Treading on Python Series

EFFECTIVE PANDAS 2

Opinionated Patterns for Data Manipulation

Matt Harrison

Effective Pandas 2

Opinionated Patterns for Data Manipulation

Effective Pandas 2

Opinionated Patterns for Data Manipulation

Matt Harrison

hairysun.com

Contents

Foreword

Many different libraries can be used for data analysis tasks in Python, but Pandas has been the cornerstone of the Python data science and data analysis ecosystem for many years now, The library was built over the last 15 years by a huge open-source community, and it is used by almost everyone who works with data in Python. It is a library that data scientists, data engineers and data analysts use. The huge API surface of Pandas makes it possible to solve almost any data analysis problem with it.

My journey into the Python world started many years ago while I was building a data analysis platform for a previous client. Similar to many of the newer Pandas users, my entry point into the Python world and PyData ecosystem was Pandas. I quickly started to love the library and its capabilities. I was able to accomplish complex tasks with very few lines of code. I especially appreciated the documentation that made the library easily accessible for beginners like myself and helped me get started very quickly.

After working with the PyData stack for a while, I found my way into Open Source development roughly four years ago. Pandas, again, opened the door for me. It was the library that I was most familiar with back then, which made me choose to start my journey as a contributor with it. I started to realize how much work goes into building and maintaining a library like Pandas, which made me appreciate the work done by the community even more. However, the community can't do everything, which is where this book comes in.

I've met Matt for the first time last year at a conference where we got talking. I greatly admire the perspective that this book brings to the table. The style is very hands-on and practical; it avoids unnecessary details and focuses on the important parts of the library. The Pandas API is huge, but Matt did an excellent job covering almost all of it. The book is very close to the pulse of Pandas, integrating the Arrow backend for DataFrames that was released in the latest major version. It is an excellent resource for users of all levels, from beginners to advanced users who want to apply the latest improvements to their work. Matt aims to get the reader to the practical part as fast as possible in all of his books, and this book is no exception. It includes many helpful exercises and focuses on readers who want to apply the knowledge they learn in the book to their own work.

Pandas has a vast API, and it is easy to get lost in the details or stumble into inefficient patterns while getting started. This book will help you avoid these pitfalls and start your journey into the Data Analysis world with Python. If you get through all of it, you will be well-prepared to efficiently leverage Pandas in whatever situation necessary.

Happy Coding!

Patrick Hoefler

Chapter 1

Introduction

I have been using Python in some professional capacity or another since the turn of the century. Over these years, I've witnessed Python's rising prominence in numerous data science domains - from data gathering and cleansing to analysis, machine learning, and visualization. The pandas library has seen much uptake in this area.

pandas[1] is a data analysis library for Python that has exploded in popularity over the past years. The website describes it like this:

> "pandas is an open-source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language."
>
> -pandas.pydata.org

My description of pandas is: pandas is an in-memory analysis tool with SQL-like constructs, essential statistical and analytic support, and graphing capability. Owing to its foundation on Cython and NumPy, pandas offers reduced memory overhead and enhanced speed compared to standard Python code. Many people use pandas to replace Excel, perform ETL (extract transform load processing to move data from one place to another), process tabular data, load CSV or JSON files, prep for machine learning, and more. Though it grew out of the financial sector (for time series analysis), it is now a general-purpose data manipulation library.

With its NumPy lineage, pandas adopts some NumPy'isms that regular Python programmers may not be aware of or familiar with. Yes, one could go out and use Cython to perform fast typed data analysis with a Python-like dialect, but you don't need to with pandas. This work is done for you. If you use pandas and vectorized operations, you are getting close to C-level speeds for numeric work but writing Python.

[1] pandas (http://pandas.pydata.org) refers to itself in lowercase, so this book will follow suit. When I'm referring to specific code, I will set it in a monospace font.

1.1 Who is this book for

This guide is intended to introduce pandas and patterns for best practices. If you work with tabular data and need capabilities beyond Excel, this is for you. This book covers many (but not all) aspects of the library, as well as some gotchas or details that may be counter-intuitive or even non-pythonic to longtime users of Python.

This book assumes a basic knowledge of Python. The author has written *Learning Python for Data* that provides all the background necessary.

1.2 Data in this Book

Every attempt has been made to use data that illustrates real-world pandas usage. As a visual learner, I appreciate seeing where data is coming and going. As such, I avoid just showing tables of random numbers with no meaning. I will show the best practices gleaned from years of using pandas.

I have selected a variety of datasets to show that the advice given in this book is applicable in most situations you may encounter.

1.3 Hints, Tables, and Images

The hints, tables, and graphics in this book have been collected over my years of using pandas. They come from hang-ups, notes, and cheat sheets that I have developed after using pandas and teaching others how to use the library.

In the physical version of this book, there is an index that has also been battle-tested during development. Inevitably, when I was doing analysis for consulting or clients, I would check that the index had the information I needed. If it didn't, I added it.

If you enjoy this book, please consider writing a review on Amazon. That is one of the best ways to thank an author.

Chapter 2

Installation

This book will use Python 3 throughout! Please do not use Python 2 unless you have a compelling reason to. Python 3 is the future of the language, and the current pandas releases do not support Python 2.

2.1 Anaconda

With that out of the way, let's address the installation of pandas. One way to install pandas on most platforms is to use the Anaconda distribution[1]. Anaconda is a meta-distribution of Python, which contains many additional packages that have traditionally been annoying to install unless you have the necessary toolchains to compile Fortran and C code. Anaconda allows you to skip the compile step because it provides binaries for most platforms. The Anaconda distribution is freely available, though commercial support is also available.

After installing the Anaconda package, you should have a `conda` executable. Running the following command will install pandas:

```
$ conda install pandas pyarrow
```

Note

This book shows commands run from the UNIX command prompt. They are prefixed by the prompt `$`. Unless otherwise noted, these commands will also run on the Windows command prompt. Do not type the prompt. It is included to distinguish commands run via a terminal or command prompt from Python code.

We can verify that this works by trying to import the `pandas` package:

[1] https://anaconda.com/downloads

```
$ python
>>> import pandas
>>> pandas.__version__
'2.2.0'
```

Note

The command above shows a Python prompt, `>>>`. Do not type the Python prompt. It is included to make it easy to distinguish Python code from the output of Python code. For example, the output of the above, `'2.1.3'` does not have the prompt in front of it. The book also includes the secondary Python prompt, `...` for longer than a single line code.

Note that Jupyter does not use the Python prompt in its cells.

If the library successfully imports, you should be good to go.

2.2 Pip

If you aren't using Anaconda, I recommend you use pip[1] to install pandas. The pandas library will install on Windows, Mac, and Linux via pip.

It may be necessary to prepare the operating system for building pandas from source by installing dependencies and the proper header files for Python. On Ubuntu, this is straightforward, other environments may be different:

```
$ sudo apt-get install build-essential python-all-dev
```

Using a *virtual environment* will alleviate the need for superuser access during installation. This lets us download and install newer releases of pandas even if the version found on the distribution is lagging.

On Mac and Linux platforms, the following commands create a virtualenv sandbox and install the latest pandas in it (assuming that the prerequisite files are also installed):

```
$ python3 -m venv pandas-env
$ source pandas-env/bin/activate
(pandas-env)$ pip install pandas pyarrow
```

Once you have pandas installed, confirm that you can import the library and check the version:

```
(pandas-env)$ python
>>> import pandas
>>> pandas.__version__
'2.2.0'
```

[1] http://pip-installer.org/

On Windows, you will open a Command Prompt and run the following to create a virtual environment:

```
> python -m venv pandas-env
> pandas-env/Scripts/activate
(pandas-env)> pip install pandas pyarrow
```

Note

The Windows command prompt, `>`, is shown in the previous command. Do not type it. Only type the commands following the prompt.

Try to import the library and check the version:

```
(pandas-env)> python
>>> import pandas
>>> pandas.__version__
'2.1.3'
```

2.3 Summary

In this chapter, we saw how to set up a Python environment using Anaconda or Pip.

2.4 Exercises

1. Install pandas on your machine (using Anaconda or pip).

Chapter 3

Using Jupyter

A standard tool for engineers, scientists, and data people is Jupyter [1]. Jupyter is an open-source web application that allows you to create and share documents containing live code. It is an excellent platform for using a Python REPL. Jupyter supports various programming languages, in addition to Python, including R and Julia.

3.1 Using a REPL in Jupyter

Jupyter can be incredibly useful for data analysis and exploration. You can input code snippets to test their functionality or to experiment with a data set quickly. Because the results of each command are displayed immediately, you can quickly iterate through your analysis until you find the desired output. Additionally, using a REPL from Jupyter makes documenting your progress and thought process more manageable as you work through an analysis, as you can intersperse text and code cells.

Jupyter is a generic term for a notebook interface. The original tool was called Jupyter Notebook (actually, it was called iPython Notebook and was renamed Jupyter to emphasize that it works with the Julia, Python, and R languages). An updated version called Jupyter Lab was introduced later.

Jupyter Notebook and Jupyter Lab are open-source web applications that allow you to create and share live documents containing code, equations, visualizations, and narrative text. However, there are some significant differences between the two.

Jupyter Notebook is the classic Jupyter application that has existed for a long time. It has a simpler user interface.

JupyterLab is the newer version of Jupyter and is designed to be more versatile and extensible. It offers a more feature-rich and customizable interface that can be extended to fit your specific needs. For example, with JupyterLab, you can drag and drop the tabs to customize your workspace according to your workflow.

[1] https://jupyter.org/

Both Notebook and Lab have code cells; from this book's point of view, you can use either tool to follow along.

3.2 Using Notebook

To install Jupyter Notebook, run the command:

```
$ pip install notebook
```

You can launch the Jupyter Notebook with the command:

```
$ jupyter notebook
```

As of version 7, Jupyter Notebook launches you into a stripped-down version of the Lab interface. Go to the View -> "Open in NbClassic" menu to access the old Notebook interface.

I talk about Lab and Notebook because many services still use the classic notebook interface rather than the lab interface. However, many of the commands are the same in lab and notebook.

To create a new notebook, click the "New" button and select "Notebook". You should see a new notebook window pop up. This window is both an editor and a Python REPL.

Type your code into the first cell of the notebook. Since you are doing the hello world example, type:

```
print("hello world")
```

To run the code, click on the "Run" button or press the "Control" and "Enter" keys together. Jupyter Notebook will immediately evaluate your code and display the output below the cell.

This might seem trivial, but the notebook now has the state of your code. In this case, you only printed to the screen, so there is little state. In future examples, you will see how you can use Jupyter to quickly try out code, see the results, and inspect the output of a program.

3.3 Using Lab

To install JupyterLab, run the command:

```
$ pip install jupyterlab
```

You can launch the tool with the command:

```
$ jupyter lab
```

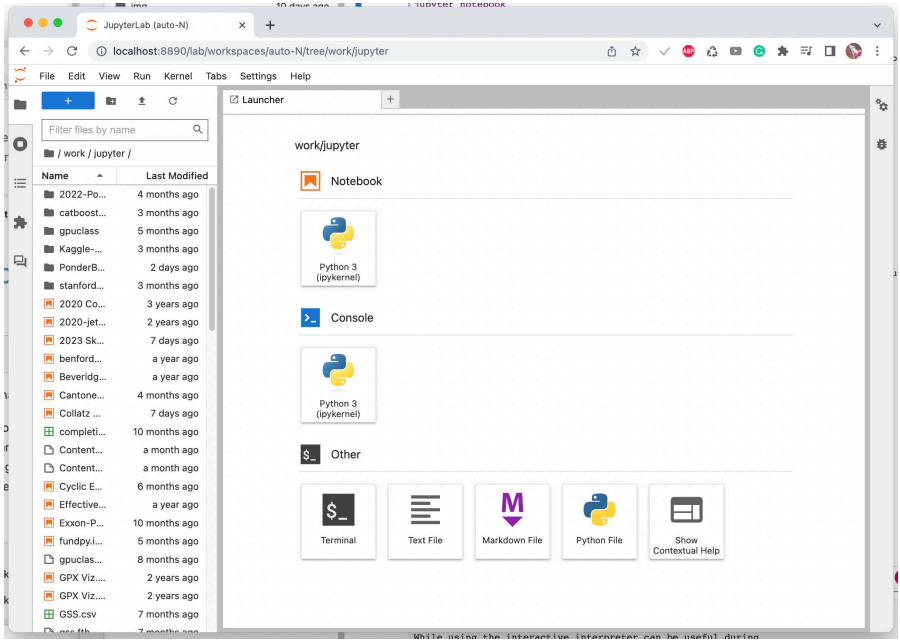


Figure 3.1: Landing page after starting JupyterLab.

When JupyterLab launches, you will see the dashboard where you can navigate to your files and create new notebooks. To create a new notebook, click the "Notebook" button in the "Launcher" pane or the + button above the file navigator. You should see a new notebook window pop up. This window is both an editor and a Python REPL.

Type your code into the first cell of the notebook. Since you are doing the hello world example, type:

```
print("hello world")
```

To run the code, click on the "Run" button or press the "Control" and "Enter" keys together. Jupyter Notebook will immediately evaluate your code and display the output below the cell.

3.4 Jupyter Modes

Both Lab and Notebook have two modes: Edit mode and Command mode.

Edit mode is where you can edit the contents of a cell. Jupyter Notebook denotes this mode by a green box around the current cell. In JupyterLab, you will see a cursor in the cell, and the background color of the cell will change from grey to white.

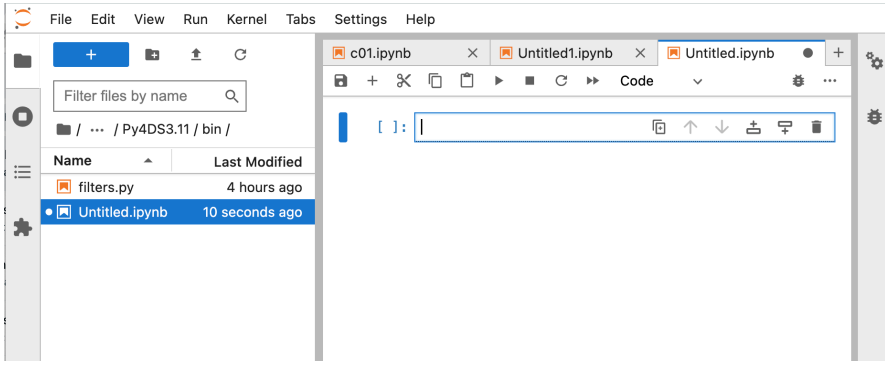


Figure 3.2: Jupyter Lab has been launched and a new notebook window has been created.

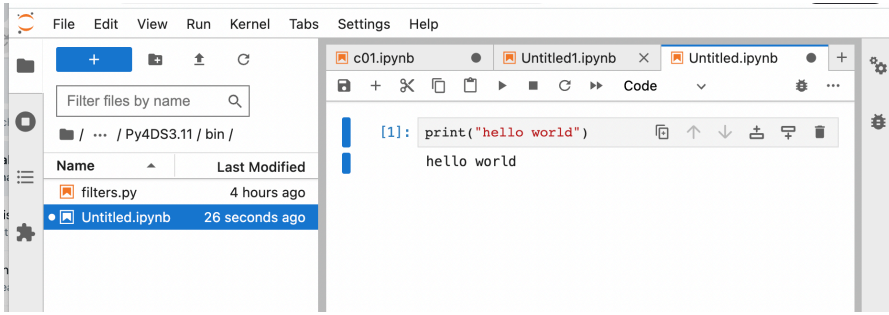


Figure 3.3: Code typed into the notebook and executed.

In edit mode, you can type into the cell and edit its contents. You can also use keyboard shortcuts within edit mode, such as *Ctrl + Enter* to run the cell or *Ctrl + Shift + -* to split the cell at the current cursor position.

To enable edit mode, you can type Enter when a cell is highlighted or double-click on the content of the cell. To exit edit mode and return to command mode, you can execute the cell (*Ctrl + Enter*) or hit Escape.

Command mode, on the other hand, is where you can perform various actions on the notebook without editing its contents. A blue box around the current cell indicates this mode. In command mode, you can navigate between cells using the arrow keys, delete cells by pressing 'dd' (the d key twice), create new cells using the 'a' or 'b' keys, and more.

The main difference between these two modes is that edit mode focuses on editing a cell's contents. In contrast, command mode manages the notebook's structure and organization. Knowing how to navigate between these modes

and use their respective actions effectively is key to working efficiently in Jupyter.

3.5 Common Commands

Here is a brief description of some of the keyboard shortcuts that are helpful commands in command mode in Jupyter:

h Show keyboard shortcuts.
a Inserts a new cell **a**bove the current cell.
b Inserts a new cell **b**elow the current cell.
x Cuts the current cell.
c Copies the current cell.
v Pastes the cell that was cut or copied.
dd Deletes the current cell. (You need to hit d twice.)
m Changes the current cell to a markdown cell.
y Changes the current cell to a code cell.
ii Interrupts the kernel running in the current notebook session.
00 Restarts the kernel running in the current notebook session.
Ctrl-Enter Runs the current cell.
Enter Go into edit mode.

These commands can save a lot of time when you're working within Jupyter notebooks. You can quickly create new cells, change cell types, delete cells, and interrupt or restart the kernel using keyboard shortcuts. This can help you focus more on coding and data analysis and less on navigating the notebook interface.

3.6 Line and Cell Magics

Jupyter has a concept of *magics* which are special commands you can run in a cell. These commands are prefixed with a `%` or `%%` and are called *line magics* and *cell magics*, respectively. Line magics are commands run on a single line, while cell magics are run on the entire cell.

For example, the `%timeit` line magic can be used to time how long it takes to run a line of code. You can use it like this:

```
%timeit x = range(10000)
```

This command will run the `range(10000)` command some number of times and time how long it takes to run.

A line magic only works on a single line of code. A cell magic, on the other hand, works on the entire cell. Here is an example of using the `%%timeit` cell magic:

```
%%timeit
x = range(10000)
total = 0
for i in x:
    total += i
mean = total / len(x)
```

This command runs the whole cell multiple times and returns the average time it took to run the cell.

There are many other line and cell magics that you can use in Jupyter. You can see a list of them by running the `%lsmagic` command in a cell. This will print out a list of all the magics available in your notebook.

Some of the most useful magics are:

`%matplotlib inline` This line magic will display matplotlib plots inline in the notebook. (This is the default behavior in JupyterLab and is not needed there.)
`%timeit` This line magic will time how long it takes to run a line of code.
`%%timeit` This cell magic will time how long it takes to run a cell of code.
`%debug` This line magic will start the interactive debugger.
`%pdb` This line magic will enable the interactive debugger.
`%%html` This cell magic will render the cell as HTML.
`%%javascript` This cell magic will run the cell's contents as JavaScript code.
`%%writefile` This cell magic will write the cell's contents to a file.

3.7 Summary

In this chapter, you learned how to install and use Jupyter Notebook and JupyterLab. You learned how to create a new notebook, run code, and navigate between cells. You also learned about the difference between edit mode and command mode and how to use keyboard shortcuts to perform actions in each mode.

3.8 Exercises

1. What is the difference between Jupyter Notebook and JupyterLab?
2. How do you create and run a new code cell in Jupyter?
3. How do you change a cell's type from code to markdown?
4. How can you save your work in a Jupyter Notebook?
5. What happens when you interrupt the kernel running in a Jupyter notebook?
6. How do you restart the kernel running in a Jupyter notebook?

Chapter 4

Data Structures

This chapter will introduce the two main data structures in pandas. Understanding these data structures isn't just academic; it is essential to using pandas effectively. The data structures are the `Series` and the `DataFrame`.

4.1 Series and DataFrame

One of the keys to understanding pandas is to understand the data model. At the core of pandas are two data structures. The most widely used data structures are the `Series` and the `DataFrame` for dealing with array and tabular data. A series is a one-dimensional array representing a single data column in a DataFrame. A DataFrame is a two-dimensional array used to represent a tabular, spreadsheet-like data structure. This table shows their analogs in the spreadsheet and database world.

Table 4.1: Different dimensions of pandas data structures

Data Structure	Dimensionality	Spreadsheet Analog	Database Analog	Linear Algebra
`Series`	1D	Column	Column	Column Vector
`DataFrame`	2D	Single Sheet	Table	Matrix

An analogy with the spreadsheet world illustrates the fundamental differences between these types. A `DataFrame` is similar to a sheet with rows and columns, while a `Series` is similar to a single column of data (when we refer to a column of data in this text, we refer to a `Series`).

Diving into these core data structures is helpful because a bit of understanding goes a long way toward better library use. We will spend a good portion of time discussing the `Series` and `DataFrame`. Both the `Series` and `DataFrame` share features. For example, they both have an index, which we will need to examine to understand how pandas works.

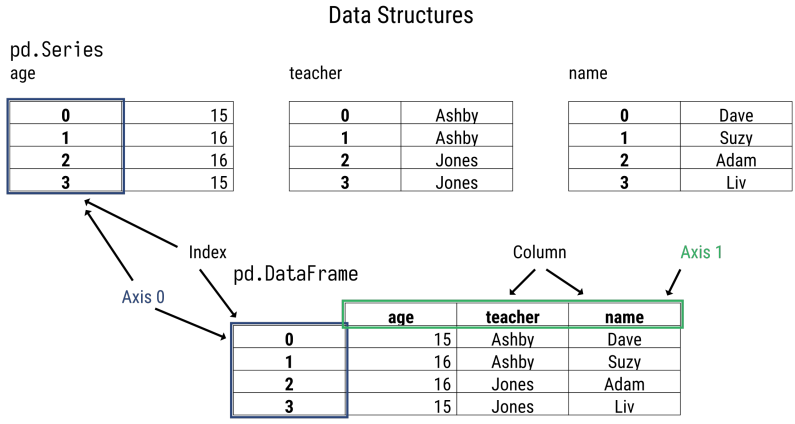


Figure 4.1: Figure showing the relation between the main data structures in pandas. Namely, a dataframe can have one or many series.

Also, because the `DataFrame` can be thought of as a collection of columns that are really `Series` objects, it is imperative that we have a comprehensive study of the `Series` first. Additionally (and perhaps odd to some), we will see this when we iterate over rows, and the rows are represented as `Series` (however, if you find yourself consistently dealing with rows instead of columns, you are probably not using pandas optimally).

Some have compared the data structures to Python lists or dictionaries, and I think this is a stretch that doesn't provide much benefit. Mapping the list and dictionary methods on top of pandas' data structures leads to confusion.

4.2 Summary

The pandas library includes two primary data structures and associated functions for manipulating them. This book will focus on the `Series` and `DataFrame`. First, we will look at the `Series` as the `DataFrame` can be considered a collection of columns represented as `Series` objects.

4.3 Exercises

1. If you had a spreadsheet with data, which pandas data structure would you use to hold the data? Why?
2. If you had a database with data, which pandas data structure would you use to hold the data? Why?

Chapter 5

Of Types Python, NumPy, and PyArrow (And Pandas)

This chapter will seem unrelated at first, but it's one of the most critical chapters in the book. It's about types. Specifically, it's about the types of data you'll encounter in the data world of Python. We are going to open the Pandora's box of Python types. We will see the benefits and significant drawbacks of how Python handles types. We will also see how Pandas uses types to make your life easier.

If you want to get the most out of Pandas, you need to understand the types of data that it uses.

