pd`.

```
# Add the Pandas dependency.
import pandas as pd
```

> **IMPORTANT!**
> It's a best practice to shorten the dependency name, or give it an **alias.** This makes it easier to use the dependency in any preceding code that you write.

In the next cell, type `school_series = pd.Series(high_schools)`. This will create a Pandas Series for the list of high schools.

```
# Create a Pandas Series from a list.
school_series = pd.Series(high_schools)
```

In this code, we assign the variable `school_series` to the conversion of the list of high schools to a Pandas Series by using `pd.Series()` instead of `pandas.Series()`. This is how we use the alias `pd` in our code.

When we run the cell, there is no direct output. The Series we just created is in computer memory. To see the output, we need to retrieve it by calling the variable `school_series`, like this:

```
# Create a Pandas Series from a list.
school_series = pd.Series(high_schools)
school_series
```

When we run the cell, the output looks like it did in the Excel file, but it's formatted differently. There is a default row of index numbers, which is a sequence of incremental numbers starting from 0. In each row, there is a value for each index, which, in this case, is each high school in the list.

```
0          Huang High School
1       Figueroa High School
2        Shelton High School
3      Hernandez High School
4        Griffin High School
5         Wilson High School
6        Cabrera High School
7         Bailey High School
8         Holden High School
9           Pena High School
10        Wright High School
11     Rodriguez High School
12       Johnson High School
13          Ford High School
14        Thomas High School
dtype: object
```

Congratulations, you have created your first Pandas Series!

> **Skill Drill**
>
> Like a list in Python, indexing can be used to get specific items from a Pandas Series.
>
> Iterate through the `school_series` and print out each high school.

Pandas DataFrames

A Pandas **DataFrame** is a two-dimensional labeled data structure, like a dictionary, with rows and columns of potentially different data types such as strings, integers, and floats (decimal point numbers), where data is aligned in a table.

For example, the following image of the Excel spreadsheet is a two-dimensional labeled data structure where each row and column contains different data types.

In essence, a Pandas DataFrame contains multiple Pandas Series, or lists. Each column in the Excel file is a Series. In the previous section, we worked with a Series, a list of high schools. That Series is like column B in this Excel file.

We can convert the data in this Excel file to a Pandas DataFrame using the `pandas.DataFrame()` method from the Pandas library.

	A	B	C
1	School ID	school_name	type
2	0	Huang High School	District
3	1	Figueroa High School	District
4	2	Shelton High School	Charter
5	3	Hernandez High School	District
6	4	Griffin High School	Charter
7	5	Wilson High School	Charter
8	6	Cabrera High School	Charter
9	7	Bailey High School	District
10	8	Holden High School	Charter
11	9	Pena High School	Charter
12	10	Wright High School	Charter
13	11	Rodriguez High School	District
14	12	Johnson High School	District
15	13	Ford High School	District
16	14	Thomas High School	Charter

Convert a List of Dictionaries to DataFrame

To creat the dictionaries, in the `pandas_practice.ipynb` file, copy the code into a new cell.

```
# A dictionary of high schools
high_school_dicts = [{"School ID": 0, "school_name": "Huang High    School",
                     {"School ID": 1, "school_name": "Figueroa High School", "
                      {"School ID": 2, "school_name":"Shelton High School", "t
                      {"School ID": 3, "school_name":"Hernandez High School",
                      {"School ID": 4, "school_name":"Griffin High School", "t
```

The column headers in the Excel file are the keys in the dictionary. The value for each of the keys corresponds to a row in the Excel file.

> **Rewind**
>
> A Python dictionary has a **key** and a **value**, or **key-value pairs**.

In the next cell, we will convert the array, or list of dictionaries, to a DataFrame using `school_df = pd.DataFrame(high_school_dicts)`. In a new cell, type and run the following code:

```
school_df = pd.DataFrame(high_school_dicts)
school_df
```

This will create a Pandas DataFrame for the list of dictionaries.

	School ID	school_name	type
0	0	Huang High School	District
1	1	Figueroa High School	District
2	2	Shelton High School	Charter
3	3	Hernandez High School	District
4	4	Griffin High School	Charter

Congratulations on creating your first Pandas DataFrame!

Just like in the image of the Excel file, there are three columns with the corresponding headers “School ID,” “school_name,” and “type.” Also, like in the Excel file, there is an index for each row. However, in Python, indexing starts from 0.

> **Note**
> The `df` in `school_df` is short for “DataFrame.” It’s a best practice to add this `df` to a named DataFrame to help distinguish between DataFrames, Series, and variables.

Convert a List or Series to a DataFrame

We can also create the same DataFrame by adding each column as a list, or Series, to an empty DataFrame.

In a new cell, add the following lists and run the cell.

```
# Three separate lists of information on high schools
school_id = [0, 1, 2, 3, 4]

school_name = ["Huang High School", "Figueroa High School",
"Shelton High School", "Hernandez High School","Griffin High School"]

type_of_school = ["District", "District", "Charter", "District","Charter"]
```

Next, in a new cell, initialize an empty DataFrame, like this:

```
# Initialize a new DataFrame.
schools_df = pd.DataFrame()
```

Now we will add each list to the empty `schools_df` DataFrame by typing the column name in quotes within brackets, like we do when we create a key for a dictionary.

> **Rewind**
> The standard format for creating a key in a dictionary is to put the key in single or double quotes and inside brackets, e.g., `counties_dict["Arapahoe"] = 422829`.

Instead of using the dictionary name, we'll use the name of the DataFrame and make it equal to the list, like this:

```
# Add the list to a new DataFrame.
schools_df["School ID"] = school_id

# Print the DataFrame.
schools_df
```

Skill Drill

Add the `school_name` list as “School Name" and the `type_of_school` list as "Type" to the `schools_df` DataFrame. Then print out the `schools_df` DataFrame.

When you run the cells, your output should look like this:

	School ID	school_name	type
0	0	Huang High School	District
1	1	Figueroa High School	District
2	2	Shelton High School	Charter
3	3	Hernandez High School	District
4	4	Griffin High School	Charter

Another way to create this DataFrame is to create a Python dictionary from the three lists, like this:

```
# Create a dictionary of information on high schools.
high_schools_dict = {'School ID': school_id, 'school_name':school_name, 'typ
```

Now we can convert the `high_schools_dict` to a DataFrame.

Skill Drill

Convert the dictionary `high_schools_dict` to a DataFrame.

The Anatomy of a DataFrame

Before we move on, let's look at three main parts of a DataFrame, using the `school_df` as an example:

1. **Columns**: The top, or header, rows
2. **Index**: The numbers that run down the left-hand margin
3. **Values**: The values in the columns (the data)

See the following image for a visual representation of a DataFrame's anatomy:

We can access these three main components of a DataFrame with the `columns`, `index`, and `values` attributes.

The Columns Attribute

To get the column names of a DataFrame, use `df.columns`. Here's how you would apply the `columns` attribute to the `school_df` DataFrame:

```
school_df.columns
```

When we run this cell, the output will be a list of the column names:

```
Index(['School ID', 'school_name', 'type'], dtype='object')
```

The Index Attribute

To get the indices of the DataFrame, use `df.index`. Here's how you would apply the `index` attribute to the `school_df` DataFrame:

```
school_df.index
```

When you run this cell, the output is a `RangeIndex` that contains the first and last index value, as well as the "step=1," or how often the index increments. In this case, the index increments a value of 1 from the beginning index to the ending index.

```
RangeIndex(start=0, stop=5, step=1)
```

The Values Attribute

To get the values of the DataFrame, use `df.values`. Here's how you would apply the `values` attribute to the `school_df` DataFrame:

```
school_df.values
```

When you run this cell, the output will be an array of all the values, but without the column names:

```
array([[0, 'Huang High School', 'District'],
       [1, 'Figueroa High School', 'District'],
       [2, 'Shelton High School', 'Charter'],
       [3, 'Hernandez High School', 'District'],
       [4, 'Griffin High School', 'Charter']], dtype=object)
```

Skill Drill

Based on the following image, create a list for each column and add each list to a new Pandas DataFrame.

A	B	C
School ID	school_name	type
0	Huang High School	District
1	Figueroa High School	District

2	Shelton High School	Charter
3	Hernandez High School	District
4	Griffin High School	Charter
5	Wilson High School	Charter
6	Cabrera High School	Charter
7	Bailey High School	District
8	Holden High School	Charter
9	Pena High School	Charter
10	Wright High School	Charter
11	Rodriguez High School	District
12	Johnson High School	District
13	Ford High School	District
14	Thomas High School	Charter

Note

For more information, see the documentation on Series and DataFrames:

- **Pandas Series** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.sample.html)
- **Pandas DataFrame** (https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.DataFrame.html)

4.4.1: Import and Inspect CSV Files

Narrative Now that you have learned about the Pandas library and what it can do, Maria is going to share the datasets with you, which are in CSV format. After you import these CSV files, you will need to inspect them. Then you'll be given a set of tasks to complete for the analysis.

Import the Data

Create a folder named "Resources" in your "School_District_Analysis" folder. Click the following links to download the datasets into the Resources folder.

Download schools_complete.csv
(https://courses.bootcampspot.com/courses/138/files/15826/download?wrap=1)
(https://courses.bootcampspot.com/courses/138/files/15826/download?wrap=1)

Download students_complete.csv
(https://courses.bootcampspot.com/courses/138/files/15828/download?wrap=1)
(https://courses.bootcampspot.com/courses/138/files/15828/download?wrap=1)

You should now have two CSV files in your Resources folder:

- `schools_complete.csv`
- `students_complete.csv`

Inspect the Data

Now that the CSV files downloaded and imported into the correct folder, let's inspect the data in the files.

Rewind

Remember, when inspecting the data, focus on the following questions:

- How many columns and rows are there?
- What types of data are present?
- Is the data readable, or does it need to be converted in some way?

When we open `schools_complete.csv`, we can see that there is a top header row followed by 15 rows of data, labeled with the names "School ID," "school_name," "type," "size," and "budget." See the following image for reference.

	A	B	C	D	E
1	School ID	school_name	type	size	budget
2	0	Huang High School	District	2917	1910635
3	1	Figueroa High School	District	2949	1884411
4	2	Shelton High School	Charter	1761	1056600
5	3	Hernandez High School	District	4635	3022020
6	4	Griffin High School	Charter	1468	917500
7	5	Wilson High School	Charter	2283	1319574
8	6	Cabrera High School	Charter	1858	1081356
9	7	Bailey High School	District	4976	3124928
10	8	Holden High School	Charter	427	248087
11	9	Pena High School	Charter	962	585858
12	10	Wright High School	Charter	1800	1049400
13	11	Rodriguez High School	District	3999	2547363
14	12	Johnson High School	District	4761	3094650
15	13	Ford High School	District	2739	1763916
16	14	Thomas High School	Charter	1635	1043130

Take a few moments to read through the data. There are no misspellings, changes of case, duplicates, or special characters; the dataset looks clean! This means that this dataset will not require any cleaning once loaded into the DataFrame, which is convenient.

Next, open `students_complete.csv`. This is a large file! Here are the first 10 rows:

	A	B	C	D	E	F	G
1	Student ID	student_name	gender	grade	school_name	reading_score	math_score
2	0	Paul Bradley	M	9th	Huang High School	66	79
3	1	Victor Smith	M	12th	Huang High School	94	61
4	2	Kevin Rodriguez	M	12th	Huang High School	90	60
5	3	Dr. Richard Scott	M	12th	Huang High School	67	58
6	4	Bonnie Ray	F	9th	Huang High School	97	84
7	5	Bryan Miranda	M	9th	Huang High School	94	94
8	6	Sheena Carter	F	11th	Huang High School	82	80
9	7	Nicole Baker	F	12th	Huang High School	96	69
10	8	Michael Roth	M	10th	Huang High School	95	87

Now go to the last row of the file.

> **Rewind**
> To get to the last row of an Excel file, place the cursor in a column that doesn't have any empty cells and press Command + the down-arrow key (on a Mac) or CTRL + the down-arrow key (on Windows).

39167	39165	Donna Howard	F	12th	Thomas High School	99	90
39168	39166	Dawn Bell	F	10th	Thomas High School	95	70
39169	39167	Rebecca Tanner	F	9th	Thomas High School	73	84
39170	39168	Desiree Kidd	F	10th	Thomas High School	99	90
39171	39169	Carolyn Jackson	F	11th	Thomas High School	95	75

When we get to the last row, we see there are 39,170 rows (the first row is the header row, so it's not counted). Also, each row contains a student ID that is associated with a student's name, gender, grade in school, the name of the school they attend, and their reading and math scores. Students are grouped by school. Like `schools_complete.csv`, this file also includes the "school_name" in the header.

What anomaly do you notice about the students_complete.csv file?

- ○ There are no anomalies in this file.
- ○ Richard Scott received a 58 on his math test.
- ○ No one got a 100 on a reading or math test.
- ○ Richard Scott has the prefix "Dr." in front of his name.

Check Answer

4.4.2: Overview of the School Data Analysis Project

After you have downloaded and inspected the files, you meet with Maria to go over the tasks you'll need to complete for the analysis. There's a lot of coding that needs to be done, but she says that she'll help you if you need it. Also, after you complete each task, you'll need to add your code to a GitHub repository, which Maria will download and review to make sure you are on the right track.

Here is the list of deliverables for the analysis of the school district:

- A high-level snapshot of the district's key metrics, presented in a table format
- An overview of the key metrics for each school, presented in a table format
- Tables presenting each of the following metrics:
  - Top 5 and bottom 5 performing schools, based on the overall passing rate
  - The average math score received by students in each grade level at each school
  - The average reading score received by students in each grade level at each school
  - School performance based on the budget per student
  - School performance based on the school size
  - School performance based on the type of school

Before we can begin these tasks, we need to import the datasets into Jupyter Notebook using Python.

4.4.3: Load and Read CSV Files

One of the first things you will need to do to get started is to load the datasets into a Jupyter Notebook file. You recall how to do this using Python, but you're not sure how to load datasets using Jupyter Notebook. Say hello to your good friend Google! You're going to research how to import a CSV file into Jupyter Notebook. Maria will be available to assist when you need it.

So far, we've practiced using Jupyter Notebook as well as the Pandas library. Now we'll put these skills together to perform our analysis of school and student data.

To get started, activate the PythonData environment. If you're using a Mac, use the command line to navigate to the "School_District_Analysis" folder and activate the PythonData environment. If you're on a Windows machine, open the PythonData Anaconda Prompt for the PythonData environment and navigate to the "School_District_Analysis" folder.

Create a new Jupyter Notebook file in the "School_District_Analysis" folder and rename it `PyCitySchools.ipynb`.

Load the CSV Files

In the first cell of your `PyCitySchools.ipynb` file, import the Pandas library as the dependency and run the cell.

```
# Add the Pandas dependency.
import pandas as pd
```

In the next cell, declare two variables: one assigned to the `schools_complete.csv` file and one assigned to the `students_complete.csv` file. (These files are located in the Resources folder.) Your code should look like this:

```
# Files to load
school_data_to_load = "Resources/schools_complete.csv"
student_data_to_load = "Resources/students_complete.csv"
```

Alternatively, you can use the indirect path method to access the `schools_complete.csv` and `students_complete.csv` files.

> **Rewind**
> When we want to get the indirect path to a file, we use `os.path.join()` to load a file from somewhere in our directory.

If you decide to use this approach, you will need to import the `os` module with your Pandas dependency using the following code:

```
# Add the dependencies.
import pandas as pd
import os
```

Then, use the `os.path.join()` method to connect to the CSV files:

```
# Files to load
school_data_to_load = os.path.join("Resources", "schools_complete.csv")
student_data_to_load = os.path.join("Resources", "students_complete.csv")
```

Read the School Data File

Now we'll read each CSV file with the Pandas function `read_csv()`. Inside this function, we'll add the file we want to read, which is one of many parameters that we can add to this function.

> **Note**
>
> For more information, see the [Pandas documentation on the read_csv() function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html).

Add the following code to a new cell and run the cell. This will allow us to read `schools_complete.csv` and store it in a DataFrame.

```
# Read the school data file and store it in a Pandas DataFrame.
school_data_df = pd.read_csv(school_data_to_load)
school_data_df
```

Previously, we used the `pd.DataFrame()` function to convert a list of dictionaries, as we have in these CSV files. The `read_csv()` function makes it easier for us by converting the CSV file to a DataFrame.

Your results should look something like this:

	School ID	school_name	type	size	budget
0	0	Huang High School	District	2917	1910635
1	1	Figueroa High School	District	2949	1884411
2	2	Shelton High School	Charter	1761	1056600
3	3	Hernandez High School	District	4635	3022020
4	4	Griffin High School	Charter	1468	917500
5	5	Wilson High School	Charter	2283	1319574
6	6	Cabrera High School	Charter	1858	1081356
7	7	Bailey High School	District	4976	3124928
8	8	Holden High School	Charter	427	248087
9	9	Pena High School	Charter	962	585858
10	10	Wright High School	Charter	1800	1049400
11	11	Rodriguez High School	District	3999	2547363
12	12	Johnson High School	District	4761	3094650
13	13	Ford High School	District	2739	1763916
14	14	Thomas High School	Charter	1635	1043130

> **Caution!**
>
> If you see the error `FileNotFoundError` in your output, this means that the CSV file was not found in the Resources subfolder inside the School_District_Analysis folder.

To fix this error, add the CSV file to the Resources subfolder. Make sure the Resources subfolder is located in the School_District_Analysis folder, or you can use the indirect path approach with `os.path.join()` method.

SHORTCUT

To view the first five rows of a DataFrame, use the `df.head()` method after the DataFrame name. For the above DataFrame, the code looks like this:

`school_data_df.head()`

To view the last five rows of a DataFrame, use the `df.tail()` method after the DataFrame name.

To view any number of rows in a DataFrame, place a number inside the parentheses. For example, to get the top 10 rows, use `df.head(10)`. To get the bottom 10 rows, use `df.tail(10)`.

Read the Student Data File

Now we'll read the student data file and store it in a Pandas DataFrame by adding the following code to a new cell:

```
# Read the student data file and store it in a Pandas DataFrame.
student_data_df = pd.read_csv(student_data_to_load)
student_data_df.head()
```

After running the code, the output should look like this:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score
0	0	Paul Bradley	M	9th	Huang High School	66	79
1	1	Victor Smith	M	12th	Huang High School	94	61
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60
3	3	Dr. Richard Scott	M	12th	Huang High School	67	58
4	4	Bonnie Ray	F	9th	Huang High School	97	84

You have now loaded and read the CSV files—nice work! Now is a good time to save your work in your GitHub repository.

4.4.4: Commit Your Code

Congratulations! You got a lot done today. Before moving on, take a moment to commit your code to GitHub. Make sure you share your code with Maria so she can look over your work.

Before we can update the GitHub repository, we need to make sure our work is saved properly on our computers. Save `PyCitySchools.ipynb` to the School_District_Analysis folder. Make sure the Resources folder containing the CSV files is also in the School_District_Analysis folder.

How you commit changes to GitHub from the command line depends on whether you're using macOS or Windows. Follow the instructions below that correspond to your operating system.

macOS

1. Launch the command line.
2. Navigate to the School_District_Analysis folder on your computer. Your command line should look like this:

   ```
   <your_computer_name>:School_District_Analysis <your_home_folder>$
   ```

3. Type `git status` and press Enter. You should see something like this in your command line:
   - The `PyCitySchools.ipynb` file
   - The Resources folder
   - The `jupyter_practice.ipynb` file
   - The `student_practice.ipynb` file

For example:

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)


PyCitySchools.ipynb
      /Resources
      jupyter_practice.ipynb
      student_practice.ipynb

nothing added to commit but untracked files present (use "git add" to tr
```

4. Type `git add PyCitySchools.ipynb Resources/` and press Enter to add the CSV files and the `PyCitySchools.ipynb` file only. After you press Enter, you will see something like this:

```
On branch master
Your branch is up to date with 'origin/master'.

Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)


new file:   PyCitySchools.ipynb
      new file:   Resources/school_complete.csv
      new file:   Resources/student_complete.csv

Untracked files:
  (use "git add <file>..." to include in what will be committed)

      jupyter_practice.ipynb
      student_practice.ipynb
```

5. Type `git commit -m "Adding PyCitySchools.ipynb file and Resources folder."` and press Enter to commit the files to be added to the repository. The output should look similar to the following:

```
[master e3ee6a3] Adding PyCitySchools.ipynb file and Resources folder.
 3 files changed, 39437 insertions(+)
 create mode 100644 PyCitySchools.ipynb
 create mode 100644 Resources/school_complete.csv
 create mode 100644 Resources/student_complete.csv
```

6. Type `git push` and press Enter to add the file to your repository. The output should look like this:

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 466.25 KiB | 6.66 MiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/<your_GitHub_account>/School_District_Analysis.git
   ab468df..e3ee6a3  master -> master
```

7. Refresh your GitHub page to see the changes to your repository.

Windows

1. Launch Git Bash.
2. Navigate to the School_District_Analysis folder on your computer. Your command line should look like this:

```
<your_computer_name>@<your_home_folder> MINGW32 ~/Class/School_District_
$
```

3. Type `git status` and press Enter. You might see something like this in your command line:
    1. The `PyCitySchools.ipynb` file
    2. The Resources folder
    3. The `jupyter_practice.ipynb` file

4. The `student_practice.ipynb` file

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)


PyCitySchools.ipynb
      /Resources
      jupyter_practice.ipynb
      student_practice.ipynb

nothing added to commit but untracked files present (use "git add" to tr
```

4. Type `git add PyCitySchools.ipynb Resources/` and press Enter to add the CSV files and the `PyCitySchools.ipynb` file only. After you press Enter, you will see something like the following:

```
On branch master
Your branch is up to date with 'origin/master'.

Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)


new file:   PyCitySchools.ipynb
      new file:   Resources/school_complete.csv
      new file:   Resources/student_complete.csv

Untracked files:
  (use "git add <file>..." to include in what will be committed)

      jupyter_practice.ipynb
      student_practice.ipynb
```

5. Type `git commit -m "Adding PyCitySchools.py file and Resources folder."` and press Enter to commit the files to be added to the repository. The output should look like this:

```
[master e3ee6a3] Adding PyCitySchools.py file and Resources folder.
 3 files changed, 39437 insertions(+)
 create mode 100644 PyCitySchools.ipynb
 create mode 100644 Resources/school_complete.csv
 create mode 100644 Resources/student_complete.csv
```

6. Type `git push` and press Enter to add the files and folder to your repository. The output should like the following:

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 466.25 KiB | 6.66 MiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/<your_GitHub_account>/School_District_Analysis.git
   ab468df..e3ee6a3  master -> master
```

7. Refresh your GitHub page to see the changes to your repository.

Congratulations—you added the updated files to your GitHub repository!

> **Note**
> For more information about Python modules and packages, reading CSV files, and using the `head()` and `tail()` methods in Pandas, refer to the following documentation:
>
> - **Importing Python modules and packages (https://www.pythonlikeyoumeanit.com/Module5_OddsAndEnds/Modules_and_Packages.html)**
> - **Reading a CSV file into a Pandas DataFrame (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html)**
> - **Using .head() on a DataFrame (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.head.html)**

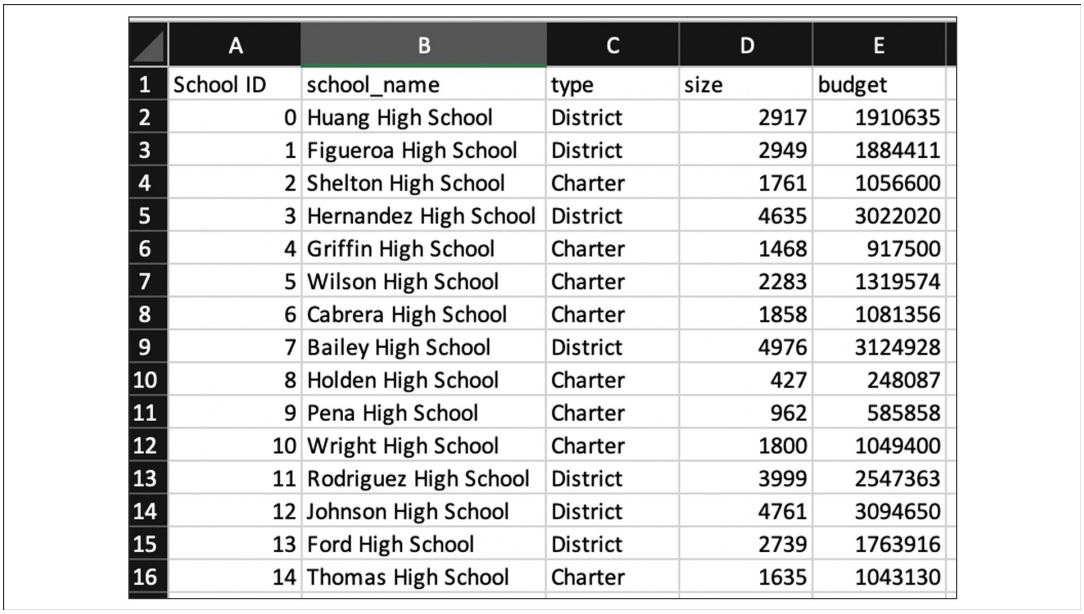
- **Using .tail() on a DataFrame (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.tail.html)**

4.5.1: Find Missing Values

Maria is aware that at least one of the datasets needs to be cleaned before any analysis can be performed. She would like you to use Pandas to do a more thorough inspection of the datasets than you did when you opened them with Excel. Cleaning the data is essential because any missing, malformed, or incorrect data in the rows can jeopardize the analysis.

As a first step in the data-cleaning process, we'll determine if there are missing values in the rows of the CSV files. Let's look at each CSV file separately.

First, open `schools_complete.csv`. We can see that each row contains a School ID, school name, type of school, student size, and budget. Therefore, there are no missing values in any of the rows, which are also called rows with **null values**. See the following image:



	A	B	C	D	E
1	School ID	school_name	type	size	budget
2	0	Huang High School	District	2917	1910635
3	1	Figueroa High School	District	2949	1884411
4	2	Shelton High School	Charter	1761	1056600
5	3	Hernandez High School	District	4635	3022020
6	4	Griffin High School	Charter	1468	917500
7	5	Wilson High School	Charter	2283	1319574
8	6	Cabrera High School	Charter	1858	1081356
9	7	Bailey High School	District	4976	3124928
10	8	Holden High School	Charter	427	248087
11	9	Pena High School	Charter	962	585858
12	10	Wright High School	Charter	1800	1049400
13	11	Rodriguez High School	District	3999	2547363
14	12	Johnson High School	District	4761	3094650
15	13	Ford High School	District	2739	1763916
16	14	Thomas High School	Charter	1635	1043130

While `schools_complete.csv` has only 15 rows of data and one row for the headers, the `student_complete.csv` has 39,170 rows. It would be very time-consuming to find missing values in a file so large. Luckily, Pandas has a few methods that can help us determine whether there are missing values in large datasets: the `count()` method, `isnull()` method, and `notnull()` method.

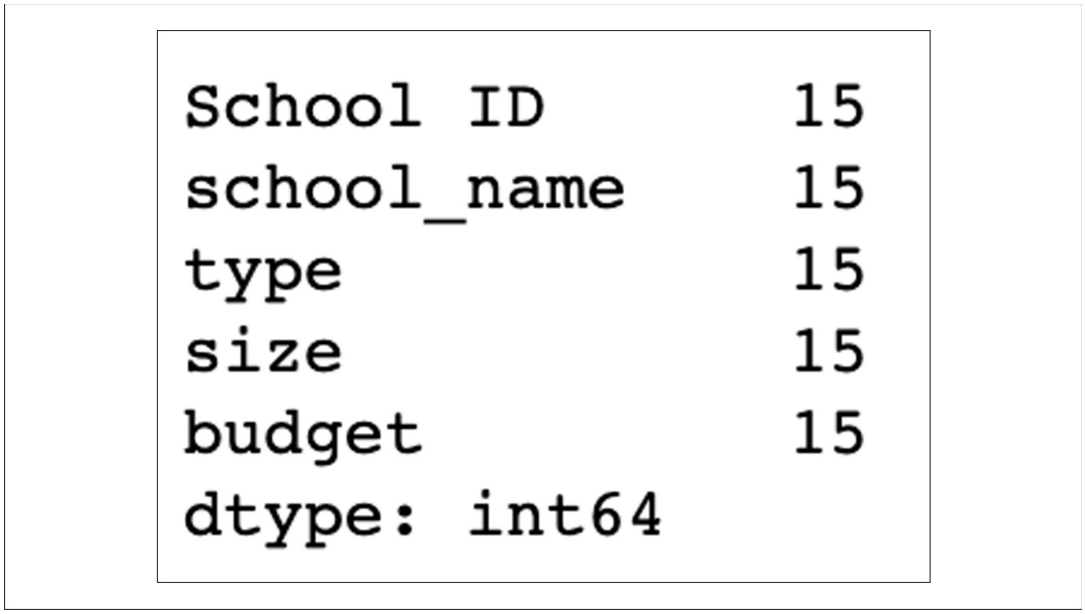
The count() Method

With the `count()` method, we can get a count of the rows for each column containing data. By default, "null" values are not counted, so you can often quickly identify which columns have missing data.