5.1 Table of Types

Here is a table of the types that we will be discussing in this chapter.

Table 5.1: Table of Pandas 1 and Pandas 2 types

Data Type	SQL	Python	Pandas 2	Pandas 1
Integer	`INT`	`int`	`'int64[pyarrow]'`	`'int64'`
Float	`FLOAT`	`float`	`'float64[pyarrow]'`	`'float64'`
String	`VARCHAR`	`str`	`pd.ArrowDType(pa.String())` [1]	`object`, `str`
Date	`DATE`	`datetime`	`'timestamp[ns][pyarrow]'`	`'datetime64[ns]'`
Categorical	N/A	N/A	`'dictionary'` [2]	`'category'`
Boolean	`BOOLEAN`	`bool`	`'bool[pyarrow]'`	`bool`

[1] Sadly, 'string[pyarrow]' came out before the Pandas 2 pyarrow transition and many operations with this type return legacy NumPy types.

[2] Pyarrow has a dictionary type, but it is not exposed easily in Pandas. I recommend using the `category` type instead.

5.2 Python Types

When we are dealing with Series and DataFrames, you will mostly find data that consists of numbers, strings, and dates. Let's go through each of these types and see how they are represented in Python.

Numbers in Python are represented by the `int` and `float` types. The `int` type is for integers, and the `float` type is for floating point numbers. At the implementation level, Python creates an object for each number you create. This object contains the type of the number (`int` or `float`), the number's value, and some other information. It can hold an arbitrarily large number of digits. This differs from other languages like C, where numbers are stored directly in memory. When you create an integer in C, the computer allocates a certain number of bytes in memory and stores the number directly in those bytes. If you specify that the number is an `int`, the computer will allocate 4 bytes of memory, and the largest number you can store in that memory is $2^{32} - 1$.

To create an eight-bit integer in C, you would use the following code:

```
int8_t x = 127;
```

The maximum value a signed eight-bit integer can hold is 127. The `x` variable takes up one byte of memory. If you try to store a larger number in it, you will get a value of `-128`. This is called **overflow**. If you want to store a larger number, you must use a larger type.

In Python, if you create a variable to store 127, you just do the following:

```
>>> x = 127
```

If you add one to `x`, you get the following:

```
>>> x + 1
128
```

You don't have to worry about overflow. Python will automatically allocate more memory for the number. If you want to see how much memory Python is using to store the number, you can use the `sys.getsizeof` function:

```
>>> import sys
>>> sys.getsizeof(x)
28
```

You can see that Python uses 28 bytes to store the number.

This behavior of Python has benefits and drawbacks. The benefit is that you don't have to worry about the size of the number you create. You can create a number with 100 digits, and Python will happily store it. The drawback is that Python has to do more work to store and manipulate numbers. This means that Python is slower than C when it comes to numbers. It also uses more memory.

For a single number, it doesn't matter if something takes a long time to compute or uses a lot of memory. But when dealing with large amounts of data, these things can make a big difference. If you have a million numbers, it will take much longer to compute something with Python than with C. It will also take up a lot more memory.

You might wonder why Python is so popular for data science if it is slower than C. The answer is a library called NumPy. If it weren't for NumPy, I wouldn't be writing this book, and you probably wouldn't be interested in using Python for data work.

5.3 NumPy

NumPy is the unsung hero of the Python data science world. I blame it for the popularity of Python in data science. It can sidestep Python's problems with numbers and make it fast and efficient.

How does NumPy address Python's problems? It does so by creating a new type called `ndarray`. This type is similar to Python's `list` type but is much more efficient. It is more efficient because it stores the data in a contiguous memory block. Instead of creating an object for each number, it creates an object for the entire array. This means it can do things like vectorized operations, which are much faster than looping over each element in the array.

When we create an array of integers in NumPy, we must specify the number of bytes we want to allocate for each integer. This is called the **data type** of the array. A single-character code specifies the data type. The code for an integer is `i`, and the code for a floating point number is `f`. A number specifies the number of bytes. For example, if we want to create an array of 32-bit integers, we would use the code `i4`. If we want to create an array of 64-bit floating point numbers, we would use the code `f8`.

Let's compare and contrast this with C. To create an array of ten 32-bit integers in C, we would do the following:

```c
// Define an array of 10 32-bit integers
int array[10];

// Initialize the array with values from 0 to 9
for (int i = 0; i < 10; i++) {
    array[i] = i;
}
```

The `array` variable points to a block of 40 bytes of memory. Each integer takes up 4 bytes of memory. The `array` variable is a pointer to the first element in the array. The `array[0]` variable is the first element in the array. The `array[1]` variable is the second element in the array. And so on.

If we wanted to add one to each element in the array, we would do the following:

```
    // add one to each element in the array
    for (int i = 0; i < 10; i++) {
        array[i] += 1;
    }
```

Let's look at how to do the same thing in NumPy. First, we need to import the NumPy library:

```
>>> import numpy as np
```

Next, we create an array of ten 32-bit integers:

```
>>> array = np.arange(10, dtype='int32')
```

Finally, we add one to each element in the array:

```
>>> array += 1
```

This last part is fantastic. We don't even need to use a loop to add one to each element in the array. NumPy does it for us. This is called **vectorization**. It is one of the most important features of NumPy. It allows us to add one to each element in an array without writing a loop. This makes our code much more readable and maintainable. It also makes our code run very fast. Modern CPUs are designed to do vectorized operations. They can quickly take a memory buffer and add one to each element in the buffer. This is called **SIMD** (Single Instruction Multiple Data). It is one of the most important features of modern CPUs. (Of course, we could also program the CPU to do SIMD in C.)

Here is how SIMD in C might look:

```
    #include <immintrin.h>  // For AVX intrinsics

    // ...

    int main() {
        int array[10];

        // ... (initialize your array or whatever you need)

        // Load a vector of ones
        __m256i ones = _mm256_set1_epi32(1);

        for (int i = 0; i < 10; i += 8) {
            // Load 8 integers from the array into a 256-bit register
            __m256i vec = _mm256_loadu_si256((__m256i*)&array[i]);

            // Add ones to the vector
```

```
            vec = _mm256_add_epi32(vec, ones);

            // Store the result back into the array
            _mm256_storeu_si256((__m256i*)&array[i], vec);
        }

        // Handle any leftover elements
        // (in this particular case, it will exceed array bounds, so be
        //  cautious)

        return 0;
    }
```

I'm not going deep into this code, but I want you to know the work you would need to do that only took one line in NumPy. This is why NumPy is so popular. It allows you to do things that would be quite tedious in C. But it runs almost as fast as C and uses almost as little memory.

5.4 Differences in NumPy and Python

NumPy allows us to create arrays and matrices of numbers. If you need to work on a large matrix of numbers, NumPy is the way to go. NumPy supports basic integer and floating point types. Unlike Python, you need to specify the size of the number when you create the array. This is because NumPy doesn't use Python numbers internally. It uses C numbers. So, it is possible to have an overflow in NumPy.

```
>>> import numpy as np
>>> n1 = np.array([1], dtype='uint8')
>>> n255 = np.array([255], dtype='uint8')
>>> n1 + n255
array([0], dtype=uint8)
```

If you are used to Python's behavior of *just working* with large integers, you must be aware that NumPy can't hold integers larger than 64 bits. If you try to create an array with a larger integer, NumPy will put a Python object inside of the array's values. This will make your code run much slower. So, if you need to work with large integers, you should use Python's `int` type instead of NumPy's `uint64` type.

Another thing to note is that NumPy integer arrays do not like to have missing values. If you try to create an array with a missing value, NumPy will convert the missing value to a `nan` (not a number) value and then change the type of the array to a floating-point type. This doesn't necessarily make it run slower, but you need to know that it will be doing floating-point operations instead of integer ones.

```
>>> import numpy as np
>>> demo = np.array([1, 2, 3, np.nan])
>>> demo
array([ 1.,  2.,  3., nan])

>>> demo.dtype
dtype('float64')
```

These are tradeoffs that many scientific programmers are willing to make. NumPy is a very powerful tool for working with large matrices of numbers. It is also very fast and efficient. If I needed to deal with large blobs of homogenous values, I would use NumPy.

However, in practice, I tend to work with tabular data. This is data that has a mixture of different types. For example, a table of people might have a name, age, and height. The name is a string, the age is an integer, and the height is a floating-point number. We could write pure Python code to deal with data like this, but it would be very slow. We could use NumPy, but it doesn't deal with heterogeneous data very well. It is meant for dealing with arrays or matrices of homogeneous data.

But pandas can still leverage NumPy to make it run fast. Pandas 1.x uses NumPy under the hood to store the numeric data. It also uses NumPy to do vectorized operations on the data. This makes Pandas very fast. Pandas 1.x supported string columns and date columns. However, string columns in Pandas are not very optimized, but they are more convenient than using NumPy or pure Python.

In this book, when I refer to Pandas 1.x, I'm referring to Pandas using NumPy as the backend to store data.

5.5 PyArrow and the Future

To overcome the annoyances and issues of NumPy in Pandas, for Pandas 2.0, the developers added a new backend, PyArrow. You can think of PyArrow as a next-generation NumPy. It is designed to work with tabular data and heterogeneous data. It also handles missing values in integers and has better support for strings. If you turn on the PyArrow backend, Pandas will be faster and more efficient. However, there might be some rough edges where PyArrow is missing features that NumPy has. I will try and point out these differences in the book.

PyArrow also allows us to create arrays and tables similar to NumPy. However, we are going to use the library indirectly through Pandas.

For the remainder of the chapter, I want to review the common types and show what they look like in pure Python, Pandas 1.x and Pandas 2.0. I will also show how to convert between the types.

5.6 Integer Types

Here are three Python lists of integers: one has small values, one has large values, and one has missing values:

```
>>> small_values = [1, 99, 127]
>>> large_values = [2**31, 2**63, 2**100]
>>> missing_values = [None, 1, -45]
```

In Pandas 1.x, we could use the `pd.Series` constructor to create a series of integers:

```
>>> import pandas as pd
>>> small_ser = pd.Series(small_values)
>>> small_ser
0      1
1     99
2    127
dtype: int64
```

Note that the type of this series is `int64`. This is a NumPy type. Pandas 1.x uses NumPy to store the data. Because these numbers don't exceed 127, we could use the `int8` type instead. This would save memory and make the code run faster. We can use the `astype` method to convert the series to `int8`:

```
>>> small_ser.astype('int8')
0      1
1     99
2    127
dtype: int8
```

We can also specify the type when we create the series:

```
>>> small_ser = pd.Series(small_values, dtype='int8')
>>> small_ser
0      1
1     99
2    127
dtype: int8
```

If you are using pandas 2, I recommend you use PyArrow types instead of NumPy types. PyArrow has better support for integers and missing values. However, you might still need to understand NumPy types because they are required for backward compatibility with some libraries or if some of the functionality you need has not yet been implemented in PyArrow.

Let's create the same series using PyArrow types:

```
>>> small_ser_pa = pd.Series(small_values, dtype='int8[pyarrow]')
>>> small_ser_pa
0      1
1     99
2    127
dtype: int8[pyarrow]
```

We add the string `[pyarrow]` to the dtype to indicate that we want to use the PyArrow extension type.

Let's convert the `large_values` list to both a NumPy and a PyArrow-backed series. Here's NumPy:

```
>>> large_ser = pd.Series(large_values)
>>> large_ser
0                          2147483648
1                 9223372036854775808
2    12676506002282294014967032053760
dtype: object
```

And here's PyArrow:

```
>>> large_ser = pd.Series(large_values, dtype='int64[pyarrow]')
Traceback (most recent call last):
    ...
OverflowError: Python int too large to convert to C long
```

Note that the NumPy-backed series has a `type` of `object`. Because the numbers are larger than an int64, NumPy will store the values but store them as Python objects. This will make the code run slower and use more memory.

The PyArrow backend doesn't gracefully fall back to Python objects. Instead, it will raise an error.

Now let's look at the `missing_values` list. We will again convert it to both a NumPy and PyArrow-backed series:

```
>>> missing_ser = pd.Series(missing_values)
>>> missing_ser
0     NaN
1     1.0
2   -45.0
dtype: float64
```

Note the type of the NumPy backed series is `float64`. This is because NumPy doesn't support missing values in integers. Instead, it converts the integers to floating-point numbers. This doesn't necessarily make the code run slower, but it does mean that you are doing floating-point operations instead of integer operations, and you should be aware of that.

In fact, in pandas 1.x, this comes in handy as you can infer from looking at an `int64` series that it has no missing values. We can't make such a strong assumption when we come across a `float64` series. There could be a floating point series with no missing values. But there could also be a floating point series with missing values. But there is also a third possibility. It could be an integer series with missing values. We can't tell the difference between these three cases by looking at the type. We need to look at the data itself.

Here's the PyArrow-backed series of the `missing_values` list:

```
>>> missing_ser_pa = pd.Series(missing_values, dtype='int8[pyarrow]')
>>> missing_ser_pa
0    <NA>
1       1
2     -45
dtype: int8[pyarrow]
```

In this case, PyArrow does support missing values in integers.

One more thing to note about type conversion. The NumPy series will let you convert to other sizes, but the value will overflow if it is too large:

```
>>> medium_values = [2**15+5, 2**31-8, 2**63]
>>> medium_ser = pd.Series(medium_values)
>>> medium_ser
0                  32773
1             2147483640
2    9223372036854775808
dtype: uint64

>>> medium_ser.astype('int8')
0    5
1   -8
2    0
dtype: int8
```

However, the PyArrow series will raise an error if the value is too large:

```
>>> medium_ser.astype('int8[pyarrow]')
Traceback (most recent call last):
    ...
pyarrow.lib.ArrowInvalid: Integer value 32773 not in range: 0 to 127
```

5.7 Floating Point Types

Here are some Python lists with floating-point data. One has missing values, one doesn't, and a final one has text values (like meteorological data with `'T'` for trace amounts of rain):

```
>>> float_vals = [1.5, 2.7, 127.0]
>>> float_missing = [None, 1.5, -45.0]
>>> float_rain = [1.5, 2.7, 0.0, 'T', 1.5, 0]
```

Let's convert the first two to Pandas 1.x series:

```
>>> pd.Series(float_vals)
0      1.5
1      2.7
2    127.0
dtype: float64

>>> pd.Series(float_missing)
0     NaN
1     1.5
2   -45.0
dtype: float64
```

There's nothing too surprising here. Both values are converted to `float64`. Let's convert the `missing_values` list to both a PyArrow and backed series:

```
>>> pd.Series(float_vals, dtype='float64[pyarrow]')
0      1.5
1      2.7
2    127.0
dtype: double[pyarrow]

>>> pd.Series(float_missing, dtype='float64[pyarrow]')
0    <NA>
1     1.5
2   -45.0
dtype: double[pyarrow]
```

A couple of things to note. The missing values in NumPy are converted to `NaN` values. PyArrow has a different missing value representation. It uses `<NA>` instead of `NaN`. However, in practice, most operations treat these two values the same, and you shouldn't worry about the difference. The type of the PyArrow series is `double[pyarrow]`, which represents a double precision floating point number (that uses 64 bits). You can also specify the type as `'double[pyarrow]'` when you create the series.

Let's look at the `float_rain` list. It has the string value `'T'` for trace amounts of rain. Let's try to convert it to a numeric series.

```
>>> pd.Series(float_rain)
0    1.5
1    2.7
2    0.0
```

```
3      T
4    1.5
5      0
dtype: object
```

If you casually look at the output, it looks like it worked. But if you look at the type, you'll see that it is `object`. This means that the values are stored as Python objects.

We must manually convert the `'T'` values to a numeric value. I'm not a meteorologist, but I've talked to one about this issue. The `'T'` stands for a *trace* amount of rain. That means it is less than the smallest amount that can be measured. His advice was to convert it to 0. Let's do this:

```
>>> pd.Series(float_rain).replace('T', '0.0').astype('float64')
0    1.5
1    2.7
2    0.0
3    0.0
4    1.5
5    0.0
dtype: float64
```

Let's do the same for PyArrow:

```
>>> pd.Series(float_rain).replace('T', 0).astype('float64[pyarrow]')
0    1.5
1    2.7
2    0.0
3    0.0
4    1.5
5    0.0
dtype: double[pyarrow]
```

I'm not sure how to go directly from a string to a PyArrow floating point value. I will pass it through NumPy first:

```
>>> (pd.Series(float_rain)
...  .replace('T', '0.0')
...  .astype('float')
...  .astype('float64[pyarrow]')
... )
0    1.5
1    2.7
2    0.0
3    0.0
4    1.5
5    0.0
dtype: double[pyarrow]
```

Both NumPy and PyArrow support 32-bit floating point numbers. When should you use 32-bit floating point numbers instead of 64-bit ones? You can use 32-bit floating point numbers when you need to save memory, and you don't need the extra precision. For example, if you are working with a large array of floating point numbers and don't need extra precision, you can save memory using 32-bit floating point numbers. Also, on some hardware, SIMD instructions can process twice as many 32-bit floating point numbers as 64-bit floating point numbers at a time. This can make your code run faster.

Another possibility is to convert the floating point numbers to integers to save even more memory. If you don't need the precision beyond whole numbers and all values are less than 127, you can convert the floating point numbers to 8-bit integers.

5.8 String Data

String columns are ubiquitous in tabular datasets. These string columns can hold different types of data, including free-form text, categorical data, dates, numbers represented as strings (or missing values represented as strings), and more. For columns with strings that hold dates or numbers, we generally want to convert them to a more appropriate type. We might want to do text analysis or natural language processing (NLP) on columns with free-form text. We will often convert columns with categorical data to a categorical type. In this section, we will focus on columns with free-form text.

Sadly, we can't just use `'string[pyarrow]'` as a type to get the new Pandas 2 pyarrow types. This is because this type was introduced back in Pandas 1.5 era and the operations on it will generally return legacy NumPy typed data. Rather we should import the `pyarrow` library and use the type: `pd.ArrowDtype(pyarrow.string())`. Since, I'm a lazy programming, this is slightly too much typing (and memorization for me. I will make an alias for the type and use the alias:

```
>>> import pyarrow as pa
>>> import pandas as pd
>>> string_pa = pd.ArrowDtype(pa.string())
```

Here is some text data to illustrate the differences:

```
>>> text_freeform = ['My name is Jeff', 'I like pandas',
...                  'I like programming']
>>> text_with_missing = ['My name is Jeff', None, 'I like programming']
```

Let's convert these to Pandas 1.x series and Pandas 2.0 series. First, the Pandas 1.x series:

```
>>> pd.Series(text_freeform)
0       My name is Jeff
```

```
1         I like pandas
2    I like programming
dtype: object
```

```
>>> pd.Series(text_with_missing)
0       My name is Jeff
1                  None
2    I like programming
dtype: object
```

This works, and we don't need to use `.astype(str)` to convert the values to strings. However, the type of the series is `object`. This is because the series is storing Python objects. Pandas 1.x stores the `str` type as Python objects. This is because NumPy doesn't support strings.

Let's use the new Pandas 2.0 string type:

```
>>> tf1 = pd.Series(text_freeform, dtype=string_pa)
>>> tf1
0       My name is Jeff
1         I like pandas
2    I like programming
dtype: string[pyarrow]
```

Note, I will use the `'string[pyarrow]'` type and you will see that the type is different.

```
>>> tf2 = pd.Series(text_freeform, dtype='string[pyarrow]')
>>> tf2
0       My name is Jeff
1         I like pandas
2    I like programming
dtype: string
```

Let's compare the types:

```
>>> tf1.dtype == tf2.dtype
False
```

Let's convert the text with missing data:

```
>>> pd.Series(text_with_missing, dtype=string_pa)
0       My name is Jeff
1                  <NA>
2    I like programming
dtype: string[pyarrow]
```

Notice that the type of the series is `string[pyarrow]`. This is a big improvement over Pandas 1.x. It uses less memory, and it is faster than Pandas 1.x.

5.9 Categorical Data

Categorical data is string data that has a *low cardinality*. This means that there are a small number of unique values. For example, a column that holds the names of the 50 states in the United States would be categorical data. There are only 50 unique values. A column that contains the names of all the people in the United States would not be categorical data. There are over 300 million people in the United States, and while there are some repeated names, there are still a lot of unique names.

Pandas stores categorical data in a particular way. It keeps the unique values in a separate array and then stores the values as integers that refer to the individual values. This can save memory and can make operations faster. There is a crossover point where storing the data as categorical data instead of strings is more efficient. This crossover point depends on the number of unique values and the length of the series.

Additionally, categorical data can be ordered or unordered. For example, the names of the 50 states in the United States would be unordered. The names of the months of the year would be ordered. Pandas supports both ordered and unordered categorical data.

```
>>> states = ['CA', 'NY', 'TX']
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
...           'Sep', 'Oct', 'Nov', 'Dec']
```

NumPy doesn't support categorical data natively, but because it is so common, pandas 1.x added support for categorical data. Let's convert a list of strings to a categorical series:

```
>>> pd.Series(states, dtype='category')
0    CA
1    NY
2    TX
dtype: category
Categories (3, object): ['CA', 'NY', 'TX']
```

Let's make an ordered categorical series from the months of the year:

```
>>> pd.Series(months, dtype='category')
0     Jan
1     Feb
2     Mar
3     Apr
4     May
5     Jun
6     Jul
7     Aug
8     Sep
```

```
9     Oct
10    Nov
11    Dec
dtype: category
Categories (12, object): ['Apr', 'Aug', 'Dec', 'Feb', ..., 'May',
     'Nov', 'Oct', 'Sep']
```

This is not ordered. If we sort the series, it will sort alphabetically:

```
>>> pd.Series(months, dtype='category').sort_values()
3     Apr
7     Aug
11    Dec
1     Feb
0     Jan
6     Jul
5     Jun
2     Mar
4     May
10    Nov
9     Oct
8     Sep
dtype: category
Categories (12, object): ['Apr', 'Aug', 'Dec', 'Feb', ..., 'May',
     'Nov', 'Oct', 'Sep']
```

To sort the data, we need to first create an ordered categorical type and pass that type in as the `dtype` parameter:

```
>>> month_cat = pd.CategoricalDtype(categories=months, ordered=True)
>>> pd.Series(months, dtype=month_cat).sort_values()
0     Jan
1     Feb
2     Mar
3     Apr
4     May
5     Jun
6     Jul
7     Aug
8     Sep
9     Oct
10    Nov
11    Dec
dtype: category
Categories (12, object): ['Jan' < 'Feb' < 'Mar' < 'Apr' ... 'Sep' <
     'Oct' < 'Nov' < 'Dec']
```

PyArrow doesn't have "categorical" type, but it does have a "dictionary" type. However, the dictionary type isn't exposed directly in pandas 2. Instead, if you want to use a categorical type, I suggest you use the Pandas 1.x categorical type.

```
>>> pd.Series(months, dtype=string_pa).astype(month_cat)
0     Jan
1     Feb
2     Mar
3     Apr
4     May
5     Jun
6     Jul
7     Aug
8     Sep
9     Oct
10    Nov
11    Dec
dtype: category
Categories (12, object): ['Jan' < 'Feb' < 'Mar' < 'Apr' ... 'Sep' <
     'Oct' < 'Nov' < 'Dec']
```

Here is an example of using the PyArrow dictionary type. In my year teaching and using pandas 2, I haven't needed to use this type. In fact, I can't even find a reference to it in the pandas documentation.

```
>>> pd.Series(months,
...   dtype=pd.ArrowDtype(pa.dictionary(pa.int64(), pa.string())))

0     Jan
1     Feb
2     Mar
3     Apr
4     May
5     Jun
6     Jul
7     Aug
8     Sep
9     Oct
10    Nov
11    Dec
dtype: dictionary<values=string, indices=int64, ordered=0>[pyarrow]
```

5.10 Dates and Times

Pandas makes it convenient to work with dates and times. It has several data types that can be used to represent dates and times. However, we generally

want to use `datetime64[ns]` for Pandas 1.x and `'timestamp[ns][pyarrow]'` for Pandas 2.0.

Let's create a few lists of dates represented as Python lists. One will have `datetime` objects, one will have strings, and one will have seconds since the epoch:

```
>>> import datetime as dt
>>> dt_list = [dt.datetime(2020, 1, 1, 4, 30 ), dt.datetime(2020, 1, 2),
...            dt.datetime(2020, 1, 3)]
>>> string_dates = ['2020-01-01 04:30:00', '2020-01-02 00:00:00',
...                 '2020-01-03 00:00:00']
>>> string_dates_missing = ['2020-01-01 4:30', None, '2020-01-03']
>>> epoch_dates = [1577836800, 1577923200, 1578009600]
```

Let's try and convert these to Pandas 1.x series. Most of these convert with little issue:

```
>>> pd.Series(dt_list)
0   2020-01-01 04:30:00
1   2020-01-02 00:00:00
2   2020-01-03 00:00:00
dtype: datetime64[ns]

>>> pd.Series(string_dates, dtype='datetime64[ns]')
0   2020-01-01 04:30:00
1   2020-01-02 00:00:00
2   2020-01-03 00:00:00
dtype: datetime64[ns]

>>> pd.Series(string_dates_missing, dtype='datetime64[ns]')
0   2020-01-01 04:30:00
1                   NaT
2   2020-01-03 00:00:00
dtype: datetime64[ns]
```

Let's convert the seconds since 1970 to dates. If we use `datetime64[ns]`, it appears we get results, but upon closer inspection, these are dates from the 1970s. This usually means that our granularity is off.

```
>>> # using the 'ns' for nanoseconds gives us erroneous results
>>> pd.Series(epoch_dates, dtype='datetime64[ns]')
0   1970-01-01 00:00:01.577836800
1   1970-01-01 00:00:01.577923200
2   1970-01-01 00:00:01.578009600
dtype: datetime64[ns]
```

Let's use `datetime64[s]` to convert from seconds (instead of nanoseconds):

```
>>> # note the 's' for seconds
>>> pd.Series(epoch_dates, dtype='datetime64[s]')
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[s]
```

Let's convert these to Pandas 2.0 series:

```
>>> pd.Series(dt_list, dtype='timestamp[ns][pyarrow]')
0    2020-01-01 04:30:00
1    2020-01-02 00:00:00
2    2020-01-03 00:00:00
dtype: timestamp[ns][pyarrow]

>>> pd.Series(string_dates, dtype='timestamp[ns][pyarrow]')
0    2020-01-01 04:30:00
1    2020-01-02 00:00:00
2    2020-01-03 00:00:00
dtype: timestamp[ns][pyarrow]
```

We also want to be sure that we specify that the epoch conversion is using seconds instead of nanoseconds.

```
>>> # make sure to use the [s] for seconds
>>> pd.Series(epoch_dates).astype('timestamp[s][pyarrow]')
0    2020-01-01 00:00:00
1    2020-01-02 00:00:00
2    2020-01-03 00:00:00
dtype: timestamp[s][pyarrow]
```

5.11 Summary

This chapter looked at the main datatypes you will run across in tabular datasets. These are numbers, booleans, strings, and dates. There are some differences between pandas 1.x and pandas 2.0. In general, we will want to use the new pandas 2.0 types. They are faster and use less memory. However, pandas 2.0 doesn't support all of the operations that pandas 1.x supports, so we will need to use pandas 1.x types in some cases.

5.12 Exercises

1. What type would you use to represent the number of people in the United States? What type would you use to describe the number of people worldwide?

2. What type would you use to describe a product? What type would you use to represent the name of a product? What type would you use to represent the price of a product?

3. What type would you use to represent the date and time of a stock trade? What type would you use to represent the date of birth of a person?

Chapter 6

Series Introduction

This chapter introduces the `Series` object, the first of the two core pandas objects. The `Series` is a one-dimensional array-like object that is used to model a single column or row of data. The `Series` object is the building block of the `DataFrame` object, which is the second core pandas object. We need to understand the `Series` object so that we can effectively manipulate columns of data.

6.1 A Simple Series Object

A `Series` is used to model one-dimensional data. The `Series` object also has a few more bits of data, including an index and a name. A common idea through pandas is the notion of an axis. Because a series is one-dimensional, it has a single *axis*—the index.

Below is a table of the number of songs artists have composed. We will use this to explore the series:

Table 6.1: Count of songs by each artist

Artist	Data
0	145
1	142
2	38
3	13

If you wanted to represent this data in pure Python, you could use a data structure similar to the following one. The dictionary, `series`, has a list of the data points stored under the `'data'` key. In addition to an entry in the dictionary for the actual data, there is an explicit entry for the corresponding index values for the data (in the `'index'` key), as well as an entry for the name of the data (in the `'name'` key):

```
>>> series = {
...   'index':[0, 1, 2, 3],
...   'data':[145, 142, 38, 13],
...   'name':'songs'
...   }
```

The get function defined below can pull items out of this data structure based on the index:

```
>>> def get(series, idx):
...     value_idx = series['index'].index(idx)
...     return series['data'][value_idx]

>>> get(series, 1)
142
```

Note

The code samples in this book are shown as if they were typed directly into an interpreter. Lines starting with `>>>` and `...` are interpreter markers for the *input prompt* and *continuation prompt*, respectively. Lines not prefixed by one of those sequences are the output from the interpreter after running the code.

In Jupyter (and IPython), you do not see the prompts. I include them to help distinguish between code and output.

The Python interpreter will automatically print the last invocation's return value (even if the `print` statement is missing). If you want to use the code samples in this book, leave the interpreter prompts out.

6.2 The Index Abstraction

This double abstraction of the index seems unnecessary at first glance—a list already has integer indexes. But there is a trick up pandas' sleeves. By allowing non-integer values, the data structure supports other index types, such as strings and dates, arbitrarily ordered indices, or even duplicate index values.

Below is an example that has string values for the index:

```
>>> songs = {
...   'index':['Paul', 'John', 'George', 'Ringo'],
...   'data':[145, 142, 38, 13],
...   'name':'counts'
...   }

>>> get(songs, 'John')
142
```

The index is a core feature of pandas' data structures, given the library's past in analysis of financial data or *time-series data*. Many operations performed on a `Series` operate directly on the index or by index lookup.

6.3 The pandas Series

With that background in mind, let's create a `Series` in pandas. It is easy to create a `Series` object from a list:

```
>>> import pandas as pd
>>> songs2 = pd.Series([145, 142, 38, 13],
...       name='counts')

>>> songs2
0    145
1    142
2     38
3     13
Name: counts, dtype: int64
```

Pandas 2 introduced a new backend, the pyarrow backend. By default, pandas still uses the NumPy backend, which is the `int64` type. To use pyarrow we need to specify the type, `'int64[pyarrow]'`.

```
>>> songs3 = pd.Series([145, 142, 38, 13],
...       name='counts', dtype='int64[pyarrow]')

>>> songs3
0    145
1    142
2     38
3     13
Name: counts, dtype: int64[pyarrow]
```

You will want to use the pyarrow backend as it is more efficient for both computation and memory usage. The pyarrow backend allows for missing values, a native string type, speed enhancements, and memory optimizations. We will discuss these benefits more throughout this book.

When the interpreter prints our series, pandas makes the best effort to format it for the current terminal size. The series is one-dimensional. However, this looks like it is two-dimensional. The leftmost column is the *index*, which contains entries for the index. The index is not part of the values. The generic name for an index is an *axis*, and the index values—0, 1, 2, 3—are called *axis labels*. The data—145, 142, 38, and 13—is also called the *values* of the series. The two-dimensional structure in pandas—a `DataFrame`—has two axes, one for the rows and another for the columns.

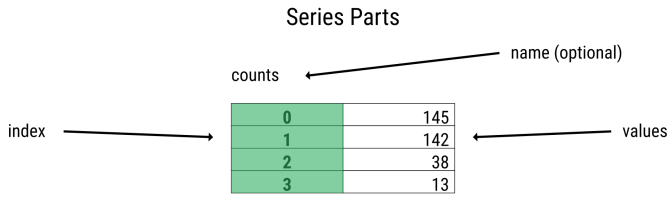


Figure 6.1: The parts of a Series.

The rightmost column in the output contains the *values* of the series—145, 142, 38, and 13. In this case, they are integers (the console representation says `dtype: int64[pyarrow]`, `dtype` meaning data type, and `int64[pyarrow]` meaning 64-bit integer stored in a pyarrow datastructure), but in general, the values of a `Series` can hold strings, floats, booleans, or arbitrary Python objects. The values should be the same type to get the best speed (and leverage vectorized operations), though this is not required.

It is easy to inspect the index of a series (or data frame) as it is an attribute of the object:

```
>>> songs2.index
RangeIndex(start=0, stop=4, step=1)
```

The default values for an index are monotonically increasing integers. `songs2` has an integer-based index.

Note

The index can be string-based as well, in which case pandas indicates that the datatype for the index is `object` (not string):

```
>>> songs3 = pd.Series([145, 142, 38, 13],
...       name='counts',
...       index=['Paul', 'John', 'George', 'Ringo'],
...       dtype='int64[pyarrow]')
```

Note that the `dtype` we see when printing a `Series` is the type of the values, not the index. Even though this looks two-dimensional, remember that the index is not part of the values:

```
>>> songs3
Paul      145
John      142
```

```
George     38
Ringo      13
Name: counts, dtype: int64[pyarrow]
```

When we inspect the index attribute, we see that the `dtype` is `object`:

```
>>> songs3.index
Index(['Paul', 'John', 'George', 'Ringo'], dtype='object')
```

A series's actual data (or values) does not have to be numeric or homogeneous. We can insert Python objects into a series:

```
>>> class Foo:
...     pass

>>> ringo = pd.Series(
...      ['Richard', 'Starkey', 13, Foo()],
...      name='ringo')

>>> ringo
0                                 Richard
1                                 Starkey
2                                      13
3    <__main__.Foo object at 0x11fa975d0>
Name: ringo, dtype: object
```

In the above case, the `dtype`-*datatype*-of the `Series` is `object` (meaning a Python object). This can be good or bad.

In pandas 2, the `object` data type is used for types not natively supported by the pyarrow backend. It is also used for values that have heterogeneous or mixed types. If you have just numeric data in a series, you wouldn't want it stored as a Python object but as an `int64[pyarrow]` or `float64[pyarrow]`, allowing you to do vectorized numeric operations.

If you have time data and it says it has the `object` type, you probably have strings for the dates. Using strings instead of date types is bad as you don't get the date operations you would get if the type were `datetime64[ns]`. On the other hand, a series with string data has the type of `object`. Don't worry; we will see how to convert types later in the book.

6.4 The NA value

A value that may be familiar to NumPy users but not Python users, generally, is `<NA>`. When pandas determines that a series holds numeric values but cannot find a number to represent an entry, it will use `<NA>` when using pyarrow types. This value stands for *Not A Number* and is usually ignored in arithmetic operations. (Similar to NULL in SQL).

If you are using the NumPy backend, you will see `NaN` instead of `<NA>`. The NumPy backend only supports missing values for floating point types.

Here is a series that has `NaN` in it because it is using the NumPy backend:

```
>>> import numpy as np
>>> nan_series = pd.Series([2, np.nan],
...     index=['Ono', 'Clapton'])
>>> nan_series
Ono        2.0
Clapton    NaN
dtype: float64
```

Note

One thing to note is that the type of this series is `float64`, not `int64`! The type is a float because `float64` supports `NaN` while `int64` does not. When pandas sees the numeric data (`2`) and the `np.nan`, it coerces the `2` to a float value.

If we create this same series using the pyarrow backend, the integer type is preserved.

```
>>> import numpy as np
>>> nan_series2 = pd.Series([2, np.nan],
...     index=['Ono', 'Clapton'], dtype='int64[pyarrow]')
>>> nan_series2
Ono           2
Clapton    <NA>
dtype: int64[pyarrow]
```

Below is an example of how pandas ignores `<NA>`. The `.count` method, which counts the number of values in a series, disregards `<NA>`. In this case, it indicates that the count of items in the series is one, one for the value of `2` at index location `Ono`, ignoring the `<NA>` value at index location `Clapton`:

```
>>> nan_series2.count()
1
```

You can inspect the number of entries (including missing values) with the `.size` property. The `.size` property returns the number of entries in the series, including missing values:

```
>>> nan_series2.size
2
```

Note

If you load data from a CSV file, an empty value for an otherwise numeric column will become `<NA>`. Later, methods such as `.fillna` and `.dropna` will explain how to deal with `<NA>`. With pyarrow, empty values for string columns become the empty string, `''`.

`None`, `NaN`, `nan`, `<NA>`, and null are synonyms in this book when referring to empty or missing data found in a pandas series or dataframe.

6.5 Similar to NumPy

If you are familiar with NumPy, you will find that the `Series` object behaves similarly to a NumPy array. As shown below, both types respond to index operations. However, because pandas supports indexing by name and position, we want to be specific and indicate that we are indexing by position by using `.iloc`:

```
>>> import numpy as np
>>> numpy_ser = np.array([145, 142, 38, 13])
>>> songs3.iloc[1]
142

>>> numpy_ser[1]
142
```

They both have methods in common:

```
>>> songs3.mean()
84.5

>>> numpy_ser.mean()
84.5
```

We can use set operations to determine the methods that are common to both types:

```
>>> len(set(dir(numpy_ser)) & set(dir(songs3)))
112
```

They also both have a notion of a *boolean array*. A boolean array is a series with the same index as the series you are working with that has boolean values, and it can be used as a mask to filter out items. Normal Python lists do not support such fancy index operations as sticking a list into an index operation.

In this example, we will make a mask:

```
>>> mask = songs3 > songs3.median()  # boolean array
```

```
>>> mask
Paul       True
John       True
George    False
Ringo     False
Name: counts, dtype: bool[pyarrow]
```

Once we have a mask, we can use that as a filter. We just need to pass the mask into an index operation. The value is kept if the mask has a `True` value for a given index. Otherwise, the value is dropped. The mask above represents the locations with a value higher than the median value of the series.

```
>>> songs3[mask]
Paul    145
John    142
Name: counts, dtype: int64[pyarrow]
```

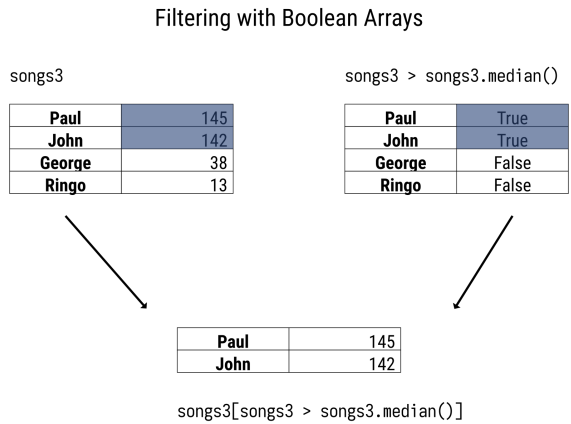


Figure 6.2: Filtering a series with a boolean array.

NumPy also supports filtering with boolean arrays but lacks the `.median` method on an array. Instead, NumPy provides a `median` function in the NumPy namespace. The equivalent version in NumPy looks like this:

```
>>> numpy_ser[numpy_ser > np.median(numpy_ser)]
array([145, 142])
```

Note

NumPy and pandas have adopted the convention of using import statements combined with an `as` statement to rename their imports to two-letter acronyms. This is called *aliasing*:

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

Renaming imports provides a slight typing benefit (four fewer characters) while still allowing the user to be explicit with their namespaces.

Be careful, as you may see the following cast about in code samples, blogs, or documentation:

```
>>> from pandas import *
```

Though you see *star imports* frequently used in examples online, I would advise not to use star imports. I never use them in my book examples or the code I write for clients. They can clobber items in your namespace and make tracing the source of a definition more difficult (especially if you have multiple star imports). As the Zen of Python states, "Explicit is better than implicit"[a].

[a] Type `import this` into an interpreter to see the Zen of Python. Or search for "PEP 20".

6.6 Categorical Data

When you load data, you can indicate that the data is categorical. If we know that our data is limited to a few values; we might want to use categorical data. Categorical values have a few benefits:

- Use less memory than strings
- Improve performance
- Can have an ordering
- Can perform operations on categories
- Enforce membership on values

Categories are not limited to strings; we can also convert numbers or datetime values to categorical data.

To create a category, we pass `dtype="category"` into the `Series` constructor. Alternatively, we can call the `.astype("category")` method on a series:

```
>>> s = pd.Series(['s', 'm', 'l'], dtype='category')
>>> s
0    s
1    m
```

```
2    l
dtype: category
Categories (3, object): ['l', 'm', 's']
```

Note

Pyarrow has a native category type, called a `dictionary`, but there is no convenient way to create one. I recommend you use the pandas `'category'` type.

Here is an example of creating a pyarrow category type:

```
>>> import pyarrow as pa
>>> dict_type = pd.ArrowDtype(pa.dictionary(pa.int64(), pa.utf8()))
>>> s = pd.Series(['m', 'l', 'xs', 's', 'xl'], dtype=dict_type)
>>> s
0     m
1     l
2    xs
3     s
4    xl
dtype: dictionary<values=string, indices=int64, ordered=0>[pyarrow]
```

Also, note that if you save a Feather file with a categorical column, you will get the `dictionary` type when you read it back in:

```
>>> (pd.Series(['sm', 'm', 'l'], dtype='category')
...   .rename('size')
...   .to_frame()
...   .to_feather('/tmp/cat.ft')
... )
>>> (pd.read_feather('/tmp/cat.ft', dtype_backend='pyarrow')
...  .loc[:, 'size']
...  .dtype
... )
dictionary<values=string, indices=int8, ordered=0>[pyarrow]
```

If this series represents the size, there is a natural ordering as a small is less than a medium. By default, categories don't have an ordering. We can verify this by inspecting the `.cat` attribute that has various properties:

```
>>> s.cat.ordered
False
```

We can create a type with the `CategoricalDtype` constructor and the appropriate parameters to convert a non-categorical series to an ordered category. Then we pass this type into the `.astype` method:

```
>>> s2 = pd.Series(['m', 'l', 'xs', 's', 'xl'], dtype='string[pyarrow]')
>>> size_type = pd.CategoricalDtype(
...     categories=['s','m','l'], ordered=True)
>>> s3 = s2.astype(size_type)
...
>>> s3
0      m
1      l
2    NaN
3      s
4    NaN
dtype: category
Categories (3, object): ['s' < 'm' < 'l']
```

In this case, we limited the categories to `'s'`, `'m'`, and `'l'`, but the data had values that were not in those categories. Converting the data to a category type replaces those extra values with `NaN`.

If we have ordered categories, we can make comparisons on them:

```
>>> s3 > 's'
0     True
1     True
2    False
3    False
4    False
dtype: bool
```

The prior example created a new `Series` from existing data that was not categorical. We can also add ordering information to categorical data. We need to make sure that we specify all of the members of the category, or pandas will throw a `ValueError`:

```
>>> s = pd.Series(['s', 'm', 'l'], dtype='category')
>>> s.cat.reorder_categories(['xs', 's', 'm', 'l', 'xl'], ordered=True)
Traceback (most recent call last):
  ...
ValueError: items in new_categories are not the same as in old categories
```

This error is because we are adding ordering to new categories. We can list the current categories with the `.cat.categories` attribute:

```
>>> s.cat.categories
Index(['l', 'm', 's'], dtype='object')
```

We need to make sure that the category is aware of all of the valid values. We can do this by calling the `.add_categories` method before calling `.reorder_categories`:

```
>>> (s
...  .cat.add_categories(['xs', 'xl', ])
...  .cat.reorder_categories(['xs','s','m','l', 'xl'],
...                          ordered=True)
... )
0    s
1    m
2    l
dtype: category
Categories (5, object): ['xs' < 's' < 'm' < 'l' < 'xl']
```

Note

String and datetime series have an `str` and a `dt` attribute. These attributes allow us to perform common operations specific to that type. If we convert these types to categorical types, we can still use the `str` or `dt` attributes on them:

```
>>> s3.str.upper()
0      M
1      L
2    NaN
3      S
4    NaN
dtype: object
```

Table 6.2: Table of functionality explored in this chapter.

Method	Description
`pd.Series(data=None, index=None, dtype=None, name=None, copy=False)`	Create a series from `data` (sequence, dictionary, or scalar).
`s.index`	Access index of series.
`s.astype(dtype, errors='raise')`	Cast a series to `dtype`. To ignore errors (and return the original object), use `errors='ignore'`.
`s[boolean_array]`	Return values from `s` where `boolean_array` is `True`.
`s.cat.ordered`	Determine if a categorical series is ordered.
`s.cat.reorder_categories( new_categories, ordered=False)`	Add categories (potentially ordered) to the series. `new_categories` must include all categories.
`s.cat.categories`	Return the categories of a categorical series.

Method	Description
`s.cat.add_categories(new_categories)`	Add new categories to a categorical series.

6.7 Summary

The `Series` object is a one-dimensional data structure. It can hold numerical data, time data, strings, or arbitrary Python objects. Using pandas rather than a Python list will benefit you if you are dealing with numeric data. Pandas is faster, consumes less memory, and comes with built-in methods that are very useful for manipulating the data. The pyarrow types in pandas 2 make these even faster and more memory efficient. Also, the index abstraction allows for accessing values by position or label. A `Series` can also have empty values and has some similarities to NumPy arrays. It is the primary workhorse of pandas; mastering it will pay dividends.

6.8 Exercises

1. Using Jupyter, create a series with the temperature values for the last seven days. Filter out the values below the mean.
2. Using Jupyter, create a series with your favorite colors. Use a categorical type.

Chapter 7

Series Deep Dive

Once you are familiar with a `Series`, you will be able to use a `DataFrame` with ease. The `Series` is one of the two core data structures in pandas (the `DataFrame` is the other). There are many operations you can do with a `Series`. In this chapter, we will introduce many of them.

We will pull data from the US Fuel Economy website[1]. This site has data on the efficiency of makes and models of cars sold in the US since 1984.

7.1 Loading the Data

I have a copy of this data in my GitHub repository. One of the nice features of pandas is that the `read_csv` function can accept not only URLs but also ZIP files. We can use this function because this ZIP file contains only a single file. If it were a ZIP file with multiple files, we would need to decompress the data to pull out the file we were interested in.

I specified the `dtype_backend` parameter so that pandas uses pyarrow types instead of NumPy types.

I use the `engine` backend so that pandas uses the pyarrow library to parse the CSV. This library tends to read files more quickly than the standard pandas implementation.