Let's use the `count()` method on the `school_data_df` DataFrame. Add the following code to a new cell and run the cell:

```
# Determine if there are any missing values in the school data.
school_data_df.count()
```

The output returns the name of the columns and the number of rows that are not null. For the `school_data_df` DataFrame, there are no missing values, because there are 15 rows that contain data in `schools_complete.csv`. In the output, the number 15 is next to each column header, as shown in the following image:

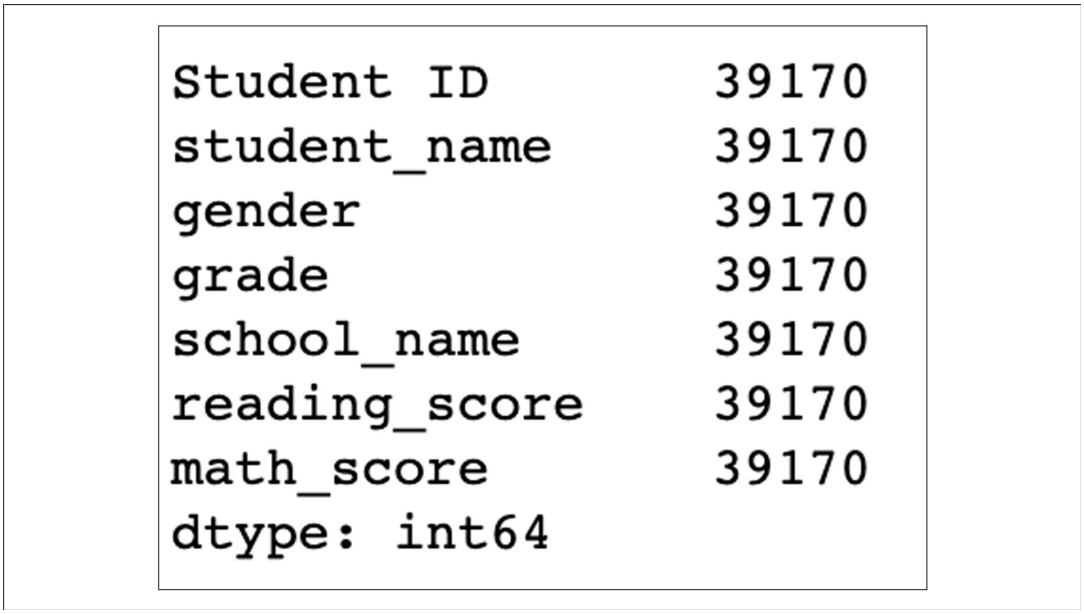


These results confirm what we observed when we looked at the `schools_complete.csv` file.

Now let's use the same method on the `student_data_df` DataFrame. Add the following code to a new cell and run the cell:

```
# Determine if there are any missing values in the student data.
student_data_df.count()
```

Like in the `schools_complete.csv` file, there are no missing values in any of the columns because the output shows 39,170 rows for the `student_complete.csv` file:



> **Note**
> For more information, see the [Pandas documentation on the count() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.count.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.count.html).

The isnull() Method

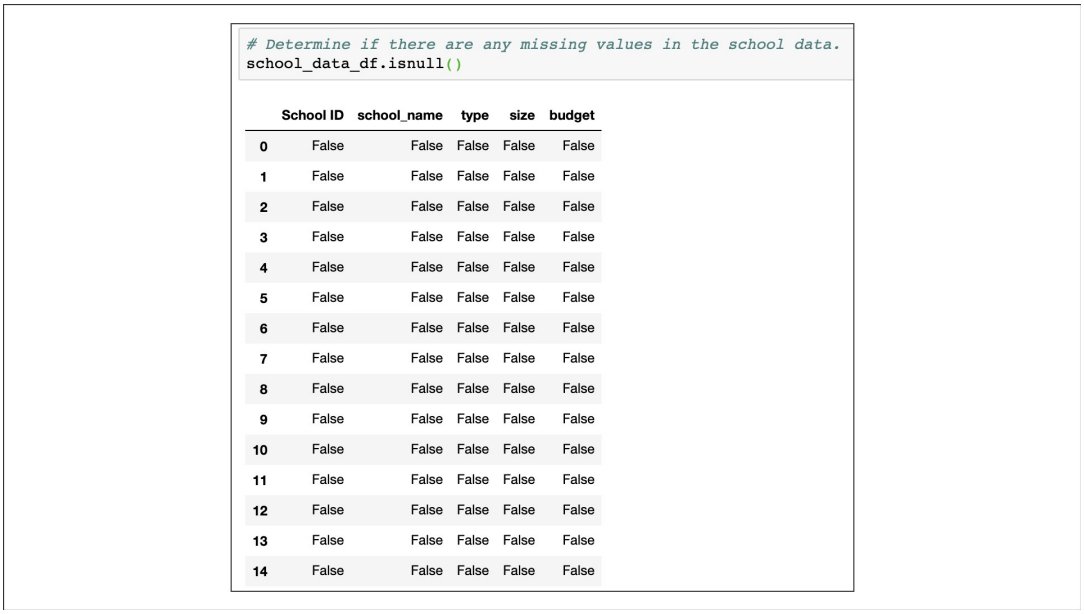
The Pandas library also has the `isnull()` method for determining empty rows. When you apply the `isnull()` method to a column, Series, or a DataFrame, a Boolean value

will be returned, either “True” for the row or rows that are empty, i.e., null, or “False” for the rows that are not empty.

Let’s use the `isnull()` method on the `school_data_df` DataFrame.

```
# Determine if there are any missing values in the school data.
school_data_df.isnull()
```

When we execute this code, we see that every row that is not empty is given the Boolean value “False,” which tells us that there are no missing values. See the following screenshot:



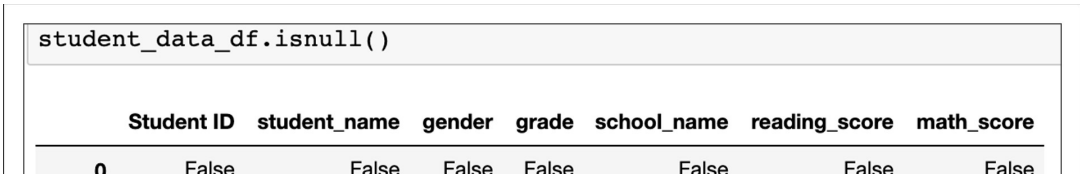
```
# Determine if there are any missing values in the school data.
school_data_df.isnull()
```

	School ID	school_name	type	size	budget
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
5	False	False	False	False	False
6	False	False	False	False	False
7	False	False	False	False	False
8	False	False	False	False	False
9	False	False	False	False	False
10	False	False	False	False	False
11	False	False	False	False	False
12	False	False	False	False	False
13	False	False	False	False	False
14	False	False	False	False	False

Now we’ll use the same method on the `student_data_df` DataFrame. Add the following code to a new cell and run the cell:

```
# Determine if there are any missing values in the student data.
student_data_df.isnull()
```

The output shows that there are no rows that contain missing values, as they are all labeled “False.” See the following image:



```
student_data_df.isnull()
```

	Student ID	student_name	gender	grade	school_name	reading_score	math_score
0	False	False	False	False	False	False	False

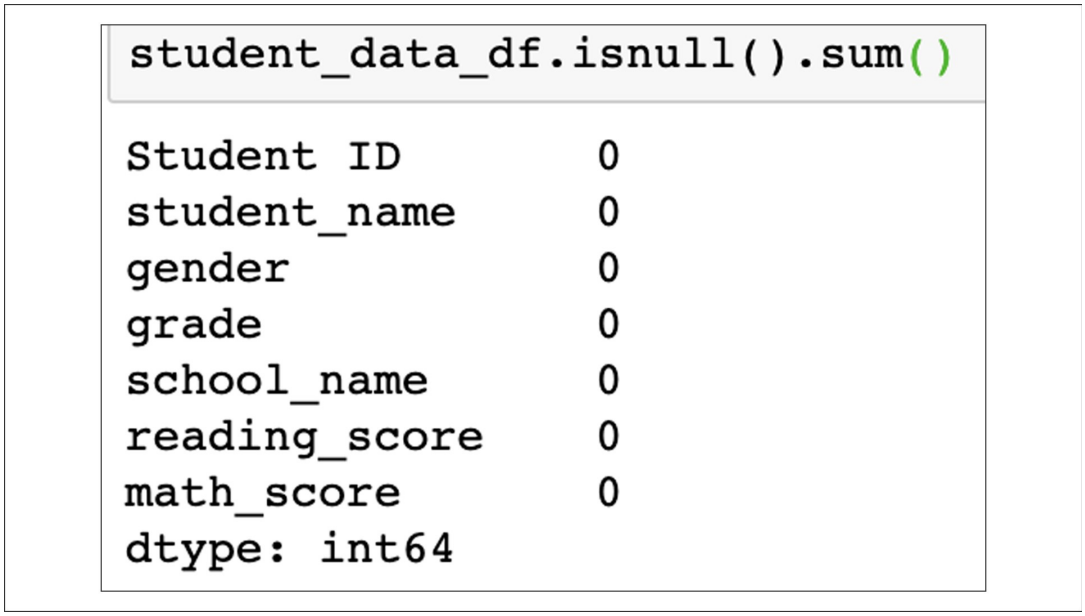
1	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False

To get the total number of empty rows, or rows that are "True," we can use the Pandas `sum()` method after the `isnull()` method, like this:

```
# Determine if there are any missing values in the student data.
student_data_df.isnull().sum()
```

> **Rewind**
> The process of joining two or more methods or functions together that are separated by a period is called **chaining**.

The output after running this code shows the total number of rows that are empty is zero for each column:



This output allows us to more easily determine at a glance how many rows are empty in the `student_data_df` DataFrame (zero). This is more straightforward than output that shows “False” in thousands of rows.

> **Note**
>
> For more information, see the [Pandas documentation on the isnull() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html).

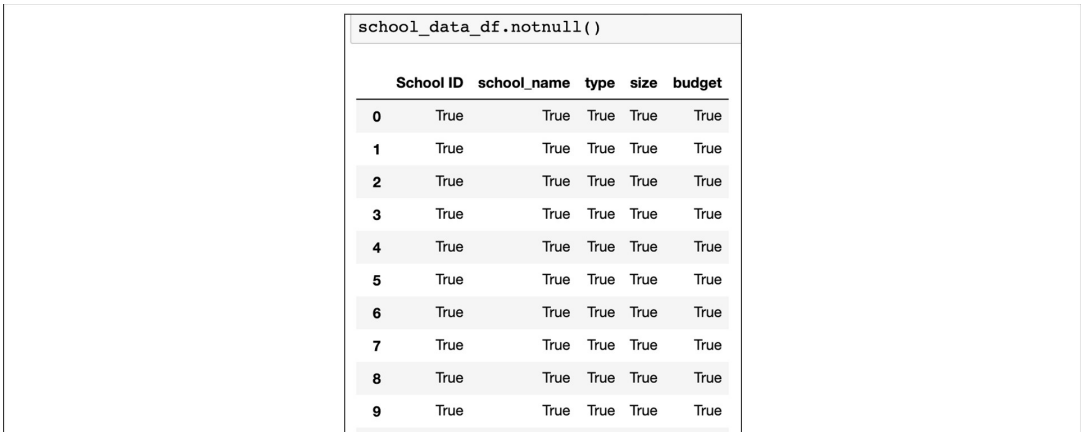
The notnull() Method

Another method that we can use to find missing values is the `notnull()` method. When you apply the `notnull()` method to a column, Series, or a DataFrame, a Boolean will be returned: “True” for the row or rows that are not empty, or “False” for the row or rows that are empty. This method returns the opposite output of the `isnull()` method.

Let’s use the `notnull()` method on the `school_data_df` DataFrame. Run the following code:

```
# Determine if there are not any missing values in the school data.
school_data_df.notnull()
```

When we run this code, the output returns a copy of our `school_data_df` DataFrame, where all the rows that do not have any missing values are labeled “True.” See the following image.



	School ID	school_name	type	size	budget
0	True	True	True	True	True
1	True	True	True	True	True
2	True	True	True	True	True
3	True	True	True	True	True
4	True	True	True	True	True
5	True	True	True	True	True
6	True	True	True	True	True
7	True	True	True	True	True
8	True	True	True	True	True
9	True	True	True	True	True

10	True	True	True	True	True
11	True	True	True	True	True
12	True	True	True	True	True
13	True	True	True	True	True
14	True	True	True	True	True

Like we did with the `student_data_df` DataFrame, we can chain the `notnull()` method and the `sum()` method to get the sum of all the columns that are "True."

```
# Determine if there are not any missing values in the student data.
student_data_df.notnull().sum()
```

When we execute this code, we get the number of rows that are not null, which is 39,170 for each column.

```
student_data_df.notnull().sum()

Student ID        39170
student_name      39170
gender            39170
grade             39170
school_name       39170
reading_score     39170
math_score        39170
dtype: int64
```

> **Note**
>
> For more information, see the [Pandas documentation on the notnull() method (http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.notnull.html)](http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.notnull.html).

4.5.2: Handle Missing Data

Maria is thrilled that there is no missing data in the CSV files. However, as a precautionary measure, she would like you to understand the options available for handling missing data in case the need arises. She's given you a CSV file that has some missing data so that you can practice.

There are a few options for handling missing data:

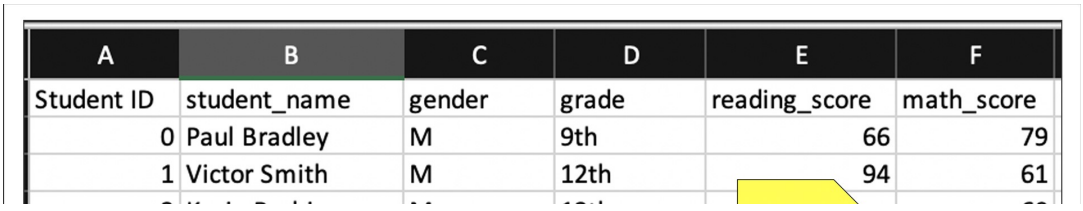
- Do nothing.
- Drop the row that has the missing value.
- Fill in the row that has the missing value.

It's not always obvious which of these options you should use, so let's review each one.

Click the following link to download the `missing_grades.csv` file into your Resources folder. (This is the practice dataset from Maria.)

Download missing_grades.csv
(https://courses.bootcampspot.com/courses/138/files/15833/download?wrap=1)
(https://courses.bootcampspot.com/courses/138/files/15833/download?wrap=1)

Open the `missing_grades.csv` file. You will notice there are two cells, E4 and E7, that are empty. The empty cells are shown in the following image:



A	B	C	D	E	F
Student ID	student_name	gender	grade	reading_score	math_score
0	Paul Bradley	M	9th	66	79
1	Victor Smith	M	12th	94	61

2	Kevin Rodriguez	M	12th		60
3	Dr. Richard Scott	M	12th	67	58
4	Bonnie Ray	F	9th	97	84
5	Bryan Miranda	M	9th	94	
6	Sheena Carter	F	11th	82	80
7	Nicole Baker	F	12th	96	69

To handle this, first create a new Jupyter Notebook file in the School_District_Analysis folder and rename it `cleaning_data.ipynb`.

In the first cell, import the Pandas library as the dependency and run the cell.

```
# Add the Pandas dependency.
import pandas as pd
```

In the next cell, declare a variable and assign it to the `missing_grades.csv` file that is located in the Resources folder.

```
# Files to load
file_to_load = "Resources/missing_grades.csv"

# Read the CSV into a DataFrame
missing_grade_df = pd.read_csv(file_to_load)
missing_grade_df
```

When we run this cell, the empty rows will be represented by “NaN," as shown in the following image:

	Student ID	student_name	gender	grade	reading_score	math_score
0	0	Paul Bradley	M	9th	66.0	79.0
1	1	Victor Smith	M	12th	94.0	61.0
2	2	Kevin Rodriguez	M	12th	NaN	60.0
3	3	Dr. Richard Scott	M	12th	67.0	58.0
4	4	Bonnie Ray	F	9th	97.0	84.0
5	5	Bryan Miranda	M	9th	94.0	NaN
6	6	Sheena Carter	F	11th	82.0	80.0
7	7	Nicole Baker	F	12th	96.0	69.0

IMPORTANT!

NaN means “not a number” and cannot be equal to zero.

Now let's review our options for handling the missing data.

Option 1: Do Nothing

If we do nothing, when we sum or take the averages of the reading and math scores, those NaNs will not be considered in the sum or the averages (just as they are not considered in the sum or the averages in an Excel file). In this situation, the missing values have no impact.

However, if we multiply or divide with a row that has a NaN, the answer will be NaN. This can cause problems if we need the answer for the rest of our code.

Option 2: Drop the Row

Another option is to drop the row where there are NaNs. When we remove the row containing the NaN, we will also remove all the data associated with that row. This can cause problems later if there is data in the other rows that we need.

To drop a row with NaNs, Pandas has the `dropna()` method. Use this method on the `missing_grade_df` DataFrame like this:

```
# Drop the NaNs.
missing_grade_df.dropna()
```

When we execute this code, the DataFrame will look like the following image. The rows with missing values have been dropped, as indicated by the index numbers, 0, 1, 3, 4, 6, and 7. After these rows are dropped, the index of the DataFrame is not automatically reset to number the rows in consecutive order.

	Student ID	student_name	gender	grade	reading_score	math_score
0	0	Paul Bradley	M	9th	66.0	79.0

1	1	Victor Smith	M	12th	94.0	61.0
3	3	Dr. Richard Scott	M	12th	67.0	58.0
4	4	Bonnie Ray	F	9th	97.0	84.0
6	6	Sheena Carter	F	11th	82.0	80.0
7	7	Nicole Baker	F	12th	96.0	69.0

IMPORTANT!

Dropping rows can affect the story you are trying to tell with the data. Before removing rows with NaN, you should ask yourself two key questions:

1. How much data would be removed if NaNs are dropped?
2. How would this impact the analysis?

   These questions need to be addressed for every dataset you work with.

NOTE

For more information, see the [Pandas documentation on the dropna() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html) .

Option 3: Fill in the Row

The third option is to fill in all the NaNs with a value.

IMPORTANT!

Filling in an empty row must be used with caution. For example, filling in a row with "0" can impact arithmetic calculations. If you decide to fill in empty rows, the values you insert must be carefully considered for every downstream analysis you perform.

In Pandas, we use the `fillna()` method to fill in a row. If we want to fill all empty rows with zero in our DataFrame, we pass the "0" in parentheses like this: `df.fillna(0)`.

Let's convert the `missing_grades.csv` file to a DataFrame again and fill in the NaNs with the number 85, using the following code:

```
# Fill in the empty rows with "85".
missing_grade_df.fillna(85)
```

When we execute this cell, the cells with NaN will be filled with 85, as shown in the following image:

	Student ID	student_name	gender	grade	reading_score	math_score
0	0	Paul Bradley	M	9th	66.0	79.0
1	1	Victor Smith	M	12th	94.0	61.0
2	2	Kevin Rodriguez	M	12th	85.0	60.0
3	3	Dr. Richard Scott	M	12th	67.0	58.0
4	4	Bonnie Ray	F	9th	97.0	84.0
5	5	Bryan Miranda	M	9th	94.0	85.0
6	6	Sheena Carter	F	11th	82.0	80.0
7	7	Nicole Baker	F	12th	96.0	69.0

> **Note**
>
> For more information, see the **Pandas documentation on the fillna() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html)** and how to **work with missing data (https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html)**.

4.5.3: Determine Data Types

You have been tasked with getting key metrics from the datasets. This will involve performing calculations to get sums, averages, and percentages. Before you perform these calculations, though, you should check if the numbers in these datasets are the correct data type for making those calculations, as this is considered a best practice.

Here are six common data types that we may encounter in this module and that you may come across in the future:

Data Type	Pandas Name	Example
Boolean	`bool`	“True” and “False”
Integer	`int32`	Values from –2,147,483,648 to 2,147,483,647
Integer	`int64`	Values from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Float	`float64`	Floating decimal
Object	`O, object`	Typically strings; often used as a catchall for columns with different data types or other Python objects like tuples, lists, and dictionaries
Datetime	`datetime64`	Specific moment in time with nanosecond precision, i.e., 2019-06-03 16:04:00.465107

Rewind

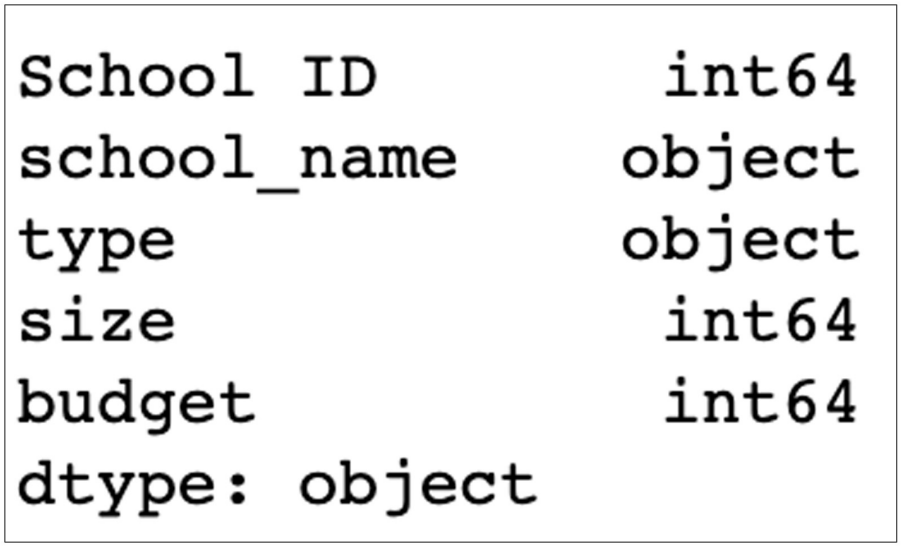
- If integers store values from −32,768 to 32,767, they are 16 bits or `int16`.
- If integer store values from −2,147,483,648 to 2,147,483,647, they are 32 bits or `int32`.
- If integer store values from
- −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, they are 64 bits, or `int64`.

With the Pandas library, we can check the data types of each column by using the Pandas `dtypes` attribute on a DataFrame.

To determine the data types in `school_data.csv`, type and run the following code:

```
# Determine data types for the school DataFrame.
school_data_df.dtypes
```

This code returns a Series with the name of each column and the data type of the values in the column. As shown in the output below, “School ID,” “size,” and “budget” are integers, which is the correct data type for performing arithmetic calculations.



We can also use the `dtype` attribute on a single column. There are two ways to perform this operation. We can use `df.column.dtype` if the column name doesn’t have any spaces in the name, like “school_name”. If there are spaces, we must use `df["column"].dtype`.

How would you find the data type of the budget column in the school data DataFrame? (Select all that apply.)

- [ ] `df.column="budget".dtype`
- [ ] `school_data_df.column="budget".dtype`
- [ ] `school_data_df."budget".dtype`
- [ ] `school_data_df.budget.dtype`
- [ ] `school_data_df["budget"].dtype`

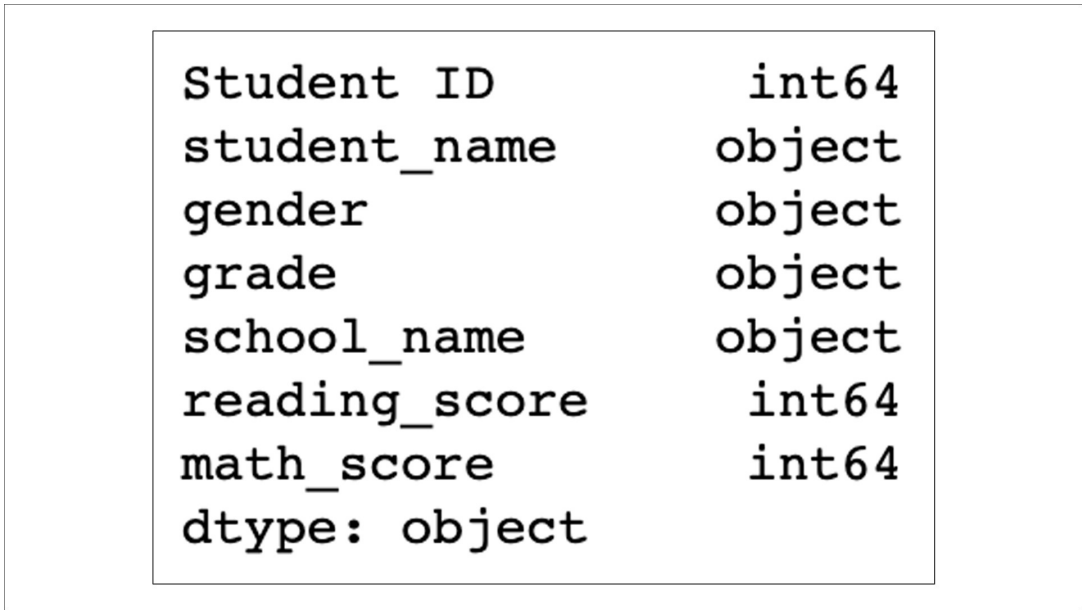
Check Answer

Finish ▸

Next, determine the data types for the student data file. Type and run the following code:

```
# Determine data types for the student DataFrame.
student_data_df.dtypes
```

The code returns each column name and the data type of each column. As shown in the output below, “Student ID,” “reading_score,” and “math_score” are integers, which is the correct data type for performing arithmetic calculations.



Based on the output, we determined that all of the columns we need to use for calculations are integers. Therefore, we won't need to change the data types for these columns. However, there may be instances in which it’s necessary to change the data type. Some CSV and text files, for example, may contain numbers as strings (or objects) rather than integers. These numbers would need to be converted to integers or floats.

> **Note**
>
> For more information, see the Pandas documentation on the following topics:
>
> - Get the data types of a DataFrame (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dtypes.html)
> - Changing data types (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html)

4.5.4: Get the Incorrect Student Names

Remember when Maria asked what anomaly you noticed in the `students_complete.csv` file? You found that one of the students' names had the prefix "Dr." Now Maria wants you to find all the students' names that have a professional prefix or suffix. Once you get all the incorrect student names, your next task is to clean the names by removing these professional prefixes and suffixes. Maria is not sure how these names came to have professional prefixes or suffixes, but it's important they are removed. These names are not coinciding with the school records, so the students' scores can't be added their student portfolio and may cause problems when they apply to colleges. Moreover, we can't drop these names because this would affect the total student count of the district as well as the individual schools, which will negatively impact the analysis.

In the `students_complete.csv` file, we will check for all the professional prefixes and suffixes that need to be removed. However, we will not remove family-related suffixes, such as Jr., II, III, and so on.

To begin, we need to do some exploratory analysis and testing to determine how to "clean" a student's name. When cleaning the data, it's a best practice to create a new Jupyter Notebook file rather than using the `PyCitySchools.ipynb` file.

Launch Jupyter Notebook and create a new file named `cleaning_student_names.ipynb`. Another option is to make a copy of the `PyCitySchools.ipynb` file and rename the copy.

We will only need to import `students_complete.csv` and convert it into a DataFrame. Here's the specific code you should be copying (or keeping, if you made a copy of the `PyCitySchools.ipynb` file):

```
# Import dependencies
import pandas as pd
```

```
student_data_to_load = "Resources/students_complete.csv"

student_data_df = pd.read_csv(student_data_to_load)
student_data_df.head()
```

	Student ID	student_name	gender	grade	school_name	reading_score	math_score
0	0	Paul Bradley	M	9th	Huang High School	66	79
1	1	Victor Smith	M	12th	Huang High School	94	61
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60
3	3	Dr. Richard Scott	M	12th	Huang High School	67	58
4	4	Bonnie Ray	F	9th	Huang High School	97	84

> **IMPORTANT!**
> The best practice for troubleshooting and testing new coding solutions before you apply them to your original file is to create a copy of the current notebook.
>
> If you do your testing on your original Jupyter Notebook file, the chances of changing the code of the original file is high, which may cause errors that were not present before.

Next, we need to get the students' names in a separate file. This will make it easier to determine how many students' names are affected, as well as what prefixes and suffixes need to be removed. First, we need to get all the students' names in a separate file so we don't affect the DataFrame.

The tolist() Method

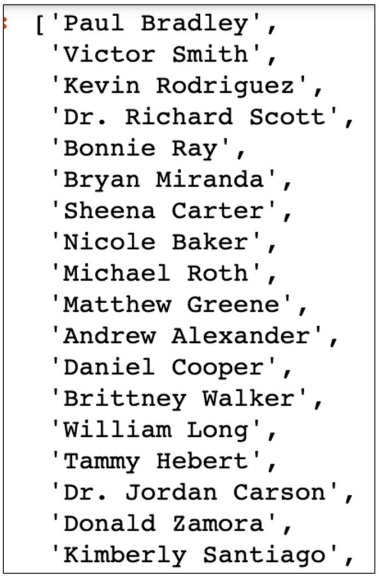
To get all the students' names in a separate file, we'll use the Pandas `tolist()` method on a DataFrame with reference to the `student_name` column. Using the `tolist()` method on the `student_name` column will add all the names to a list.

> **Rewind**
>
> Recall that every column in the CSV file and DataFrame is a Series or a list.

In the `cleaning_student_names.ipynb` file, add the following code and run the cell:

```
# Put the student names in a list.
student_names = student_data_df["student_name"].tolist()
student_names
```

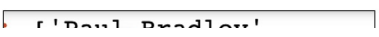
When we run this file, the first few lines in the output will look like this:

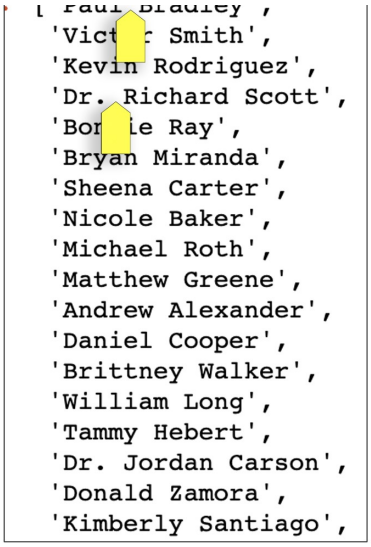


If we scroll through the list of 39,170 students, we'll see that many of the students' names have the "Dr." prefix, and there are other prefixes and suffixes as well. However, picking out all of the different prefixes and suffixes this way is not an efficient use of our time, and we may not catch all of them. Let's try a different method.

Look carefully at the list of names. Between the prefix and the first name, and between the first and last name, there are whitespaces—spaces with no text. Scrolling down the list, we'll observe the same situation for last name and suffix.

In Python, we can split the names on these whitespaces using the `split()` method. Look at the following image to see an example of whitespace.





The split() Method

In Python, the `split()` method will split a Python string object on the whitespace, or where there is no text. If we split each name, we can count the length of each name. A student with a first and last name only will return a length of 2, whereas a name with a prefix or suffix only will have a length of 3.

Try the following example.

In a new cell, type `name = "Mrs. Linda Santiago"` and do the following:

1. Print out the name.
2. Apply the `split()` method to the "name" variable, i.e., `name.split()`.
3. Get the length of the name after using the `split()` method.

What is the length of the name "Mrs. Linda Santiago"?

- ○ 2
- ○ 3
- ○ 1
- ○ 4

Check Answer

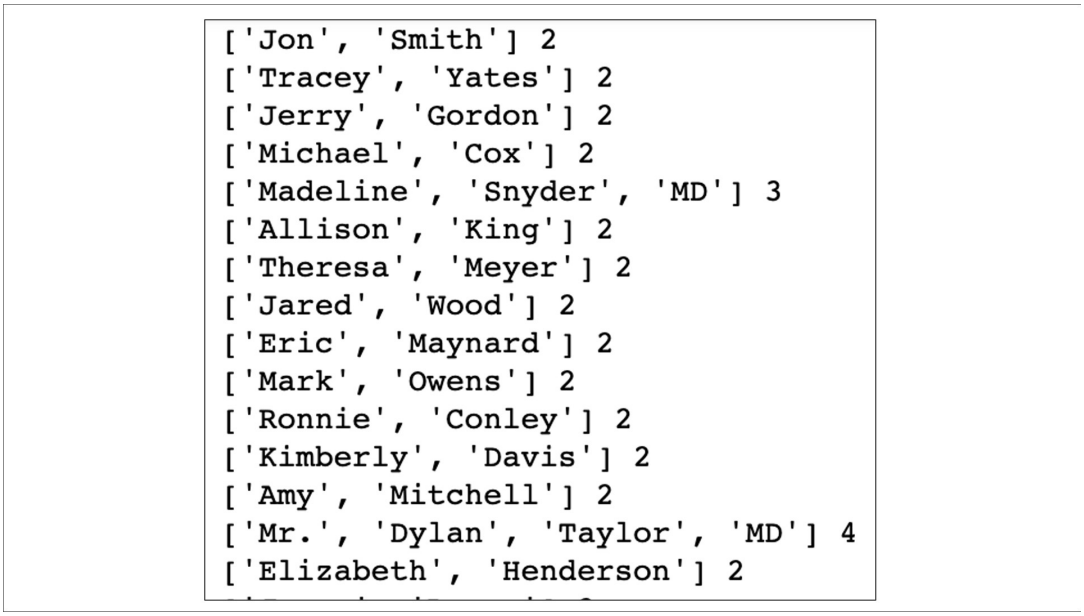
Finish ▸

In the `student_names` list, we can use a `for` loop to iterate through the list, split each name, and print out the name and the length of the name.

In a new cell, add the following code and run the cell.

```
# Split the student name and determine the length of the split name.
for name in student_names:
    print(name.split(), len(name.split()))
```

A sample of the output shows that the length of a majority of the names is 2, which corresponds to names that have only a first and last name. Names with a length greater than 2 have a prefix and/or suffix in addition to the first and last name. The output sample is shown below:



We can filter this list using a conditional statement. If the length of the name is equal to or greater than 3, we can add it to a new list. By getting the length of this new list, we can determine how many students are affected. Your code should look like this:

```
# Create a new list and use it for the for loop to iterate through the list.
students_to_fix = []

# Use an if statement to check the length of the name.
```

```
# If the name is greater than or equal to "3", add the name to the list.

for name in student_names:
    if len(name.split()) >= 3:
        students_to_fix.append(name)

# Get the length of the students whose names are greater than or equal to "3
len(students_to_fix)
```

When you print the length of the `students_to_fix` list, how many students names need to be fixed?

- ○ 189
- ○ 1,342
- ○ 1,531
- ○ 0

Check Answer

Finish ▸

The output should show 1,531 students:

```
# Get the length of the students whose names are greater than or equal to "3".
len(students_to_fix)
```

```
151
```

If we print the list of students, we can see the different prefixes and suffixes:

```
print(students_to_fix)
```

```
['Dr. Richard Scott', 'Dr. Jordan Carson', 'Madeline Snyder MD', 'Mr. Dylan Taylor MD', 'Dr. Scott Gill', 'Miss Madis
on Everett', 'Virginia Ramirez MD', 'Joseph Morales III', 'Angela Perkins DVM', 'Heather Allen MD', 'Luke Lowery MD',
'Dr. Larry Hines', 'Emily Cardenas MD', 'Mrs. Elizabeth Espinoza DDS', 'Mrs. Lisa Becker', 'Mr. Travis Young', 'Paula
Fernandez DVM', 'Dr. Xavier Bell II', 'Andrew Hanson MD', 'Mr. Roberto Wright MD', 'Mr. Dale Miller', 'Mary Gonzalez
DDS', 'Mrs. Cindy Brown', 'James White MD', 'Donald Gill DDS', 'Mrs. Catherine Williams MD', 'Ashley Caldwell MD', 'M
r. Jared Campbell', 'Michael Moreno PhD', 'Tara Parker MD', 'Shawn Williams DVM', 'Dr. Alicia Martinez', 'Mr. Timothy
Anderson', 'Mrs. Paula Lee', 'Mr. Elijah Hall', 'Mr. Raymond Stone', 'John Dickerson MD', 'Mr. Alex Franklin', 'Mrs.
Jessica Dunn', 'Mr. Michael Stevenson', 'Jason Jones DDS', 'Adriana Wade MD', 'Erica Hurley MD', 'Dr. Garrett Wrigh
t', 'Mr. James Allen', 'Mr. Timothy Harper MD', 'Dale Brown MD', 'Dr. Christina Moore', 'Angela Peterson MD', 'Jennif
er Ferguson MD', 'Anthony Wilson DDS', 'Dr. Jessica Lopez', 'David Miller Jr.', 'Dr. John Jones', 'Ms. Samantha Brow
n', 'Cynthia Hansen DVM', 'Rebecca Montoya MD', 'Dr. Jacob Matthews', 'Dr. Jacob Bell', 'Mr. Bruce Thomas', 'Robert M
artin MD', 'Mrs. Marcia Jones PhD', 'Reginald Garcia IV', 'Patricia Smith MD', 'Amy Dyer DDS', 'Brian Hall DDS', 'Dou
glas Levy DDS', 'Mr. Eric Ibarra', 'Benjamin Ramirez MD', 'Raymond Freeman DDS', 'Carla Reeves MD', 'Dr. Donald Willi
ams', 'Mr. John Carter', 'William Jackson DVM', 'Mr. Mitchell Cooper', 'Kevin Brown IV', 'Mrs. Kathleen Reed', 'Donna
Robinson MD', 'Mrs. Dawn Olson DDS', 'Mr. Dustin Glover', 'Alexis French MD', 'Lisa Guerrero MD', 'David Nguyen DVM',
'William Washington Jr.', 'Julie Patterson MD', 'Mr. James Kelly', 'Dr. Robert Martin', 'Darlene Thomas PhD', 'Mr. Ju
an Jenkins', 'Mr. Juan Bryant II', 'Mr. Matthew Coleman', 'Bonnie Hammond MD', 'Kathleen Harris DDS', 'Steven Velez D
DS', 'Tammy Cruz DDS', 'Dr. Marie Williams', 'Kenneth Paul Jr.', 'Mr. Jose Morris', 'Scott Rivers III', 'Melissa Rodr
iguez MD', 'Geoffrey Ortiz MD', 'Matthew Stewart MD', 'Sandra Thompson DDS', 'Mark Myers Jr.', 'Mrs. Maria Smith DD
```

Great job! Using the `split()` method is much more efficient than scrolling through all 39,170 names!

By glancing at the output, we can see that prefixes include Dr., Mr., Mrs., and Miss, and the suffixes include MD, DDS, DVM, Ph.D., Jr., II, III, and IV. However, there could be more.

Going through this list name by name to find all of the prefixes and suffixes would be time-consuming. Instead, we can write an algorithm to get the length of each name, including the prefix or suffix from the list. This will allow us to filter out prefixes and suffixes.

Get All the Prefixes

To get all of the prefixes in the student data, we're going to write an algorithm that will iterate through the `students_to_fix` list and determine if the length of the prefix, or the first item in the list after using `name.split()`, is less than or equal to 4. We'll use 4 as the upper limit because the longest prefix is "Miss," which is four characters long. If the length is less than or equal to 4, we'll add it to a new list named `prefixes`.

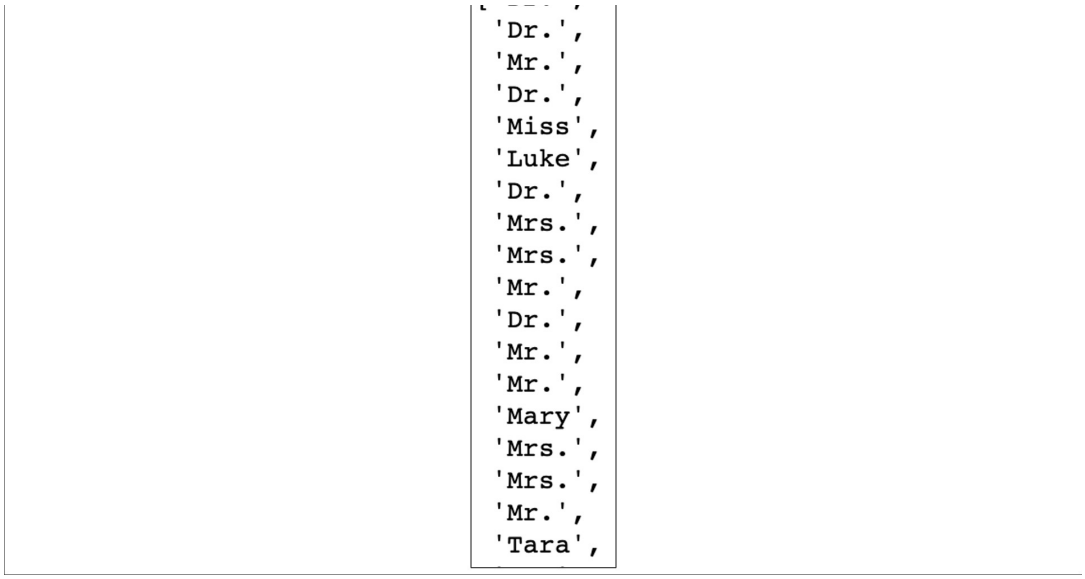
In a new cell, add the following code and run the cell:

```
# Add the prefixes less than or equal to 4 to a new list.
prefixes = []
for name in students_to_fix:
    if len(name.split()[0]) <= 4:
        prefixes.append(name.split()[0])

print(prefixes)
```

With the `split()` method, we can get the length of not only each name in the `students_to_fix` list but also the length of each **substring** in the list. In this case, a substring can be either the prefix, first name, last name, or suffix.

In `len(name.split()[0])`, we're getting the length of the first item, or prefix. When we print out the `prefixes` list, we'll see first names and prefixes that are less than or equal to 4:

```
['Dr.',
```



Get All the Suffixes

Let's apply the same methodology to get the suffixes. We'll iterate through the `students_to_fix` list and determine if the last item `[-1]` after using `name.split()` is less than or equal to 3.

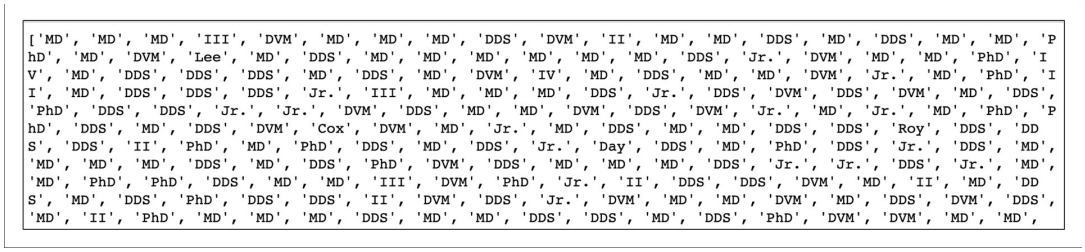
> **Rewind**
> We can use negative indexing to get the last item in the list.

We'll use 3 as the upper limit because the longest suffixes (DDS, DVM, and Ph.D.) are three characters long. If the length is less than or equal to 3, we'll add it to a new list named `suffixes`.

In a new cell, add the following code and run the cell.

```
# Add the suffixes less than or equal to 3 to a new list.
suffixes = []
for name in students_to_fix:
    if len(name.split()[-1]) <= 3:
        suffixes.append(name.split()[-1])

print(suffixes)
```

When we print out the `suffixes` list, we'll see all the suffixes that are less than or equal to 3:



Both of the lists, `prefixes` and `suffixes`, are long. Fortunately, we can make these lists shorter by getting the unique items in each list with a Python method called `set()`.

> **Note**
>
> For more information, read the [Python documentation on splitting strings (https://docs.python.org/3/library/stdtypes.html#string-methods)](https://docs.python.org/3/library/stdtypes.html#string-methods).

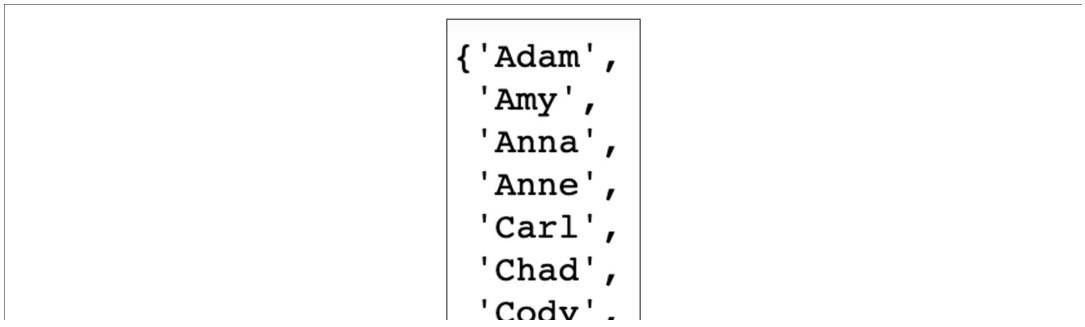
The set() Method

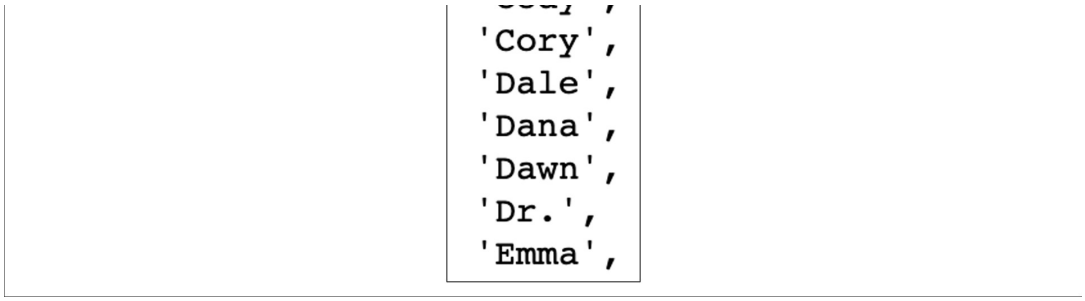
The `set()` method returns all unique items in a list when that list is added inside the parentheses. If the list contains strings, they will be ordered alphabetically. Let's apply this method to the `prefixes` list first.

In a new cell, add the following code and run the cell.

```
# Get the unique items in the "prefixes" list.
set(prefixes)
```

Looking at the output, we can pick out the prefixes Dr., Mr., Mrs., Ms., and Miss:





Now let's apply the same method to the suffixes. In a new cell, add the following code and run the cell.

```
# Get the unique items in the "suffixes" list.
set(suffixes)
```

In the output, you should see the suffixes MD, DDS, DVM, Ph.D., Jr., II, III, IV, and V.

```
 'PhD',
 'Roy',
 'V'}
```

Congratulations on finding the unique prefixes and suffixes from the names! Our exploratory analysis was a success. Now we can remove Dr., Mr., Mrs., Ms., Miss, MD, DDS, DVM, and Ph.D. from the students' names in `student_data_df`.

4.5.5: Remove Strings: strip() vs. replace()

Maria appreciates your help finding all of the prefixes and suffixes in students' names and would now like your help removing them. This is all part of the data cleaning process. To clean the students' names, you'll be using Python string methods.

To clean the students' names, you will have to remove the unique prefixes and suffixes. To recap our findings from the last section, the unique prefixes we need to remove are:

- Dr.
- Mr.
- Mrs.
- Miss

And, the unique suffixes we need to remove are:

- MD
- DDS
- DVM
- Ph.D

Remember, we don't want to remove the family-related suffixes like Jr., II, III, IV, and V.

We have two options to remove these prefixes and suffixes: the `strip()` method and the `replace()` method. Let's compare them to figure out which one we should use.

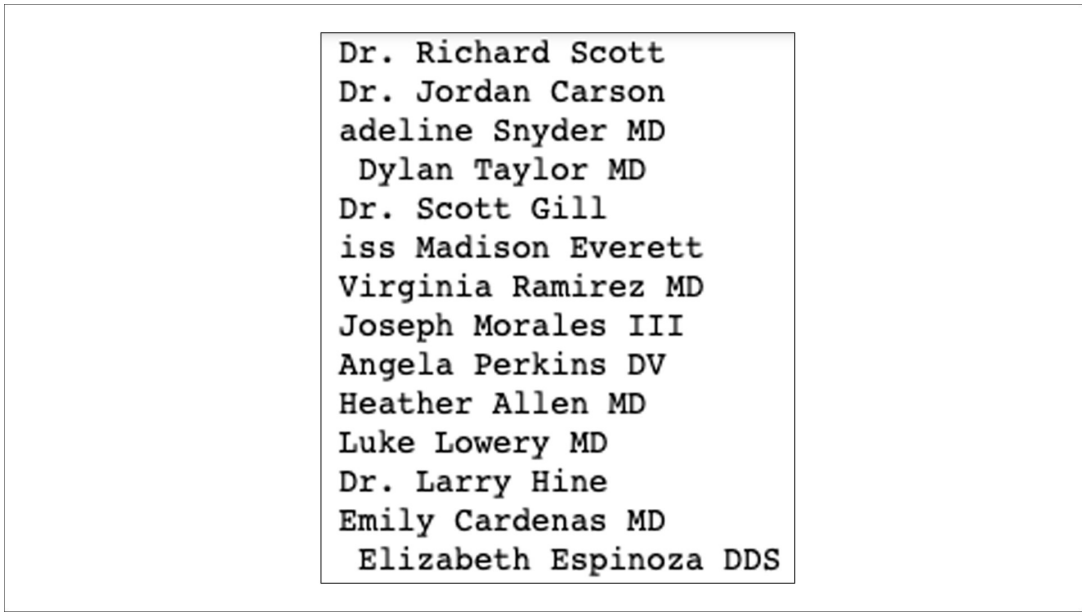
The strip() Method

The `strip()` method removes any combination of letters and words that are inside the parentheses. For example, if we put “Mrs.” inside the parentheses of the `strip()` method, any combination of the letters “M,” “r,” and “s” will be stripped from the student names.

Let’s test this in the `cleaning_student_names.ipynb` file. Strip “Mrs.” from the students’ names using the following code:

```
# Strip "Mrs." from the student names
for name in students_to_fix:
    print(name.strip("Mrs."))
```

You may notice something odd in the results:



In the third line, the “M” has been removed from “Madeline.” In the fourth line, “Mr.” has been removed from “Dylan Taylor MD.” In the sixth line, the “M” in “Miss” has been removed, and in the last line, the prefix “Mrs.” has been removed from “Elizabeth Espinosa DDS.” From these results, we can see that the `strip()` method will not work for us because it removes letters from the students' names, not just the prefix "Mrs."

> **Note**

> For more information, read the [Python documentation on the strip() method (https://docs.python.org/3/library/stdtypes.html#string-methods)](https://docs.python.org/3/library/stdtypes.html#string-methods) .

The replace() Method

When we use the `replace()` method, we can replace an "old" phrase, or string, with a new phrase, or string, inside the parentheses.

In our case, we want to replace prefixes with an empty string, or "nothing." For example, to replace "Dr." with an empty string, we would use the following syntax: `replace("Dr.", "")`.

If you initialized a "name" variable to "Mrs. Linda Santiago," how would you convert the name to "Dr. Linda Santiago"?

- ○ `name.replace("Mrs.","Dr.")`
- ○ `name.str.replace("Mrs.","Dr.")`
- ○ `name.split().replace("Mrs.","Dr.")`
- ○ `name.strip(Mrs., Dr.)`

Check Answer

Finish ▸

To change "Mrs. Linda Santiago" to "Dr. Linda Santiago" you would write `name.replace("Mrs.","Dr.")`.

Now let's replace "Dr." in "Dr. Linda Santiago" with an empty string. Add the following code to a new cell and run the cell.

```
# Replace "Dr." with an empty string.
name = "Dr. Linda Santiago"
name.replace("Dr.", "")
```

The output after running this cell will be the following:

```
# Replace "Dr." with an empty string.
name = "Dr. Linda Santiago"
name.replace("Dr.", "")
```

```
' Linda Santiago'
```

You may have noticed in the output that there is a white space before “Linda Santiago.” We can remove this white space by adding it to “Dr.” while we are using the `replace()` method, like this:

`name.replace("Dr. ", "")`.

> **Note**
>
> For more information, read the **Python documentation on the replace() method (https://docs.python.org/3/library/stdtypes.html#string-methods)** .

4.5.6: Replace Substrings

Now that you have found a way to replace the suffixes and prefixes in the students' names, Maria wants you to test how to remove them before you apply the procedure to the DataFrame.

We have determined that we can use the `replace()` method to remove prefixes and suffixes from the students' names. However, we don't want to write a different `replace()` statement for each prefix and suffix; this is not efficient or practical programming.

A more efficient way to remove the prefixes and suffixes is to do the following:

1. Declare a list named `prefixes_suffixes` that holds the common prefixes and suffixes as strings.
2. Iterate through this list with a `for` loop.
3. For each item in the list, use the `replace()` method on the student_name column.

Let's start cleaning! In the `cleaning_student_names.ipynb` file, type the following code in a new cell and run the cell.

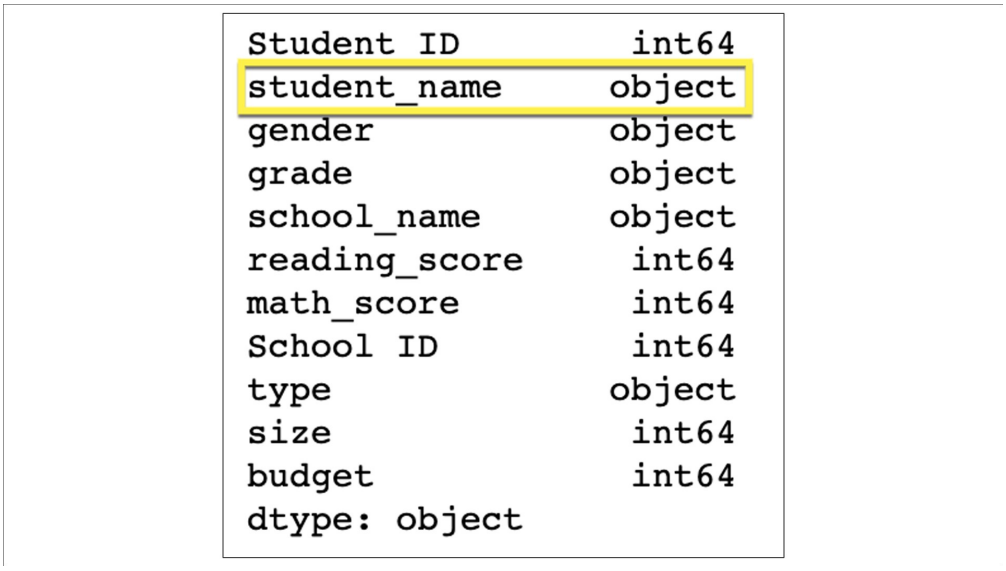
```
# Add each prefix and suffix to remove to a list.
prefixes_suffixes = ["Dr. ", "Mr. ","Ms. ", "Mrs. ", "Miss ", " MD", " DDS",
```

Remember that there is a whitespace between the prefix and the first name, as well as between the last name and the suffix, as shown in the above image. We're also going

to add a whitespace after each prefix and before each suffix in our list of prefixes and suffixes.

> **Note**
> The `replace()` method can be used only on Python strings.

> **Rewind**
> Remember, we used the `dtypes` method to determine the data types of each column in a DataFrame. For the `student_data_df` DataFrame, we determined that the `student_name` column was an object, not a string.



Even though an object is typically a string, we need to convert the object data type to a string while using the `replace()` method. Luckily, Python provides us with a way to access and manipulate strings with the `str` attribute.

We can chain the `str` method with the `replace()` method on the `student_name` column in the `student_data_df`, like this:

```
student_data_df["student_name"].str.replace()
```

Next, iterate through the list of prefixes and suffixes by passing the prefix or suffix as `word` in the `replace()` method, as shown in the following code.

```
# Iterate through the "prefixes_suffixes" list and replace them with an empt
for word in prefixes_suffixes:
    student_data_df["student_name"] = student_data_df["student_name"].str.re
```

There's a lot going on in this code, so let's break it down.

- First, we iterate through the `prefixes_suffixes` list.
- In the first part of the `for` loop, we assign each student name in the `student_name` column with the same student name after we replace the prefix or suffix with an empty string.
- To replace the prefix or suffix with an empty string, we convert the name of the student to a string with `student_data_df["student_name"].str`. For every `word` in the `prefixes_suffixes` list, we replace that prefix or suffix, if the name has one, with an empty string using `replace(word,"")`.

To check if this works, let's print the first 10 rows of `student_data_df`. The results should look like this:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score
0	0	Paul Bradley	M	9th	Huang High School	66	79
1	1	Victor Smith	M	12th	Huang High School	94	61
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60
3	3	Richard Scott	M	12th	Huang High School	67	58
4	4	Bonnie Ray	F	9th	Huang High School	97	84
5	5	Bryan Miranda	M	9th	Huang High School	94	94
6	6	Sheena Carter	F	11th	Huang High School	82	80
7	7	Nicole Baker	F	12th	Huang High School	96	69
8	8	Michael Roth	M	10th	Huang High School	95	87
9	9	Matthew Greene	M	10th	Huang High School	96	84

It looks like it worked! Now we have to confirm that all of the prefixes and suffixes were caught in our script.

Reusing some of our previous code to check the length of each name, we will:

1. Declare a new variable and assign it to the `student_data_df["student_name"]`.
2. Convert the `student_data_df["student_name"]` to a list using `.tolist()`.

Add the following code to a new cell and run the cell.

```
# Put the cleaned students' names in another list.
student_names = student_data_df["student_name"].tolist()
student_names
```

Now, add all the students' names that are greater than or equal to 3 to a list. Add the following code in a new cell and run the cell.

```
# Create a new list and use it for the for loop to iterate through the lis
students_fixed = []

# Use an if statement to check the length of the name.

# If the name is greater than or equal to 3, add the name to the list.

for name in student_names:
    if len(name.split()) >= 3:
        students_fixed.append(name)

# Get the length of the students' names that are greater than or equal to
len(students_fixed)
```

How many students are in the `students_fixed` list?