The first columns in the dataset we will investigate are *city08* and *highway08*, which provide information on miles per gallon usage while driving around in the city and highway, respectively:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/' \
...       'vehicles.csv.zip'
>>> df = pd.read_csv(url, dtype_backend='pyarrow',
...                  engine='pyarrow')
>>> city_mpg = df.city08
>>> highway_mpg = df.highway08
```

[1] https://www.fueleconomy.gov/feg/download.shtml

Let's look at the data:

```
>>> city_mpg
0        19
1         9
2        23
         ..
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int64[pyarrow]

>>> highway_mpg
0        25
1        14
2        33
         ..
41141    24
41142    24
41143    21
Name: highway08, Length: 41144, dtype: int64[pyarrow]
```

It looks like each series has around 40,000 integer entries.

7.2 Series Attributes

The pandas library provides a lot of functionality. The built-in `dir` function will list the attributes of an object. Let's examine how many attributes there are on a series:

```
>>> len(dir(city_mpg))
457
```

Wow! There are over 400 attributes in a series. In contrast, a Python list or dictionary has around 40 attributes. Do not fret; you will not need to memorize all of these if you get comfortable with a tool like Jupyter. If you have a `Series` object, you can hit TAB after a period, and a list of completions will pop up. (Other tools are also able to do this for Python objects).

What functionality do all of these attributes provide? Here is a summary. There are many ways to categorize these, and I'm roughly going to do it by what the result of the method is:

- Dunder methods (`.__add__`, `.__iter__`, etc) provide many numeric operations, looping, attribute access, and index access. For the numeric operations, these return `Series`.

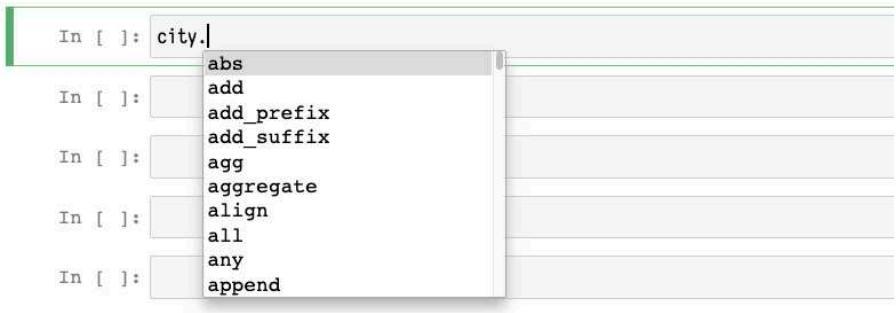


Figure 7.1: Jupyter will pop up a list of options for completions when you hit TAB following a period.

- Corresponding operator methods for many of the numeric operations allow us to tweak the behavior (there is an `.add` method in addition to `.__add__`).
- Aggregate methods and properties that reduce or aggregate the values in a series down to a single scalar value. The `.mean`, `.max`, and `.sum` methods, and `.is_monotonic` property are all examples.
- Conversion methods. Some of these start with `.to_` and export the data to other formats.
- Manipulation methods such as `.sort_values`, `.drop_duplicates`, that return `Series` objects with the same index.
- Indexing and accessor methods and attributes such as `.loc` and `.iloc`. These return `Series` or scalars.
- String manipulation methods using `.str`.
- Date manipulation methods using `.dt`.
- Plotting methods using `.plot`.
- Categorical manipulation methods using `.cat`.
- Transformation methods such as `.unstack` and `.reset_index`, `.agg`, `.transform`.
- Attributes such as `.index` and `.dtype`.
- A bunch of *private* attributes that we will ignore (around 130 of them).

We will cover many of these in the following chapters. Remember that memorizing all 400+ attributes is not necessary. You are not training to be a pandas encyclopedia. I will show you the most useful ones.

7.3 Summary

This chapter introduced the notion that pandas objects have many attributes and methods. Do not let this overwhelm you. You don't need to memorize all of the methods.

7.4 Exercises

1. Explore the documentation for five attributes of a series from Jupyter.
2. How many attributes are found on the `.str` attribute? Look at the documentation for three of them.
3. How many attributes are found on the `.dt` attribute? Look at the documentation for three of them.

Chapter 8

Operators (& Dunder Methods)

8.1 Introduction

This chapter will review operators, also called magic or *dunder methods* found in the series. These are not named after the popular TV show *The Office*, but instead is a shortcut for saying "double underscore." These dunder methods are the sorcery that happens under the covers when you add two numbers with a plus sign.

In short, these are the protocols that determine how the Python language reacts to operations. For example, when you use the `+` operation, Python is dispatching to the `.__add__` method. When you use a loop with a `for` statement, Python dispatches to the `.__iter__` method.

This will not be a deep treatise on the dunder methods (double underscore methods) or magic methods.

Let's examine how Python's hidden mechanics work with a pandas series.

8.2 Dunder Methods

Here is an example of pure Python. When you run this code:

```
>>> 2 + 4
6
```

Under the covers, Python runs this:

```
>>> (2).__add__(4)
6
```

In Python, when you use the `+` operator with integers, Python internally calls the `.__add__` method. This is called a *dunder method* because it starts with a double underscore. Because a `Series` object has the `.__add__` method, you can call `+` on it. There is also a `.__div__` method that supports division. One way to calculate the average of the two series is the following:

```
>>> (city_mpg + highway_mpg)/2
0        22.0
1        11.5
2        28.0
         ...
41141    21.0
41142    21.0
41143    18.5
Length: 41144, dtype: double[pyarrow]
```

Note that the type of the result is `double[pyarrow]`.

8.3 Index Alignment

Let's align our understanding of index alignment. Pandas ensures that the index of the two series is aligned before performing math operations. Of note, you can apply most math operations on a series with another series in addition (no pun intended) to using a scalar (as we did with the division). When you operate with two series, pandas will *align* the index before performing the operation. Aligning will take each index entry in the left series and match it up with every entry with the same name in the index of the right series. In the above case, values with the same index name are added together and then divided by 2. These operations return a `Series` object.

Because of index alignment, you will want to make sure that the indexes:

- Are unique (no duplicates)
- Are common to both series

If these situations do not exist, you will get missing values or a combinatoric explosion of results. Here is a simple example of two series that have repeated index entries as well as non-common entries:

```
>>> s1 = pd.Series([10, 20, 30], index=[1,2,2])
>>> s2 = pd.Series([35, 44, 53], index=[2,2,4], name='s2')
>>> s1
1    10
2    20
2    30
dtype: int64

>>> s2
2    35
2    44
4    53
Name: s2, dtype: int64
```

```
>>> s1 + s2
1     NaN
2    55.0
2    64.0
2    65.0
2    74.0
4     NaN
dtype: float64
```

Note that index names 1 and 4 have `NaN` while index name 2 has four results—every 2 from `s1` is matched up with every 2 from `s2`.

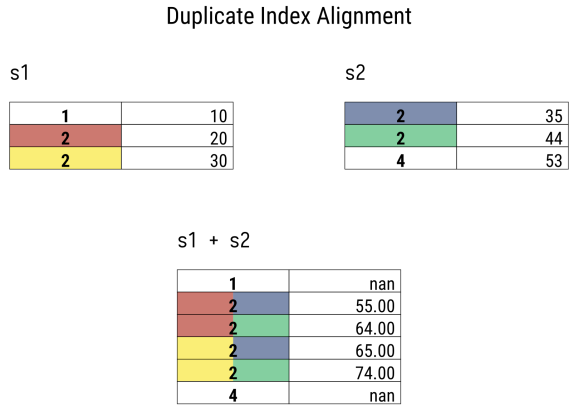


Figure 8.1: The index entries align before operating. If they are not unique, you will get a combinatoric explosion of index entries. Notice that each 2 name from `s1` matches each 2 name from the index in `s2`.

Remember that the index entries align before performing math operations.

8.4 Broadcasting

When you perform math operations with a scalar, pandas *broadcasts* the operation to all values.

If we add 5 to the series, pandas will add 5 to every value in the series.

```
>>> s2 + 5
2    40
2    49
4    58
Name: s2, dtype: int64
```

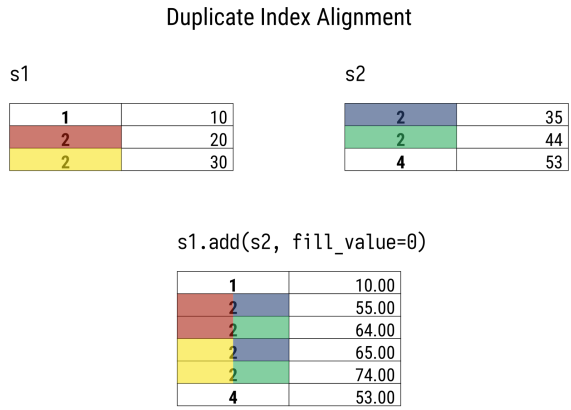


Figure 8.2: One upside to the operation methods like `.add` is that you can specify a fill value. The index entries will still align before performing the operation.

This makes it easy to write mathematical operations. It also makes the code easy to read.

There is another advantage to broadcasting. With many math operations, these are optimized and happen very quickly in the CPU. This is called *vectorization*. (A numeric pandas series is a block of memory, and modern CPUs leverage a technology called Single Instruction/Multiple Data (SIMD) to apply a math operation to the block of memory.)

Operations that are available include: `+`, `-`, `/`, `//` (floor division), `%` (modulus), `@` (matrix multiplication), `**` (power), `<`, `<=`, `==`, `!=`, `>=`, `>`, `&` (binary and), `^` (binary xor), `|` (binary or).

8.5 Iteration

Note that there is also a `.__iter__` method on a series; you can loop over the items in a series. This is like taking the scenic route through the series. You can use a `for` loop to iterate over the items.

I recommend avoiding using a `for` loop with a series. That is a *code smell*, indicating that you are probably doing things incorrectly. You are removing one of the benefits of pandas—vectorization and operating at the C level. If you use a loop to search or filter for values, we will see that there are other ways to do that that are usually faster and make the code easier to understand.

8.6 Operator Methods

You might wonder why pandas also provides methods for the standard operators. Sometimes, you want to dive below the surface and control an operation. In general, functions and methods have parameters to allow you to *parameterize* or change the behavior based on the parameters. The dunder methods generally fill in `NaN` (or `<NA>` for `Int64`) when one of the operands is missing following index alignment. The operator methods have a `fill_value` parameter that changes this behavior. If one of the operands is missing, it will use the `fill_value` instead.

If we call the `.add` method with the default parameters, we will have the same result as the `+` operator:

```
>>> s1 + s2
1     NaN
2    55.0
2    64.0
2    65.0
2    74.0
4     NaN
dtype: float64

>>> s1.add(s2)
1     NaN
2    55.0
2    64.0
2    65.0
2    74.0
4     NaN
dtype: float64
```

However, we can use the `fill_value` parameter to specify that we use zero instead:

```
>>> s1.add(s2, fill_value=0)
1    10.0
2    55.0
2    64.0
2    65.0
2    74.0
4    53.0
dtype: float64
```

Operator methods in pandas give you the control knob to fine-tune your data operations, offering flexibility beyond the standard operators.

8.7 Chaining

Another stylistic reason to prefer the method to the operator is that it makes *chaining* manipulations easier. Because most pandas methods do not mutate data in place but instead return a new object, we can keep tacking on method calls to the returned object. We will see many examples of this throughout the book. Chaining makes the code easy to read and understand. We can chain with operators as well, but we must wrap the operation with parentheses.

Below, we calculate the average city and highway mileage using operators:

```
>>> ((city_mpg +
...   highway_mpg)
...   / 2
... )
0        22.0
1        11.5
2        28.0
         ...
41141    21.0
41142    21.0
41143    18.5
Length: 41144, dtype: double[pyarrow]
```

Here is an example of chaining to calculate the average of city and highway mileage:

```
>>> (city_mpg
...   .add(highway_mpg)
...   .div(2)
... )
0        22.0
1        11.5
2        28.0
         ...
41141    21.0
41142    21.0
41143    18.5
Length: 41144, dtype: double[pyarrow]
```

This is a simple example, but I like how chaining can lead to understanding your code. I want to put these operations in their own line. I read this as, "we are taking the *city_mpg* series, then adding the *highway_mpg* series to it. Finally, we are dividing by two."

Chaining is not just a stylistic choice. It is also not suggested just to write less or more compact code; it's a strategic approach in pandas to build clear and efficient code. Code that is easy to read, understand, maintain, and

debug. In future chapters, we will see how chaining can be used to build complex data manipulations.

Table 8.1: Math Methods and Operators

Method	Operator	Description
`s.add(s2)`	`s + s2`	Adds series
`s.radd(s2)`	`s2 + s`	Adds series
`s.sub(s2)`	`s - s2`	Subtracts series
`s.rsub(s2)`	`s2 - s`	Subtracts series
`s.mul(s2)` `s.multiply(s2)`	`s * s2`	Multiplies series
`s.rmul(s2)`	`s2 * s`	Multiplies series
`s.div(s2)` `s.truediv(s2)`	`s / s2`	Divides series
`s.rdiv(s2)` `s.rtruediv(s2)`	`s2 / s`	Divides series
`s.mod(s2)`	`s % s2`	Modulo of series division
`s.rmod(s2)`	`s2 % s`	Modulo of series division
`s.floordiv(s2)`	`s // s2`	Floor divides series
`s.rfloordiv(s2)`	`s2 // s`	Floor divides series
`s.pow(s2)`	`s ** s2`	Exponential power of series
`s.rpow(s2)`	`s2 ** s`	Exponential power of series
`s.eq(s2)`	`s2 == s`	Elementwise equals of series
`s.ne(s2)`	`s2 != s`	Elementwise not equals of series
`s.gt(s2)`	`s > 2`	Elementwise greater than of series
`s.ge(s2)`	`s >= 2`	Elementwise greater than or equals of series
`s.lt(s2)`	`s < 2`	Elementwise less than of series
`s.le(s2)`	`s <= 2`	Elementwise less than or equals of series
`np.invert(s)`	`~s`	Elementwise inversion of boolean series (no pandas method).
`np.logical_and(s, s2)`	`s & s2`	Elementwise logical and of boolean series (no pandas method).
`np.logical_or(s, s2)`	`s | s2`	Elementwise logical or of boolean series (no pandas method).

8.8 Summary

Pandas series respond to most common math operations. You can use the operator directly, and pandas will broadcast the operation to all the values. Alternatively, you can also call the corresponding method for the operator if you want to make chaining easier or parameterize the behavior of the operation.

8.9 Exercises

With a dataset of your choice:

1. Add a numeric series to itself.
2. Add 10 to a numeric series.
3. Use the `.add` method to add a numeric series.
4. Read the documentation for the `.add` method.

Chapter 9

Aggregate Methods

Aggregate methods collapse the values of a series down to a scalar. Aggregations are the numbers that your boss wants to be reported. If you worked at a burger joint and the boss came in and asked how the restaurant was doing, you wouldn't answer, "Sally ordered a burger and fries. Joe ordered a cheeseburger and shake. Tom ordered …".

Your boss doesn't care about that level of detail. They care about the juiciest bits: the totals, averages, and trends:

- How many people came in (count)
- How much food was ordered (count)
- What was the total revenue (sum)
- When did people come (skew)
- What was the average purchase amount (mean)

With this tasty analogy in mind, let's dive into how to sizzle down your data to what's needed to do aggregations in pandas.

9.1 Aggregations

There are many aggregation methods available on series. These methods collapse the values of a series down to a single value. One of the most common aggregations is the `.mean` method, which calculates the average value of a series:

```
>>> city_mpg.mean()
18.369045304297103
```

There are also a few aggregate properties. These start with `.is_`. You do not call them; they will evaluate to `True` or `False`:

```
>>> city_mpg.is_unique
False
```

```
>>> city_mpg.is_monotonic_increasing
False
```

One method to be aware of is the `.quantile` method. By default, it returns the 50% quantile. You can specify another level or pass in a list of levels. In the latter case, the result of calling `.quantile` no longer returns a scalar but a `Series` object:

```
>>> city_mpg.quantile()
17.0

>>> city_mpg.quantile(.9)
24.0

>>> city_mpg.quantile([.1, .5, .9])
0.1    13.0
0.5    17.0
0.9    24.0
Name: city08, dtype: double[pyarrow]
```

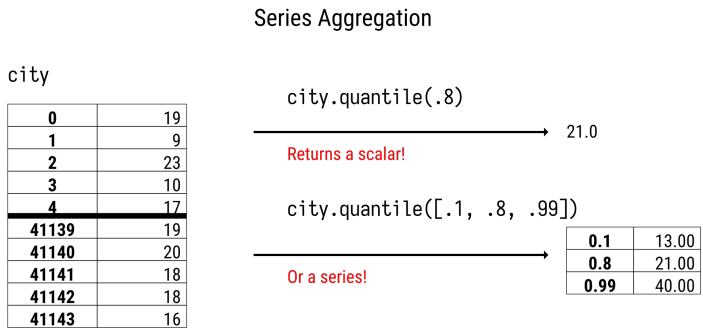


Figure 9.1: Aggregation collapses a series to a scalar value. However, the `.quantile` method also accepts a list of quantile levels and will return a `Series` object in that case.

A pandas series can aggregate with methods and properties. Some of the methods, like `.quantile`, can accept parameters to change their behavior.

9.2 Count and Mean of an Attribute

Next up, we have a pandas hat-trick: using `.sum` and `.mean` to count and calculate percentages. If you want the count of values that meet some

criteria, you can use the `.sum` method. For example, if we want the count and percentage of cars with mileage greater than 20, we can use the following code:

```
>>> (city_mpg
...  .gt(20)
...  .sum()
... )
10272
```

In this case, 10,272 cars have a city mileage greater than 20.

If you want to calculate the percentage of values that meet some criteria, you can apply the `.mean` method:

```
>>> (city_mpg
...  .gt(20)
...  .mul(100)
...  .mean()
... )
24.965973167412017
```

It looks like 25% of cars have a city mileage greater than 20.

This trick comes from the fact that Python treats `True` as `1` and `False` as `0`. (In earlier versions of the language, `True` and `False` did not exist, so programmers used `1` and `0` as stand-ins for them). To maintain backward compatibility, the language maintained math operations on booleans. If you sum up a series of boolean values, the result is the `True` values count. If you take the mean of a series of boolean values, the result is the fraction of values that are `True`. You can use this trick with any series of boolean values.

9.3 .agg and Aggregation Strings

Next up, we have the Swiss Army knife of aggregation methods: `.agg`. This allows you to combine multiple aggregations into a single call.

The `.agg` method does aggregations (not too much of a surprise, given the name). But like `.quantile`, it also transforms the data in other ways depending on how it is called.

You can use `.agg` to calculate the mean:

```
>>> city_mpg.agg('mean')
18.369045304297103
```

However, that is easier with `city_mpg.mean()`. Where `.agg` shines is in the ability to perform multiple aggregations. In that case, it returns a series. You can pass in the names of aggregation methods, NumPy reduction functions, Python aggregations, or define your own aggregation function. Here is an example calling all of these types of reductions:

```
>>> import numpy as np
>>> def second_to_last(s):
...     return s.iloc[-2]

>>> city_mpg.agg(['mean', np.var, max, second_to_last])
mean              18.369045
var               62.503036
max              150.000000
second_to_last    18.000000
Name: city08, dtype: float64
```

Below are strings that the `.agg` method accepts. You can pass in other strings as well, but they will return non-aggregating results. When you pass in a string to `.agg` pandas will map it to a method found on the `Series`:

Table 9.1: Aggregation strings for `.agg` method

Method	Description
`'all'`	Returns `True` if every value is truthy.
`'any'`	Returns `True` if any value is truthy.
`'autocorr'`	Returns Pearson correlation of series with shifted self. Can override `lag` as keyword argument(default is 1).
`'corr'`	Returns Pearson correlation of series with other series. Need to specify `other`.
`'count'`	Returns count of non-missing values.
`'cov'`	Return covariance of series with other series. Need to specify `other`.
`'dtype'`	Type of the series.
`'dtypes'`	Type of the series.
`'empty'`	`True` if no values in series.
`'hasnans'`	`True` if missing values in series.
`'idxmax'`	Returns index value of maximum value.
`'idxmin'`	Returns index value of minimum value.
`'is_monotonic'`	`True` if values always increases. Can also use `'is_monotonic_increasing'` or `'is_monotonic_decreasing'`.
`'kurt'`	Return "excess" kurtosis (0 is normal distribution). Values greater than 0 have more outliers than normal.
`'mad'`	Return the mean absolute deviation.
`'max'`	Return the maximum value.
`'mean'`	Return the mean value.
`'median'`	Return the median value.
`'min'`	Return the minimum value.
`'nbytes'`	Return the number of bytes of the data.
`'ndim'`	Return the number of dimensions (1) of the data.
`'nunique'`	Return the count of unique values.

Method	Description
`'quantile'`	Return the median value. Can override `q` to specify other quantile.
`'sem'`	Return the unbiased standard error.
`'size'`	Return the size of the data.
`'skew'`	Return the unbiased skew of the data. Negative indicates tail is on the left side.
`'std'`	Return the standard deviation of the data.
`'sum'`	Return the sum of the series.

Below is a table of various aggregation methods and properties.

Table 9.2: Aggregation methods and properties

Method	Description
`s.agg(func=None, axis=0, *args, **kwargs)`	Returns a scalar if `func` is a single aggregation function. Returns a series if a list of aggregations are passed to `func`.
`s.all(axis=0, bool_only=None, skipna=True, level=None)`	Returns `True` if every value is truthy. Otherwise `False`
`s.any(axis=0, bool_only=None, skipna=True, level=None)`	Returns `True` if at least one value is truthy. Otherwise `False`
`s.autocorr(lag=1)`	Returns Pearson correlation between `s` and shifted `s`
`s.corr(other, method='pearson')`	Returns correlation coefficient for `'pearson'`, `'spearman'`, `'kendall'`, or a callable.
`s.cov(other, min_periods=None)`	Returns covariance.
`s.max(axis=None, skipna=None, level=None, numeric_only=None`	Returns maximum value.
`s.min(axis=None, skipna=None, level=None, numeric_only=None)`	Returns minimum value.
`s.mean(axis=None, skipna=None, level=None, numeric_only=None)`	Returns mean value.
`s.median(axis=None, skipna=None, level=None, numeric_only=None)`	Returns median value.
`s.prod(axis=None, skipna=None, level=None, numeric_only=None, min_count=0)`	Returns product of `s` values.
`s.quantile(q=.5, interpolation='linear')`	Returns 50% quantile by default. *Note* returns `Series` if `q` is a list.

Method	Description
`s.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased standard error of mean.
`s.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns sample standard deviation.
`s.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased variance.
`s.skew(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased skew.
`s.kurtosis(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased kurtosis.
`s.nunique(dropna=True)`	Returns count of unique items.
`s.count(level=None)`	Returns count of non-missing items.
`s.size`	Number of items in series. (Property)
`s.is_unique`	`True` if all values are unique
`s.is_monotonic`	`True` if all values are increasing
`s.is_monotonic_increasing`	`True` if all values are increasing
`s.is_monotonic_decreasing`	`True` if all values are decreasing

9.4 Summary

As we've seen, aggregate methods are the power tools of data analysis, helping you distill complex datasets into meaningful, digestible statistics. In this chapter, we discussed ways to summarize data in a series. As you begin to analyze data, many of these keep popping up. One thing to keep in mind is that they also apply to a `DataFrame`. You only have to learn them once.

9.5 Exercises

With a dataset of your choice:

1. Find the count of non-missing values of a series.
2. Find the number of entries of a series.
3. Find the number of unique entries of a series.
4. Find the mean value of a series.
5. Find the maximum value of a series.
6. Use the `.agg` method to find all of the above.

Chapter 10

Conversion Methods

I come across messy data or data read from a CSV file frequently. This data is often not in the format that I want. Sometimes, you will need to change the type of the data. This may be due to formats that do not include type information, or it may be that you can eke out better performance (more manipulation options or use less memory) by changing types.

In this chapter, we will look at various conversions that you might want to do to a `Series`.

10.1 Type Conversion

To specify a type for a series, you can try to use the `.astype` method. Our city mileage can be held in an 8-bit unsigned or 16-bit integer. However, an 8-bit signed integer will not work, as the maximum value for that signed type is 127, and we have some cars with a value of 150:

```
>>> city_mpg.astype('int16[pyarrow]')
0        19
1         9
2        23
         ..
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int16[pyarrow]

>>> city_mpg.astype('int8[pyarrow]')
ArrowInvalid            Traceback (most recent call last)
  ...
ArrowInvalid: Integer value 132 not in range: -128 to 127
```

Using the correct type can save significant amounts of memory. The default numeric type is 8 bytes wide (64 bits, ie `int64` or `float64`). If you can

use a narrower type, you can cut back on memory usage, allowing you to have more memory to process more data.

You can use the NumPy `iinfo` and `finfo` functions to inspect limits on integer and float types:

```
>>> import numpy as np
>>> np.iinfo('int64')
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)

>>> np.iinfo('uint8')
iinfo(min=0, max=255, dtype=uint8)

>>> np.finfo('float32')
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38,
      dtype=float32)
```

If you know the range of values that you need to store, you can use an appropriate type to save memory without losing information.

10.2 Memory Usage

Now, let's talk about the diet plan for your data: reducing memory usage.

To calculate memory usage of the `Series`, you can use the `.nbytes` property or the `.memory_usage` method. The latter is useful when dealing with `object` types as you can pass `deep=True` to include the amount of memory used by the Python objects in the `Series`.

Here we compare memory usage of default numeric integers to `Int16`:

```
>>> city_mpg.nbytes
329152

>>> city_mpg.astype('Int16').nbytes
123432
```

Using `.nbytes` with `object` types only shows how much memory the Pandas object is taking. The *make* of the autos has pyarrow strings. If we convert it back to a Pandas 1 string column, the type becomes `object`.

Here we inspect the Pandas 2 memory usage:

```
>>> make = df.make
>>> make.nbytes
425635

>>> make.memory_usage()
425767

>>> make.memory_usage(deep=True)
425767
```

Now, let's convert it to Pandas 1. To get the true amount of memory that includes the strings, we need to use the `.memory_usage` method with `deep=True`:

```
>>> make.astype(str).memory_usage()
329284
```

It may appear that the Pandas 2 *make* series uses more memory than the Pandas 1 *make* series. However, `.memory_usage` with `deep=True` shows the real amount of data in Pandas 1 is much larger.