- ○ 151
- ○ 348
- ○ 1,531
- ○ 0

Check Answer

Finish ▸

Here's the printed list:

```
print(students_to_fix)
```

```
['Joseph Morales III', 'Xavier Bell II', 'David Miller Jr.', 'Reginald Garcia IV', 'Kevin Brown IV', 'William Washing
ton Jr.', 'Juan Bryant II', 'Kenneth Paul Jr.', 'Scott Rivers III', 'Mark Myers Jr.', 'Michael Norton Jr.', 'Sean Pen
a Jr.', 'Cody Mueller Jr.', 'Mathew White Jr.', 'Jason Daugherty Jr.', 'Leonard Webster II', 'Michael Stein Jr.', 'Ry
an Phillips Jr.', 'Allen Snyder Jr.', 'Calvin Williams Jr.', 'Ronald Torres Jr.', 'Gabriel Smith III', 'Brian Pitts J
r.', 'Bruce Thompson II', 'Jeremy Sanders II', 'Austin Johnson II', 'Bryan Conway Jr.', 'Ronald Moore II', 'Robert Ga
rrison IV', 'William Gonzalez Jr.', 'Brian Matthews Jr.', 'Eric Richards Jr.', 'Kenneth Munoz Jr.', 'Jon Delgado J
r.', 'Andrew English Jr.', 'Raymond Cox Jr.', 'Jeffrey Mcclain Jr.', 'Brian Poole V', 'Jeffrey Kim II', 'Matthew Alex
ander Jr.', 'Kevin Reynolds Jr.', 'Jacob Skinner Jr.', 'Dennis Gibson Jr.', 'Jimmy Shea Jr.', 'James Reed Jr.', 'Jeff
ery Baker Jr.', 'Derrick Hughes III', 'Joshua Rivera Jr.', 'Timothy Guzman Jr.', 'Lonnie Allen II', 'Tyrone Clark II
I', 'Jeffrey Nelson Jr.', 'Michael Mann II', 'David Johnson Jr.', 'Timothy Chavez Jr.', 'Brian Powell III', 'Lance Ra
ngel Jr.', 'Noah Herrera Jr.', 'Eric Johnson Jr.', 'Victor Chavez II', 'Richard Ellis Jr.', 'Charles Goodman Jr.', 'M
ichael Singh Jr.', 'Jonathan Smith Jr.', 'Allen Morris Jr.', 'Aaron Solis Jr.', 'Daniel Mckinney Jr.', 'Christopher P
etty Jr.', 'Michael Garcia Jr.', 'Charles Short II', 'Derek Barker Jr.', 'Zachary Garrett III', 'Thomas Harrison J
r.', 'Andrew Mitchell V', 'Aaron Davis Jr.', 'Brian Bailey Jr.', 'Jeremiah Morris IV', 'David Morales Jr.', 'Larry Ch
andler Jr.', 'James Clark II', 'Cameron Roman Jr.', 'Christopher Smith Jr.', 'Todd Grant Jr.', 'Kenneth Avila Jr.',
'Jason Lopez Jr.', 'Dwayne Mathis Jr.', 'Colton Collins II', 'Chris Palmer III', 'Tyler Randall Jr.', 'Joseph Morgan
IV', 'Carlos Adkins Jr.', 'Mark Livingston Jr.', 'Michael Williams III', 'Eric Evans III', 'Christopher Thornton J
r.', 'Adam Hayes III', 'Michael Moore Jr.', 'Joseph Hudson II', 'William Brooks Jr.', 'Daniel Welch II', 'Roger Horto
n Jr.', 'Cory Marshall II', 'Joshua Thompson Jr.', 'Jon Flores II', 'Philip Jones Jr.', 'Greg Joyce Jr.', 'Thomas Fry
e Jr.', 'Joseph Marquez Jr.', 'Robert Smith Jr.', 'David Beck Jr.', 'Ryan Gomez II', 'Erik Smith Jr.', 'Omar Thompson
Jr.', 'Leonard Roberts Jr.', 'Duane Li Jr.', 'Richard Lucero Jr.', 'Seth Medina Jr.', 'Anthony Carlson Jr.', 'Edward
Dudley II', 'Matthew Lewis Jr.', 'Christopher Bowers Jr.', 'Timothy Hughes Jr.', 'Kevin Hoover IV', 'David Butler J
r.', 'Michael Wilson Jr.', 'George Hess Jr.', 'Kevin Briggs Jr.', 'Joseph Matthews III', 'Dylan Bailey Jr.', 'Anthony
Woods Jr.', 'Michael Harris Jr.', 'Daniel Diaz Jr.', 'Taylor Thompson Jr.', 'Daniel Robinson Jr.', 'Ronald Phelps J
r.', 'Jonathan Reid II', 'Joshua Smith Jr.', 'Joseph Herrera III', 'Tony Robertson II', 'James Fowler III', 'William
Wright II', 'Thomas Carrillo II', 'Matthew Bennett Jr.', 'Mark Lucas IV', 'Keith Newman Jr.', 'Matthew Mitchell III',
'Colin Schultz II', 'Terry Scott Jr.', 'Jason Patel Jr.', 'Steven Lewis Jr.', 'Lucas Williams Jr.']
```

There are now only names with familial designations like Jr., II, III, IV, and V in the `students_fixed` list, which confirms that we caught all the professional prefixes and suffixes. Congratulations on cleaning the data! Let's apply the same process to the data in the `PyCitySchools.ipynb` file.

4.5.7: Fix the Students' Names

Maria is impressed by your success in testing how to remove the prefixes and suffixes from the students' names. Now she wants you to apply the same method to the "student name" column in the `student_data_df` DataFrame. Doing so will correct the students' names so they align with their school records.

Using the same code that we tested for cleaning the students' names, we're going to turn to the `PyCitySchools.ipynb` file to fix the students' names so they match their school records.

> **Rewind**
> For the previous data-cleaning step, we created a list to hold the common prefixes and suffixes as strings like this:
>
> `prefixes_suffixes = ["Dr. ", "Mr. ","Ms. ", "Mrs. ", "Miss ", " MD", " DDS", " DVM", " PhD"]`
>
> We iterated through the list and used the `replace()` method on the "student_name" column, replacing every occurrence of each prefix or suffix as they occurred with an empty string.

To initialize the list, add the following code to a new cell in your `PyCitySchools.ipynb` file. Make sure that there is a space after the prefix and before the suffix.

```
# Add each prefix and suffix to remove to a list.
prefixes_suffixes = ["Dr. ", "Mr. ","Ms. ", "Mrs. ", "Miss ", " MD", " DDS",
```

Next, add the following code below the `prefixes_suffixes` list. This is similar to the code we used in the `cleaning_student_names.ipynb` file.

```
# Iterate through the words in the "prefixes_suffixes" list and replace them
for word in prefixes_suffixes:
    student_data_df["student_name"] = student_data_df["student_name"].str.re
```

When we execute the cell, the results should look like what we observed in the `cleaning_student_names.ipynb` file:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score
0	0	Paul Bradley	M	9th	Huang High School	66	79
1	1	Victor Smith	M	12th	Huang High School	94	61
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60
3	3	Richard Scott	M	12th	Huang High School	67	58
4	4	Bonnie Ray	F	9th	Huang High School	97	84
5	5	Bryan Miranda	M	9th	Huang High School	94	94
6	6	Sheena Carter	F	11th	Huang High School	82	80
7	7	Nicole Baker	F	12th	Huang High School	96	69
8	8	Michael Roth	M	10th	Huang High School	95	87
9	9	Matthew Greene	M	10th	Huang High School	96	84

Now we have a clean `student_data_df` DataFrame, which means we can finally start our analysis!

4.5.8: Commit Your Code

Congratulations! You got a lot done today. Now is a good time to update your GitHub repository with all the work you just did so that Maria and other stakeholders can review your code.

Save `PyCitySchools.ipynb` to your School_District_Analysis folder, and follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder using the necessary commands.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "cleaning student data"` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.6.1: Download a Working Version of the Code

We've covered a lot of ground so far. Take a moment to check your work against the file provided and verify that your work matches. You can also use the following file as your starting point to help complete the rest of the module. Download `clean_students_complete.csv` into your Resources folder to complete the module. This file contains a clean version of the data.

Download clean_students_complete.csv
(https://courses.bootcampspot.com/courses/138/files/15831/download?wrap=1)
(https://courses.bootcampspot.com/courses/138/files/15831/download?wrap=1)

4.7.1: Merge DataFrames

As you sit down to get started on your work, Maria presents your task for today. She tells you that the first report you need to generate is the school district summary, which will be based on several key metrics.

The school district summary will be a high-level snapshot of the district's key metrics:

- Total number of students
- Total number of schools
- Total budget
- Average math score
- Average reading score
- Percentage of students who passed math
- Percentage of students who passed reading
- Overall passing percentage

We'll find this information and visualize the data with a table like the following:

	Total Schools	Total Students	Total Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
0	15	39,170	$24,649,428.00	79.0	81.9	75	86	80

In order to get all of this data organized in one table, we'll need to merge the two DataFrames and perform analysis on the single, merged DataFrame. Although we'll be merging the DataFrames, it may be more efficient to use either `school_data_df` or `student_data_df` when performing certain calculations.

> **Important!**
> Even though we can perform calculations on both DataFrames to get the information we need, it's a best practice to create a new DataFrame to do calculations. This way, the original data is not affected.

To merge two DataFrames, there must be a column in each of the DataFrames with the same name. So before we merge, let's review the column names in each DataFrame.

The columns in `school_data_df` are:

- School ID
- school_name
- type
- size
- budget

The columns in the `student_data_df` are:

- Student ID
- student_name
- gender
- grade
- school_name
- reading_score
- math_score

We'll merge `school_data_df` and `student_data_df` on a shared column using the `merge()` method. The column that these DataFrames have in common is `school_name`. Inside the parentheses of the `merge()` method, we'll do the following:

- Add the DataFrames to be merged.
- Add the shared column to each DataFrame so that the merge can occur.
- Define how the DataFrames should be merged: left, right, inner, or outer. The default is inner. (You will learn more about merging later in this course.)

> **Note**
>
> For more information, see the following documentation:
>
> - **Merging Pandas DataFrames** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html)
> - **Merging, joining, and concatenating Pandas DataFrames** (https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html)

In a new cell, add the following code and run the cell.

```
# Combine the data into a single dataset.
school_data_complete_df = pd.merge(student_data_df, school_data_df, on=["sch
school_data_complete_df.head()
```

Let's break down how the DataFrames are being merged with this code.

- We create a new DataFrame for the merged DataFrames, called `school_data_complete_df`.
- The new DataFrame is created as a result of merging DataFrame #2 (`school_data_df`), which is the "right" DataFrame, into DataFrame #1 (`student_data_df`), which is the "left" DataFrame. We refer to the DataFrames as "left" and "right" to reflect the order they appear inside the parentheses.
- We use the parameter "on," which is equal to a list of the columns that are identical from each DataFrame, in this case, "school_name." We can also use the column name like this: `on="school_name"`.

> **Important!**
>
> There may be cases in which you want to merge on a column that has similar information in two separate DataFrames, but is named differently in each—for example, "school_name" in one DataFrame and "high_school" in the second. In these cases, you should rename the columns so that they match. This will help avoid duplicate columns or merging issues.

After you run the cell, the output should look like this:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
1	1	Victor Smith	M	12th	Huang High School	94	61	0	District	2917	1910635
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60	0	District	2917	1910635
3	3	Richard Scott	M	12th	Huang High School	67	58	0	District	2917	1910635
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635

The data from `student_data_df` takes up the first seven columns, and the four columns from the `school_data_df` are added at the end: School ID, type, size, and budget.

Caution!

Did you get output that looks like this?

	School ID	school_name	type	size	budget	Student ID	student_name	gender	grade	reading_score	math_score
0	0	Huang High School	District	2917	1910635	0	Paul Bradley	M	9th	66	79
1	0	Huang High School	District	2917	1910635	1	Victor Smith	M	12th	94	61
2	0	Huang High School	District	2917	1910635	2	Kevin Rodriguez	M	12th	90	60
3	0	Huang High School	District	2917	1910635	3	Richard Scott	M	12th	67	58
4	0	Huang High School	District	2917	1910635	4	Bonnie Ray	F	9th	97	84

If so, this is probably because the "left" DataFrame was `school_data_df` and the "right" DataFrame was `student_data_df` inside the `merge()` method. You'll need to reverse the order of the DataFrames and run your code again to avoid errors later.

Now that we have our merged DataFrame, we can start getting some of the school district's key metrics.

4.7.2: Get the Number of Students

Now that you have merged the DataFrames, Maria would like you to get the total number of students in the school district. She reminds you that if you had dropped the students whose names were incorrect instead of correcting them, the total number of students would be incorrect.

> **Rewind**
> Recall that we counted the number of missing values in each DataFrame using the `count()` method. We can use the same method on the `school_data_complete_df` DataFrame to get the counts of specific columns.

If we add the `count()` method to the `school_data_complete_df` DataFrame, we will count all the items in each row for each column that is not null. Run the following code:

```
# Get the total number of students.
student_count = school_data_complete_df.count()
student_count
```

Your results should be the following:

```
Student ID        39170
student_name      39170
gender            39170
grade             39170
school_name       39170
reading_score     39170
```

```
math_score          39170
School ID           39170
type                39170
size                39170
budget              39170
dtype: int64
```

In the output, we see that each column has 39,170 rows, or values. Therefore, we can choose any one of the columns and get the student count using the following format:

```
school_data_complete_df[column].count()
```

To assign the "student_count" to a column that identifies with students, we will use the "Student ID" column.

Edit the code in the previous cell to look like this:

`school_data_complete_df["Student ID"].count()`

Run the cell.

FINDING

The number of students in the district is 39,170.

```
# Get the total number of students.
student_count = school_data_complete_df["Student ID"].count()
student_count
```

```
39170
```

CAUTION

If you got a `SyntaxError: invalid syntax`, as shown in the following image, you didn't put single or double quotation marks around the column name.

```
  File "<ipython-input-56-059a362bee76>", line 1
    student_count = school_data_complete[Student ID].count()
                                                ^
```

```
SyntaxError: invalid syntax
```

4.7.3: Get the Number of Schools

Great work on getting the total number of students! It's a good thing you didn't drop those incorrect names, or you would still be cleaning data and behind in your work. Now it's time to get the total number of schools in the school district.

> **Rewind**
> Remember we found the total number of schools by using the `count()` method on the `school_data_df` DataFrame. The output was 15.

There are two ways we can get the number of schools. One is to use the `school_data_df` DataFrame and assign a variable to one of the columns in the DataFrame, like this:

```
# Calculate the total number of schools.
school_count = school_data_df["school_name"].count()
school_count
```

After running the code, the output is 15:

```
# Calculate the total number of schools.
school_count = school_data_df["school_name"].count()
school_count
```

```
15
```

We can't use the `count()` method on the `school_data_complete_df["school_name"]` column because this would give us a value of 39,170. If we want to use `school_data_complete_df`, we first need to get the unique items in the `["school_name"]` column by using the `unique()` method. This method will return an array, or list, of all the unique values of that column.

Add the following code to a new cell and run the cell. The output is shown below.

```
# Calculate the total number of schools
school_count_2 = school_data_complete_df["school_name"].unique()
school_count_2
```

```
array(['Huang High School', 'Figueroa High School', 'Shelton High School',
       'Hernandez High School', 'Griffin High School',
       'Wilson High School', 'Cabrera High School', 'Bailey High School',
       'Holden High School', 'Pena High School', 'Wright High School',
       'Rodriguez High School', 'Johnson High School', 'Ford High School',
       'Thomas High School'], dtype=object)
```

The results are an array, or Python list, of all the high schools—and you know how to get the number of items in a list!

Using Python, how would you get the number of high schools from the array `school_count_2`?

- ○ Count the high schools and set a variable equal to 15.
- ○ Running the code: `school_data_complete_df["school_name"].unique().count()`
- ○ Running the code: `len(school_data_complete_df["school_name"].unique())`
- ○ Running the code: `school_data_complete_df["school_name"].unique().value_counts()`

Check Answer

Finish ▸

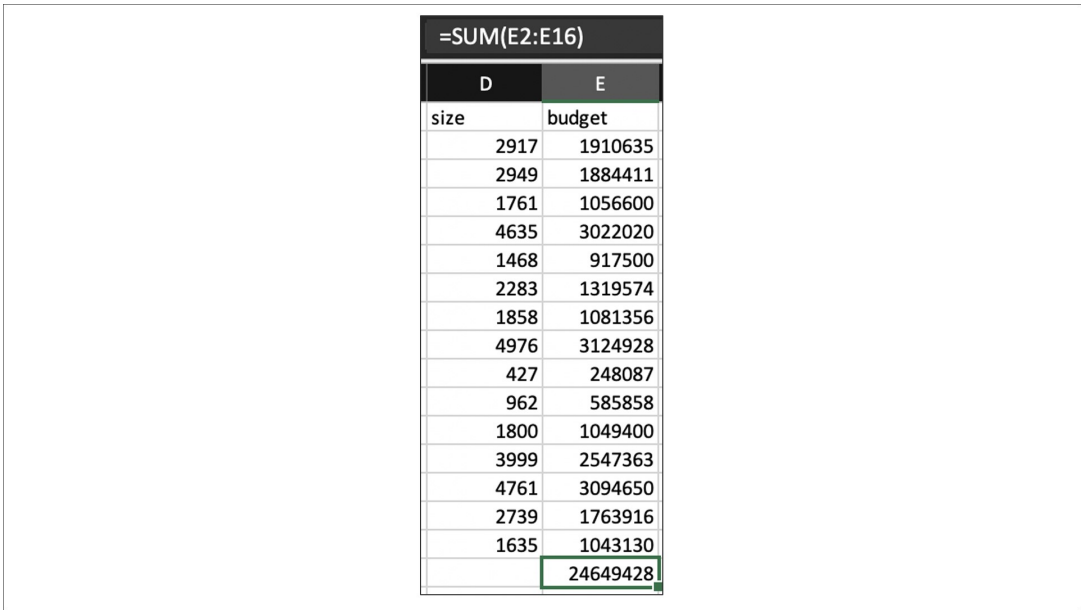
NOTE

> For more information, see the **Pandas documentation on the unique() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.unique.html)** .

4.7.4: Get the Total Budget

Since you successfully found the total number of schools, Maria has asked you to calculate the total budget for the whole district.

If we were working in Excel, we could find the sum of the budget column using the `=sum()` function on `schools_complete.csv` like this:



But, because we're using Pandas, we'll use the `sum()` method on the budget column.

Look at `school_data_complete_df`. Notice that each student is associated with a school and all the data related to that school, including type, size, and budget:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
1	1	Victor Smith	M	12th	Huang High School	94	61	0	District	2917	1910635
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60	0	District	2917	1910635
3	3	Richard Scott	M	12th	Huang High School	67	58	0	District	2917	1910635

4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635

When we apply the `sum()` method on the `school_data_complete_df["budget"]` column, we get the following output:

```
# Calculate the Total Budget.
total_budget = school_data_complete_df["budget"].sum()
total_budget
```

```
82932329558
```

This is a huge number, and based on our Excel calculation, it's incorrect. Our code summed all 39,170 rows in the "budget" column.

Instead of using the `sum()` on the `school_data_complete_df["budget"]` column, we should use `sum()` on the `school_data_df["budget"]` column.

Add the following code to a new cell and run the cell.

```
# Calculate the total budget.
total_budget = school_data_df["budget"].sum()
total_budget
```

Finding

The total budget of the schools in the district is $24,649,428. Your output should match the sum of `E3:E16` in the `schools_complete.csv` file. This is a good way to check that our answer matches the CSV file.

```
# Calculate the Total Budget.
total_budget = school_data_df["budget"].sum()
total_budget
```

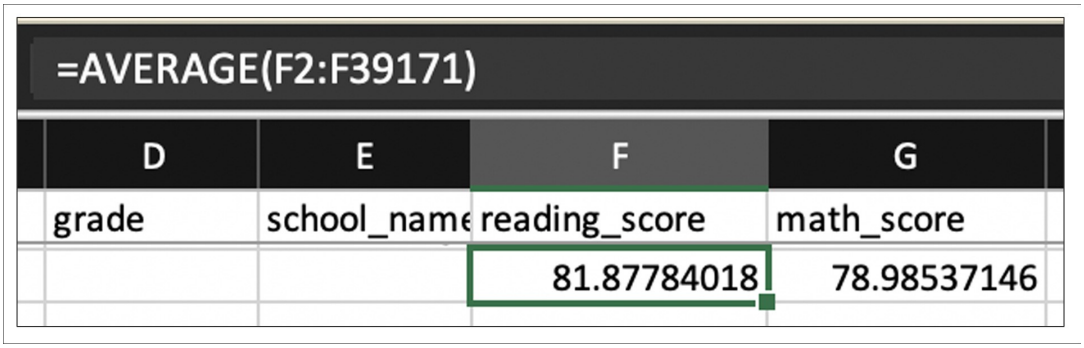
```
24649428
```

4.7.5: Get the Score Averages

You're doing a great job completing these tasks. So far, you have calculated the total number of students, the total number of schools, and the total budget of the district. Now you need to get the average math and reading scores.

Before we get the average math and reading scores for all the students in the district, let's open `student_data.csv` file and look at the data.

The averages can be determined by clicking inside an empty cell and adding the `=average()` function for the "reading_score" and "math_score" columns, as shown in the following image of the CSV file:



The Pandas method for getting the average of columns is the `mean()` method.

Find the Average Reading Score

Using the `school_data_complete_df` DataFrame, we can get the average reading score using the following code:

```
# Calculate the average reading score.
average_reading_score = school_data_complete_df["reading_score"].mean()
average_reading_score
```

When this code is executed, we get the following results:

```
# Calculate the average reading score.
average_reading_score = school_data_complete_df["reading_score"].mean()
average_reading_score
```

```
81.87784018381414
```

The average, 81.87784018 is the same as in `student_data.csv` when we use Excel.

Find the Average Math Score

Now we'll repurpose the code for the average reading score to find the average math score, replacing `average_reading_score` with `average_math_score`.

Run the following code:

```
# Calculate the average math score.
average_math_score = school_data_complete_df["math_score"].mean()
average_math_score
```

The results are the following:

```
# Calculate the average math score.
average_math_score = school_data_complete_df["math_score"].mean()
average_math_score
```

```
78.98537145774827
```

The average, 78.9853714, is the same as in `student_data.csv`.

4.7.6: Get the Passing Percentages

You're almost done compiling the data for the district summary. Now it's time to get the percentage of students who passed math and reading, as well as the overall passing percentage.

To get the percentage of students who passed math and reading, we will write code to:

1. Determine the passing grade.
2. Get the number of students who passed math and reading in separate DataFrames.
3. Calculate the number of students who passed math and reading.
4. Calculate the percentage of students who passed math and reading.

To get the overall passing percentage, we will write code to:

1. Get the number of students who passed both math and reading in a DataFrame.
2. Calculate the number of students who passed both math and reading.
3. Calculate the percentage of students who passed both math and reading.

Determine the Passing Grade

For math and reading assessment tests in this school district, the passing score was 70. Therefore, we need to get all the math and reading scores that are greater than or equal to 70. To do this, in a new cell, assign a `passing_math` variable to the math_score column in `school_data_complete_df`, where all the math scores are equal to or greater than 70.

```
passing_math = school_data_complete_df["math_score"] >= 70
passing_reading = school_data_complete_df["reading_score"] >= 70
```

To find the `passing_math` variable, run `passing_math` in a new cell. The result is Boolean values for the rows, where "True" means the score is equal to or greater than 70, and "False" means the score is not equal to or greater than 70.

```
passing_math = school_data_complete_df["math_score"] >= 70
passing_math
```

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

When we execute the code for `passing_reading`, we will get Boolean values for each row, where "True" means the score is equal to or greater than 70, and "False" means the score is not equal to or greater than 70.

Get the Number of Students Who Passed Math and Reading

To get all the students who passed math and all the students who passed reading, we need to filter our `school_data_complete_df` DataFrame for the "True" cases. In other words, get only the students who have a grade is equal or greater to 70.

We can filter the `school_data_complete_df` DataFrame by adding the `school_data_complete_df["math_score"] >= 70` within brackets, like this:

```
# Get all the students who are passing math in a new DataFrame.
passing_math = school_data_complete_df[school_data_complete_df["math_score"]
passing_math.head()
```

When we run this cell, the output will be a new DataFrame with all the rows that contain students who passed math.

```
# Get all the students that passed math in a new DataFrame.
passing_math = school_data_complete_df[school_data_complete_df["math_score"] >= 70]
passing_math.head()
```

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635
5	5	Bryan Miranda	M	9th	Huang High School	94	94	0	District	2917	1910635
6	6	Sheena Carter	F	11th	Huang High School	82	80	0	District	2917	1910635
8	8	Michael Roth	M	10th	Huang High School	95	87	0	District	2917	1910635

Now we can repurpose the code we wrote to calculate `passing_math` in order to find `passing_reading`, but we'll switch out `math_score` with `reading_score`, as shown in the following code:

```
# Get all the students that are passing reading in a new DataFrame.
passing_reading = school_data_complete_df[school_data_complete_df["reading_s
```

Now that you have found all the students who have passed math and reading, you need to get the *number* of students who passed each subject.

How would you get the number of students who passed math?

- ○ Count them and set a variable equal to the number.
- ○ Run the code: `passing_math["student_name"].count()`
- ○ Run the code: `len(passing_math)`
- ○ Run the code: `passing_math.count()`

Check Answer

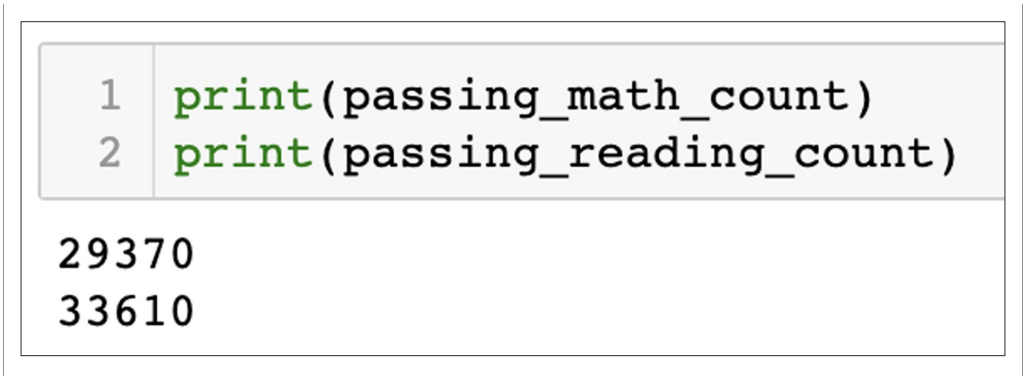
Finish ▸

To get the number of students who passed math and reading, apply the `count()`

method to the student_name column of the `passing_math` and `passing_reading` DataFrames, like this:

```
# Calculate the number of students passing math.
passing_math_count = passing_math["student_name"].count()

# Calculate the number of students passing reading.
passing_reading_count = passing_reading["student_name"].count()
```

When we execute this cell and print out the `passing_math_count` and the `passing_reading_count`, we get the number of students who passed math and the number of students who passed reading:



Great job on getting the `passing_math_count` and the `passing_reading_count`! Now we need to get the percentage of students who passed math and reading.

Get the Percentage of Students Who Passed Math and Reading

To get the percentage of students who passed math and reading, divide the `passing_math_count` and the `passing_reading_count` by the total number of students, and then multiply by 100.

> **Rewind**
>
> Recall that the student count was calculated using this code:
> `school_data_complete_df["Student ID"].count()`

This gave us 39,170 students.

```
# Get the total number of students.
student_count = school_data_complete_df["Student ID"].count()
student_count
```

```
39170
```

Since we are calculating a percentage, we need to convert the `student_count` to a number with a decimal, or floating-point decimal, by using `float()`.

The final calculation should look like this:

```
# Calculate the percent that passed math.
passing_math_percentage = passing_math_count / float(student_count) * 100

# Calculate the percent that passed reading.
passing_reading_percentage = passing_reading_count / float(student_count) *
```

When we execute this cell and print out the `passing_math_percentage` and `passing_reading_percentage`, we get the percentage of students who passed math and the percentage of students who passed reading:

```
1 print(passing_math_percentage)
2 print(passing_reading_percentage)
```

```
74.9808526933878
85.80546336482001
```

We're almost done! Next, we need to calculate the percentage of students who passed both math and reading.

Calculate the Overall Passing Percentage

To get the overall passing percentage, we need to get all the students who passed *both* math and reading and divide by the total number of students.

We can filter the `school_data_complete_df` DataFrame by adding the `school_data_complete_df["math_score"] >= 70` and `school_data_complete_df["reading_score"] >= 70` with the logical operator "`&`" within brackets, like this:

```
# Calculate the students who passed both math and reading.
passing_math_reading = school_data_complete_df[(school_data_complete_df["mat

passing_math_reading.head()
```

When we run this cell, the output will be a new DataFrame with all the columns that contain students who passed both math and reading.

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635
5	5	Bryan Miranda	M	9th	Huang High School	94	94	0	District	2917	1910635
6	6	Sheena Carter	F	11th	Huang High School	82	80	0	District	2917	1910635
8	8	Michael Roth	M	10th	Huang High School	95	87	0	District	2917	1910635
9	9	Matthew Greene	M	10th	Huang High School	96	84	0	District	2917	1910635

Next, we'll get the total number of students who passed both math and reading.

What is the total number of students who passed both math and reading?

Check Answer

Finish ▸

To get the total number of students who passed both math and reading, we apply the `count()` method to the `passing_math_reading` DataFrame with the following code.

```
# Calculate the number of students who passed both math and reading.
overall_passing_math_reading_count = passing_math_reading["student_name"].co
overall_passing_math_reading_count
```

Finally, we calculate the percentage of students who passed both math and reading by dividing the total number of students and multiplying by 100, using the following code.

```
# Calculate the overall passing percentage.
overall_passing_percentage = overall_passing_math_reading_count / student_co
overall_passing_percentage
```

> **Finding**
> When we run this cell, the percentage of students who passed both math and reading is 65.17232575950983.

This completes the data we need for the district summary. Now we'll add it to a new DataFrame.

4.7.7: Create a District Summary DataFrame

Maria is eager to see the district summary so that she can pass the information along to stakeholders. You'll need to combine all of the metrics we just calculated and put them in a new DataFrame to provide Maria with a table that contains all the data.

Now that we have performed all the calculations needed for the district summary, let's add the following values and columns to a new DataFrame named `district_summary_df`.

- Total number of schools in the column "Total Schools"
- Total number of students in the column "Total Students"
- Total budget in the column "Total Budget"
- Average reading score in the column "Average Reading Score"
- Average math score in the column "Average Math Score"
- Percentage of students passing reading in the column "% Passing Reading"
- Percentage of students passing math in the column "% Passing Math"
- Overall passing percentage in the column "% Overall Passing"

> **Rewind**
> Remember, one way to create a new DataFrame is to convert a list of dictionaries to a DataFrame.

To create `district_summary_df` DataFrame, we can create a list of dictionaries, where the keys are column names and the values are the metrics we calculated. We do this because the DataFrame has no index, and lists have natural indexing.

Add the following code to a new cell and run the cell.

```
# Adding a list of values with keys to create a new DataFrame.
district_summary_df = pd.DataFrame(
          [{"Total Schools": school_count,
          "Total Students": student_count,
          "Total Budget": total_budget,
          "Average Math Score": average_math_score,
          "Average Reading Score": average_reading_score,
          "% Passing Math": passing_math_percentage,
         "% Passing Reading": passing_reading_percentage,
        "% Overall Passing": overall_passing_percentage}])
district_summary_df
```

When we execute this code, we get the following district summary DataFrame.

	Total Schools	Total Students	Total Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
0	15	39170	24649428	78.985371	81.87784	74.980853	85.805463	65.172326

4.7.8: Format Columns

Maria is impressed by your summary DataFrame but wants to add some formatting to make the DataFrame look more professional. You'll need to format the budget to two decimal places; format the grade averages to one decimal place and grade percentages to the nearest whole number percent; and add a thousands separator for numbers greater than 1,000.

To clean up the `district_summary_df` DataFrame, we will format dollar amounts to two decimal places, and format the grade averages to one decimal place and percentages to the nearest whole number percent.

> **Note**
> Grade averages are usually formatted to one decimal place because averages taken to the hundredths (range: 0.01–0.09) or lower do not impact an average as much as tenths of a grade point, (range: 0.1–0.9). When tenths of a grade point equal or are greater than 0.5, you should usually round up to the next whole number.
>
> Grade percentages are formatted to the whole percentage because the tenths of a percent is equivalent to thousandths of a value, or grade. Like averages, thousandths of a grade don't have an impact on the overall percentage because they are so small.

This type of formatting can be done with the built-in Python `map()` function. The `map()` function returns a list of the results after applying a given function to each item that we can iterate over, like a list or a DataFrame column.

In this module, as well as future modules, we'll be using and writing functions. So before we use the `map()` function for formatting, let's gain a better understanding of how functions work.

Overview of Functions

When writing algorithms that perform a repetitive task, like getting grade averages over and over again, it's a best practice to convert the algorithm to a function. A **function** is a smaller, more manageable piece of code.

Long algorithms can increase the chance of syntax errors and repeated variable assignments, which can lead to programming headaches—especially if your code is more than 200 lines. Breaking your code into manageable pieces has many advantages. You can:

- Debug and fix errors faster.
- Make your code more readable.
- Reuse code by importing functions into other algorithms.
- Speed up programming development. Multiple team members can work on separate functions of a complex algorithm to speed up the development of the project.

A hallmark of professional programmers is that they can take long blocks of code and write smaller pieces of that code as functions.

There are four basic parts to a function:

1. The name, which is what we call the function
2. The parameters, which are values we send to the function
3. The code block, which are the statements under the function that perform the task
4. The return value, which is what the function gives back, or "returns," to use when the task is complete

Naming Functions

The name of a function can be almost anything you want, but it should be descriptive enough so that anyone reading your code can reasonably guess what the function does. It's generally thought of as a best coding practice to write the name of the function as a verb, as functions perform actions. However, Python requires that you follow the same rules that you follow when naming variables.

> **Rewind**
> Variable names in Python can be any length and can consist of the following:
> - Uppercase letters (A–Z)
> - Lowercase letters (a–z)
> - Digits (0–9)
> - Underscores (_). Also, the first character of a variable name cannot be a digit.

When it comes to naming functions, there are two additional rules and two caveats to the variable naming rules.

- The name cannot be a Python keyword.
- The name cannot contain spaces.
- The first character of the name must be an uppercase or lowercase letter or an underscore.
- After the first character, you can use uppercase and lowercase letters, digits 0 through 9, or an underscore.

Parts of a Function

Let's review the parts of a function. We'll use the simple function below as an example.

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")
```

Let's go over what is happening in this function.

- The `def` keyword, which is short for "define," and is used to create the function.
- The function name, `say_hello(),` is always written after `def`.
- There is a colon after the function name, which tells Python that a code block or statement will follow. This is similar to conditional and repetition statements. All code within the function should be indented.
- The indented statement that follows is `print("Hello!")`.

Calling a Function

Now we'll practice calling a function. Create a new Jupyter Notebook file and name it `function.ipynb`. Copy the following code for the `say_hello()` function into the file.

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")
```

When you run the code, nothing appears to happen because we don't see an output error. To get the function to print "Hello!" in the output cell, we have to **call** the function.

To call the `say_hello()` function, type `say_hello()` in a new cell and run the cell.

```
# Call the function.
say_hello()
```

This will print "Hello!" to the output cell:

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")
```

```
# Call the function.
say_hello()
```

```
Hello!
```

Now we'll build on our knowledge of functions and add a parameter inside the parentheses of a new function, `say_something()`. A **parameter** is a value that we send to the function.

Add the following code to a new cell and run the cell.

```
# Define the function "say_something" so it prints whatever is passed as the
def say_something(something):
    print(something)
```

Inside the parentheses, we added the variable `something`. When this function gets called, it will print whatever we have added inside the parentheses of the function. This doesn't have to be the word "something"; it can be whatever you want it to be.

In the next cell, add "Hello World" inside the parentheses of the function, `say_something()`.

```
# Call the function.
say_something("Hello World")
```

When we call the function, i.e., run the cell, "Hello World" is printed to the output.

Like the string "Hello World," we can also pass a variable inside the function and have that variable printed to the output. In the following code, Jane introduces herself with the string "Hi, my name is Jane. I'm learning Python!" We can print her introduction by adding the variable `Jane_says` inside the `say_something()` function:

```
Jane_says = "Hi, my name is Jane. I'm learning Python!"
say_something(Jane_says)
```

When we run this cell, we get "Hi, my name is Jane. I'm learning Python!" in the output.

```
Jane_says = "Hi, my name is Jane. I learning Python!"
say_something(Jane_says)
```

```
Hi, my name is Jane. I learning Python!
```

Writing Functions for the School District Data

We've reviewed some basics about functions. Now how do we apply functions to our data analysis? For example, consider if you were in charge of calculating passing percentages for not just the school district we've been focusing on, but many school districts. It wouldn't be efficient to perform the steps of the calculation for each school district. A better method would be to create functions for this calculation and pass variables into the function.

Let's say you need to write a function to get the percentage of students who passed math when we know the number of students that passed math, `pass_math_count`, and the total number of students for a school district, `student_count`. We can define this function as `passing_math_percent` and pass the two values inside the function. Here's what this might look like:

```
# Define a function that calculates the percentage of students that passed b
# function is called.
def passing_math_percent(pass_math_count, student_count):
    return pass_math_count / float(student_count) * 100
```

Let's go over what is happening in this function.

1. We added two values to the `passing_math_percent` function: `pass_math_count` and `student_count`.
2. We added `return` in front of the calculation for the passing percentage.

The `return` statement has a unique purpose. It causes the function to end and literally "returns" what is in front of the statement back to the caller, which is the function. Let's run through an example of what happens to better illustrate the process.

In a new cell, assign the `passing_math_count` and the `total_student_count` variables 29,730 and 39,170, respectively. When we run this cell, that nothing will happen until we call the function.

```
passing_math_count = 29370
total_student_count = 39170
```

Now we can call the function in a new cell with the two values, like this:

```
# Call the function.
passing_math_percent(passing_math_count, total_student_count)
```

When we run this cell, the calculated percentage is printed to the output window.

```
# Call the function.
passing_math_percent(passing_math_count,total_student_count)
```

```
74.9808526933878
```

> **IMPORTANT!**
> If you don't add the number of parameters that are assigned to the function when calling the function, you'll get a `TypeError:`. This means that you need to add one or more of the parameters when calling the function.

Now that you have a good handle on how to write and use functions, let's apply this knowledge to the `map()` function, which we'll use to format our data.

The map() Function

Let's look at an example of the `map()` function in two steps.

First, in `functions.ipynb`, add the following code:

```
# A basic function that squares a number.
#1 def sqr(x):
#2     return x**2
#3
#4 sqr(2)
```

Let's break down what is happening in this code.

- In line #1, we define a function named `sqr()`. Inside the `sqr()` function, we pass a variable, x.
- In line #2, we write a statement to have the function do something, namely, square "x," and return it to the output when the function is called.
- In line #4, we call the function by writing it out and add a number, or argument, inside the function.

> **REWIND**
> A parameter is the input given to the function.

When we execute this code, we get 4.

Next, to illustrate how the `map()` function works, we'll take the `sqr` function and use it inside the `map()` function, as shown in the following example:

```
# Example of the map function.
#1 numbers = [1, 2, 3, 4, 5]
#2
#3 def sqr(x):
#4    return x**2
#5
#6 list(map(sqr, numbers))
```

Here's what's happening in this code, line by line:

- In line #1, we create a list of numbers.
- In lines #3 and #4, we add the `sqr` function and everything below it as we did before.
- In line #6, we create a new list using the `list()` function. Inside the `list()` function, we add the `map()` function. Inside the `map()` function, we add the `sqr` function and the list of numbers.

When we execute this code, we get a new list that has each number from the `numbers` list squared:

```
numbers = [1, 2, 3, 4, 5]

def sqr(x):
    return x**2

list(map(sqr, numbers))
```

```
[1, 4, 9, 16, 25]
```

To get the same results without using the `map()` function, we would have to iterate through the `numbers` list, square each number, and then add it to a new list. In fact, using the `map()` function is equivalent to using for loops, like this:

```
# A for loop that squares a number.
numbers = [1, 2, 3, 4, 5]
numbers_sqr = []

for number in numbers:
    numbers_sqr.append(number**2)

print(numbers_sqr)
```

This is a little more coding, and it gets the same results. However, using a function to do this is more compact, cleaner, and can be used many times.

> **Note**
> For more information, see the **Python documentation on the map() function (https://docs.python.org/3/library/functions.html#map)** .

The format() Function

The `format()` function is used to format a value to a specific format, such as one or two decimal places, or adding a thousands separator. This function follows this syntax:

`"{value:format specification}".format(value)`

The “value” within the curly braces represents the value that we want to format. Usually, the value in the `format(value)` is a variable. After the colon, the `format specification` will designate how the value should be formatted.

Let’s look at an example of the `format()` function.

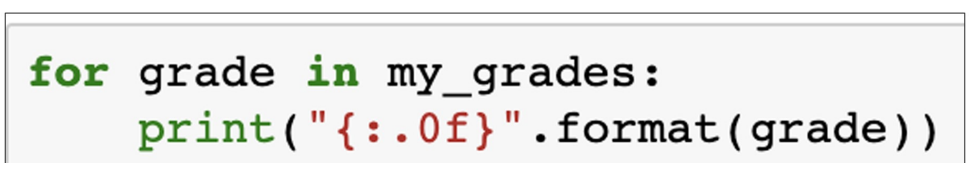
In this example, we are going to iterate through a list of grades and format each grade to the nearest whole number percent. Copy the following code, add to a new cell in the `functions.ipynb` file, and run it.

```
# Using the format() function.
my_grades = [92.34, 84.56, 86.78, 98.32]

for grade in my_grades:
    print("{:.0f}".format(grade))
```

In this code, there is a list of grades formatted to two decimal places. To format these grades to the nearest whole number percent, we will do the following:

- Iterate through the grades.
- Pass the `grade` variable inside the `format()` function.
- Specify the format for the `grade` variable we would like by using `{:.0f}`.
  - In this format, the `grade` variable is referenced in front of the colon, so there is no need to add the `grade` variable.
  - After the colon, the `.0f` means to format the grade with no decimal place, where the “period” is for the decimal place, the “0” is for “no” decimal place, and the “f” means floating-point decimal. If we wanted to format to one decimal place, we would use 1 instead of 0, and 2 for two decimal places, and so on.

When we run the example code, we get the grades formatted to the nearest whole number percent:



```
92
85
87
98
```

> **Note**
> For more information about formatting, read the following Python documentation:
>
> - **Format Examples** (https://docs.python.org/3.4/library/string.html#format-examples)
> - **Format Specification** (https://docs.python.org/3.4/library/string.html#format-specification-mini-language)

Chaining the map() and format() Functions

Now we will format the dollar amounts in the `district_summary_df` columns to two decimal places with a U.S. dollar sign, and numbers larger than 999 with a thousands separator. To do this, we'll chain the `map()` and `format()` functions to make our code more concise.

Let's apply this formatting technique to format the Total Students column with a thousands separator using the basic syntax:

```
df["column"] = df["column"].map("{:,}".format)
```

Now let's replace `df["column"]` with `district_summary_df["Total Students"]`, so our code looks like the following. Add this code to you `PyCitySchools.ipynb` file below the output of your `district_summary_df` DataFrame, and run the cell.

```
# Format the "Total Students" to have the comma for a thousands separator.
district_summary_df["Total Students"] = district_summary_df["Total Students"
```

```
district_summary_df["Total Students"]
```

When we execute this code, we get the following output:

```
1  district_summary_df["Total Students"]
```

```
0     39,170
Name: Total Students, dtype: object
```

One benefit of using the `format()` function is that we can add other format specifications, like a U.S. dollar sign or other characters. For example, we will format the Total Budget column in the `district_summary_df` DataFrame with a U.S. dollar sign, and then format the numbers with a thousands separator and to two decimal places using the following syntax: `"${:,.2f}".format`.

Add the following code to a new cell in our `PyCitySchools.ipynb` file and run the cell.

```
# Format "Total Budget" to have the comma for a thousands separator, a decim

district_summary_df["Total Budget"] = district_summary_df["Total Budget"].ma

district_summary_df["Total Budget"]
```

When we execute this code, we get the following output:

```
1  district_summary_df["Total Budget"]
```

```
0     $24,649,428.00
Name: Total Budget, dtype: object
```

We need to format the remaining columns accordingly:

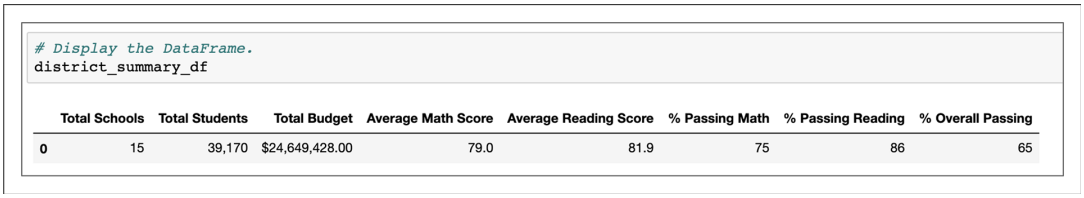
- The “Average Reading Score” column will be formatted to one decimal place.

- The “Average Math Score” column will be formatted to one decimal place.
- The “% Passing Reading” column will be formatted to the nearest whole number percentage.
- The “% Passing Math” column will be formatted to the nearest whole number percentage
- The “% Overall Passing” column will be formatted to the nearest whole number percentage.

Here’s what the code for these columns will look like:

```
# Format the columns.
district_summary_df["Average Math Score"] = district_summary_df["Average Mat

district_summary_df["Average Reading Score"] = district_summary_df["Average

district_summary_df["% Passing Math"] = district_summary_df["% Passing Math"

district_summary_df["% Passing Reading"] = district_summary_df["% Passing Re

district_summary_df["% Overall Passing"] = district_summary_df["% Overall Pa
```

When you run the code, the `district_summary_df` DataFrame should look like this:



```
# Display the DataFrame.
district_summary_df
```

	Total Schools	Total Students	Total Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
0	15	39,170	$24,649,428.00	79.0	81.9	75	86	65

> **Note**
>
> If the columns in your `district_summary_df` DataFrame are not in the order as shown, then you will have to put them in the correct order.

4.7.9: Reorder Columns

After you format the columns, Maria requests that you put them in top-down order. Her preference is to have the DataFrame show the school information first—like total schools, total students, and total budget—and then show the student data, such as average scores and the percentage passing.

If the order of your columns in the district summary DataFrame is not in top-down order, you need to reorder them. If the columns are already in this order, you can use this time to gain familiarity with how to reorder columns in case the need arises.

The order of the columns in the `district_summary_df` DataFrame should be as follows:

- Total Schools
- Total Students
- Total Budget
- Average Math Score
- Average Reading Score
- % Passing Math
- % Passing Reading
- % Overall Passing

To reorder columns using Pandas, we can pass a list of columns to a current DataFrame using **square bracket notation**. This tells Pandas to select those specific columns and put them in the DataFrame in the same order that they appear in the list.

Here is the standard format for reordering columns:

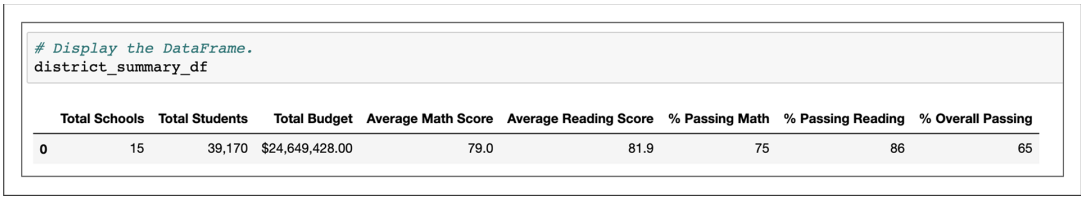
```
# Reorder the columns in the order you want them to appear.
new_column_order = ["column2", "column4", "column1"]

# Assign a new or the same DataFrame the new column order.
df = df[new_column_order]
```

This technique can also be used to filter out columns that you don't need and select the columns in the order that you want. For example, we didn't include `column3` in the new order; this is how we can filter out columns we don't want.

To get the columns in the correct order, add the following code and run the cell.

```
# Reorder the columns in the order you want them to appear.
new_column_order = ["Total Schools", "Total Students", "Total Budget","Avera

# Assign district summary df the new column order.
district_summary_df = district_summary_df[new_column_order]
district_summary_df
```

The DataFrame in the output cell should look like this:



```
# Display the DataFrame.
district_summary_df
```

	Total Schools	Total Students	Total Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
0	15	39,170	$24,649,428.00	79.0	81.9	75	86	65

Your district summary columns are in the correct order!

4.7.10: Commit Your Code

You just completed a lot of work, so it's time to update your GitHub repository. This will allow Maria and other stakeholders, such as district officials, to stay up-to-date on your progress.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "district summary complete."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

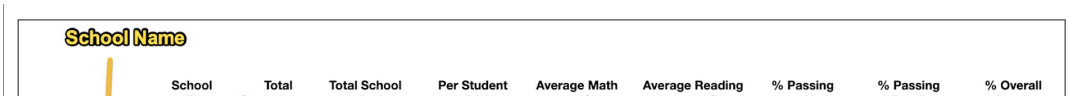
4.8.1: Set the Index to the School Name

Maria is impressed with the work you have done generating the district summary. Now she would like to generate a similar summary for each school in the district.

This next project requires you to get the following key metrics for each school and place them in a school summary DataFrame. As a reminder, here are the key metrics you're working with:

- School name
- School type
- Total students
- Total school budget
- Per student budget
- Average math score
- Average reading score
- % passing math
- % passing reading
- % overall passing

The school summary DataFrame should look like the image below. You'll see that the `school_name` is now the index for this DataFrame. This is important, because as we perform calculations and gather data, we will be creating either Series or DataFrames that need to have the `school_name` as the index.



	Type	Students	Budget	Budget	Score	Score	Math	Reading	Passing
Bailey High School	District	4976	$3,124,928.00	$628.00	77.048432	81.033963	66.680064	81.933280	54.642283
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476
Ford High School	District	2739	$1,763,916.00	$644.00	77.102592	80.746258	68.309602	79.299014	54.289887
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455

Now we'll create a new DataFrame. As mentioned, the index needs to be the `school_name` with "School Type" as the first column.

We can use the data from the `school_data_df` DataFrame because the "school_name" and "type" columns are both Series:

	School ID	school_name	type	size	budget
0	0	Huang High School	District	2917	1910635
1	1	Figueroa High School	District	2949	1884411
2	2	Shelton High School	Charter	1761	1056600
3	3	Hernandez High School	District	4635	3022020
4	4	Griffin High School	Charter	1468	917500
5	5	Wilson High School	Charter	2283	1319574
6	6	Cabrera High School	Charter	1858	1081356
7	7	Bailey High School	District	4976	3124928
8	8	Holden High School	Charter	427	248087
9	9	Pena High School	Charter	962	585858
10	10	Wright High School	Charter	1800	1049400
11	11	Rodriguez High School	District	3999	2547363
12	12	Johnson High School	District	4761	3094650
13	13	Ford High School	District	2739	1763916
14	14	Thomas High School	Charter	1635	1043130

We'll take these Series and use one as the index and the other as the first column. Add the following code to a new cell and run the cell.

```
# Determine the school type.
per_school_types = school_data_df.set_index(["school_name"])["type"]
per_school_types
```

In this code, we are setting the index to the `school_name` column with the `set_index` method. This method will return a Series with the index as the `school_name` and a column with the type of school, like this:

```
school_name
Huang High School        District
```

```
Figueroa High School     District
Shelton High School       Charter
Hernandez High School    District
Griffin High School       Charter
Wilson High School        Charter
Cabrera High School       Charter
Bailey High School       District
Holden High School        Charter
Pena High School          Charter
Wright High School        Charter
Rodriguez High School    District
Johnson High School      District
Ford High School         District
Thomas High School        Charter
Name: type, dtype: object
```

Now we'll create a new DataFrame by converting this Series to a DataFrame as follows:

```
# Add the per_school_types into a DataFrame for testing.
df = pd.DataFrame(per_school_types)
df
```

When we run this cell, we get a DataFrame with "school_name" as the index and "School Types" as a column.

school_name	School Types
Huang High School	District
Figueroa High School	District
Shelton High School	Charter
Hernandez High School	District
Griffin High School	Charter
Wilson High School	Charter
Cabrera High School	Charter
Bailey High School	District
Holden High School	Charter
Pena High School	Charter
Wright High School	Charter
Rodriguez High School	District
Johnson High School	District
Ford High School	District
Thomas High School	Charter

You created the school summary DataFrame—nice work! Now we can start adding the other columns to this DataFrame.

Important!

As we get the rest of the data for this DataFrame, we need to make sure the index for the data being added as columns is always "school_name."

Note

For more information, see the **Pandas documentation on the set_index() method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html)** .

4.8.2: Get the Student Count Per School

Next to the "type" of school column, Maria wants you to add the total number of students in each school. However, you have the number of students in two DataFrames: `school_data_df` and `school_data_complete_df`. Which one do you use? You'll need to get the number of students from both DataFrames and find which one has "school_name" as the index.

There are two DataFrames that have the total number of students per school: `school_data_df` and `school_data_complete_df`. First, we will use the `school_data_df` DataFrame.

Rewind

Recall that the `school_data_df` DataFrame looks like this:

	School ID	school_name	type	size	budget
0	0	Huang High School	District	2917	1910635
1	1	Figueroa High School	District	2949	1884411
2	2	Shelton High School	Charter	1761	1056600
3	3	Hernandez High School	District	4635	3022020
4	4	Griffin High School	Charter	1468	917500
5	5	Wilson High School	Charter	2283	1319574
6	6	Cabrera High School	Charter	1858	1081356
7	7	Bailey High School	District	4976	3124928
8	8	Holden High School	Charter	427	248087
9	9	Pena High School	Charter	962	585858
10	10	Wright High School	Charter	1800	1049400
11	11	Rodriguez High School	District	3999	2547363
12	12	Johnson High School	District	4761	3094650

13	13	Ford High School	District	2739	1763916
14	14	Thomas High School	Charter	1635	1043130

Looking at the `school_data_df` DataFrame, we can see the student count is in the “size” column.

Now let’s get the student count in the `school_data_df` DataFrame. Add the following code to a new cell and run the cell.

```
# Calculate the total student count.
per_school_counts = school_data_df["size"]
per_school_counts
```

When we run this cell, we get the following output: a Series showing the number of students in each school with a numerical index (0–14), which is the same as the index of the `school_data_df` DataFrame.

```
0     2917
1     2949
2     1761
3     4635
4     1468
5     2283
6     1858
7     4976
8      427
9      962
10    1800
11    3999
12    4761
13    2739
14    1635
Name: size, dtype: int64
```

Unfortunately, this Series doesn’t have an index with "school_name." Therefore, we can’t use the "size" column from `school_data_df` to get the count of the student population.

To fix this, we can use the `set_index()` method on the "school_name" column, and then select the "size" column to display the student count for each school.

Add the following code to a new cell and run the cell.

```
# Calculate the total student count.
per_school_counts = school_data_df.set_index(["school_name"])["size"]
per_school_counts
```

When we run this cell, the output shows the `school_name` as the index and has the number of students as the data.

```
school_size = school_data_df.set_index(["school_name"])["size"]
school_size
```

```
school_name
Huang High School         2917
Figueroa High School      2949
Shelton High School       1761
Hernandez High School     4635
Griffin High School       1468
Wilson High School        2283
Cabrera High School       1858
Bailey High School        4976
Holden High School         427
Pena High School           962
Wright High School        1800
Rodriguez High School     3999
Johnson High School       4761
Ford High School          2739
Thomas High School        1635
Name: size, dtype: int64
```

Next, we'll use the `school_data_complete_df` DataFrame.

Rewind

Recall that the `school_data_complete_df` DataFrame looks like this:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
1	1	Victor Smith	M	12th	Huang High School	94	61	0	District	2917	1910635
2	2	Kevin Rodriguez	M	12th	Huang High School	90	60	0	District	2917	1910635
3	3	Richard Scott	M	12th	Huang High School	67	58	0	District	2917	1910635
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635

To get the number of students from the `school_data_complete_df` DataFrame, we can count the number of times a high school appears using `value_counts()` on the "school_name" column. The `value_counts()` method will return a Series of data with the number of times each `school_name` appears in a row.

To get the total student count per high school, add the following code to a new cell:

```
# Calculate the total student count.
per_school_counts = school_data_complete_df["school_name"].value_counts()
per_school_counts
```

This code returns a Series with the number of times the school appeared in the "school_name" column.

```
Bailey High School       4976
Johnson High School      4761
Hernandez High School    4635
Rodriguez High School    3999
Figueroa High School     2949
Huang High School        2917
Ford High School         2739
Wilson High School       2283
Cabrera High School      1858
Wright High School       1800
Shelton High School      1761
Thomas High School       1635
Griffin High School      1468
Pena High School          962
Holden High School        427
Name: school_name, dtype: int64
```

Now we have two methods to get the student count. As long as we make sure the index is the "school_name" column, we can use either method.

> **Note**
>
> For more information, see the [Pandas documentation on value_counts() (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html) .

4.8.3: Get the Budget Per Student

Next to the Total Students column, Maria wants you to add the budget per student for each school. First you'll need to get the budget for each school and then divide by the total students per school, which you already calculated as `per_school_counts`.

In order to find the budget per student for each school, we need to divide the budget for each school by the number of students at each school.

If we look at the `school_data_df` DataFrame. The budget for each school is listed in the `budget` column.

> **Rewind**
> We can use the `set_index()` method on the `school_name` column of the `school_data_df` DataFrame to get the `school_name` as the index, like we did when we calculated the total students from each school, with `school_data_df.set_index(["school_name"])["size"]`.

Add the following code to a new cell and run the cell.

```
# Calculate the total school budget.
per_school_budget = school_data_df.set_index(["school_name"])["budget"]
per_school_budget
```

After we execute this code for the `per_school_budget`, the results should look like this:

```
per_school_budget = school_data_df.set_index(["school_name"])["budget"]
per_school_budget
```

```
school_name
Huang High School         1910635
Figueroa High School      1884411
Shelton High School       1056600
Hernandez High School     3022020
Griffin High School        917500
Wilson High School        1319574
Cabrera High School       1081356
Bailey High School        3124928
Holden High School         248087
Pena High School           585858
Wright High School        1049400
Rodriguez High School     2547363
Johnson High School       3094650
Ford High School          1763916
Thomas High School        1043130
Name: budget, dtype: int64
```

The data type for the `budget` column is `int64`, which is suitable for calculations that we need to perform in order to find the budget per student.

To get the budget per student, we'll divide the `per_school_budget` by the `per_school_counts`.

Add the following code to a new cell and run the cell.