```
>>> make.astype(str).memory_usage(deep=True)
2606399
```

Notice that the Pandas 1 memory usage is much higher than the Pandas 2. In this example, it is using almost 6 times the memory!

The value of `.nbytes` is just the memory that the data is using and not the ancillary parts of the `Series`. The `.memory_usage` includes the index memory and can include the contribution from `object` types.

In the next section, we discuss converting to a categorical. We can see that we will save a lot of memory for the `make` data:

10.3 String and Category Types

In the memory-saving game, we can get quick savings by converting to a category. This is because the category type stores the unique values in a dictionary and then uses integers to reference the dictionary. Because it doesn't have to store a copy of the string for each value, it can save a lot of memory.

You can use the `.astype` method to convert to a category:

```
>>> (make
...  .astype('category')
...  .memory_usage(deep=True)
... )
88701
```

We save another 5x versus the pyarrow string type when we convert the *make* column to a category.

When we convert strings to categories, we retain the ability to use the string methods via the `.str` accessor. We can also use the `.cat` accessor to use category methods.

We can also convert numeric types to categories. An example where this might be useful is to represent shoe sizes. You could save a lot of memory. However, you would lose the ability to do math on the values.

```
(city_mpg
.astype('category')
.cat.as_ordered()
)
```

10.4 Ordered Categories

Pandas also supports ordered categories. This is useful when you have data that has a natural order. For example, you might have a series of shoe sizes. You could compare the sizes to see which is larger or smaller. You can also use ordered categories to sort the data. You could get the frequency of each size with the `.value_counts` method. However, you cannot do math, like calculating the average size of shoes.

To create ordered categories, you need to define your own `CategoricalDtype`. Here we convert the city mileage to an ordered category:

```
>>> values = pd.Series(sorted(set(city_mpg)))
>>> city_type = pd.CategoricalDtype(categories=values,
...     ordered=True)
>>> city_mpg.astype(city_type)
0        19
1         9
2        23
3        10
4        17
         ..
41139    19
41140    20
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: category
Categories (105, int64): [6 < 7 < 8 < 9 ... 137 < 138 < 140 < 150]
```

You can also use the `.cat.as_ordered` method to convert to an ordered category:

```
>>> city_mpg.astype('category').cat.as_ordered()
0        19
1         9
2        23
         ..
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: category
Categories (105, int64[pyarrow]): [6 < 7 < 8 < 9 ... 137 < 138 < 140
    < 150]
```

Note that there is a distinct pyarrow category type, `dictionary`; however, I recommend just using `'category'` as we can't easily access the `dictionary` type. We don't tack a `[pyarrow]` onto the category type when converting it. If you

save data with categorical strings, both the feather and parquet formats will support that.

However, converting a numeric column into a category will be converted to a pyarrow `dictionary` type for feather exporting. The dictionary type does not have a `.cat` accessor. For parquet, it will use the underlying type when you save it. For example, creating a category from an integer will save it as an integer.

Ordered categories bring structure to your data and also enable comparisons and sorting.

The section on categories below will discuss more of their features.

The following table lists the types that you can pass into `.astype`.

Table 10.1: Type and strings for column conversion

String or Type	Description
`str` or `'str'`	Convert type to Pandas 1 string
`'string[pyarrow]'`	Convert type to Pandas 1.5 pyarrow string (supports `pd.NA`). Don't use this!
`pd.ArrowDtype(pa.string())`	Convert to Pandas 2 PyArrow string
`int`, `'int'`, or `'int64'`	Convert type to NumPy int64
`'int32'` or `'uint32'`	Convert type to 32 signed or unsigned NumPy integer (can also use 16 and 8).
`'Int64'`	Convert type to pandas Int64 (supports `pd.NA`). Might complain when you convert floats or strings.
`'INTTYPE[pyarrow]'`	Convert to pyarrow where *INTTYPE* is one of the following: `int8` … `int64` or `uint8` … `uint64`
`float`, `'float'`, or `'float64'`	Convert type to NumPy float64 (can also support 32 or 16).
`'float[pyarrow]'`	Convert type to pyarrow float64
`'category'`	Convert type to categorical (supports `pd.NA`). You can also use an instance of `CategoricalDtype`.
`'datetime64[ns]'`	Convert to datetime. Use `pd.to_datetime` for more control.
`'timestamp[ns][pyarrow]'`	Convert to PyArrow datetime.

10.5 Converting to Other Types

Sometimes, you need to step outside of the types that Pandas supports. Let's explore how to transform a column into arrays, lists, or other types.

The `.to_numpy` method (or the `.values` property) will give us a NumPy array of values, and the `.to_list` will return a Python list of values. I recommend staying away from these unless necessary. Sometimes, there is

a speed increase if you use straight NumPy, but there are drawbacks. I find pandas data structures much more user-friendly, and the code reads easier. Using Python lists will slow down your code significantly.

As was mentioned before, a `Series` object is a column from a `DataFrame`. However, you might need to turn a `Series` back into a `DataFrame`. When we discuss dataframes, we will show how to add columns to them, but if you want a dataframe with a single column, you can use the `.to_frame` method:

```
>>> city_mpg.to_frame()
       city08
0          19
1           9
2          23
3          10
4          17
...       ...
41139      19
41140      20
41141      18
41142      18
41143      16

[41144 rows x 1 columns]
```

Also, many conversion methods exist to export data into other formats, including CSV, Excel, HDF5, SQL, JSON, Parquet, Feather, and more. These also exist on dataframes, and I find that I use them there and never use them on a `Series` object. We will talk more about them in the dataframe serialization chapter. Be aware of these methods, and realize that if you understand how they work with dataframes, that knowledge will map back to series.

Finally, to convert to a datetime, use the `to_datetime` function in pandas. It is a little more involved if you want to add time zone information. The section on dates will discuss this.

Table 10.2: Aggregation methods and properties

Method	Description
`s.convert_dtypes(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True)`	Convert types to appropriate pandas 1 types (that support `NA`). Doesn't try to reduce size of integer or float types.
`s.astype(dtype, copy=True, errors='raise')`	Cast series into particular type. If `errors='ignore'` then return original series on error.

Method	Description
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, format=None, exact=True, unit=None, infer_datetime_format=False, origin='unix', cache=True)`	Convert `arg` (a series) into datetime. Use `format` to specify strftime string.
`s.to_numpy(dtype=None, copy=False,na_value=object, **kwargs)`	Convert the series to a NumPy array.
`s.values`	Convert the series to a NumPy array.
`s.to_frame(name=None)`	Return a dataframe representation of the series.
`pd.CategoricalDtype(categories=None, ordered=False)`	Create a type for categorical data.
`pd.ArrowDtype(pa.string())`	Create a PyArrow string type for casting

10.6 Summary

Having the correct types is very convenient. Not only does it save memory, but it also enables operations that are otherwise tedious. Whenever I teach students the fundamentals of data analysis, I make sure that they go through each column and determine the correct type for that column. The right type can be a game-changer, optimizing memory usage and enhancing data manipulation.

10.7 Exercises

With a dataset of your choice:

1. Convert a numeric column to a smaller type.
2. Calculate the memory savings by converting to smaller numeric types.
3. What is the proper type to cast into String types?
4. Convert a string column into a categorical type.
5. Calculate the memory savings (or losses) by converting to a categorical type.

Chapter 11

Manipulation Methods

Data manipulation has been compared to janitorial work. Janitors sweep up the merriment of the night before, pick up the garbage that folks leave behind, wipe up spills, throw out trash, sprinkle sawdust on vomit, mop up body fluids, wipe down tables, and more. They know which tool in their arsenal to use for each job. They act methodically, performing each task in a specific order.

You might feel like you are doing the same thing with data. The messy data you get from the real world must be cleaned up and then cleaned up again. Luckily, pandas has a lot of tools to help you with this.

I consider manipulation methods to be the workhorses of pandas. When I have a dataset I am trying to understand, clean up, and model, I use methods that operate on a series and return a new series (usually with the same index) to stick it back in the dataframe I'm working on. Most methods we discuss here manipulate the series values but preserve the index. In this chapter, we will explore these methods.

11.1 .apply and .where

Let's start our discussion with an oft-misunderstood and very often misapplied method, `.apply`. The `.apply` method is a curious method, and I often tell my students to avoid it, but it seems to rear its ugly head in many places. I suspect this is because many SEO-focused blog posts pretend it is the cure for all data manipulation problems.

It is not a cure, and in my experience, it is often misused.

Having said that, sometimes it comes in handy. `.apply` allows you to apply a function to an entire series or element-wise to every value. And this is where the problem begins. If it does the latter, you are taking the data out of the optimized and fast storage and pulling it into Python. This is a slow operation. Then, you pass the data to a Python function, which is also slow. Then, you return the data back to pandas, which is slow. This is a lot of slow operations , which can often be avoided.

The function will be called one million times if you have one million values in a series. It breaks out of the fast vectorized code paths we can leverage in pandas, and puts us back to using slow Python code.

For example, we previously checked whether the values in the mileage were greater than 20. We can also do this with the `.apply` method. I'll use the Jupyter `%%timeit` cell magic to microbenchmark this (note this will only work in Jupyter or IPython):

```
>>> def gt20(val):
...     return val > 20

>>> %%timeit
>>> city_mpg.apply(gt20)
7.32 ms ± 390 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In contrast, if we use the broadcasted `.gt` method, it runs almost 50 times faster:

```
>>> %%timeit
>>> city_mpg.gt(20)
156 µs ± 30.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Here's another example. I will look at the *make* column from my dataset. This is the company that produced each car. There are quite a few makes in there. I might want to limit my dataset to show the top five makes and label everything else as *Other*. To do that, I would use the `.value_counts` method to get the frequencies:

```
>>> make = df.make
>>> make
0        Alfa Romeo
1           Ferrari
2             Dodge
             ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]

>>> make.value_counts()
make
Chevrolet           4003
Ford                3371
Dodge               2583
                    ...
General Motors         1
Goldacre               1
```

```
Isis Imports Ltd       1
Name: count, Length: 136, dtype: int64[pyarrow]
```

The first five entries in the index are the values I want to keep. I want to replace everything else with *Other*. Here is an example using `.apply`:

```
>>> top5 = make.value_counts().index[:5]
>>> top5
Index(['Chevrolet', 'Ford', 'Dodge', 'GMC', 'Toyota'],
     dtype='string[pyarrow]', name='make')


>>> def generalize_top5(val):
...     if val in top5:
...         return val
...     return 'Other'

>>> make.apply(generalize_top5)
```

Note that when we have already defined a function, `generalize_top5`, to pass into `.apply` that we do not call that function. In the above example, we are not calling `generalize_top5`; we are just passing it into `.apply`. The `.apply` method will call the function for us.

In the above example, `generalize_top5` is called once for every value. A faster, more conversational manner of doing this is using the `.where` method. This method takes a *boolean array* to mark where a condition is true. The `.where` method keeps values from the series it is called on (`make` in the example below) where the boolean array is true, and if the boolean array is false, it uses the value of the second parameter, `other`:

```
>>> make.where(make.isin(top5), other='Other')
0        Other
1        Other
2        Dodge
         ...
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: string[pyarrow]
```

The `.where` method is optimized, and if you look at the timings, it is about fifteen times faster for my data size:

```
>>> %%timeit
>>> make.apply(generalize_top5)
21.6 ms ± 306 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

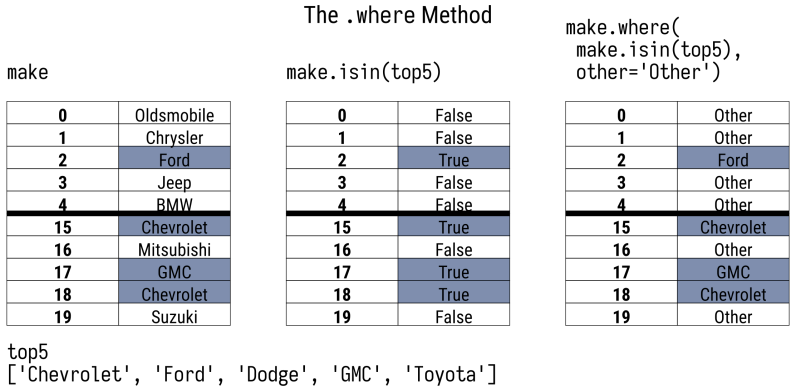


Figure 11.1: The `.where` method keeps the values where the index is `True` and uses the `other` parameter to specify values for `False`

```
>>> %%timeit
>>> make.where(make.isin(top5), 'Other')
1.04 ms ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
    each)
```

The complement of the `.where` method is the `.mask` method. Wherever the condition is `False` it keeps the original values; if it is `True` it replaces the value with the `other` parameter. Here is the `.mask` version of our where statement:

```
>>> make.mask(~make.isin(top5), other='Other')
0        Other
1        Other
2        Dodge
         ...
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: string[pyarrow]
```

The tilde, `~`, performs an inversion of the boolean array, switching all true values to false and vice versa.

In pandas, there is often more than one way to do something. My take is to prefer using `.where` and ignoring `.mask` since it is the complement.

11.2 Apply with NumPy Functions

There are cases where `.apply` is helpful. If you are working with a NumPy function that works on arrays, then `.apply` will broadcast the operation to the entire series. I don't see many of these blog posts exhibiting or even mentioning this. They just show you how to make your pandas code slow.

Let's demonstrate this with the NumPy `np.log` function. This function takes the natural log of each value in an array. It is a universal function (*ufunc*) and works on arrays. We can use it with `.apply` to the *city_mpg* column.

Let's compare using `np.log` and the `math.log` function. The `math.log` function only works on a single value and not on an array. Let's import the libraries:

```
>>> import numpy as np
>>> import math

>>> %%timeit
>>> city_mpg.apply(np.log)
155 µs ± 1.2 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
     each)

>>> %%timeit
>>> city_mpg.apply(math.log)
3.83 ms ± 40.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops
     each)
```

On my machine, the NumPy version is about 14 times faster than the Python version. This is one of the few cases where `.apply` is appropriate.

11.3 If Else with Pandas

I will show one more piece of code that illustrates what I consider a shortcoming of pandas. If I wanted to keep the top five makes and use *Top10* for the remainder of the top ten makes, with *Other* for the rest, there is no built-in pandas method to do that easily. I could use the following function in combination with `.apply`:

```
>>> vc = make.value_counts()
>>> top5 = vc.index[:5]
>>> top10 = vc.index[:10]
>>> def generalize(val):
...     if val in top5:
...         return val
...     elif val in top10:
...         return 'Top10'
...     else:
...         return 'Other'
```

```
>>> make.apply(generalize)
0        Other
1        Other
2        Dodge
         ...
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: object
```

One of the new features of pandas 2.2 is the `.case_when` method. Let's take it for a spin. To use this method, we provide a list of tuples for the `caselist` parameter. The first item in the tuple is a boolean array or a function that takes the series and returns a boolean array. The second item in the tuple is the values when that boolean array is true. You can have as many tuples as you want. When one sets values for an index, subsequent tuples cannot override the values.

I wish you could provide a final *else* or default value wi th this interface, but it is not available. In my example, for the `else` part of the code, I'm passing a true boolean array for the default value.

```
>>> (make
...   .case_when(caselist=[(make.isin(top5), make),
...       (make.isin(top10), 'Top10'),
...       (pd.Series(True, index=make.index), 'Other')])
... )

0        Other
1        Other
2        Dodge
         ...
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: object
```

On my machine (and using this data), the `.case_when` code runs about 6x fasxter.

We can also use the `.where` method to replicate this in pandas. I would need to chain calls to `.where`. I find this harder to read because it isn't like a tradition *if then* in programming. It is more like an if this boolean array not true use this value.

```
>>> (make
...   .where(make.isin(top5), 'Top10')
```

```
...    .where(make.isin(top10), 'Other')
... )
0         Other
1         Other
2         Dodge
          ...
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: string[pyarrow]
```

11.4 Missing Data

Missing data can be a nightmare to deal with. Especially given that many machine learning algorithms do not work if there is missing data. Also, knowing how much data is missing is prudent to ensure you get the full story from your data.

The *cylinders* column has missing values. Remember our trick to calculate the count of items that have some property? We can use it here to determine the count of missing entries. We convert the property to booleans (using `.isna`), then call `.sum` on it:

```
>>> cyl = df.cylinders
>>> (cyl
...  .isna()
...  .sum()
... )
206
```

From the *cylinders* series alone, it is hard to determine why these values are missing. Typically, we will need more context, and the corresponding values from a dataframe will give that to us. We will use the *make* column, which corresponds with the cylinder values, to give us some insight. First, let's find the index where the values are missing in the *cylinders* column and then show what those makes are:

```
>>> missing = cyl.isna()
>>> make.loc[missing]
7138      Nissan
7139      Toyota
8143      Toyota
           ...
34565      Tesla
34566      Tesla
34567      Tesla
Name: make, Length: 206, dtype: string[pyarrow]
```

Note

We often use the same term to represent different items. In pandas, both a series and a data frame have an *index*, the value naming each row. In addition, we use an *index operation*, performed with square brackets ([and]), to select values from a series or a data frame.

I will try to use the noun "index" to discuss the member of the series or data frame. If I use "index" as a verb or say "index operation", it refers to selecting out subsets of data. Below, I am indexing off of the `.loc` attribute. I could also say that I'm doing an indexing operation:

```
make.loc[missing]
```

We will talk about the `.loc` attribute when we discuss indexing. For now, realize that if we index `.loc` with a boolean array, it returns the rows where the boolean array is true.

11.5 Filling In Missing Data

It looks like the cylinder information is missing from electric cars. A Tesla car-because it has an electric engine, not a combustion engine-has zero cylinders. The `.fillna` method allows you to specify a replacement value for any missing data. To fill in the missing values with `0`, we can do the following:

```
>>> cyl[cyl.isna()]
7138     <NA>
7139     <NA>
8143     <NA>
         ...
34565    <NA>
34566    <NA>
34567    <NA>
Name: cylinders, Length: 206, dtype: int64[pyarrow]

>>> cyl.fillna(0).loc[7136:7141]
7136    6
7137    6
7138    0
7139    0
7140    6
7141    6
Name: cylinders, dtype: int64[pyarrow]
```

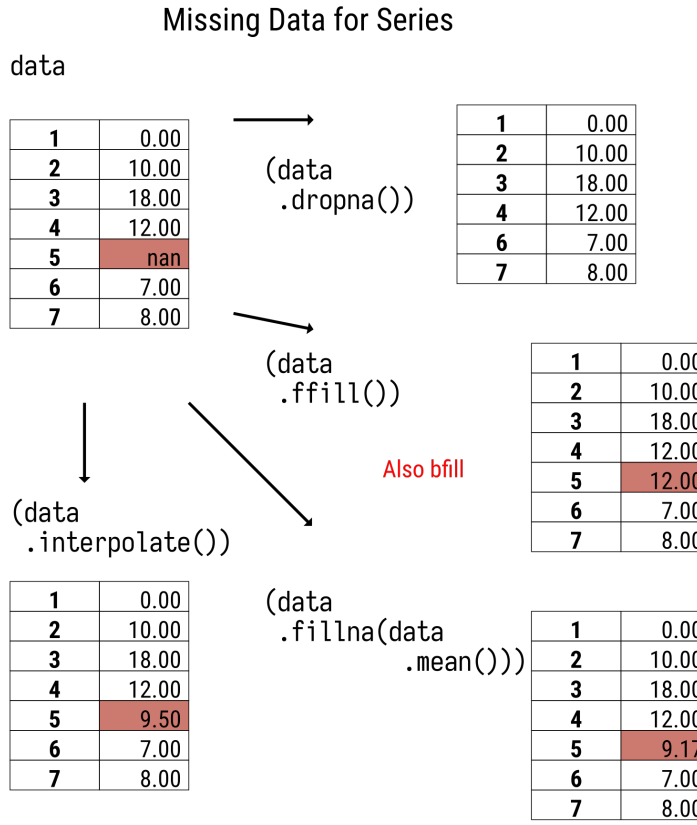


Figure 11.2: We can drop missing data or fill it in with other values.

Note

Almost every operation that I show in this book does not mutate data. In other words, the above operation returns a new series with the missing values replaced by zero. If I want to update my `cyl` variable, I need to assign it to this new result. Usually, I chain each command and build up a sequence of operations.