```
# Calculate the per capita spending.
per_school_capita = per_school_budget / per_school_counts
per_school_capita
```

We can perform this calculation because the `per_school_budget` and `per_school_counts` are Series, both data types are `int64`, and both have the same index.

When we execute this code, we get a Series with the school name as the index with a column showing the budget per student.

```
per_school_capita
```

```
Bailey High School       628.0
Cabrera High School      582.0
Figueroa High School     639.0
Ford High School         644.0
Griffin High School      625.0
Hernandez High School    652.0
Holden High School       581.0
```

```
Huang High School         655.0
Johnson High School       650.0
Pena High School          609.0
Rodriguez High School     637.0
Shelton High School       600.0
Thomas High School        638.0
Wilson High School        578.0
Wright High School        583.0
dtype: float64
```

Nice work! Next, we will get the grade averages from each school.

4.8.4: Get the Score Averages Per School

You're almost done with getting all the data for the school summary! Now you need to calculate the average math score and the average reading score for each school.

Next, we need to perform a few more calculations for the final data to be added to the school summary DataFrame.

First, let's get the average reading and math scores for each school.

To calculate the average math and reading test scores for each school, what method should you use for the math or reading score columns?

- ○ `count()` method
- ○ `sum()` method
- ○ `mean()` method
- ○ `avg()` method

Check Answer

Finish ▸

Rewind

Make sure that the averages have an index of `school_name` so the data can be added to the DataFrame.

We've used the `set_index()` method on the school_name column in `student_data_df` to get data from another column, just like how we retrieved the school budget using `school_data_df.set_index(["school_name"])["budget"]`.

Let's use this procedure to replace `budget` with `math_score`. Add the following code to a new cell and run the cell.

```
# Calculate the math scores.
student_school_name = student_data_df.set_index(["school_name"])["math_score
```

The output from the code will look like the following, where we get every occurrence of the high school as the index, and the math grade from each student in that school.

```
student_school_math = student_data_df.set_index(["school_name"])["math_score"]
student_school_math
```

```
school_name
Huang High School     79
Huang High School     61
Huang High School     60
Huang High School     58
Huang High School     84
Huang High School     94
Huang High School     80
Huang High School     69
```

Unfortunately, we can't use the `school_data_df` DataFrame, as there aren't any columns containing grades. We also can't use the `set_index()` method on the school_name column in `student_data_df` because there are too many occurrences of the school_name column.

Instead, we need to use the Pandas `groupby()` function. The `groupby()` function will split an object (like a DataFrame), apply a mathematical operation, and combine the results. This can be used to group large amounts of data when we want to compute mathematical operations on these groups.

The mathematical operation we will apply to the `groupby()` function is the `mean()` method. Let's see how this will look when we apply it to `school_data_complete_df` to get the grade averages for each column. Add the following code to a new cell and run the cell.

```
# Calculate the average math scores.
per_school_averages = school_data_complete_df.groupby(["school_name"]).mean(
per_school_averages
```

The output will be the average of each column in the `school_data_complete_df` DataFrame:

```
per_school_averages = school_data_complete_df.groupby(["school_name"]).mean()
per_school_averages
```

school_name	Student ID	reading_score	math_score	School ID	size	budget
Bailey High School	20358.5	81.033963	77.048432	7.0	4976.0	3124928.0
Cabrera High School	16941.5	83.975780	83.061895	6.0	1858.0	1081356.0
Figueroa High School	4391.0	81.158020	76.711767	1.0	2949.0	1884411.0
Ford High School	36165.0	80.746258	77.102592	13.0	2739.0	1763916.0
Griffin High School	12995.5	83.816757	83.351499	4.0	1468.0	917500.0
Hernandez High School	9944.0	80.934412	77.289752	3.0	4635.0	3022020.0
Holden High School	23060.0	83.814988	83.803279	8.0	427.0	248087.0
Huang High School	1458.0	81.182722	76.629414	0.0	2917.0	1910635.0
Johnson High School	32415.0	80.966394	77.072464	12.0	4761.0	3094650.0
Pena High School	23754.5	84.044699	83.839917	9.0	962.0	585858.0
Rodriguez High School	28035.0	80.744686	76.842711	11.0	3999.0	2547363.0
Shelton High School	6746.0	83.725724	83.359455	2.0	1761.0	1056600.0
Thomas High School	38352.0	83.848930	83.418349	14.0	1635.0	1043130.0
Wilson High School	14871.0	83.989488	83.274201	5.0	2283.0	1319574.0
Wright High School	25135.5	83.955000	83.682222	10.0	1800.0	1049400.0

But we don't want all of this data in the school summary DataFrame, just the reading and math scores. To get the average math score and reading score for each school, we can add the `math_score` and `reading_score` columns at the end. Add the following code to a new cell and run the cell.

```
# Calculate the average test scores.
per_school_math = school_data_complete_df.groupby(["school_name"]).mean()["m

per_school_reading = school_data_complete_df.groupby(["school_name"]).mean()
```

When we run this cell and reference each Series, we get a Series like the other Series we have created, where the index is on the `school_name`, and the column is the average `math_score` or average `reading_score`.

The Series with the average math scores for each school will look like this:

```
per_school_math

school_name
Bailey High School       77.048432
Cabrera High School      83.061895
Figueroa High School     76.711767
Ford High School         77.102592
Griffin High School      83.351499
Hernandez High School    77.289752
Holden High School       83.803279
Huang High School        76.629414
Johnson High School      77.072464
Pena High School         83.839917
Rodriguez High School    76.842711
Shelton High School      83.359455
Thomas High School       83.418349
Wilson High School       83.274201
Wright High School       83.682222
Name: math_score, dtype: float64
```

The `per_school_reading` results will have the same format, with the column being the average `reading_score`.

> **Note**
>
> For more information, read the [Pandas documentation on the groupby() function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html).

4.8.5: Get the Passing Percentages Per School

Great work on finding the average math and reading scores for every high school. Now Maria would like you to continue gathering key data by calculating the passing percentages for math and reading for each school, as well as get the overall passing percentage for each school. She goes over what you will need in order to perform these calculations, writing the pseudocode on a whiteboard.

Here's the pseudocode from Maria:

```
# To get the passing percentages, we need to:
# 1. Determine what is the passing grade.
# 2. Get the number of students who passed math and reading.
# 3. Get the students who passed math and passed reading
```

We're in luck! We have already determined the passing grade as well as calculated the number of students who passed math and reading.

Determine the Passing Grade

> **Rewind**
> Remember, for assessment tests, the passing score is 70, with the `>= 70` statement to filter the grades that are passing.

Get the Number of Students Who Passed Math and Reading

> **Rewind**
>
> We determined the number of students who passed math and reading for the district summary using the following code:
>
> `passing_math = school_data_complete_df[school_data_complete_df["math_score"] >= 70]`
>
> and
>
> `passing_reading = school_data_complete_df[school_data_complete_df["reading_score"] >= 70]`

So variables don't get reassigned, we'll use the same code but assign each calculation to new variables that reflect the students who passed math or reading for each school.

Add the following code to a new cell and run the cell.

```
# Calculate the passing scores by creating a filtered DataFrame.
per_school_passing_math = school_data_complete_df[(school_data_complete_df["

per_school_passing_reading = school_data_complete_df[(school_data_complete_d
```

When we execute this cell and print the `per_school_passing_math`, we get a DataFrame that looks like this:

```
per_school_passing_math.head()
```

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635
5	5	Bryan Miranda	M	9th	Huang High School	94	94	0	District	2917	1910635
6	6	Sheena Carter	F	11th	Huang High School	82	80	0	District	2917	1910635
8	8	Michael Roth	M	10th	Huang High School	95	87	0	District	2917	1910635

However, we need to get the average reading and math scores for each school. So, the index needs to be the `school_name`, and we need to get the number of students in the `per_school_passing_math` and the `per_school_passing_reading` DataFrames.

From the `per_school_passing_math` and the `per_school_passing_reading` DataFrames. How can we get the index to be the `school_name`?

- ○ Use `index(["school_name"])` method.
- ○ Use `groupby(["school_name"])`
- ○ Use `sort(["school_name"])`.
- ○ Use `set_index(["school_name"])`

Check Answer

Finish ▸

To create a Series that has `school_name` as the index, we'll use `groupby(["school_name"])` on the `per_school_passing_math` and the `per_school_passing_math` DataFrames.

> **Rewind**
> Using the `groupby()` function will split an object (like a DataFrame) and apply a mathematical operation.

Next, we need to perform a mathematical operation on the `groupby()` object. Remember, we need to get the number of students who passed, so we'll use the `count()` method on the student_name column. Add the following code to a new cell and run the cell.

```
# Calculate the number of students passing math and passing reading by schoo
per_school_passing_math = per_school_passing_math.groupby(["school_name"]).c

per_school_passing_reading = per_school_passing_reading.groupby(["school_nam
```

When we print the `per_school_passing_math`, the output will be a Series where the index is the school name and the column is the number of students who passed math for each school:

```
per_school_passing_math

school_name
Bailey High School       3318
Cabrera High School      1749
Figueroa High School     1946
Ford High School         1871
Griffin High School      1371
Hernandez High School    3094
Holden High School        395
Huang High School        1916
Johnson High School      3145
Pena High School          910
Rodriguez High School    2654
Shelton High School      1653
Thomas High School       1525
Wilson High School       2143
Wright High School       1680
Name: student_name, dtype: int64
```

The `per_school_passing_reading` results will have the same format, with the column being the number of students who passed reading for each school.

Determine the Percentage of Students Passing Math and Reading

To determine the percentage of students passing math and reading, we must divide `per_school_passing_math` and `per_school_passing_reading` by the `per_school_counts`, and then multiply by 100, as shown in the following code:

```
# Calculate the percentage of passing math and reading scores per school.
per_school_passing_math = per_school_passing_math / per_school_counts * 100

per_school_passing_reading = per_school_passing_reading / per_school_counts
```

This time, when we print the `per_school_passing_math`, the output will be a Series where the index is the school name and the column is the percentage of students who passed math for each school.

```
per_school_passing_math
```

```
Bailey High School       66.680064
Cabrera High School      94.133477
Figueroa High School     65.988471
Ford High School         68.309602
Griffin High School      93.392371
Hernandez High School    66.752967
Holden High School       92.505855
Huang High School        65.683922
Johnson High School      66.057551
Pena High School         94.594595
Rodriguez High School    66.366592
Shelton High School      93.867121
Thomas High School       93.272171
Wilson High School       93.867718
Wright High School       93.333333
dtype: float64
```

The `per_school_passing_reading` results will have the same format, with the column being the percentage of students who passed reading.

Get the Overall Passing Percentage for All Students for Each School

> **Rewind**
>
> To get the overall passing percentage, we need to get all the students who passed both math and reading and then divide by the total number of students.

The code to calculate the overall passing percentage is as follows:

```
# Calculate the students who passed both math and reading.
per_passing_math_reading = school_data_complete_df[(school_data_complete_df[

per_passing_math_reading.head()
```

When we run this cell, the output will be a new DataFrame with all the columns that contain students who passed both math and reading.

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635

5	5	Bryan Miranda	M	9th	Huang High School	94	94	0	District	2917	1910635
6	6	Sheena Carter	F	11th	Huang High School	82	80	0	District	2917	1910635
8	8	Michael Roth	M	10th	Huang High School	95	87	0	District	2917	1910635
9	9	Matthew Greene	M	10th	Huang High School	96	84	0	District	2917	1910635

Next, we'll get the total number of students from each school who passed both math and reading. To do this, we need to set the index as `school_name`, and then we need to get the number of students.

Complete the code to set the index to school_name.

```
per_passing_math_reading =

per_passing_math_reading.                    (["school_name"])
```

Check Answer

Finish ▸

After setting the index to school_name, complete the code to get the number of students who passed both math and reading.

```
per_passing_math_reading =

per_passing_math_reading.groupby(["school_name"]).                    ["student_name"]
```

Check Answer

Finish ▸

To get the total number of students who passed both math and reading on the `per_passing_math_reading` DataFrame, we use the following code, which sets the index to `school_name`, and then use the `count()` method for the `student_name`.

```
# Calculate the number of students who passed both math and reading.
per_passing_math_reading = per_passing_math_reading.groupby(["school_name"])
```

Finally, we calculate the percentage of students who passed math and reading by dividing the the total number of students and multiplying by 100, using the following code.

```
# Calculate the overall passing percentage.
per_overall_passing_percentage = per_passing_math_reading / per_school_count
```

When we execute this cell, we get a Series with the overall passing percentage for each school.

```
per_overall_passing_percentage

Bailey High School       54.642283
Cabrera High School      91.334769
Figueroa High School     53.204476
Ford High School         54.289887
Griffin High School      90.599455
Hernandez High School    53.527508
Holden High School       89.227166
Huang High School        53.513884
Johnson High School      53.539172
Pena High School         90.540541
Rodriguez High School    52.988247
Shelton High School      89.892107
Thomas High School       90.948012
Wilson High School       90.582567
Wright High School       90.333333
dtype: float64
```

Congratulations––you've gathered all of the data you need! Now we can add the data to a new DataFrame.

4.8.6: Create the School Summary DataFrame

Maria is eager to see the school summary, as are other district officials. You'll need to combine all the data you retrieved and calculated and put this data in a new DataFrame.

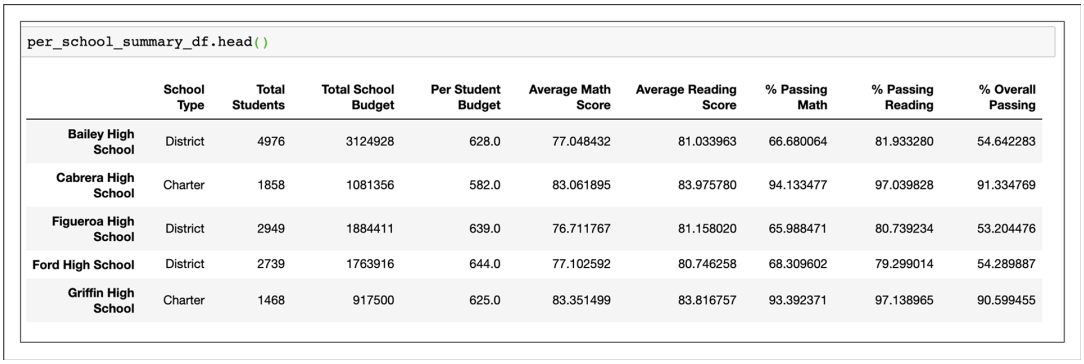
Now that we have performed all the calculations needed for the school summary, we can add the following values to a new DataFrame named `per_school_summary_df`. The data and columns of the DataFrame will be:

- Type of school in the "School Type" column
- Total students per school in the "Total Students" column
- Total budget per school in the "Total School Budget" column
- Total budget per student for each school in the "Per Student Budget" column
- Average math score for each school in the "Average Math Score" column
- Average reading score for each school in the "Average Reading Score" column
- Percentage of students passing math for each school in the "% Passing Math" column
- Percentage of students passing reading for each school in the "% Passing Reading" column
- Overall passing percentage for each school in the "% Overall Passing" column

> **Rewind**
>
> To create `per_school_summary_df`, the column names will be the keys, and the values will be each piece of data we retrieved or calculated.

```
# Adding a list of values with keys to create a new DataFrame.

per_school_summary_df = pd.DataFrame({
            "School Type": per_school_types,
            "Total Students": per_school_counts,
            "Total School Budget": per_school_budget,
            "Per Student Budget": per_school_capita,
            "Average Math Score": per_school_math,
          "Average Reading Score": per_school_reading,
          "% Passing Math": per_school_passing_math,
          "% Passing Reading": per_school_passing_reading,
          "% Overall Passing": per_overall_passing_percentage})
per_school_summary_df.head()
```

When we execute this code, we get the following DataFrame:



per_school_summary_df.head()

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Bailey High School	District	4976	3124928	628.0	77.048432	81.033963	66.680064	81.933280	54.642283
Cabrera High School	Charter	1858	1081356	582.0	83.061895	83.975780	94.133477	97.039828	91.334769
Figueroa High School	District	2949	1884411	639.0	76.711767	81.158020	65.988471	80.739234	53.204476
Ford High School	District	2739	1763916	644.0	77.102592	80.746258	68.309602	79.299014	54.289887
Griffin High School	Charter	1468	917500	625.0	83.351499	83.816757	93.392371	97.138965	90.599455

The columns are in the correct order, but we still need to apply formatting to the Total School Budget and Per Student Budget columns, as well as apply a thousands separator to the Total Students column. Let's clean up this DataFrame!

4.8.7: Clean Up the DataFrame

Just like you did for the district summary DataFrame, Maria wants you to format the budget columns by adding dollar signs and converting the amounts to two decimal places.

In the `per_school_summary_df` DataFrame, we will format the `Total School Budget` and `Per Student Budget` columns to include:

- A U.S. dollar sign
- Two decimal places
- A thousands separator

Format Columns

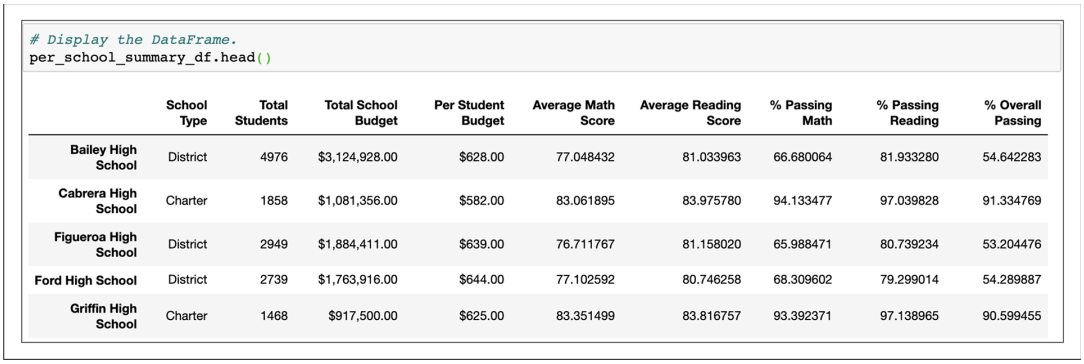
> **Rewind**
> To format every row in the column, use the `map()` function. Inside the parentheses, apply formatting with `{" "}.format`. Inside the quotations, pass the formatting specification to the value in the row.

To format the `Total School Budget` and `Per Student Budget` columns, add the following code to a new cell and run the cell.

```
# Format the Total School Budget and the Per Student Budget columns.
per_school_summary_df["Total School Budget"] = per_school_summary_df["Total

per_school_summary_df["Per Student Budget"] = per_school_summary_df["Per Stu
```

```
# Display the data frame
per_school_summary_df.head()
```

When we execute this code, we get the following DataFrame:



```
# Display the DataFrame.
per_school_summary_df.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Bailey High School	District	4976	$3,124,928.00	$628.00	77.048432	81.033963	66.680064	81.933280	54.642283
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476
Ford High School	District	2739	$1,763,916.00	$644.00	77.102592	80.746258	68.309602	79.299014	54.289887
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455

If your DataFrame looks like this, nice work! However, if the columns are in a different order, you'll need to reorder them.

Reorder Columns

If the columns were not ordered correctly, put them in the following order:

- School Type
- Total Students
- Total School Budget
- Per Student Budget
- Average Math Score
- Average Reading Score
- % Passing Math
- % Passing Reading
- % Overall Passing

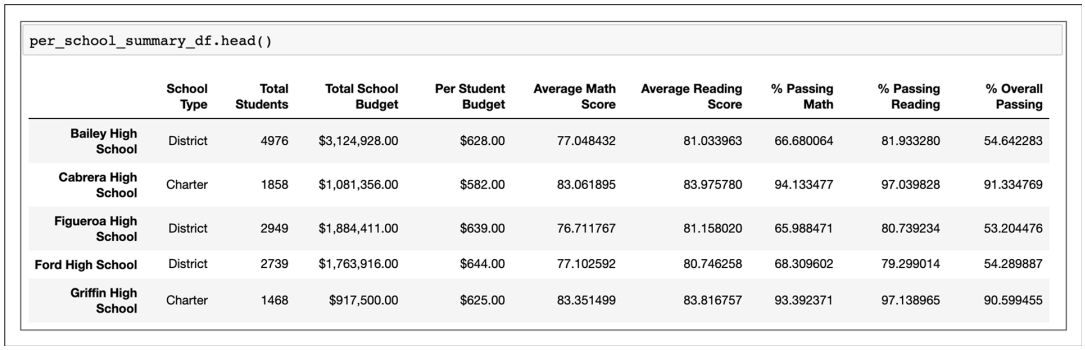
> **Rewind**
>
> To reorder columns, use the following format:
>
> ```
> new_column_order = ["column2", "column4", "column1"]
> ```

> Assign the same DataFrame to the new column order:
>
> `df = df[new_column_order]`

The code for reordering the columns will look like this:

```
# Reorder the columns in the order you want them to appear.
new_column_order = ["School Type", "Total Students", "Total School Budget",

# Assign district summary df the new column order.
per_school_summary_df = per_school_summary_df[new_column_order]

per_school_summary_df.head()
```

When we run this code, the columns will be in the correct order, as shown below:



```
per_school_summary_df.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Bailey High School	District	4976	$3,124,928.00	$628.00	77.048432	81.033963	66.680064	81.933280	54.642283
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476
Ford High School	District	2739	$1,763,916.00	$644.00	77.102592	80.746258	68.309602	79.299014	54.289887
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455

Now commit `PyCitySchool.ipynb` to your GitHub repository.

4.8.8: Commit Your Code

You just completed a lot of work, so it's time to update your GitHub repository. This will allow Maria and other stakeholders and district officials to stay up-to-date on your progress.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "school summary complete."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.9.1: Find the Highest-Performing Schools

In a two-hour morning meeting with Maria and her supervisor, you decide that you need to find which schools are the highest-performing based on the overall percentage of passing students. This will help the district determine how much money should be allocated to and spent on each school; this will also allow the school board to set the budget for the upcoming school year. While the meeting is in progress, Maria asks you to work on finding the data, so you open your computer, start Jupyter Notebook, and get to work.

To display the top five highest-performing schools, we can sort the `per_school_summary_df` DataFrame on the `% Overall Passing` column.

> **Rewind**
>
> To view the first five rows of a DataFrame, use the `df.head()` method.
>
> To view the last five rows of a DataFrame, use the `df.tail()` method.

Using the Pandas `sort_values()` function, we can sort a DataFrame or Series for a given text, index, or column that is passed within the parentheses. In addition, we can add the parameter `ascending=False` to sort from highest to lowest, or `ascending=True` to sort from lowest to highest the value referenced in the `sort_values()` function. If we don't add the `ascending` parameter, the default is `ascending=True`.

To sort the schools on the `% Overall Passing` column from highest to lowest, add the following code and run the cell.

```
# Sort and show top five schools.
top_schools = per_school_summary_df.sort_values(["% Overall Passing"], ascen

top_schools.head()
```

When we execute the code, we are going to sort the `per_school_summary_df` DataFrame on the `% Overall Passing` column from the highest `% Overall Passing` to the lowest.

Finding

The top five highest-performing schools based on the highest % Overall Passing are charter schools that have a low student population. See the following results:

```
top_schools.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769
Thomas High School	Charter	1635	$1,043,130.00	$638.00	83.418349	83.848930	93.272171	97.308869	90.948012
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455
Wilson High School	Charter	2283	$1,319,574.00	$578.00	83.274201	83.989488	93.867718	96.539641	90.582567
Pena High School	Charter	962	$585,858.00	$609.00	83.839917	84.044699	94.594595	95.945946	90.540541

Note

For more information, read the [Pandas documentation on the sort_values() function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html).

4.9.2: Find the Lowest-Performing Schools

While discussions continue in the meeting, you interrupt to show Maria and her supervisor the top-performing schools. They're both impressed, so they ask you to determine which schools are the lowest-performing based on the overall percentage of students who passed. This will help the school board determine if more money needs to be allocated to these schools, or if other solutions are needed, based on what the data shows.

To find the five lowest-performing schools based on the overall percentage of students who passed in the `per_school_summary_df` DataFrame, we can use the `ascending=True` parameter. Since `ascending=True` is the default parameter for the `sort_values()` function, we don't need to add it. However, it's beneficial to add this parameter so we know how we are sorting.

Which code would you use to sort the results in order to determine the bottom five performing schools in ascending order? (Select all that apply.)

- [ ]
```
bottom_schools = per_school_summary_df.sort_values(["% Overall Passing"])
bottom_schools.head()
```

- [ ]
```
bottom_schools = per_school_summary_df.sort_values(["% Overall Passing"],
ascending=False)
bottom_schools.head()
```

- [ ]
```
bottom_schools = per_school_summary_df.sort_values(["% Overall Passing"],
ascending=True)
bottom_schools.head()
```

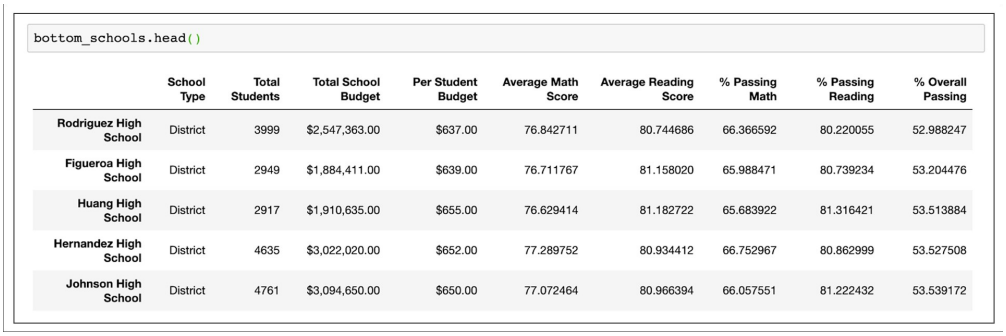
- [ ]
```
bottom_schools = per_school_summary_df.sort_values(["% Overall Passing"],
ascending=False)
bottom_schools.tail()
```

To sort for the five lowest-performing schools based on the `% Overall Passing`, add the following code to a new cell and run the cell.

```
# Sort and show top five schools.
bottom_schools = per_school_summary_df.sort_values(["% Overall Passing"], as

bottom_schools.head()
```

Finding

The five lowest-performing schools based on the lowest "% Overall Passing" are district schools that have a high student population.



```
bottom_schools.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Rodriguez High School	District	3999	$2,547,363.00	$637.00	76.842711	80.744686	66.366592	80.220055	52.988247
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476
Huang High School	District	2917	$1,910,635.00	$655.00	76.629414	81.182722	65.683922	81.316421	53.513884
Hernandez High School	District	4635	$3,022,020.00	$652.00	77.289752	80.934412	66.752967	80.862999	53.527508
Johnson High School	District	4761	$3,094,650.00	$650.00	77.072464	80.966394	66.057551	81.222432	53.539172

Now that we have the five highest- and lowest-performing schools, let's commit our work to our GitHub repository.

4.9.3: Commit Your Code

Maria asks you to push your results to GitHub. She wants to pull them onto her computer so she can begin to write a report based on your findings, as well as prepare a slideshow presentation for the school board.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder and follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "high-low performing schools"` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.10.1: Create Grade-Level DataFrames

There's still some left work to do for the school district data analysis. Maria wants you to dig a little deeper into the each school's metadata and get the average math and reading scores for each school. But first, you will need to create DataFrames for each grade level.

You are tasked with creating two new DataFrames: one that will display the average math scores for each grade, and the one that will display the average reading scores for each grade. These will be grouped by the name of the school.

In the final DataFrame, we need to have the index on the school name and the average math or reading score for each grade: 9th, 10th, 11th, and 12th. This is what the DataFrame for the average math scores should look like when we're done:

	9th	10th	11th	12th
Bailey High School	77.1	77.0	77.5	76.5
Cabrera High School	83.1	83.2	82.8	83.3
Figueroa High School	76.4	76.5	76.9	77.2
Ford High School	77.4	77.7	76.9	76.2
Griffin High School	82.0	84.2	83.8	83.4

In this DataFrame, each grade—9th, 10th, 11th and 12th—is a Series of data. To get each grade as a Series, we need to filter `school_data_complete_df` for each grade.

> **Rewind**
>
> To get the passing scores for the district and school summary, we used the following code:
>
> `passing_math = school_data_complete_df["math_score"] >= 70`
>
> When we executed this code, a Series was returned where rows with a score equal to or greater than 70 were "True," and rows with scores not equal to or greater than 70 were "False." See the following image:
>
> ```
> passing_math = school_data_complete_df["math_score"] >= 70
> passing_math
> ```
>
> ```
> 0         True
> 1        False
> 2        False
> 3        False
> 4         True
> 5         True
> 6         True
> ```
>
> We can use this same principle to get the grade level for each school in the `school_data_complete_df` DataFrame.

Let's first get the ninth graders in a Series. Set the variable `ninth_graders` equal to `school_data_complete_df["grade"] == "9th"`. This will retrieve all rows where this statement is "True."

Using the code `school_data_complete_df["grade"] == "9th"`, we can filter `school_data_complete_df` for the "True" cases as follows:

`school_data_complete_df[(school_data_complete_df["grade"] == "9th")]`

Now repurpose this code for each grade by replacing `"9th"` with each grade. Assign a grade-level variable to each grade-level DataFrame; for example, `ninth_graders` is for the ninth grade data.

Add the following code to a new cell and run the cell.