11.6 Interpolating Data

Another option for replacing missing data is the `.interpolate` method. This comes in handy if the data is ordered (as time series data often is) and there are holes in the data. For example, if you had the temperature measurements, `temp`, you could fill in the values using this:

```
>>> temp = pd.Series([32, 40, None, 42, 39, 32],
...     dtype='float[pyarrow]')
>>> temp
0    32.0
1    40.0
2    <NA>
3    42.0
4    39.0
5    32.0
dtype: float[pyarrow]
```

Let's fill in the missing value for index label 2 using the `.interpolate` method:

```
>>> temp.interpolate()
0    32.0
1    40.0
2    41.0
3    42.0
4    39.0
5    32.0
dtype: float[pyarrow]
```

Notice that the value for index label 2 was missing. However, there are values for index labels 1 and 3. After interpolation, the missing value becomes *41.0*, a linear interpolation of the values around the missing value.

11.7 Clipping Data

If you have outliers in your data, you might want to use the `.clip` method. In the example below, the first 447 entries in *city* range from 9 to 31:

```
>>> city_mpg.loc[:446]
0      19
1       9
2      23
       ..
444    15
445    15
446    31
Name: city08, Length: 447, dtype: int64[pyarrow]
```

We can trim the values to be between the 5th (11.0) and 95th quantile (27.0) with the following code:

```
>>> (city_mpg
...     .loc[:446]
...     .clip(lower=city_mpg.quantile(.05),
...           upper=city_mpg.quantile(.95))
... )
0      19
1      11
2      23
       ..
444    15
445    15
446    27
Name: city08, Length: 447, dtype: int64[pyarrow]
```

In fact, if you dig into the implementation of `.clip`, you will see a call to `.where`. Below is a portion of the `._clip_with_scalar` method that `.clip` calls:

```
result = self
mask = self.isna()

if lower is not None:
    cond = mask | (self >= lower)
    result = result.where(
        cond, lower, inplace=inplace
    )  # type: ignore[assignment]
if upper is not None:
    cond = mask | (self <= upper)
    result = self if inplace else result
    result = result.where(
        cond, upper, inplace=inplace
    )  # type: ignore[assignment]

return result
```

11.8 Sorting Values

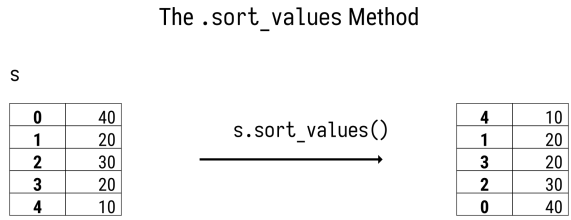


Figure 11.3: The `.sort_values` method will return a new series with the values sorted (and the original labels in the corresponding order).

Other manipulation methods might return objects with different index entries. The `.sort_values` method will sort the values in ascending order and also rearrange the index accordingly:

```
>>> city_mpg.sort_values()
7901       6
21060      6
34557      6
          ...
31256    150
32599    150
33423    150
Name: city08, Length: 41144, dtype: int64[pyarrow]
```

Note that because of index alignment, you can still do math operations (and many other operations) on a sorted series:

```
>>> (city_mpg.sort_values() + highway_mpg) / 2
0        22.0
1        11.5
2        28.0
         ...
41141    21.0
41142    21.0
41143    18.5
Length: 41144, dtype: double[pyarrow]
```

11.9 Sorting the Index

If you want to sort the index of a series, you can use the `.sort_index` method. Below, we unsort the index by sorting the values, then essentially revert that:

```
>>> city_mpg.sort_values().sort_index()
0        19
1         9
2        23
         ..
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int64[pyarrow]
```

11.10 Dropping Duplicates

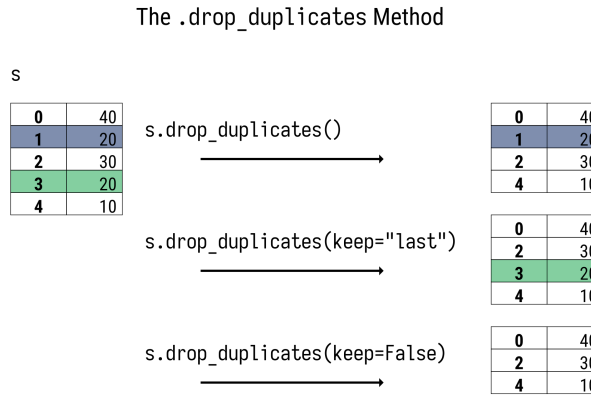


Figure 11.4: The `.drop_duplicates` method will return a new series that drops the values after they appear more than once by default. The behavior can be changed with the `keep` parameter.

Many datasets have duplicate entries. The `.drop_duplicates` method will remove values that appear more than once. You can determine whether to keep the first or last duplicate value using the `keep` parameter. If you set it to `'last'`, it will use the last value. The default value is `'first'`. If you set it to `False`, it will remove any duplicated values (including the initial value). Notice that this call keeps the original index. However, there are only 105 results (down from 41144) now that duplicates are removed:

```
>>> city_mpg.drop_duplicates()
0        19
1         9
```

```
2         23
         ...
34564    140
34565    115
34566    104
Name: city08, Length: 105, dtype: int64[pyarrow]
```

11.11 Ranking Data

The `.rank` method will return a series that keeps the original index but uses the ranks of values from the original series. You can control how ranking occurs with the `method` parameter. By default, if two values are the same, their rank will be the average of the positions they take.

```
>>> city_mpg.rank()
0        27060.5
1          235.5
2        35830.0
          ...
41141    23528.0
41142    23528.0
41143    15479.0
Name: city08, Length: 41144, dtype: double[pyarrow]
```

You can specify `'min'` to put equal values in the same rank. However, the next rank will be the number of values that were equal. For example, if the first three values are equal, they will all be ranked 1, and the next value will be ranked 4.

```
>>> city_mpg.rank(method='min')
0        25555
1          136
2        35119
          ...
41141    21502
41142    21502
41143    13492
Name: city08, Length: 41144, dtype: uint64[pyarrow]
```

If you use `method='dense'`, the ranks will be consecutive integers and *dense* (i.e., to not skip any positions). For example, if the first three values are equal, they will all be ranked 1, and the next value will be ranked 2:

```
>>> city_mpg.rank(method='dense')
0        14
1         4
```

```
2        18
         ..
41141    13
41142    13
41143    11
Name: city08, Length: 41144, dtype: uint64[pyarrow]
```

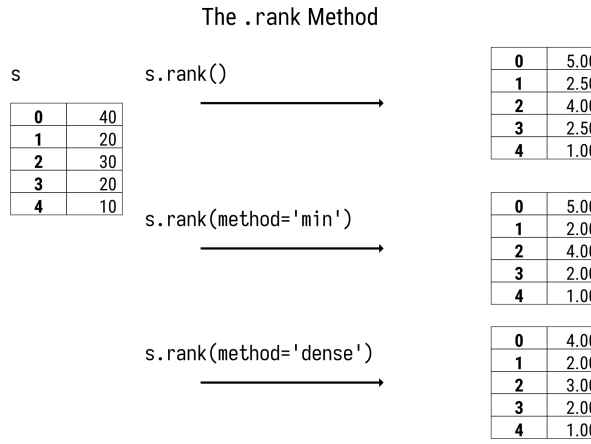


Figure 11.5: The .rank method has various options for dealing with ties.

11.12 Replacing Data

The .replace method allows you to map whole values to new values. There are many ways to specify how to replace the values. You can specify an entire string to replace a string or use a dictionary to map old to new values. This example uses the former:

```
>>> make.replace('Subaru', 'スバル')
0        Alfa Romeo
1           Ferrari
2             Dodge
            ...
41141           スバル
41142           スバル
41143           スバル
Name: make, Length: 41144, dtype: string[pyarrow]
```

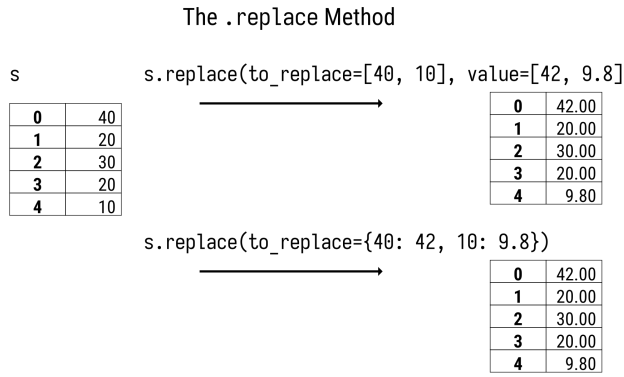


Figure 11.6: The `.replace` method illustrating lists and dictionaries.

The `to_replace` parameter's value can contain a regular expression if you provide the `regex=True` parameter. In this example, we use the regular expression *capture groups* (they are specified in the expression by the parentheses). In the `value` parameter, we refer to these groups (`\1` refers to the contents inside the first parentheses, and `\2` refers to the contents in the second parentheses) when replacing the original value:

```
>>> make.replace(r'(Fer)ra(r.*)',
...     value=r'\2-other-\1', regex=True)
0          Alfa Romeo
1        ri-other-Fer
2               Dodge
              ...
41141          Subaru
41142          Subaru
41143          Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

11.13 Binning Data

You can bin data as well. Using the `cut` function, you can create bins of equal width. For example, the following code creates 10 bins of equal width for the *city* column. Because the values range from 6 to 150, the width of each bin is 14.4:

```
>>> pd.cut(city_mpg, 10)
0        (5.856, 20.4]
```

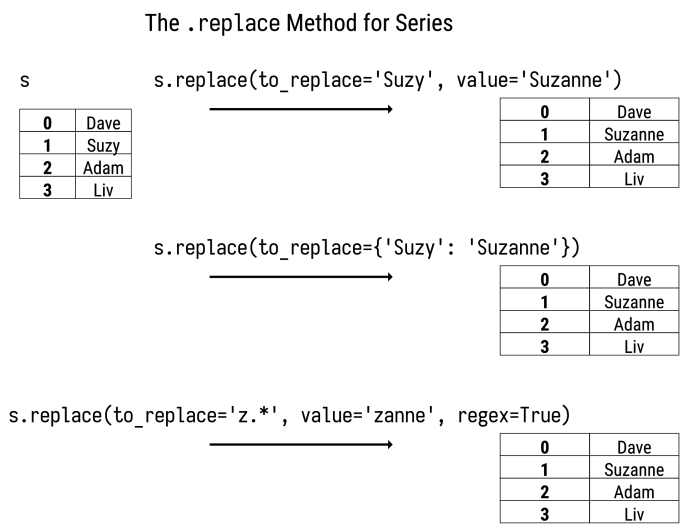


Figure 11.7: The .replace method illustrating different replacement mechanisms.

```
1         (5.856, 20.4]
2          (20.4, 34.8]
3         (5.856, 20.4]
4         (5.856, 20.4]
              ...
41139     (5.856, 20.4]
41140     (5.856, 20.4]
41141     (5.856, 20.4]
41142     (5.856, 20.4]
41143     (5.856, 20.4]
Name: city08, Length: 41144, dtype: category
Categories (10, interval[float64]): [(5.856, 20.4] < (20.4, 34.8] < ...
   (121.2, 135.6] < (135.6, 150.0]]
```

Notice that the result of this call is a series of categorical values.

If you have specific sizes for bin edges, you can specify those. In the following example, five bins are created (so you need to provide six edges). The first bin will contain values from 0 to 10, the second from 10 to 20, and so on. The last bin will contain values from 70 to 150:

```
>>> pd.cut(city_mpg, [0, 10, 20, 40, 70, 150])
0         (10, 20]
1          (0, 10]
2         (20, 40]
```

```
3         (0, 10]
4        (10, 20]
           ...
41139    (10, 20]
41140    (10, 20]
41141    (10, 20]
41142    (10, 20]
41143    (10, 20]
Name: city08, Length: 41144, dtype: category
Categories (5, interval[int64]): [(0, 10] < (10, 20] < (20, 40]
    < (40, 70] < (70, 150]]
```

Note the bins have a half-open interval. They do not have the start value but do include the end value. If the `city_mpg` series had values with 0 or values above 150, they would be missing after binning the series.

You can bin data with quantiles instead. If you wanted ten bins with approximately the same number of entries in each bin (rather than each bin width being the same), use the `qcut` function. Each of the ten bins would have approximately 10 percent of the data. The quantiles of the data determine the bin edges.

```
>>> pd.qcut(city_mpg, 10)
0         (18.0, 20.0]
1        (5.999, 13.0]
2         (21.0, 24.0]
3        (5.999, 13.0]
4         (16.0, 17.0]
             ...
41139     (18.0, 20.0]
41140     (18.0, 20.0]
41141     (17.0, 18.0]
41142     (17.0, 18.0]
41143     (15.0, 16.0]
Name: city08, Length: 41144, dtype: category
Categories (10, interval[float64, right]): [(5.999, 13.0] < (13.0, 14.0]
  < (14.0, 15.0] < (15.0, 16.0] ... (18.0, 20.0] < (20.0, 21.0]
  < (21.0, 24.0] < (24.0, 150.0]]
```

Both of these functions (`pd.cut`, and `pd.qcut`) allow you to set the labels to use instead of the labels derived from the categorical intervals they generate:

```
>>> pd.qcut(city_mpg, 10, labels=list(range(1,11)))
0        7
1        1
2        9
3        1
```

```
4        5
        ..
41139    7
41140    7
41141    6
41142    6
41143    4
Name: city08, Length: 41144, dtype: category
Categories (10, int64): [1 < 2 < 3 < 4 ... 7 < 8 < 9 < 10]
```

Table 11.1: Manipulation methods and properties

Method	Description
`s.apply(func, convert_dtype=True, args=(), **kwds)`	Pass in a NumPy function that works on the series, or a Python function that works on a single value. `args` and `kwds` are arguments for `func`. Returns a series, or dataframe if `func` returns a series.
`s.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False)`	Pass in a boolean series/dataframe, list, or callable as `cond`. If the value is `True`, keep it, otherwise use `other` value. If it is a function, it takes a series and should return a boolean sequence.
`s.case_when(caselist)`	Use `caselist` (list of tuples of (boolean array, result), to simulate if then statements
`s.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)`	Pass in a scalar, dict, series, or dataframe for `value`. If it is a scalar, use that value, otherwise use the index from the old value to the new value.
`s.interpolate( method='linear', axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)`	Perform interpolation with missing values. `method` may be `linear`, `time` among others.
`s.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)`	Return a new series with values clipped to `lower` and `upper`.

Method	Description
`s.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)`	Return a series with values sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates location of NaNs and may be `'first'` or `'last'`.
`s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)`	Return a series with index sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates location of NaNs and may be `'first'` or `'last'`.
`s.drop_duplicates( keep='first', inplace=False)`	Drop duplicates. `keep` may be `'first'`, `'last'`, or `False`. (If `False`, it removes all values that were duplicated).
`s.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)`	Return a series with numerical ranks. `method` allows you to specify tie handling. `'average'`, `'min'`, `'max'`, `'first'` (uses order they appear in series), `'dense'` (like `'min'`, but rank only increases by one after tie). `na_option` allows you to specify `NaN` handling. `'keep'` (stay at `NaN`), `'top'` (move to smallest), `'bottom'` (move to largest).
`s.replace( to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad')`	Return a series with new values. `to_replace` can be many things. If it is a string, number, or regular expression, you can replace it with a scalar `value`. It can also be a list of those things which requires `values` to be a list of the same size. Finally, it can be a dictionary mapping old values to new values.
`pd.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates='raise', ordered=True)`	Bin values from `x` (a series). If `bins` is an integer, use equal-width bins. If `bins` is a list of numbers (defining minimum and maximum positions) use those for the edges. `right` defines whether the right edge is open or closed. `labels` allows you to specify the bin names. Out of bounds values will be missing.

Method	Description
`pd.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')`	Bin values from `x` (a series) into `q` equal-sized bins (`10` for quantiles, `4`). Alternatively, can pass in a list of quantile edges. Out of bounds values will be missing.

11.14 Summary

This chapter explored many useful methods and functions for changing the data. We saw how to use function application with the `.apply` method, and we cautioned against its use when a vectorized solution is available. We saw how to use the `.where` method to replace values based on a boolean series. I prefer `.case_when` instead for flexibility and readability. We discussed various ways to deal with missing data. We saw that we could sort both the values and the index. We can replace data, and we can bin data. These operations will come in useful as you begin to analyze data.

11.15 Exercises

With a dataset of your choice:

1. Create a series from a numeric column that has the value of `'high'` if it is equal to or above the mean and `'low'` if it is below the mean using `.apply`.
2. Create a series from a numeric column that has the value of `'high'` if it is equal to or above the mean and `'low'` if it is below the mean using `.case_when`.
3. Time the differences between the previous two solutions to see which is faster.
4. Replace the missing values of a numeric series with the median value.
5. Clip the values of a numeric series between to 10th and 90th percentiles.
6. Using a categorical column, replace any value that is not in the top 5 most frequent values with `'Other'`.
7. Using a categorical column, replace any value that is not in the top 10 most frequent values with `'Other'`.
8. Make a function that takes a categorical series and a number (`n`) and returns a replace series that replaces any value not in the top `n` most frequent values with `'Other'`.
9. Using a numeric column, bin it into 10 groups with the same width.
10. Using a numeric column, bin it into 10 groups that have equal-sized bins.

Chapter 12

Indexing Operations

Indexing is an overloaded term in the pandas world. A series and a dataframe have an index (the labels down the left side for each row). Also, both types support the Python indexing operator ([]). But that is not all! They both have attributes (`.loc` and `.iloc`) that you can index against (using the Python indexing operator). This section will address changing the index and accessing parts of a series with the indexing operators.

12.1 Prepping the Data and Renaming the Index

To ease explaining the various operations, I will take the automobile mileage data series with the city miles per gallon values and insert each car's make as the index. This is because many operations work on the index position while others work on the index label. This might initially seem perplexing, especially when both indices and labels are integers, but it becomes more apparent if the index has string labels.

We will use the `.rename` method to change the index labels. We can pass in a dictionary to map the previous index label to the new label:

```
>>> city2 = city_mpg.rename(make.to_dict())
>>> city2
Alfa Romeo    19
Ferrari        9
Dodge         23
              ..
Subaru        18
Subaru        18
Subaru        16
Name: city08, Length: 41144, dtype: int64[pyarrow]
```

To view the index, you can access the `.index` attribute:

```
>>> city2.index
Index(['Alfa Romeo', 'Ferrari', 'Dodge', 'Dodge', 'Subaru', 'Subaru',
```

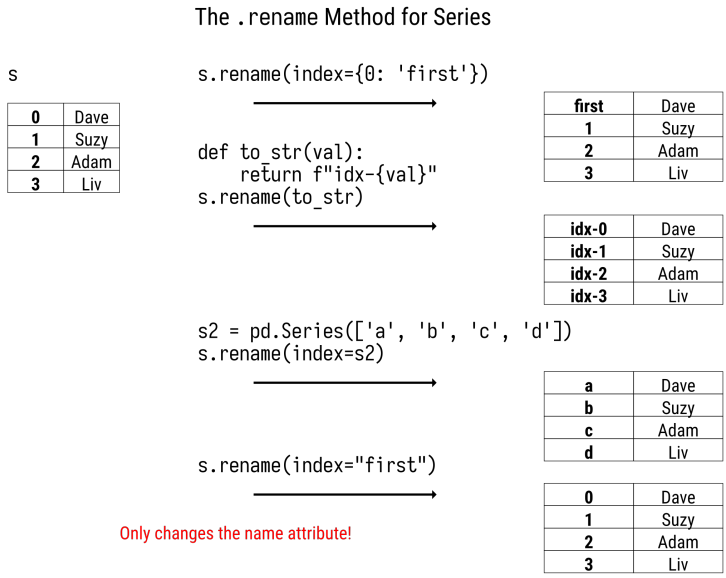


Figure 12.1: The `.rename` method will return a new series with the original values but new index labels. If you pass in a scalar value, it will change the `.name` attribute of the series on the new series it returns, leaving the index intact.

```
 'Toyota', 'Toyota', 'Toyota',
 ...
 'Saab', 'Saturn', 'Saturn', 'Saturn', 'Saturn', 'Subaru', 'Subaru',
 'Subaru', 'Subaru', 'Subaru'],
dtype='object', length=41144)
```

The `.rename` method also accepts a series, a scalar, or a function that takes an old label and returns a new label or a sequence. When we pass in a series, and the index values are the same, the values from the series that we passed in are used as the index:

```
>>> city2 = city_mpg.rename(make)
>>> city2
Alfa Romeo    19
Ferrari        9
Dodge         23
              ..
Subaru        18
Subaru        18
```

```
Subaru        16
Name: city08, Length: 41144, dtype: int64[pyarrow]
```

Careful though! If you pass a scalar value (a single string) into `.rename`, the index will stay the same, but the `.name` attribute of the series will update:

```
>>> city2.rename('citympg')
Alfa Romeo    19
Ferrari        9
Dodge         23
              ..
Subaru        18
Subaru        18
Subaru        16
Name: citympg, Length: 41144, dtype: int64[pyarrow]
```

12.2 Resetting the Index

Sometimes, you need a unique index to perform an operation. If you want to set the index to monotonic increasing, and therefore unique integers starting at zero, you can use the `.reset_index` method. By default, this method will return a dataframe, moving the current index into a new column:

```
>>> print(city2.reset_index())
            index  city08
0      Alfa Romeo      19
1         Ferrari       9
2           Dodge      23
...           ...     ...
41141      Subaru      18
41142      Subaru      18
41143      Subaru      16

[41144 rows x 2 columns]
```

To drop the current index and return a `Series`, use the `drop=True` parameter:

```
>>> city2.reset_index(drop=True)
0        19
1         9
2        23
         ..
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int64[pyarrow]
```

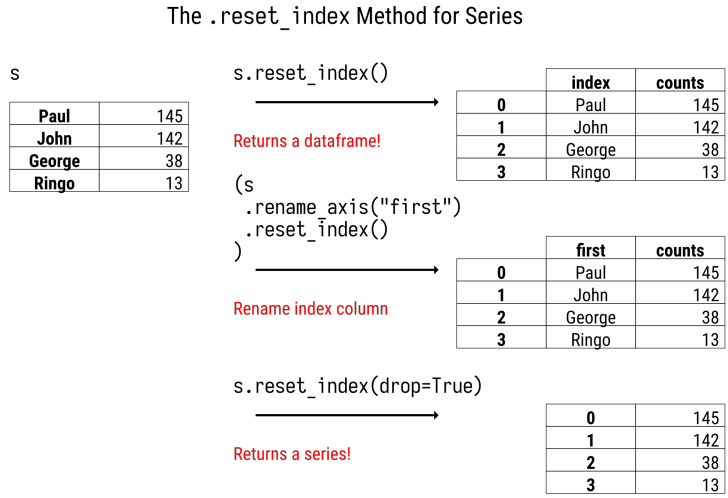


Figure 12.2: The `.reset_index` method will return a dataframe or a series with the index changed to a monotonically increasing index.

Note that you can sort the values and the index with `.sort_values` and `.sort_index` respectively. Because those keep the same index but rearrange the order, they do not impact operations that align on the index.

12.3 The .loc Attribute

Let's shift the focus onto pulling data out by using indexing operators. You can index directly on a series object, but I recommend not doing it. I prefer to be a little more explicit. I would index off of the `.loc` or `.iloc` attributes.

The `.loc` attribute deals with index *labels*. It allows you to pull out pieces of the series. You can pass in the following into an index operation on `.loc`:

- A scalar value of one of the index labels
- A list of index labels.
- A slice of labels (closed interval so it includes the stop value).
- An index.
- A boolean array (same index labels as the series, but with `True` or `False` values.
- A function that accepts a series and returns one of the above.

If you pass in a scalar with the label of an index, you need to be careful. If there are duplicate labels in the index, it will return a series, but if there is only one value for that label, it will return a scalar. In the example below

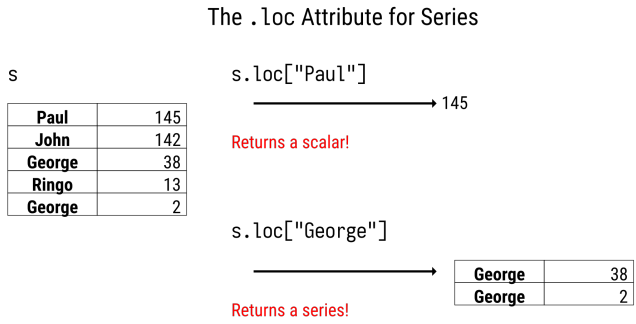


Figure 12.3: Indexing off of the `.loc` attribute will return a series if the index label is duplicated.

`'Subaru'` has multiple index entries, but `'Fisker'` only has one. Note the types they return. One returns a series, while the other returns a scalar:

```
>>> city2.loc['Subaru']
Subaru    17
Subaru    21
Subaru    22
          ..
Subaru    18
Subaru    18
Subaru    16
Name: city08, Length: 885, dtype: int64[pyarrow]

>>> city2.loc['Fisker']
20
```

If you want to guarantee that a series is returned, pass in a list rather than passing in a scalar value. It can be a list with a single value or a list with multiple values:

```
>>> city2.loc[['Fisker']]
Fisker    20
Name: city08, dtype: int64[pyarrow]

>>> city2.loc[['Ferrari', 'Lamborghini']]
Ferrari         9
Ferrari        12
Ferrari        11
               ..
```

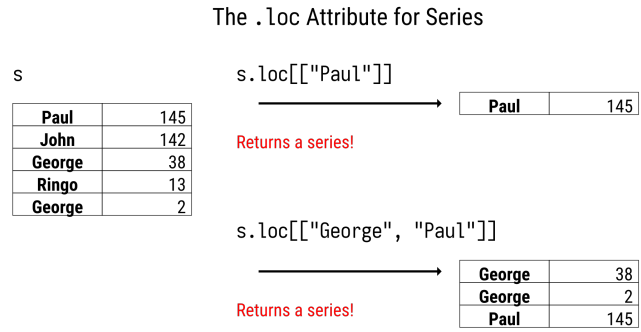


Figure 12.4: Indexing off of the `.loc` attribute with a list of index names will return a series.