```
# Create a grade level DataFrames.
ninth_graders = school_data_complete_df[(school_data_complete_df["grade"] ==

tenth_graders = school_data_complete_df[(school_data_complete_df["grade"] ==

eleventh_graders = school_data_complete_df[(school_data_complete_df["grade"]

twelfth_graders = school_data_complete_df[(school_data_complete_df["grade"]
```

When we print out the first five rows for the `ninth_graders`, the DataFrame should look like this:

	Student ID	student_name	gender	grade	school_name	reading_score	math_score	School ID	type	size	budget
0	0	Paul Bradley	M	9th	Huang High School	66	79	0	District	2917	1910635
4	4	Bonnie Ray	F	9th	Huang High School	97	84	0	District	2917	1910635
5	5	Bryan Miranda	M	9th	Huang High School	94	94	0	District	2917	1910635
12	12	Brittney Walker	F	9th	Huang High School	64	79	0	District	2917	1910635
13	13	William Long	M	9th	Huang High School	71	79	0	District	2917	1910635

Now that we have the data for each grade level, we need to get the grade averages for each school.

4.10.2: Score Averages Grouped by School Name

Now that you have the DataFrames for each grade level, Maria says you can get the average math and reading scores for each grade level and for each school. She reminds you that the school will be the index and the averages will be the column.

To get the average math and reading scores for each grade level and for each school, we need to do the following:

- Use the `groupby` function to group by the `school_name` column
- Calculate the mean `math_score` and `reading_score`

> **Rewind**
> When we use the `groupby()` function on a DataFrame, we need to perform a mathematical operation on the `groupby()` object by selecting a column from the DataFrame.

Get the Average Math Scores by School

First, we will get the average math score for each school. For each grade level DataFrame, we will use the `groupby()` function on the `school_name` column and apply the `mean()` on the `math_score` column.

To get the `math_score` averages for each grade level, add the following code in a new cell and run the cell.

```
# Group each school Series by the school name for the average math score.
ninth_grade_math_scores = ninth_graders.groupby(["school_name"]).mean()["ma

tenth_grade_math_scores = tenth_graders.groupby(["school_name"]).mean()["ma

eleventh_grade_math_scores = eleventh_graders.groupby(["school_name"]).mean

twelfth_grade_math_scores = twelfth_graders.groupby(["school_name"]).mean(
```

When we print out the first five rows for the `ninth_grade_math_scores`, the Series will look like the following image; the index is the school name, and the first column is the average math scores for ninth graders.

```
ninth_grade_math_scores

school_name
Bailey High School       77.083676
Cabrera High School      83.094697
Figueroa High School     76.403037
Ford High School         77.361345
Griffin High School      82.044010
Hernandez High School    77.438495
Holden High School       83.787402
Huang High School        77.027251
Johnson High School      77.187857
Pena High School         83.625455
Rodriguez High School    76.859966
Shelton High School      83.420755
Thomas High School       83.590022
Wilson High School       83.085578
Wright High School       83.264706
Name: math_score, dtype: float64
```

What is the average math score for the *11th graders* at Pena High School?

- ○ 84.328125
- ○ 83.372000
- ○ 84.121547
- ○ 83.625455

Check Answer

Finish ▸

Get the Average Reading Scores by School

We can repurpose this code to find the average reading scores for each grade level by replacing `math_score` with `reading_score`. Add the following code in a new cell and run the cell.

```
# Group each school Series by the school name for the average reading scor
ninth_grade_reading_scores = ninth_graders.groupby(["school_name"]).mean()

tenth_grade_reading_scores = tenth_graders.groupby(["school_name"]).mean()

eleventh_grade_reading_scores = eleventh_graders.groupby(["school_name"]).

twelfth_grade_reading_scores = twelfth_graders.groupby(["school_name"]).me
```

When we print out the first five rows for the `ninth_grade_reading_score`, the Series will look like the following image; the index is the school name, and the first column is the average reading scores for ninth graders.

```
ninth_grade_reading_scores

school_name
Bailey High School       81.303155
Cabrera High School      83.676136
Figueroa High School     81.198598
Ford High School         80.632653
Griffin High School      83.369193
Hernandez High School    80.866860
Holden High School       83.677165
Huang High School        81.290284
Johnson High School      81.260714
Pena High School         83.807273
Rodriguez High School    80.993127
Shelton High School      84.122642
Thomas High School       83.728850
Wilson High School       83.939778
Wright High School       83.833333
Name: reading_score, dtype: float64
```

What is the average reading score for the 12th graders at Shelton High School?

- ○ 84.373786

Now that we have this information, we need to generate a report showing the average math and reading scores by grade level. Let’s add all of the data to a DataFrame!

4.10.3: Combine the Series into a DataFrame

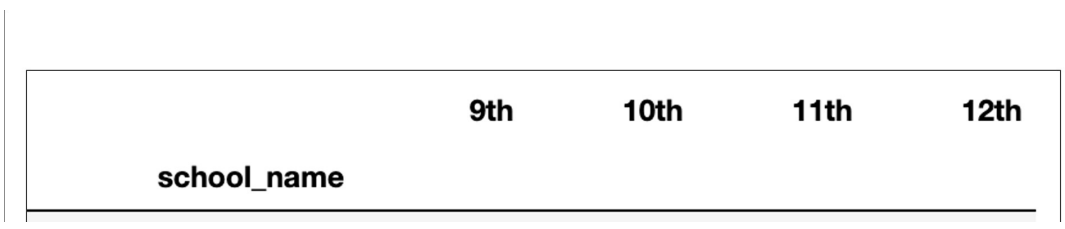
District officials and other key stakeholders are eager to see the report on average math and reading scores by grade level. Maria would like you to combine the average math scores for each grade and the average reading scores for each grade and add them to separate DataFrames.

Now that we have performed all the necessary calculations to get the average math and reading scores by grade level for each school, we can add them to separate DataFrames.

First, let's add the math scores for each grade level to a new DataFrame. To do this, add the following code in a new cell and run the cell.

```
# Combine each Series for average math scores by school into single DataFram
math_scores_by_grade = pd.DataFrame({
                "9th": ninth_grade_math_scores,
                "10th": tenth_grade_math_scores,
                "11th": eleventh_grade_math_scores,
                "12th": twelfth_grade_math_scores})

math_scores_by_grade.head()
```

When we run the cell, the `math_scores_by_grade` DataFrame will look like this:



	9th	10th	11th	12th
school_name				

Bailey High School	77.083676	76.996772	77.515588	76.492218
Cabrera High School	83.094697	83.154506	82.765560	83.277487
Figueroa High School	76.403037	76.539974	76.884344	77.151369
Ford High School	77.361345	77.672316	76.918058	76.179963
Griffin High School	82.044010	84.229064	83.842105	83.356164

We'll use the same format to add the reading scores for each grade level to a new DataFrame. To do this, add the following code in a new cell and run the cell.

```
# Combine each Series for average reading scores by school into single DataF
reading_scores_by_grade = pd.DataFrame({
              "9th": ninth_grade_reading_scores,
              "10th": tenth_grade_reading_scores,
              "11th": eleventh_grade_reading_scores,
              "12th": twelfth_grade_reading_scores})

reading_scores_by_grade.head()
```

When we run the cell, the `reading_scores_by_grade` DataFrame will look like this:

school_name	9th	10th	11th	12th
Bailey High School	81.303155	80.907183	80.945643	80.912451
Cabrera High School	83.676136	84.253219	83.788382	84.287958
Figueroa High School	81.198598	81.408912	80.640339	81.384863
Ford High School	80.632653	81.262712	80.403642	80.662338
Griffin High School	83.369193	83.706897	84.288089	84.013699

You just created the average math and reading DataFrames—great work!

4.10.4: Format the Averages and Remove the Index Name

Just as you've done for the previous DataFrames, Maria would like you to format the reading and math averages to one decimal place and removed the name of the index column. This will make the DataFrame look cleaner and more professional.

For reporting purposes, we'll format the grade-level averages to one decimal place, as well as remove the name of the index column, `school_name`.

> **Rewind**
> To format every row in a column, use the `map()` function. Inside the parentheses, apply the formatting using `"{ }".format`.
>
> To format `float64` data types to one decimal place, add `{:.1f}` inside the quotations, like this: `map("{:.1f}".format`.

To format the reading and math averages for each grade level we'll use `map("{:.1f}".format`. And we'll remove the index, `school_name`, by setting the index name to "None" with the following syntax: `index.name = None`.

Let's apply these methods to the `math_scores_by_grade` DataFrame first. Add the following code and run the cells:

```
# Format each grade column.
math_scores_by_grade["9th"] = math_scores_by_grade["9th"].map("{:.1f}".for

math_scores_by_grade["10th"] = math_scores_by_grade["10th"].map("{:.1f}".f
```

```
math_scores_by_grade["11th"] = math_scores_by_grade["11th"].map("{:.1f}".f

math_scores_by_grade["12th"] = math_scores_by_grade["12th"].map("{:.1f}".f

# Make sure the columns are in the correct order.
math_scores_by_grade = math_scores_by_grade[
               ["9th", "10th", "11th", "12th"]]

# Remove the index name.
math_scores_by_grade.index.name = None
# Display the DataFrame.
math_scores_by_grade.head()
```

After we run this code, the `math_scores_by_grade` DataFrame will look like this:

	9th	10th	11th	12th
Bailey High School	77.1	77.0	77.5	76.5
Cabrera High School	83.1	83.2	82.8	83.3
Figueroa High School	76.4	76.5	76.9	77.2
Ford High School	77.4	77.7	76.9	76.2
Griffin High School	82.0	84.2	83.8	83.4

Now apply the same formatting method and remove the index name for the `reading_scores_by_grade` DataFrame. Add the following code to a new cell and run the cell.

```
# Format each grade column.
reading_scores_by_grade["9th"] = reading_scores_by_grade["9th"].map("{:,.1

reading_scores_by_grade["10th"] = reading_scores_by_grade["10th"].map("{:,

reading_scores_by_grade["11th"] = reading_scores_by_grade["11th"].map("{:,
```

```
reading_scores_by_grade["12th"] = reading_scores_by_grade["12th"].map("{:,

# Make sure the columns are in the correct order.
reading_scores_by_grade = reading_scores_by_grade[
                ["9th", "10th", "11th", "12th"]]

# Remove the index name.
reading_scores_by_grade.index.name = None
# Display the data frame.
reading_scores_by_grade.head()
```

After running this code, the `reading_scores_by_grade` DataFrame will look like this:

	9th	10th	11th	12th
Bailey High School	81.3	80.9	80.9	80.9
Cabrera High School	83.7	84.3	83.8	84.3
Figueroa High School	81.2	81.4	80.6	81.4
Ford High School	80.6	81.3	80.4	80.7
Griffin High School	83.4	83.7	84.3	84.0

4.10.5: Commit Your Code

Maria asks that you to add the average reading and math scores by grade level to GitHub. She wants to pull them on to her computer so that she can update her presentation with this new data.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "added average math and reading scores by grade."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.11.1: Establish the Spending Ranges per Student

You meet again with Maria and her supervisor and a question arises: how does school spending per student affect the school's average scores and passing percentages? Maria tasks you with finding an answer to this question. This information will help the school board make decisions about the budget for the upcoming school year. Maria would like to see this data organized by spending ranges for the schools. You will need to create four spending bins, or ranges, to group the school spending per student.

For this analysis, we will need to sort the school spending per student into four spending "bins," or ranges. The four bins will be dollar amounts that range from the lowest amount ($578) to the highest amount ($655) a school spends on a student.

> **Rewind**
>
> We determined the budget per student in the school summary when we retrieved the `per_school_capita` Series:

```
per_school_capita
Bailey High School       628.0
Cabrera High School      582.0
Figueroa High School     639.0
Ford High School         644.0
Griffin High School      625.0
Hernandez High School    652.0
Holden High School       581.0
Huang High School        655.0
Johnson High School      650.0
Pena High School         609.0
Rodriguez High School    637.0
Shelton High School      600.0
Thomas High School       638.0
Wilson High School       578.0
```

```
Wilson High School        578.0
Wright High School        583.0
dtype: float64
```

We will need to arbitrarily determine the spending ranges in order to group the schools fairly. Each bin must include the average math and reading scores, the percentage passing math and reading, and the overall passing percentage for the spending bins, not for the schools.

The final DataFrame will look like the following image, with the spending bins included in place of the box that says "Spending Bins." (We'll determine the four spending bins later in this section.) The columns will contain the average math score, average reading score, percent passing math, percent passing reading, and percent overall passing for the schools in each spending bin.

Spending Ranges (Per Student)	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Spending Bins	83.5	83.9	93	97	90
	81.9	83.2	87	93	81
	78.5	81.6	73	84	63
	77.0	81.0	66	81	54

To create this DataFrame, we need to do the following:

1. Group the schools into four bins, or ranges, based on the budget per student.
2. For each spending range, get the following data:
    - Average math and reading scores
    - The percentage of students passing math and reading
    - The overall passing percentage, which is the average of the percentage of students passing math and reading

Before we aggregate the school spending per student into four bins, we must establish the spending ranges. We can determine the spending ranges by using the `per_school_capita` Series that was used to get the budget per student.

Get the Spending Ranges

Before we create the bins, we need to determine the distribution of spending per student. We can find the distribution for school spending per student by using the `describe()` method.

When we apply the `describe()` method to a DataFrame or Series, it will return the following descriptive statistics for the DataFrame or Series:

- The number of rows in the DataFrame or Series as `count`
- The average of the rows as `mean`
- The standard deviation of the rows as `std`
- The minimum value of the rows as `min`
- The 25th percentile as `25%`
- The 50th percentile as `50%`
- The 75th percentile as `75%`
- The minimum value of the rows as `max`

Using the `describe()` method will help us determine the spending bins we should use, based on the descriptive statistics. To get the descriptive statistics on the `per_school_capita`, add the following code to a new cell:

```
# Get the descriptive statistics for the per_school_capita.
per_school_capita.describe()
```

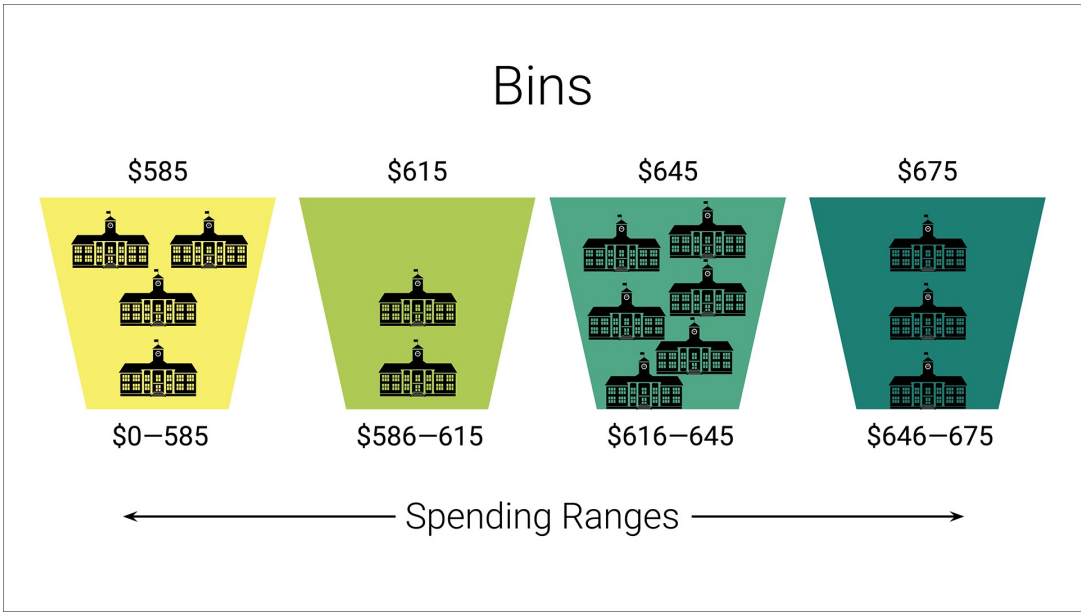
Your results should look like this:

```
per_school_capita.describe()

count     15.000000
mean     620.066667
std       28.544368
min      578.000000
25%      591.500000
50%      628.000000
75%      641.500000
```

```
max       655.000000
dtype: float64
```

As you can see, the minimum is 578 and the maximum is 655. The standard deviation is 28.5, or approximately 30. We don't want the lowest bin to be $578 because there is only one school at or below $578, which is Wilson High School. However, there are four schools that spend less than $585 per student, so $585 will be our lowest bin. Also, because the standard deviation is about 30, we will increase the bins by $30, up to $675. Therefore, the four bins will be: $585, $615, $645, and $675.

The bins will define a range of spending. For the $585 bin, the range is all values at or below $585, the range for the $615 bin is $586–615, and so on. Here's what our bins look like:



As shown in this image, each bin has a lower "edge" and an upper "edge" value. Schools that spend $0–585 will be placed in the first bin. Schools that spend $586–615 will be placed in the second bin, and so on.

In Pandas, we can write the ranges for the bins as follows: `spending_bins = [0, 585, 615, 645, 675]`.

Let's go over why these spending bins have five numbers instead of four, which is the number of bins we want. When Pandas creates the lower edge, it needs a value lower than the lowest value in the column of the `per_school_capita` Series, which happens

to be 578. In our DataFrame, we make the lower edge equal to 0. If we don't include the 0 for the spending bins, the lowest bin becomes $585–614, which means that the schools that spend less than $585 are not considered. First we'll use the correct spending bins for our ranges, and then we'll leave out the 0 to see how that affects the bins.

Let's address one more item before moving on. If you look at the `spending_bins` list, you'll notice that the dollar sign ($) isn't included with the ranges. This is because the numbers in the `per_school_capita` Series are the `float64` data type, so adding a dollar sign would cause an error.

Now let's take a look at how to use the spending bins.

Group the Series on the Spending Ranges

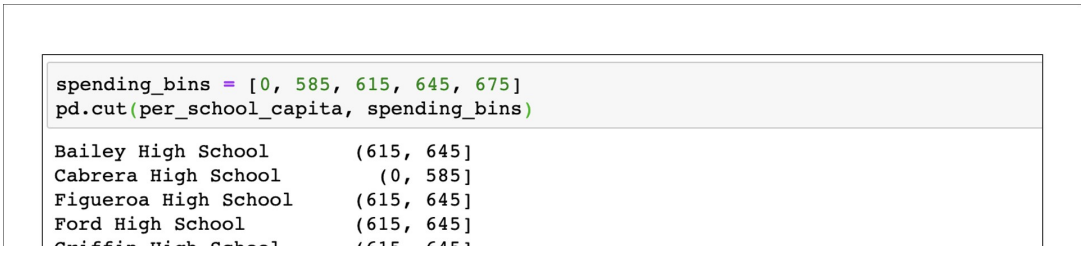
To group the spending ranges, we use the Pandas `cut()` function. The `cut()` function segments and sorts data values into bins. We'll use the `cut()` function to create our spending bins.

To apply the `cut()` function to a DataFrame or Series, we use `pd.cut(df, ranges)`. This creates a new DataFrame or Series on the given ranges.

We can "cut" the `per_school_capita` into the `spending_ranges` using the following code. Add this code to a new cell and run the cell.

```
# Cut the per_school_capita into the spending ranges.
spending_bins = [0, 585, 615, 645, 675]
pd.cut(per_school_capita, spending_bins)
```

The output shows that each school has a range for the spending per student instead of a specific amount:



```
Griffin High School      (615, 645]
Hernandez High School    (645, 675]
Holden High School         (0, 585]
Huang High School        (645, 675]
Johnson High School      (645, 675]
Pena High School         (585, 615]
Rodriguez High School    (615, 645]
Shelton High School      (585, 615]
Thomas High School       (615, 645]
Wilson High School         (0, 585]
Wright High School         (0, 585]
dtype: category
Categories (4, interval[int64]): [(0, 585] < (585, 615] < (615, 645] < (645, 675]]
```

The `cut()` function places each value of the `per_school_capita` Series into one of four bins. The bins are created by a list of five numbers given as the `spending_bins` that define the edges.

What happens if we don't add 0 as the lower edge for the first bin? Copy the following code in a new cell and run the cell.

```
# Cut the per_school_capita into the spending ranges.
spending_bins = [585, 615, 645, 675]
pd.cut(per_school_capita, spending_bins)
```

The output shows that four high schools—Cabrera, Holden, Wilson, and Wright—are not in a bin and therefore would not be in the final DataFrame.

```
spending_bins = [585, 615, 645, 675]
pd.cut(per_school_capita, spending_bins)
```

```
Bailey High School       (615.0, 645.0]
Cabrera High School                 NaN
Figueroa High School     (615.0, 645.0]
Ford High School         (615.0, 645.0]
Griffin High School      (615.0, 645.0]
Hernandez High School    (645.0, 675.0]
Holden High School                  NaN
Huang High School        (645.0, 675.0]
Johnson High School      (645.0, 675.0]
Pena High School         (585.0, 615.0]
Rodriguez High School    (615.0, 645.0]
Shelton High School      (585.0, 615.0]
Thomas High School       (615.0, 645.0]
Wilson High School                  NaN
Wright High School                  NaN
dtype: category
Categories (3, interval[int64]): [(585, 615] < (615, 645] < (645, 675]]
```

Next, we'll determine how many schools are in each range by grouping the spending bins as the index using the `groupby()` function on the `per_school_capita` Series.

Remember that we need to perform a mathematical operation when we use the `groupby()` function.

To find out how many schools are in those bins, use the `count()` method on the `groupby()` function. Edit your previous code so that it looks like the following:

```
# Cut the per_school_capita into the spending ranges.
spending_bins = [0, 585, 615, 645, 675]
per_school_capita.groupby(pd.cut(per_school_capita, spending_bins)).count()
```

When we run the cell, the output is the number of schools in each range:

```
(0, 585]      4
(585, 615]    2
(615, 645]    6
(645, 675]    3
dtype: int64
```

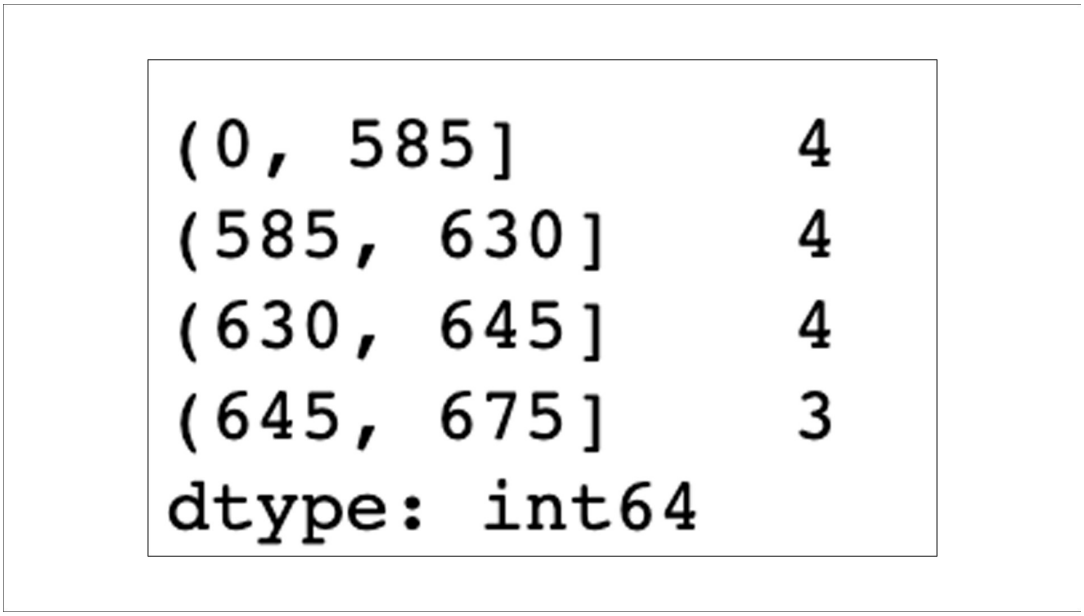
Looking at this output, it seems that we didn't group the schools fairly. We need to adjust our ranges so that we have three or four schools in each range. Let's increase the `[585, 615]` range to `[585, 630]` and change the third range to `[630, 645]`. The `spending_bins` will look like this:

`[0, 585, 630, 645, 675]`

Edit your previous code and run the cell again. The code should look like this:

```
# Cut the per_school_capita into the spending ranges.
spending_bins = [0, 585, 630, 645, 675]
per_school_capita.groupby(pd.cut(per_school_capita, spending_bins)).count()
```

Now the output shows that there is a more fair distribution of schools in each range:



These ranges look better! Now let's name each range. We can label the ranges using a list of string values. Add the following code to a new cell and run the cell.

```
# Establish the spending bins and group names.
spending_bins = [0, 585, 630, 645, 675]
group_names = ["<$584", "$585-629", "$630-644", "$645-675"]
```

Now that we have our correct spending bins and group names, we can “cut” the `per_school_capita` Series to get the four spending bins and add them to the `per_school_summary_df` DataFrame.

4.11.2: Categorize the Spending Bins

Establishing the bin ranges was quite a challenge, but it's done! Now Maria wants you to focus on grouping the schools' spending in the `per_school_summary_df` DataFrame based on the bins you just created.

Using our spending bins and ranges, we can create a new column in the `per_school_summary_df` DataFrame which will be assigned the spending bins from the `per_school_capita` Series.

To do this, we will need to do the following:

- Use the `cut()` function on the `per_school_capita` Series.
- Add the bin data to a new column in the `per_school_summary_df` DataFrame.

Add the following code to a new cell and run the cell.

```
# Categorize spending based on the bins.
per_school_summary_df["Spending Ranges (Per Student)"] = pd.cut(per_school_c

per_school_summary_df
```

There's a lot going on in this code, so let's break it down.

- **To the left of the equals sign,** we add a new column called `"Spending Ranges (Per Student)"` to the `per_school_summary_df` DataFrame. This column will contain the values specified by using the `cut()` function.

- **Inside the `cut()` function,** we add the data (`per_school_capita`) we are going to use for `"Spending Ranges (Per Student)"`. The data must be a one-dimensional array, like a list or the `per_school_capita` Series.
- **Inside the parentheses,** we add the `spending_bins` and the labels for the bins using `labels=group_names`.

When we execute the code, the results should look like this:

```
per_school_summary_df.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing	Spending Ranges (Per Student)
Bailey High School	District	4976	$3,124,928.00	$628.00	77.048432	81.033963	66.680064	81.933280	54.642283	$585-629
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769	<$584
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476	$630-644
Ford High School	District	2739	$1,763,916.00	$644.00	77.102592	80.746258	68.309602	79.299014	54.289887	$630-644
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455	$585-629

The “Spending Ranges (Per Student)” column is added as the last column. Whenever a new column is added to a DataFrame, it becomes the last column. This is because each column in the DataFrame is like an item in a list.

> **Rewind**
>
> When we add a new item to a list, that item will always be added at the end of the list unless we specify a list index where it should be placed.

Great work! Now we need to filter the `per_school_summary_df` DataFrame in order to get the average scores as well as the percentage of students who passed reading and math for each range in the `spending_bins`.

> **Note**
>
> For more information, read the [Pandas documentation on the cut() function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html).

4.11.3: Group by the Spending Ranges

Nice work grouping the schools based on the spending bins—Maria is impressed! Now she would like you get the average math and reading scores, the average percentage of students who passed math and reading, and the average overall passing percentage. Once you get this data, you need to create a new DataFrame with these averages and percentages, based on the spending bins, to show how school spending affects score averages and passing rates.

The final DataFrame will contain the average math and reading scores, the percentage of students who passed math and reading, and the overall percentage of students who passed for each spending bin. To get these averages and percentages, we'll use the `groupby()` function to group the data on the spending bins in the index column, "Spending Ranges (Per Student)."

First, let's create four new Series with the following data:

- Average Math Score
- Average Reading Score
- Average % Passing Math
- Average % Passing Reading
- Average % Overall Passing

> **Rewind**
> To create a Series that has a column as the index, use the `groupby()` function on the DataFrame and add the column inside the parentheses.

We'll create each Series using the `groupby()` function on the "Spending Ranges (Per Student)" column. For each Series, we'll use the `mean()` method to get the averages of the Average Math Score column, Average Reading Score column, % Passing Math column, and % Passing Reading column. For the average "% Overall Passing," we'll add the "% Passing Math" and "% Passing Reading" columns and then divide by 2.

To get these averages, add the following code to a new cell and run the cell.

```
# Calculate averages for the desired columns.
spending_math_scores = per_school_summary_df.groupby(["Spending Ranges (Per

spending_reading_scores = per_school_summary_df.groupby(["Spending Ranges (P

spending_passing_math = per_school_summary_df.groupby(["Spending Ranges (Per

spending_passing_reading = per_school_summary_df.groupby(["Spending Ranges (

overall_passing_spending = per_school_summary_df.groupby(["Spending Ranges (
```

What is the overall passing percentage for the $630 spending bin, or the $630–644 spending range?