```
Lamborghini     8
Lamborghini     8
Lamborghini     8
Name: city08, Length: 357, dtype: int64[pyarrow]
```

This next option might seem weird if you are used to typical list slicing with Python. When we slice sequences, we use an integer index position. However, with `.loc`, we can use a slice with string values. You should know that you must first sort the index if you are slicing with duplicate index labels. Otherwise, you will see a `KeyError`:

```
>>> city2.loc['Ferrari':'Lamborghini']
Traceback (most recent call last):
  ...
KeyError: "Cannot get left slice bound for non-unique label: 'Ferrari'"

>>> city2.sort_index().loc['Ferrari':'Lamborghini']
Ferrari        10
Ferrari        13
Ferrari        13
               ..
Lamborghini     8
Lamborghini    13
Lamborghini     8
Name: city08, Length: 11210, dtype: int64[pyarrow]
```

Note that when slicing with `.loc`, it follows the *closed interval*. The closed interval includes both the start index and the final index. This behavior differs from the *half-open interval* found in Python's slicing behavior for strings and

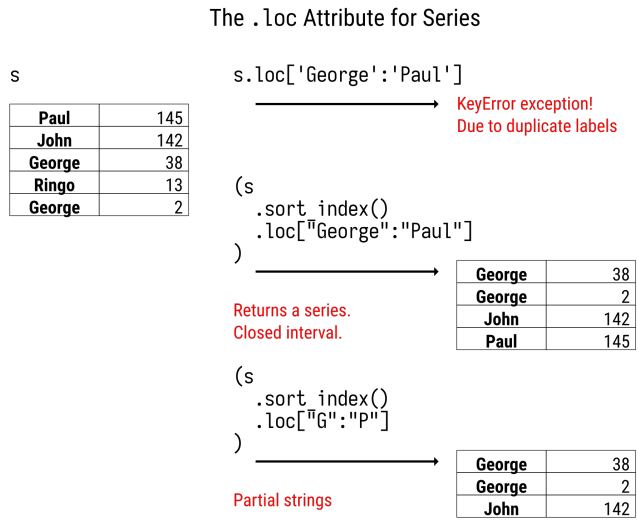


Figure 12.5: Indexing off of the `.loc` attribute with a slice will return a series. Note that slicing with labels is *closed* and includes the end value.

lists (which includes the start index, going up to but not including the final index). We will see that the `.iloc` attribute supports slicing with the half-open interval as well.

There is another trick up the label slicing sleeve. If you have a sorted index, you can slice with strings that are not actual labels. For example, if I wanted all the labels in `city2` that start with *F* and go up to those index labels that also start with *G H I*, and including precisely `'J'`, but not anything else that happens to start with *J*, I could do the following. Note that no label has the literal value of either the start or stop, so these are not included:

```
>>> city2.sort_index().loc["F":"J"]
Federal Coach    15
Federal Coach    13
Federal Coach    13
                 ..
Isuzu            15
Isuzu            27
Isuzu            18
Name: city08, Length: 9040, dtype: int64[pyarrow]
```

You can also pass in a pandas `Index` to `.loc`. This is useful when you have parallel pandas objects with the same index. If you have already filtered

one of them, you can get the other to conform by passing its index into `.loc`. However, you need to be aware of duplicate index labels.

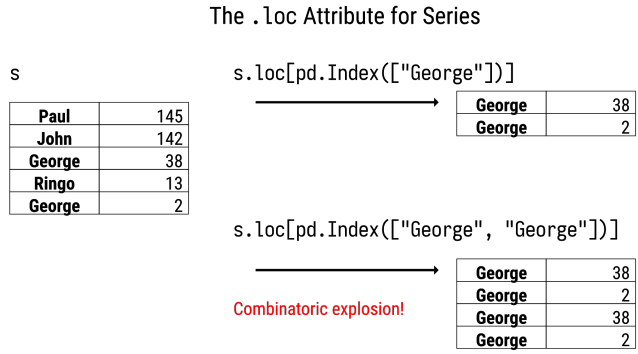


Figure 12.6: The `.loc` attribute will accept an `Index` in an indexing operation (no pun intended). Be careful with duplicate index labels, as that may lead to a combinatoric explosion.

An example will make this clearer. Our `city2` series has many duplicated index labels. If we index into `.loc` with a simple `Index` with only `'Dodge'` in it, we get back every value for the label. Using an index is useful if we want to align a series to a new index:

```
>>> idx = pd.Index(['Dodge'])
>>> city2.loc[idx]
Dodge    23
Dodge    10
Dodge    12
         ..
Dodge    14
Dodge    14
Dodge    11
Name: city08, Length: 2583, dtype: int64[pyarrow]
```

However, if we duplicate `'Dodge'` in the `Index`, the previous operation has twice as many values, a combinatoric explosion:

```
>>> idx = pd.Index(['Dodge', 'Dodge'])
>>> city2.loc[idx]
Dodge    23
Dodge    10
Dodge    12
```

```
          ..
Dodge    14
Dodge    14
Dodge    11
Name: city08, Length: 5166, dtype: int64[pyarrow]
```

You can also pass in a boolean array to `.loc`. Remember that a boolean array is a series with the same index labels as the series (or dataframe) that you are manipulating that has boolean values. If you do an indexing operation off of `.loc` with a boolean array, it will return only the values where the boolean array was true.

In the example below, we will filter out values where the city mileage is above 50. First, I will create a boolean array and store it in a variable called `mask`:

```
>>> mask = city2 > 50
>>> mask
Alfa Romeo    False
Ferrari       False
Dodge         False
              ...
Subaru        False
Subaru        False
Subaru        False
Name: city08, Length: 41144, dtype: bool[pyarrow]
```

Then I will use that boolean array in an index operation off of `.loc`:

```
>>> city2.loc[mask]
Nissan     81
Toyota     81
Toyota     81
          ...
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64[pyarrow]
```

You can see that there were only 236 entries with mileage above 50.

Note

The `.loc` attribute supports pulling out values by specifying the index name and providing a boolean array. You can extract almost any data from a series using a boolean array. This becomes even more powerful

when you use it with dataframes and combine logic based on different columns.

Finally, you can use a function with the `.loc` attribute. This will come in handy when chaining operations. After multiple operations, the intermediate object you are operating on might have a completely different index than the original object. By using a function, you will have access to the intermediate series and be able to create a row filter based on it. This might seem like overkill for series objects, but it comes in handy with dataframes.

Here is an example. I have a series with old pricing information from last year. I know there was a 10% increase in cost during that time. If I want to find all of the new prices that are above $3 after inflation, we can chain these operations together:

```
>>> cost = pd.Series([1.00, 2.25, 3.99, .99, 2.79],
...     index=['Gum', 'Cookie', 'Melon', 'Roll', 'Carrots'])
>>> inflation = 1.10
>>> (cost
...     .mul(inflation)
...     .loc[lambda s_: s_ > 3]
... )
Melon      4.389
Carrots    3.069
dtype: float64
```

If I calculate the boolean array before taking into account the inflation (i.e., using the old series instead of the chained intermediate values) I got the wrong answer:

```
>>> cost = pd.Series([1.00, 2.25, 3.99, .99, 2.79],
...    index=['Gum', 'Cookie', 'Melon', 'Roll', 'Carrots'])
>>> inflation = 1.10
>>> mask = cost > 3
>>> (cost
...     .mul(inflation)
...     .loc[mask]
... )
Melon    4.389
dtype: float64
```

Note

The correct example above uses a *lambda function*. This is a syntax that Python provides for making a function in a single line of code. We could have defined a regular Python function instead. The following are equivalent:

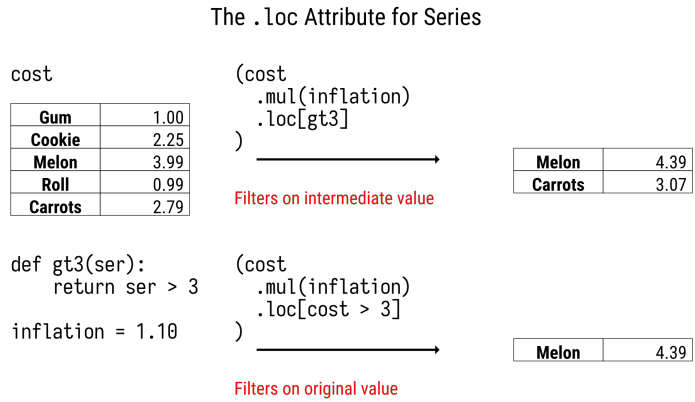


Figure 12.7: The `.loc` attribute will accept a function. This function accepts the current series it was called on and should return a scalar, list, slice, or index.

```
>>> def gt3(s):
...     return s > 3

>>> gt3 = lambda s: s > 3
```

The basic rule for creating a lambda function is that you use the `lambda` statement followed by the parameters (`s` in this case). The parameters are followed by a colon and whatever you want to return. Note that the lambda function has an implicit `return` statement. Also, you can only put an *expression* in it. You cannot have a *statement*. So, it is limited to a single line of code.

12.4 The .iloc Attribute

The series also supports indexing off of the `.iloc` attribute. This attribute is analogous to `.loc` but with a few differences. When we slice off of this attribute, we pull out items by index position. The `.iloc` attribute supports indexing with the following:

- A scalar index position (an integer)
- A list of index positions
- A slice of positions (half-open interval, which does not include stop value).
- A NumPy array (or Python list) of boolean values.
- A function that accepts a series and returns one of the above.

The examples below will pull out the first and last values by slicing off `.iloc` with a scalar. Note that because index positions are unique, we will always get the scalar value when indexing with `.iloc` at a position:

```
>>> city2.iloc[0]
19
```

We can also use negative indexing to pull out the last value:

```
>>> city2.iloc[-1]
16
```

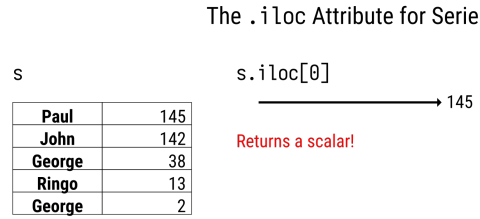


Figure 12.8: Indexing off of the `.iloc` attribute will return a scalar by location in the series.

If we want to return a series object, we can index it with a list of positions. This can be a list with a single index or multiple index values. The following code will return a series with the first, second, and last values:

```
>>> city2.iloc[[0,1,-1]]
Alfa Romeo    19
Ferrari        9
Subaru        16
Name: city08, dtype: int64
```

We can also use slices with `.iloc`. In this case, slices behave as they do in Python lists and follow the half-open interval. That is, they include the first index and go up to but do not include the last index. If we want to return the first five items, we can use the `.head` method or the following code, which takes index positions starting at 0 and includes 1, 2, 3, and 4 but does not include 5:

```
>>> city2.iloc[0:5]
Alfa Romeo    19
Ferrari        9
```

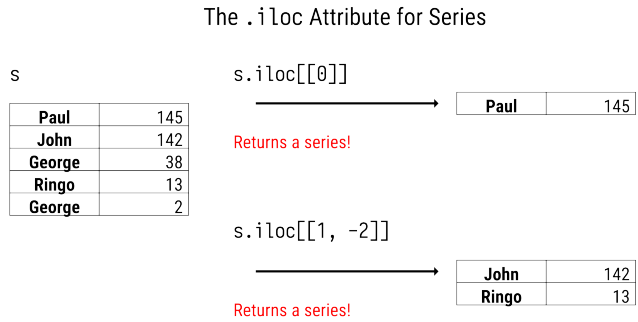


Figure 12.9: Indexing off of the `.iloc` attribute with a list will return a series of values at the locations in the list.

```
Dodge          23
Dodge          10
Subaru         17
Name: city08, dtype: int64[pyarrow]
```

To return the last eight values, you could use the following code. In Python, negative index positions start counting from the end. The position -1 is the last index, -2 is the second to last, etc. If we do not include a final index, the slice goes up to the end:

```
>>> city2.iloc[-8:]
Saturn    21
Saturn    24
Saturn    21
          ..
Subaru    18
Subaru    18
Subaru    16
Name: city08, Length: 8, dtype: int64[pyarrow]
```

You can also use a NumPy array of booleans (or a Python list), but if you use what we call a boolean array (a pandas series with booleans), this will fail:

```
>>> mask = city2 > 50
>>> city2.iloc[mask]
Traceback (most recent call last):
  ...
ValueError: iLocation based boolean indexing cannot use an indexable as a mask
```

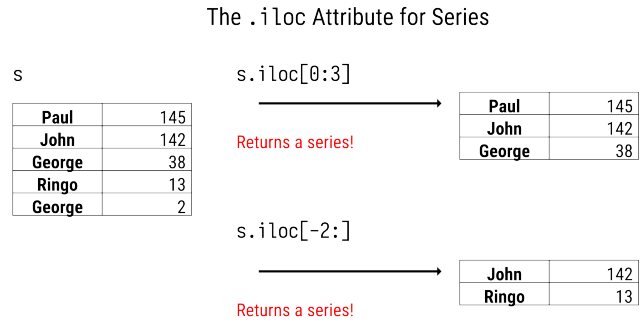


Figure 12.10: Indexing off of the `.iloc` attribute with a slice uses the *half-open interval* of positions.

We can convert the mask to a NumPy array or Python list, and the `.iloc` selection will work:

```
>>> mask = city2 > 50
>>> city2.iloc[mask.to_numpy()]
Nissan     81
Toyota     81
Toyota     81
          ...
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64[pyarrow]

>>> city2.iloc[list(mask)]
Nissan     81
Toyota     81
Toyota     81
          ...
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64[pyarrow]
```

Finally, you can pass in a function to `.iloc` that accepts the series on which it is called. This function can return any of the above options for `.iloc`. I have not found a real-life use case for passing in a function. Because I would use such functionality to pull out values on the result of a chained method call, using `.loc` is preferred as it accepts a boolean array.

Ok, we just spent a lot of time discussing `.iloc` and now I'm going to burst your bubble and advise you not to use it for production code. The `.loc` attribute is more explicit and makes your code easier to read. If you really do need to pull out values that are at the start or end of the series, then use the `.head` and `.tail` methods, which we will discuss next.

12.5 Heads and Tails

The `.head` and `.tail` methods help pull out values at the start or end of the series, respectively. These methods are used to inspect a chunk of the data quickly.

I would be careful about assuming that a series's first values represent the entire series. I have found that the first values of a series might be incomplete as more columns are added over time, or they might even be test values. I prefer the `.sample` method to understand better what data is in the series.

The following code inspects the three values at the start and end:

```
>>> city2.head(3)
Alfa Romeo    19
Ferrari        9
Dodge         23
Name: city08, dtype: int64[pyarrow]

>>> city2.tail(3)
Subaru    18
Subaru    18
Subaru    16
Name: city08, dtype: int64[pyarrow]
```

12.6 Sampling

While the previous two methods, `.head` and `.tail` allow us to inspect the data, sampling the data can be a better choice. The first few data entries may often be incomplete, test data, or not representative of all values. Sampling might be a better option. The code below randomly pulls out six values:

```
>>> city2.sample(6, random_state=42)
Volvo         16
Mitsubishi    19
Buick         27
Jeep          15
Land Rover    13
Saab          17
Name: city08, dtype: int64[pyarrow]
```

12.7 Filtering Index Values

The `.filter` method will filter index labels by exact match, substring, or regular expression. These are controlled with the mutually exclusive `items`, `like`, and `regex` parameters, respectively.

Note that exact match (with `items`) fails with duplicate index labels:

```
>>> city2.filter(items=['Ford', 'Subaru'])
Traceback (most recent call last):
  ...
ValueError: cannot reindex from a duplicate axis
```

Using `like`, we can do substring matches:

```
>>> city2.filter(like='rd')
Ford    18
Ford    16
Ford    17
        ..
Ford    21
Ford    18
Ford    19
Name: city08, Length: 3371, dtype: int64[pyarrow]
```

We can also specify a regular expression to match against index values:

```
>>> city2.filter(regex='(Ford)|(Subaru)')
Subaru    17
Subaru    21
Subaru    22
          ..
Subaru    18
Subaru    18
Subaru    16
Name: city08, Length: 4256, dtype: int64[pyarrow]
```

12.8 Reindexing

The `.reindex` method lets you pull out values by index label. It will *conform* the series or return a series with the order of the index labels provided. Unlike `.loc` and `.filter`, you can pass in labels that are not in the index, and it will not throw an error. Rather it will insert missing values. However, the `.reindex` method does not like duplicate index labels in the series and will throw an error if you have them:

```
>>> city2.reindex(['Missing', 'Ford'])
Traceback (most recent call last):
```

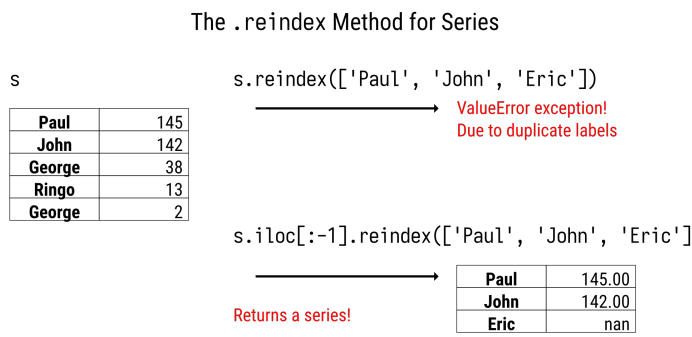


Figure 12.11: The .reindex method will *conform* an index to a new index.