- ○ 90.37
- ○ 84.39
- ○ 73.484209
- ○ 62.86

Check Answer

Finish ▸

Now that we have all the averages, we can add them to a new DataFrame.

4.11.4: Create a DataFrame for the Scores by School Spending

Your next task is to create a new DataFrame for the average math and reading scores, average percentage of students who passed math and reading, and the average overall percentage of passing students. You take a break to stretch your legs and grab a cup of coffee. You're feeling pretty confident about this next task, since you've done it a few times already.

Now that we have the average math and reading scores, the average percentage of students who passed math and reading, and the average overall percentage as a Series, let's add this data to a new DataFrame.

> **Rewind**
> To add a list or Series to a DataFrame, use the following format:
>
> `df = pd.DataFrame({"column name": column values})`

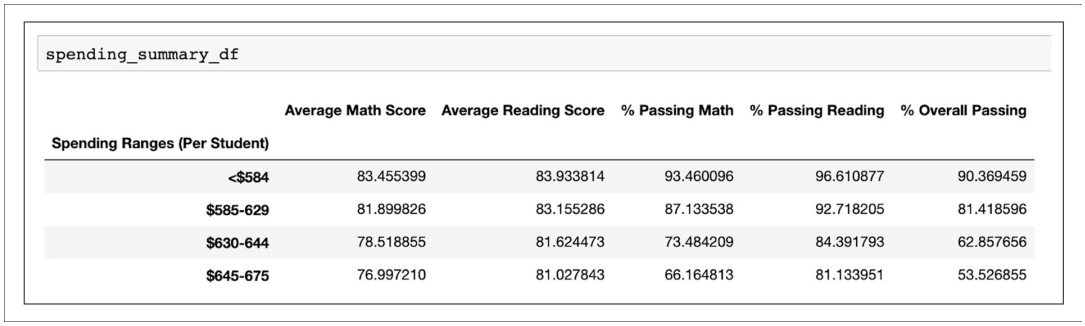
Create the Spending Summary DataFrame

To create the spending summary DataFrame, add the following code to a new cell and run the cell.

```
# Assemble into DataFrame.
spending_summary_df = pd.DataFrame({
          "Average Math Score" : spending_math_scores,
          "Average Reading Score": spending_reading_scores,
```

```
            "% Passing Math": spending_passing_math,
            "% Passing Reading": spending_passing_reading,
            "% Overall Passing": overall_passing_spending})

spending_summary_df
```

When we run the cell, results in the output window will look like this:



spending_summary_df

Spending Ranges (Per Student)	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
<$584	83.455399	83.933814	93.460096	96.610877	90.369459
$585-629	81.899826	83.155286	87.133538	92.718205	81.418596
$630-644	78.518855	81.624473	73.484209	84.391793	62.857656
$645-675	76.997210	81.027843	66.164813	81.133951	53.526855

Nice work! But we need to get our DataFrame to adhere to grade-reporting standards by following certain formatting. We need to apply the following changes:

- Format the average math and reading scores to one decimal place
- Format the percent passing math and reading to the nearest whole number
- Format the overall passing percentage to the nearest whole number

---

Format the DataFrame

To format the `spending_summary_df` DataFrame, add the following code to a new cell and run the cell:

```
# Formatting
spending_summary_df["Average Math Score"] = spending_summary_df["Average Mat

spending_summary_df["Average Reading Score"] = spending_summary_df["Average

spending_summary_df["% Passing Math"] = spending_summary_df["% Passing Math"

spending_summary_df["% Passing Reading"] = spending_summary_df["% Passing Re

spending_summary_df["% Overall Passing"] = spending_summary_df["% Overall Pa
```

```
spending_summary_df
```

The formatted `spending_summary_df` DataFrame should look like this:

Spending Ranges (Per Student)	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
<$584	83.5	83.9	93	97	90
$585-629	81.9	83.2	87	93	81
$630-644	78.5	81.6	73	84	63
$645-675	77.0	81.0	66	81	54

4.11.5: Commit Your Code

Maria needs to get the latest data from the “School_District_Analysis” repository. She wants you to send her a direct message when you push the updated `PyCitySchools.ipynb` file to the repository.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and then follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder**.**
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "scores by school spending."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.12.1: Create Bins for School Size

Creating the spending bins for the scores by school spending per student was a bit of a challenge, but you did a great job. In fact, Maria is so impressed that she now wants you to group the same averages and percentages by the school size.

Previously, we grouped the scores by school spending per student. Now we'll group the scores by school size. For this DataFrame we'll create three bins: small, medium, and large. To determine these bins, we can look at the `per_school_counts`, or total students, from each school from the `per_school_summary_df` DataFrame.

> **Rewind**
>
> We retrieved the `per_school_counts` using the following code:

```
# Calculate the total student count.
per_school_counts = school_data_complete_df["school_name"].value_counts()
per_school_counts
```

```
Bailey High School       4976
Johnson High School      4761
Hernandez High School    4635
Rodriguez High School    3999
Figueroa High School     2949
Huang High School        2917
Ford High School         2739
Wilson High School       2283
Cabrera High School      1858
Wright High School       1800
Shelton High School      1761
Thomas High School       1635
Griffin High School      1468
Pena High School          962
Holden High School        427
Name: school_name, dtype: int64
```

Looking at the `per_school_counts` Series, we can see that the highest student population is 4,976. Therefore, the upper edge will be 5,000. Because we have only three bins—small, medium, and large—here's how we can group the school sizes:

- "Small" schools have fewer than 1,000 students
- "Medium" schools have 1,000–1,999 students
- "Large" schools have 2,000–5,000 students

In the new DataFrame, calculate the average math and reading scores, math and reading passing percentages, and overall passing percentage for each bin, as shown in the following image:

School Size	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Small (<1000)	83.8	83.9	94	96	90
Medium (1000-2000)	83.4	83.9	94	97	91
Large (2000-5000)	77.7	81.3	70	83	58

To establish the bins and group names, add the following code to a new cell and run the cell.

```
# Establish the bins.
size_bins = [0, 1000, 2000, 5000]
group_names = ["Small (<1000)", "Medium (1000-2000)", "Large (2000-5000)"]
```

Now we can "cut" the `per_school_summary_df` DataFrame using the `cut` function to get the three sizing bins.

4.12.2: Categorize the Size Bins

Getting the school size bins was a lot easier than getting the spending bins. Now you have enough time left in the day to group the school sizes in the `per_school_summary_df` DataFrame based on the bins.

Using our school size bins, we can create a new column in the `per_school_summary_df` DataFrame, which will be assigned the school size bins from the `per_school_summary_df` DataFrame.

To do this, we will need to do the following:

- Use the `cut()` function on the `per_school_summary_df` DataFrame.
- Add the bin data to a new column in the `per_school_summary_df` DataFrame.

Add the following code to a new cell and run the cell.

```
# Categorize spending based on the bins.
per_school_summary_df["School Size"] = pd.cut(per_school_summary_df["Total S

per_school_summary_df.head()
```

Let's go over what is happening in this code.

- We added a new column to `per_school_summary_df` DataFrame, called `"School Size"`.
- We used the `cut()` function on the `per_school_summary_df` DataFrame column `"Total Students"` and grouped the student size in the `size_bins`, and then added the `labels=group_names`.

When we execute the code, the results should look like this:

```
per_school_summary_df.head()
```

	School Type	Total Students	Total School Budget	Per Student Budget	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing	Spending Ranges (Per Student)	School Size
Bailey High School	District	4976	$3,124,928.00	$628.00	77.048432	81.033963	66.680064	81.933280	54.642283	$585-629	Large (2000-5000)
Cabrera High School	Charter	1858	$1,081,356.00	$582.00	83.061895	83.975780	94.133477	97.039828	91.334769	<$584	Medium (1000-2000)
Figueroa High School	District	2949	$1,884,411.00	$639.00	76.711767	81.158020	65.988471	80.739234	53.204476	$630-644	Large (2000-5000)
Ford High School	District	2739	$1,763,916.00	$644.00	77.102592	80.746258	68.309602	79.299014	54.289887	$630-644	Large (2000-5000)
Griffin High School	Charter	1468	$917,500.00	$625.00	83.351499	83.816757	93.392371	97.138965	90.599455	$585-629	Medium (1000-2000)

We added the new column “School Size," which becomes the last column in the DataFrame.

4.12.3: Group by School Size

Now that you have grouped the schools based on size, Maria would like you to calculate the following, based on the school sizes: average math and reading scores, the average percentage of students who passed math and reading, and the average overall percentage. After you gather this data, you'll add it to a new DataFrame to show how school size affects score averages and passing rates.

The final DataFrame will contain the average math and reading scores, the percentage of students who passed math and reading, and the overall percentage of students who passed for each school-size bin. Just as we did when we created the DataFrame for school spending per student, we'll use the `groupby()` function to group the data on the size bins in the index column, School Size.

First, let's create four new Series. We need to get the following data:

- Average Math Score
- Average Reading Score
- Average % Passing Math
- Average % Passing Reading
- Average % Overall Passing

Create each Series using the `groupby()` function on the School Size column. The data for each column will be calculated by using the `mean()` method to get the averages of the following columns: Average Math Score, Average Reading Score, % Passing Math, and % Passing Reading. For the average "% Overall Passing," we'll add the % Passing Math and % Passing Reading columns, and then divide by 2.

To get these averages, run the following code in a new cell.

```
# Calculate averages for the desired columns.
size_math_scores = per_school_summary_df.groupby(["School Size"]).mean()["Av

size_reading_scores = per_school_summary_df.groupby(["School Size"]).mean()[

size_passing_math = per_school_summary_df.groupby(["School Size"]).mean()["%

size_passing_reading = per_school_summary_df.groupby(["School Size"]).mean()

size_overall_passing = per_school_summary_df.groupby(["School Size"]).mean()
```

Now we can add these averages to a new DataFrame!

4.12.4: Create a DataFrame for the Scores by School Size

Now it's time to add the average math and reading scores, the average percentage of students who passed math and reading, and the average overall percentage to a new DataFrame. You should get to work so that you can present your findings to Maria.

Just like we did with the school spending data, we are going to add the average math and reading scores, the average percentage of students who passed math and reading, and the average overall percentage to a new DataFrame.

Create the Size Summary DataFrame

To create the size summary DataFrame, add the following code to a new cell and run the cell.

```
# Assemble into DataFrame.
size_summary_df = pd.DataFrame({
          "Average Math Score" : size_math_scores,
          "Average Reading Score": size_reading_scores,
          "% Passing Math": size_passing_math,
          "% Passing Reading": size_passing_reading,
          "% Overall Passing": size_overall_passing})

size_summary_df
```

The results in the output window will look like this:

School Size	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Small (<1000)	83.821598	83.929843	93.550225	96.099437	89.883853
Medium (1000-2000)	83.374684	83.864438	93.599695	96.790680	90.621535
Large (2000-5000)	77.746417	81.344493	69.963361	82.766634	58.286003

Now we need to apply the proper formatting to ensure the DataFrame adheres to grade-reporting standards. We need to do the following:

- Format the average math and reading scores to one decimal place
- Format the percentage of students passing math and reading to the nearest whole number
- Format the overall passing percentage to the nearest whole number

To do this, we can repurpose some code from the `spending_summary_df` DataFrame.

Format the DataFrame

To format the `size_summary_df` DataFrame, add the following code to a new cell and run the cell.

```
# Formatting.
size_summary_df["Average Math Score"] = size_summary_df["Average Math Score"

size_summary_df["Average Reading Score"] = size_summary_df["Average Reading 

size_summary_df["% Passing Math"] = size_summary_df["% Passing Math"].map("{

size_summary_df["% Passing Reading"] = size_summary_df["% Passing Reading"].

size_summary_df["% Overall Passing"] = size_summary_df["% Overall Passing"].

size_summary_df
```

The formatted `size_summary_df` DataFrame should look like this:

size_summary_df

School Size	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing

Small (<1000)	83.8	83.9	94	96	90
Medium (1000-2000)	83.4	83.9	94	97	91
Large (2000-5000)	77.7	81.3	70	83	58

4.12.5: Commit Your Code

Maria wants to compare the school spending summary and the school size summary DataFrames, so she asks you to push the latest changes to the "School_District_Analysis" repository.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and then follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder**.**
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "scores by school size."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

4.13.1: Group By School Type

Late in the day, Maria sends you a message. She tells you how impressed she is by your work; the data you've generated will have a big impact on how the school board plans for the upcoming academic year. However, there's just one more thing: she would like you to create a DataFrame that groups the same averages and percentages by the type of school, district and charter.

For this final DataFrame, we'll group the scores by the type of school: district or charter. For each type of school, we'll calculate the average math and reading scores, math and reading passing percentages, and overall passing percentage for each bin, as shown in the following image:

```
type_summary_df
```

School Type	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Charter	83.5	83.9	94	97	90
District	77.0	81.0	67	81	54

Since the School Type column is in the `per_school_summary_df` DataFrame, we won't need to create this column as we did for the school spending and school size DataFrames. All we need to do is use the `groupby()` function to group the data on the `School Type`.

However, we will need to create four new Series, as we did for the two previous DataFrames by getting the following data:

- Average Math Score
- Average Reading Score
- Average % Passing Math
- Average % Passing Reading
- Average % Overall Passing

To get these averages, run the following code in a new cell:

```
# Calculate averages for the desired columns.
type_math_scores = per_school_summary_df.groupby(["School Type"]).mean()["Av

type_reading_scores = per_school_summary_df.groupby(["School Type"]).mean()[

type_passing_math = per_school_summary_df.groupby(["School Type"]).mean()["%

type_passing_reading = per_school_summary_df.groupby(["School Type"]).mean()

type_overall_passing = per_school_summary_df.groupby(["School Type"]).mean()
```

Let’s add all the averages to generate the final DataFrame!

4.13.2: Create a DataFrame for the Scores by School Type

Based on how quickly you're generating this data, you're working like a seasoned data analyst! There are just a few more steps you need to complete. Get ready to create the final DataFrame!

We're going to create a new DataFrame called `type_summary_df` that includes average math and reading scores, the average percentage of students who passed math and reading, and the average overall percentage.

Create the School Type DataFrame

Based on the work you've done so far, complete the following activity.

Complete the code to create the `type_summary_df` DataFrame.

```
type_summary_df = pd.DataFrame({
      "Average Math Score" : [                        ],
      "Average Reading Score": [                      ],
      "% Passing Math": [                             ],
      "% Passing Reading": [                          ],
      "% Overall Passing": [                          ])
```

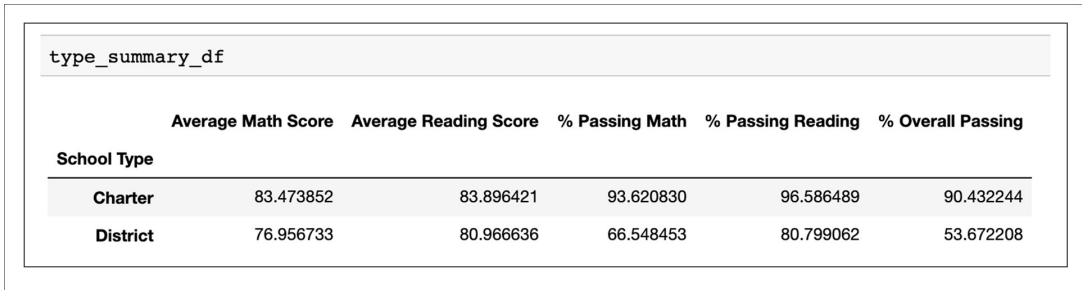
type_reading_scores type_passing_reading type_passing_math

type_math_scores type_overall_passing

To create the type summary DataFrame, add the following code to a new cell and run the cell.

```
# Assemble into DataFrame.
type_summary_df = pd.DataFrame({
          "Average Math Score" : type_math_scores,
          "Average Reading Score": type_reading_scores,
          "% Passing Math": type_passing_math,
          "% Passing Reading": type_passing_reading,
          "% Overall Passing": type_overall_passing})

type_summary_df
```

The results in the output window will look like this:



type_summary_df

School Type	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Charter	83.473852	83.896421	93.620830	96.586489	90.432244
District	76.956733	80.966636	66.548453	80.799062	53.672208

We're almost done! Like the previous DataFrames, this new DataFrame needs to adhere to grade-reporting standards. Therefore, we'll make the following formatting changes:

- Format the average math and reading scores to one decimal place
- Format the percentage of students passing math and reading to the nearest whole number
- Format the overall passing percentage to the nearest whole number

---

Format the DataFrame

To format the `type_summary_df` DataFrame, add the following code to a new cell and run the cell.

```
# Formatting
type_summary_df["Average Math Score"] = type_summary_df["Average Math Score"

type_summary_df["Average Reading Score"] = type_summary_df["Average Reading

type_summary_df["% Passing Math"] = type_summary_df["% Passing Math"].map("{

type_summary_df["% Passing Reading"] = type_summary_df["% Passing Reading"].

type_summary_df["% Overall Passing"] = type_summary_df["% Overall Passing"].

type_summary_df
```

The results should look like the following:

```
type_summary_df
```

School Type	Average Math Score	Average Reading Score	% Passing Math	% Passing Reading	% Overall Passing
Charter	83.5	83.9	94	97	90
District	77.0	81.0	67	81	54

4.13.3: Commit Your Final Code

You've finished your project! Now you need to update your GitHub repository with your latest work. This way, Maria will be able to view the most recent changes.

We need to save and commit our changes to GitHub. Save `PyCitySchools.ipynb` to the School_District_Analysis folder, and then follow these steps:

1. Launch the terminal for macOS or Git Bash for Windows.
2. Navigate to the School_District_Analysis folder.
3. Type `git status` and press Enter.
4. Type `git add .` and press Enter.
5. Check the status again with `git status` and press Enter.
6. Commit the files to be added to the repository by typing `git commit -m "scores by school type."` and press Enter.
7. Type `git push` and press Enter to add the file to your repository.
8. Refresh your GitHub page to see the repository changes.

Module 4 Challenge

Submit Assignment

Due Feb 9 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Maria and her supervisor have discovered that the score averages for ninth graders from one high school are incorrect. Maria has asked you to replace the math and reading scores for that high school, but without removing those ninth-grade students from the analysis.

In this challenge, you will use your skills using the Pandas library and Jupyter Notebook. You need to replace incorrect data in columns using logical operations with conditionals; retrieve data from DataFrames; merge, filter, slice, and sort DataFrames; and apply the `groupby()` function to a DataFrame. Using these methods, you will create a new analysis with the incorrect data removed.

Background

The grades of the ninth graders at Thomas High School have been changed. While administrators do not know the full extent of this academic dishonesty, they want to uphold the standards of state testing and have turned to you for help.

After assessing the situation with the school superintendent and Maria, you decide the best approach is to:

- Replace the ninth-grade math and reading scores from Thomas High School.
- Keep all other data associated with the ninth-grade students and Thomas High School intact.

Objectives

The goals of this challenge are for you to:

- Filter DataFrames using logical operators.

- Replace the incorrect values with NaN.
- Explain how your PyCitySchools analysis changes after you handle the incorrect data.

Instructions

1. Create a duplicate of `PyCitySchools.ipynb` and rename it `PyCitySchools_Challenge.ipynb`.
2. Correct the students' names so there are no professional prefixes or suffixes.
3. Replace the reading and math scores for ninth graders at Thomas High School with NaN.
    - Use `loc` on the `student_data_df` DataFrame to select the columns by condition.
    - Set the column you want equal to "NaN" by using `np.nan` for the reading and math scores separately.
        - You will need to import `import numpy as np` to use `np.nan`.
4. Merge the clean student data with the school dataset.
5. After replacing the reading and math scores, answer the following questions:
    - How is the district summary affected?
    - How is the school summary affected?
    - How does replacing the ninth graders' math and reading scores affect Thomas High School's performance, relative to the other schools?
    - How does replacing the ninth-grade scores affect the Math and Reading Scores by Grade, Scores by School Spending, Scores by School Size, and Scores by School Type?
6. To answer the questions above, you will need to repeat some of the steps you did during the module:
    - Recreate the district and school summary DataFrames.
    - Recalculate the top 5 and bottom 5 performing schools.
    - Recalculate the average math score received by students in each grade level at each school.

- Recalculate the average reading score received by students in each grade level at each school.
- Recalculate the school performance based on the spending per student.
- Recalculate the school performance based on the size of the school.
- Recalculate the school performance based on the type of school.

> **Hint**
>
> - [Pandas documentation on the loc method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html)]
> - [Using np.nan on a DataFrame (https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html)]

> **Add, Commit, Push**
>
> When you have completed the challenge, add the `PyCitySchools_Challenge.ipynb` file to your School_District_Analysis GitHub repository.

Submission

To submit your challenge assignment, click Submit, then provide the URL of your School_District_Analysis GitHub repository for grading.

[Module 4 Challenge Rubric]

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 4 Challenge Rubric

Criteria	Ratings					Pts
Written Report Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** Presents a cohesive written analysis that correctly answers the questions listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** Presents a cohesive written analysis that correctly answers three of the questions listed in the detailed rubric.	**10.0 pts** **Progressing** Presents a developing written analysis that correctly answers two or more of the questions listed in the detailed rubric.	**5.0 pts** **Emerging** Presents a limited written analysis or no written analysis that correctly answers two or more of the questions listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
Expected output displayed Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** Output for PyCitySchools contains all of the DataFrames listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** Output for PyCitySchools contains at least 5 of the DataFrames listed in the detailed rubric.	**10.0 pts** **Progressing** Output for PyCitySchools contains at least 4 of the DataFrames listed in the detailed rubric.	**5.0 pts** **Emerging** Output for PyCitySchools contains at least 3 of the DataFrames listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
Functions used on DataFrames Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** All of the functions listed in the detailed rubric are used on DataFrames and produce correct results.	**15.0 pts** **Approaching Mastery** At least 6 of the functions listed in the detailed rubric are used on DataFrames and produce varying results.	**10.0 pts** **Progressing** At least 4 of the functions listed in the detailed rubric are used on DataFrames and produce varying results.	**5.0 pts** **Emerging** At least 3 of the functions listed in the detailed rubric are used on DataFrames and produce varying results.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
GroupBy used Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** GroupBy is used in determining the all of the items listed on the detailed rubric.	**15.0 pts** **Approaching Mastery** GroupBy is used in determining at least 3 of the items listed on the detailed rubric.	**10.0 pts** **Progressing** GroupBy is used in determining at least 2 of the items listed on the detailed rubric.	**5.0 pts** **Emerging** GroupBy is used in determining 1 or fewer of the items listed on the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
	20.0 pts	**15.0 pts**	**10.0 pts**	**5.0 pts**	**0.0 pts**	20.0 pts

Criteria	Ratings					Pts
Drop rows or NaN to replace values	**Mastery** Replaces both the reading and math scores for the 9th graders in Thomas High school with NaN.	**Approaching Mastery** Replaces either the reading or math scores for the 9th graders in Thomas High school with NaN.	**Progressing** Replaces all the values for the 9th graders in Thomas High School with NaN.	**Emerging** Drops the rows instead of replacing the incorrect grades with NaN.	**Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty.	
						Total Points: 100.0

Rubric for PyCitySchools:

	Mastery **20 points**	**Approaching Mastery** **15 points**	**Progressing** **10 points**	**Emerging** **5-0 points**	**Incomplete**
Written Report	Presents a cohesive written analysis that correctly answers the following questions: ✓ How is the district summary affected? ✓ How is the school summary affected? ✓ How does removing the ninth graders' math and reading scores affect Thomas High School's performance, relative to the other schools? ✓ How does removing the ninth grade scores affect the following - Math and Reading Scores by Grade - Scores by School Spending - Scores by School Size - Scores by School Type	Presents a cohesive written analysis that correctly answers three of the following questions : ✓ How is the district summary affected? ✓ How is the school summary affected? ✓ How does removing the ninth graders' math and reading scores affect Thomas High School's performance, relative to the other schools? ✓ How does removing the ninth grade scores affect the following - Math and Reading Scores by Grade - Scores by School Spending - Scores by School Size - Scores by School Type	Presents a developing written analysis that correctly answers two or more of the following questions: ✓ How is the district summary affected? ✓ How is the school summary affected? ✓ How does removing the ninth graders' math and reading scores affect Thomas High School's performance, relative to the other schools? ✓ How does removing the ninth grade scores affect the following - Math and Reading Scores by Grade - Scores by School Spending - Scores by School Size - Scores by School Type	Presents a limited written analysis or no written analysis that correctly answers two or more of the following questions: ✓ How is the district summary affected? ✓ How is the school summary affected? ✓ How does removing the ninth graders' math and reading scores affect Thomas High School's performance, relative to the other schools? ✓ How does removing the ninth grade scores affect the following - Math and Reading Scores by Grade - Scores by School Spending - Scores by School Size - Scores by School Type	
Expected output displayed	Output for PyCitySchools contains all of the following DataFrames: ✓ `District Summary` ✓ `School Summary` ✓ `High and Low Performing Schools` ✓ `Math and Reading Scores by Grade` ✓ `Scores by School Spending` ✓ `Scores by School Size` ✓ `Scores by School Type`	Output for PyCitySchools contains at least 5 of the following DataFrames: ✓ `District Summary` ✓ `School Summary` ✓ `High and Low Performing Schools` ✓ `Math and Reading Scores by Grade` ✓ `Scores by School Spending` ✓ `Scores by School Size` ✓ `Scores by School Type`	Output for PyCitySchools contains at least 4 of the following DataFrames: ✓ `District Summary` ✓ `School Summary` ✓ `High and Low Performing Schools` ✓ `Math and Reading Scores by Grade` ✓ `Scores by School Spending` ✓ `Scores by School Size` ✓ `Scores by School Type`	Output for PyCitySchools contains 3 or fewer of the following DataFrames: ✓ `District Summary` ✓ `School Summary` ✓ `High and Low Performing Schools` ✓ `Math and Reading Scores by Grade` ✓ `Scores by School Spending` ✓ `Scores by School Size` ✓ `Scores by School Type`	No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty
Functions used on DataFrames	The following functions are used on DataFrames and produce correct results:	Six of the following functions are used on DataFrames and produce varying results:	Four of the following functions are used on DataFrames to produce varying results:	Three or fewer of the following functions are used on DataFrames to produce varying results:	

	✓ Mean ✓ Sum ✓ Count ✓ Cut ✓ Map ✓ Format ✓ Sort_values ✓ Merge *Note: values may be up to 1% different than the solution files.*	✓ Mean ✓ Sum ✓ Count ✓ Cut ✓ Map ✓ Format ✓ Sort_values ✓ Merge *Note: values may be up to 1% different than the solution files.*	✓ Mean ✓ Sum ✓ Count ✓ Cut ✓ Map ✓ Format ✓ Sort_values ✓ Merge *Note: values may be up to 1% different than the solution files.*	✓ Mean ✓ Sum ✓ Count ✓ Cut ✓ Map ✓ Format ✓ Sort_values ✓ Merge *Note: values may be up to 1% different than the solution files.*	
GroupBy used	GroupBy is used in determining the following: ✓ Math and Reading Scores by Grade ✓ Scores by School Spending ✓ Scores by School Size ✓ Scores by School Type *Note: while you completed the above steps during the module content, you will need to do them again to complete the challenge.*	GroupBy is used for in determining at least 3 of the following: ✓ Math and Reading Scores by Grade ✓ Scores by School Spending ✓ Scores by School Size ✓ Scores by School Type *Note: while you completed the above steps during the module content, you will need to do them again to complete the challenge.*	GroupBy is used for in determining at least 2 of the following: ✓ Math and Reading Scores by Grade ✓ Scores by School Spending ✓ Scores by School Size ✓ Scores by School Type *Note: while you completed the above steps during the module content, you will need to do them again to complete the challenge.*	GroupBy is used for in determining 1 or fewer of the following: ✓ Math and Reading Scores by Grade ✓ Scores by School Spending ✓ Scores by School Size ✓ Scores by School Type *Note: while you completed the above steps during the module content, you will need to do them again to complete the challenge.*	
Drop rows or use NaN to replace values	✓ Replaces **both** the **reading** and **math** scores for the 9th graders in Thomas High school with NaN.	✓ Replaces **either** the reading or math scores for the 9th graders in Thomas High school with NaN.	✓ Replaces **all the values** for the 9th graders in Thomas High School with NaN.	✓ **Drops the rows** instead of replacing the incorrect grades with NaN.	

Module 4 Career Connection

Highlight Your Transferable Skills

If you don't have one already, something you'll be crafting in this course is a professional brand statement. A brand statement expresses your professional value and provides a clear and concise introduction for employers. Before you start drafting your statement, it's important to understand the value you add through your transferable skills, background, and experiences. Transferable skills are the soft skills you've developed that will allow you to succeed in any role. Examples include skills like leadership, communication, and management.

For a guided experience on highlighting your transferable skills and finding ways to tell your professional story, watch this.

Career Services Next Step:

After watching the video, create a list of the following:

1. Your transferable skills
2. Relevant education
3. Your area(s) of expertise
4. Notable accomplishments

Module 4 Survey

Due No due date **Questions** 14 **Time Limit** None

Instructions

Congratulations on completing the module. Please take a moment to reflect back over the last week and provide feedback. We value this feedback as we continually enhance our program.

Take the Survey