```
  ...
ValueError: cannot reindex from a duplicate axis
```

Note that even though this will not work with duplicate index labels in a series, you can pass in the index label multiple times in the call and it will repeat that index (city has a numeric index that is unique):

```
>>> city_mpg.reindex([0,0, 10, 20, 2_000_000])
0            19
0            19
10           23
20           14
2000000    <NA>
Name: city08, dtype: int64[pyarrow]
```

This method is a lifesaver if you have series that have portions of index labels that are the same and you want one to have the index of the other:

```
>>> s1 = pd.Series([10,20,30], index=['a', 'b', 'c'])
>>> s2 = pd.Series([15,25,35], index=['b', 'c', 'd'])

>>> s2
b    15
c    25
d    35
dtype: int64

>>> s2.reindex(s1.index)
a     NaN
```

```
b    15.0
c    25.0
dtype: float64
```

Table 12.1: Indexing operation, methods, and properties

Method	Description
`s.rename(index=None, *, level=None, errors='ignore')`	Return a series with updated `.name` attribute if `index` is a scalar. If `index` is a function series, or dictionary, return a series with updated index mapped from input (functions work on index name, series and dictionaries map the index name to a new value).
`s.index`	Returns the index of the series.
`s.reset_index(level=None, drop=False, name=None, inplace=False)`	Return a dataframe (or series when `drop=True`) with a new integer index.
`s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)`	Return a series with the index sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates the location of NaNs and may be `'first'` or `'last'`.
`s.loc[idx]`	Slice series by names. `idx` can be a scalar (pull out value at that name), list of names, slice with names (including end position), a boolean array, an index, or a function (that accepts the series and returns one of the previous items).
`s.iloc[idx]`	Slice series by index position. `idx` can be a scalar (pull out value at that index), list of indices, slice with index positions (half-open including start but not end index), a list of booleans, or a function (that accepts the series and returns one of the previous items).
`s.head(n=5)`	Return a series with the first `n` values.
`s.tail(n=5)`	Return a series with the last `n` values.

Method	Description
`s.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)`	Return a series with `n` random entries. Can also specify a fraction with `frac` (if `frac` > 1 specify `replace=True`).
`s.filter(items=None, like=None, regex=None, axis=None)`	Return a series with index values from `items` list, matching `like` substring, or when `regex` (regular expression) search matches.
`s.reindex(index=None, method=None, copy=True, level=None, limit=None, tolerance=None)`	Return a series with a conformed index.

12.9 Summary

The index is a fundamental structure of pandas. Both a series and a dataframe have an index. To get the most out of pandas, you must understand how to manipulate the index. We often have two pandas objects, and if we want to perform operations on them, we might need them to have similar index values. For example, pandas will align the index values and add the corresponding values for each index entry when we add a series to another series.

We also saw that we could index from `.loc` and `.iloc` to pull out values by name and position. You will often use both attributes when dealing with pandas dataframes and series. However, I recommend to make your code easier to read and not using `.iloc`.

12.10 Exercises

With a dataset of your choice:

1. Inspect the index.
2. Sort the index.
3. Set the index to monotonically increasing integers starting from 0.
4. Set the index to monotonically increasing integers starting from 0, then convert these to the string version. Save this a `s2`.
5. Using `s2`, pull out the first five entries.
6. Using `s2`, pull out the last five entries.
7. Using `s2`, pull out one hundred entries starting at index position 10.
8. Using `s2`, create a series with values with index entries `'20'`, `'10'`, and `'2'`.

Chapter 13

String Manipulation

In this chapter, we will explore series that have string data. String data is commonly used to hold free-form, semi-structured, categorical, and data that should have another type (typically numeric or datetime). We will look at the everyday operations of textual data.

13.1 Strings and Objects

Before pandas 2.0, if you stored strings in a series, the underlying type of the series was `object`. This is unfortunate as the `object` type can be used for other series with Python types (such as a list, a dictionary, or a custom class). Also, the `object` type is used for mixed types. If you have a series with numbers and strings, the type is also `object`.

Pandas 2.0, with its integration with PyArrow, introduced the new `pd.ArrowDtype(pa.string())` type. In addition to being more explicit than `object`, it supports missing values that are not `NaN`.

Note

Again, when casting to strings, dont' use `'string[pyarrow]'` for casting to PyArrow types, use `pd.ArrowDtype(pa.string())` instead. The former was introduced during pandas 1.5 and doesn't always return PyArrow types for operations. Somewhat confusingly, it reports the type for the latter as `string[pyarrow]`.

The *make* column has the `string[pyarrow]` type because we loaded the CSV file with the `dtypes_backend` parameter set to `'pyarrow'`:

```
>>> make
0        Alfa Romeo
1           Ferrari
2             Dodge
              ...
41141         Subaru
```

```
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

We can use the `.astype` method to convert the string columns between the `object` (`str`) and `string[pyarrow]` types. Here we roundtrip the *make* column from `string[pyarrow]` to `object` and back to `string[pyarrow]`:

```
>>> make.astype(str) # go old school
0        Alfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: object

>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> make.astype(str).astype(string_pa)
0        Alfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

A few differences exist between the `'string[pyarrow]'` type and strings stored as `object` types. The PyArrow version uses less memory and is quicker, yet lacks some support for the older methods. Otherwise, the behavior is similar.

13.2 Categorical Strings

If you have low cardinality string columns, consider using a categorical type for them. You will have access to many of the same string manipulation methods (though some are not available in this case). The main advantage here is memory savings and performance improvements, as the operations need to be done only on the individual categories and not each value in the series:

```
>>> make.astype('category')
0        Alfa Romeo
```

```
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): [AM General, ASC Incorporated,
  Acura, Alfa Romeo, ..., Volvo, Wallace Environmental,
  Yugo, smart]
```

We will dive into categories later.

13.3 The .str Accessor

The `object`, `pd.ArrowDtype(pa.string())`, and `'category'` types have a `.str` accessor that provides string manipulation methods. Most of these methods are modeled after the Python string methods. If you are comfortable with the Python string methods, many pandas variants should be second nature. Here is the Python string method `.lower`:

```
>>> 'Ford'.lower()
'ford'
```

In Python, I can convert a string to lowercase with the `.lower` method:

```
>>> "Hello".lower()
'hello'
```

There is a corresponding pandas method `.lower` that works on a series. It is located on the `.str` accessor:

```
>>> make.str.lower()
0        alfa romeo
1           ferrari
2             dodge
            ...
41141        subaru
41142        subaru
41143        subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

Here is another example of the Python `.find` method:

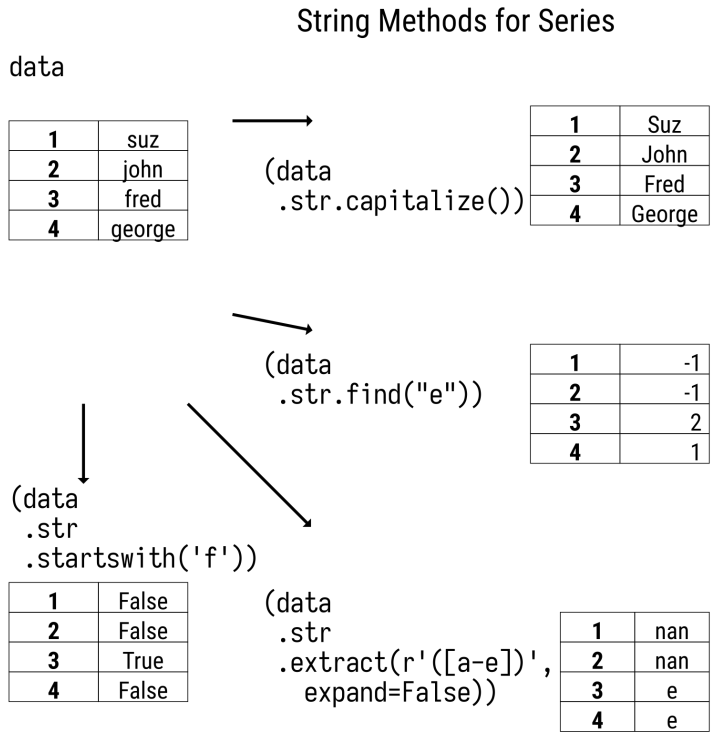


Figure 13.1: The `.str` accessor will allow you to manipulate strings in a series much like you can manipulate Python strings.

```
>>> 'Alfa Romeo'.find('A')
0
```

And here is the pandas version:

```
>>> make.str.find('A')
0         0
1        -1
2        -1
         ..
41141    -1
41142    -1
41143    -1
Name: make, Length: 41144, dtype: int32[pyarrow]
```

Many methods are common to both strings and pandas series. They are found in a table later in this chapter.

13.4 Searching

Several methods leverage regular expressions to search, replace, and split strings. This book will not go deep into regular expressions as books are solely devoted to that subject.

To find all of the non-alphabetic characters (disregarding space), you could use the `.extract`. As of pandas 2.0.2 [1], this has a new interface, and the traditional legacy call with a simple regular expression will not work.

```
>>> print(make.str.extract(r'([^a-z A-Z])'))
Traceback (most recent call last)
  ...
ValueError: pat='([^a-z A-Z])' must contain a symbolic group name.
```

We can work around this by converting the series to the `object` type or adding a regular expression named capture group:

```
>>> print(make.str.extract(r'(?P<non_alpha>[^a-z A-Z])'))
      non_alpha
0          <NA>
1          <NA>
2          <NA>
...         ...
41141      <NA>
41142      <NA>
41143      <NA>

[41144 rows x 1 columns]
```

[1] https://github.com/pandas-dev/pandas/issues/56268

This returns a dataframe with mostly missing values and, by inspection, is not very useful. If we collapse it into a series (with the parameter `expand=False`), we can chain the `.value_counts` method to view the count of non-missing values:

```
>>> (make
...    .str.extract(r'(?P<non_alpha>[^a-z A-Z])', expand=False)
...    .value_counts()
... )
non_alpha
-    1727
.      46
,       9
Name: count, dtype: int64[pyarrow]
```

Note

I like to use a similar technique to the above to search for non-numeric characters that pop up from reading a CSV file. If a column in a CSV file contains non-numeric characters, use the following code to find them:

```
(col
   .str.extract(r'(?P<non_num>[^0-9.])', expand=False)
   .value_counts()
)
```

After diagnosing the bad actors, you can replace them or drop them and convert the column to the appropriate numeric type.

13.5 Splitting

When dealing with survey data, you may come across binned numeric values. The survey probably had a drop-down of different ranges. It might have said, what is your age? And have options for *20-29*, *30-39*, *40-49*, etc. Those survey results come in as strings because pandas cannot handle the dash. Hence, we cannot perform math operations on the ages, like calculating the minimum or mean values.

Here is an example of pulling out the value before the dash and converting it to a number using the `.split` method:

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> age = pd.Series(['0-10', '11-15', '11-15', '61-65', '46-50'],
...    dtype=string_pa)
>>> age
0     0-10
```

```
1    11-15
2    11-15
3    61-65
4    46-50
dtype: string[pyarrow]
```

If we call `.split` on the series, we get back a series that has lists in it:

```
>>> age.str.split('-')
0     ['0' '10']
1    ['11' '15']
2    ['11' '15']
3    ['61' '65']
4    ['46' '50']
dtype: list<item: string>[pyarrow]
```

A series with a Python list makes it hard to manipulate the data. We can provide the `expand=True` parameter to retrieve a dataframe to remedy that. If I just wanted to use the first column as an age value, I could chain together an `.iloc` operation to pull out the first column and then convert the strings to integers with the `.astype` method:

```
>>> (age
...    .str.split('-', expand=True)
...    .iloc[:,0]
...    .astype('int8[pyarrow]')
... )
0     0
1    11
2    11
3    61
4    46
Name: 0, dtype: string[pyarrow]
```

This will bias our ages towards the low side. If you want to use the tail end of the binned value, you can use the `.slice` method or just do a slice operation off of `.str`:

```
>>> (age
...    .str.slice(-2)
...    .astype('int8[pyarrow]')
... )
0    10
1    15
2    15
3    65
4    50
dtype: int8[pyarrow]
```

```
>>> (age
...   .str[-2:]
...   .astype('int8[pyarrow]')
... )
0    10
1    15
2    15
3    65
4    50
dtype: int8[pyarrow]
```

We can take the average of the bin ranges using this code:

```
>>> (age
...   .str.split('-', expand=True)
...   .astype('int8[pyarrow]')
...   .mean(axis='columns')
... )
0     5.0
1    13.0
2    13.0
3    63.0
4    48.0
dtype: double[pyarrow]
```

We have not dived into dataframes, but in short, the above will convert the columns to numbers, then apply the `.mean` method across each row (manipulating across the row is accomplished with the `axis='columns'` parameter). This will make more sense when we discuss the dataframe axis.

Finally, if you wanted to get a random number between the ranges, you could do this:

```
>>> import random
>>> def between(row):
...     return random.randint(*row.values)

>>> (age
...   .str.split('-', expand=True)
...   .astype(int)
...   .apply(between, axis='columns')
... )
0     7
1    14
2    12
3    65
4    46
dtype: int64
```

13.6 Removing Apply

I would probably write this with the following implementation. It uses a dataframe, so you might want to revisit it after exploring dataframes. I'm showing it because it is 60x faster on my machine when I run it on a series with 150,000 values.

It splits the age column into two columns, renames the columns to make them more meaningful, converts the columns to integers, creates a new random column, and calculates the age by Add the age range's low end to the random number multiplied by the range.

```
>>> import numpy as np
>>> print(age
...  .str.split('-', expand=True)
...  .rename(columns={0:'lower', 1:'upper'})
...  .astype('int8[pyarrow]')
...  .assign(rand=np.random.rand(len(age)),
...          age=lambda df_: (df_.lower + (df_.rand *
...                (df_.upper - df_.lower)))
...                .astype('int8[pyarrow]', errors='ignore')
...   )
... )
   lower  upper      rand        age
0      0     10  0.257498   2.574976
1     11     15  0.521265   13.08506
2     11     15  0.408589  12.634357
3     61     65  0.608659  63.434635
4     46     50  0.487223   47.94889
```

13.7 Optimizing with NumPy

Let's see if we can speed up this operation using a vectorized NumPy operation. In the previous example, we called the `randint` function once for each row. That can be an expensive proposition if we have a lot of rows.

In this case, the NumPy library provides a vectorized alternative, `np.random.randint`. We need to pass in the left column for the first parameter and the right column for the second parameter:

```
>>> import numpy as np
>>> (age
...  .str.split('-', expand=True)
...  .astype(int)
...  .pipe(lambda df_: pd.Series(np.random.randint(df_.iloc[:,0],
...                                df_.iloc[:,1]), index=df_.index)
...       )
... )
```

```
0     1
1    13
2    13
3    62
4    48
dtype: int64
```

Let's benchmark these with 100,000 strings:

```
>>> age_100k = (age
...    .sample(100_000, replace=True, random_state=42)
...    .reset_index(drop=True)
... )
```

Here's the timing for the `.apply` code:

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype('int8[pyarrow]')
...    .apply(between, axis='columns')
... )
1.84 s ± 19.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Here's the timing for the vectorized NumPy code:

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype('int8[pyarrow]')
...    .pipe(lambda df_: pd.Series(np.random.randint(df_.iloc[:,0],
...                                  df_.iloc[:,1]), index=df_.index)
...        )
... )
32.1 ms ± 423 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

It appears that the NumPy solution is about sixty times faster for this dataset.

```
>>> %%timeit
>>> (age_100k
...  .str.split('-', expand=True)
...  .rename(columns={0:'lower', 1:'upper'})
...  .astype('int8[pyarrow]')
...  .assign(rand=np.random.rand(len(age_100k)),
...          age=lambda df_: (df_.lower + (df_.rand *
...                (df_.upper - df_.lower)))
```

```
...                 .astype('int8[pyarrow]', errors='ignore')
...   )
... )
32.2 ms ± 469 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

13.8 Optimizing .apply with Cython

The previous example uses `.apply`, and by now, you should know that I'm generally against that method because it is slow. Let's divert from strings for a minute and consider making the `.apply` operation quicker using Cython.

Cython is a superset of Python that can compile to native code. To enable it in Jupyter, you will need make sure Cython is installed and run the following cell magic:

```
%load_ext Cython
```

Then, you can define functions with Cython. I'm going to "cythonize" the `between` function as a first step:

```
%%cython
import random
def between_cy(row):
    return random.randint(*row.values)
```

When I benchmark this, it is no faster than my current code. You can get a speed increase if you add types to Cython code. I'll try that here:

```
%%cython
import random
import numpy as np
def between_cy3(x: np.int64, y: np.int64) -> np.int64:
    return random.randint(x, y)
```

Because I'm calling `.apply` across the columns axis, the `between` function needs to work on a row (converted into a series) of data. I'm going to use a `lambda` to pull apart the series and then call `between_cy3`:

```
>>> (age
...   .str.split('-', expand=True)
...   .astype(int)
...   .apply(lambda row: between_cy3(row[0], row[1]), axis=1)
... )
0     1
1    12
2    14
3    61
4    47
dtype: int64
```

Let's benchmark this with the larger dataset:

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype(int)
...    .apply(lambda row: between_cy3(row[0], row[1]), axis=1)
... )
342 ms ± 3.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

I'm still not getting much of a boost. Using `prun`, I see that I'm spending a good deal of time doing index operations (`row[0]` and `row[1]`):

```
%prun -l 10 (age_100k.str.split('-', expand=True).astype(int) \
       .apply(lambda row: between_cy3(row[0], row[1]), axis=1))
```

I'm going to change the plan of attack and send in two NumPy arrays and return a NumPy array:

```
%%cython

import numpy as np
import random
def apply_between_cy4(x: np.ndarray[int],
                      y: np.ndarray[int] ) -> np.ndarray[int]:
    res:  np.ndarray[int] = np.empty(len(x), dtype='int32')
    i : int
    for i in range(len(x)):
        res[i] = random.randint(x[i], y[i])
    return res
```

I can run this with the following code, and it runs 3x faster on a dataset with 100,000 values:

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype('int8[pyarrow]')
...    .pipe(lambda df_: apply_between_cy4(
...        df_.iloc[:, 0].to_numpy(dtype='int32'),
...        df_.iloc[:, 1].to_numpy(dtype='int32')))
... )
138 ms ± 612 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Let's try once more using the `np.random.randint` function instead of the Python standard library function:

```
%%cython
import numpy as np
import random
def apply_between_cy5(x: np.ndarray[int],
                      y: np.ndarray[int]) -> np.ndarray[long]:
    return np.random.randint(x,y)
```

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype('int8[pyarrow]')
...    .pipe(lambda df_: apply_between_cy5(
...       df_.iloc[:, 0].to_numpy(dtype='int32'),
...       df_.iloc[:, 1].to_numpy(dtype='int32')))
... )
31.9 ms ± 586 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

13.9 Optimizing .apply with Numba

Another mechanism to optimize pandas code is with the Numba library. Numba is an open-source just-in-time (JIT) compiler that translates a subset of Python and NumPy code into fast machine code, providing significant performance boosts. Ideal for heavy computational tasks, Numba is user-friendly, allowing you to optimize your Python applications with simple decorators while maintaining the flexibility and readability of Python.

Let's repeat the previous example but use Numba instead of Cython.

Numba is a JIT compiler for Python that translates a subset of Python and NumPy code into fast machine code.

I'm going to use Numba to optimize the `between` function as an initial step. I will call the NumPy `randint` function::

```
import numba as nb
import numpy

@nb.jit(nb.int32[:](nb.int32[:], nb.int32[:]))
def between_nb(arr1, arr2):
    return numpy.random.randint(arr1, arr2)
```

Let's run it with the benchmark.

```
>>> %%timeit
>>> (age_100k
...    .str.split('-', expand=True)
...    .astype(int)
...    .pipe(lambda df_: between_nb(df_.iloc[:, 0].to_numpy(dtype='int32'),
...                                 df_.iloc[:, 1].to_numpy(dtype='int32')))
```

```
... )
41.1 ms ± 365 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Our NumPy, Cython, and Numba versions run at approximately the same speed. This happened because they are all calling the NumPy function. However, when we don't have a predefined function that implements what we need, we might get different performance characteristics from the different optimization options.

13.10 Replacing Text

Both the series and the `.str` attribute have a `.replace` method, and these methods have overlapping functionality. If I want to replace single characters, I typically use `.str.replace`, but if I have complete replacements for many values, I use `.replace`.

If I wanted to replace a capital "A" with the Unicode letter a with a ring above it (Unicode name "'LATIN CAPITAL LETTER A WITH RING ABOVE"), I could use this code:

```
>>> make.str.replace('A', 'Å')
0        Ålfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

This would replace all the "A"s in Audi, Acura, Ashton Martin, Alfa Romeo etc.

However, the version below, calling `.replace` directly on the series, does not replace anything because it tries to replace the whole string `'A'`, and there are no makes with that name:

```
>>> make[make == 'A']
Series([], Name: make, dtype: string[pyarrow])
```

Because this code tries to replace the whole string, it doesn't find any:

```
>>> make.replace('A', 'Å')
0        Alfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
```

```
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

You can use a dictionary to specify complete replacements. (This is very explicit, but it might be problematic if you had 20,000 numeric values that had dashes in them and you wanted to strip out the dashes for all 20,000 numbers. You would have to create a dictionary with all the entries, tedious work.):

```
>>> make.replace({'Audi': 'Åudi', 'Acura': 'Åcura',
...     'Ashton Martin': 'Åshton Martin',
...     'Alfa Romeo': 'Ålfa Romeo'})
0        Ålfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

Alternatively, you can specify that you mean to use a regular expression to replace just a portion of the strings with the `regex=True` parameter:

```
>>> make.replace('A', 'Å', regex=True)
0        Ålfa Romeo
1           Ferrari
2             Dodge
            ...
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

I use `.str.replace` to replace substrings and `.replace` to replace mappings of complete strings.

Note

In pandas, we often refer to vectorized operations. It turns out that pandas is not very optimized for dealing with strings. The PyArrow types often run a bit quicker than the other types.

I'm generally against using the `.apply` method because unless you use NumPy functions, you lose vectorization, and operations take a slow path through Python rather than SIMD instructions on the CPU. Because strings are already slow, this is one place where I'm ok with `.apply`.

There are a bunch of other string operations. Below is a table with the string methods.

Table 13.1: String methods

Method	Description
`.str.capitalize()`	Capitalize strings
`.str.casefold()`	Lowercase Unicode/caseless strings.
`.str.cat( others=None, sep='', na_rep=None, join='inner')`	If `others` is `None`, return a string with values separated by `sep`. Otherwise, align the index (if `others` series) and concatenate values.
`.str.center(width, fillchar=' ')`	Center align strings
`.str.contains(pat, case=True, flags=0, na=np.nan, regex=True)`	Return a boolean array if `pat` matches values.
`.str.count(pat, flags=0)`	Return series with the count of how many times `pat` occurs in each value.
`.str.decode( encoding)`	Works with bytestrings to decode them to Unicode strings.
`.str.encode( encoding)`	Encode Unicode string to bytestring.
`.str.endswith(pat, na=np.nan)`	Return boolean array if value ends with `pat`.
`.str.extract(pat, flags=0, expand=True)`	Return a dataframe with the first match from each regular expression capture group in its own column (use named groups for column names). Returns a series if `expand=False`.
`.str.extractall( pat, flags=0)`	Return a dataframe with all matches from each regular expression capture group in its own column (use named groups for column names). The dataframe has a multiindex, where the inner index is named *match* and has match number.
`.str.find(sub, start=None, end=None)`	Return the lowest index of `sub`. -1 if not found.
`.str.findall(pat, flags=0)`	Return a series with a list of matches for each value.
`.str.get(i)`	Return a series with the result of `val[i]` for each value (`val`) in the series.
`.str.get_dummies( sep='|')`	Return a dataframe with each value in its own column and a 0/1 indicating if the value is absent/appeared for that index label. If a string has multiple values they can be separated with `sep`.

Method	Description
`.str.index(sub, start=None, end=None)`	Return the lowest index of `sub`. `ValueError` if not found.
`.str.isalnum()`	Return boolean array if characters are alphanumeric.
`.str.isalpha()`	Return boolean array if characters are alphabetic.
`.str.isdecimal()`	Return boolean array if characters are decimal.
`.str.isdigit()`	Return boolean array if characters are digits.
`.str.islower()`	Return boolean array if characters are lowercase.
`.str.isnumeric()`	Return boolean array if characters are numeric.
`.str.isspace()`	Return boolean array if characters are whitespace.
`.str.istitle()`	Return boolean array if characters are titlecase.
`.str.isupper()`	Return boolean array if characters are uppercase.
`.str.join(sep)`	Given a series with a list of strings in it, join each element with `sep`.
`.str.len()`	Return a series with length of each value (works with lists or collections).
`.str.ljust(width, fill=' ')`	Return a left justified series.
`.str.lower()`	Return a lowercase series.
`.str.lstrip( to_strip=None)`	Return a series with left stripped `to_strip` (whitespace default).
`.str.match(pat, case=True, flags=0, na=np.nan)`	Return a boolean array if `pat` matches values (anchored at the beginning). Use `.str.contains` to match anywhere in the string. (Use `.str.extract` to pull out the string.)
`.str.normalize( form)`	Return Unicode normal form for series. `form` can be `'NFC'`, `'NFKC'`, `'NFD'`, or `'NFKD'`.
`.str.pad(width, side='left', fill=' ')`	Return a padded series of length `width`. `side` can be `'left'`, `'right'`, or `'both'`.
`.str.partition( sep, expand=True)`	Return a dataframe with three columns: element before first `sep`, the `sep`, and the part after.
`.str.repeat( repeats)`	Return a series with values repeated `repeats` times. `repeats` can be a scalar or list.

Method	Description
`.str.replace(pat, repl, n=-1, case=True, flags=0, regex=True)`	Return a series where `pat` is replaced by `repl`. `n` is the number of times to replace a value. `repl` can be a string or a callable that takes a match object and returns a string.
`.str.rfind(sub, start=None, end=None)`	Return highest index of `sub`. -1 if not found.
`.str.rindex(sub, start=None, end=None)`	Return highest index of `sub`. `ValueError` if not found.
`.str.rjust(width, fill=' ')`	Return a right justified series.
`.str.rpartition( sep, expand=True)`	Return a dataframe with three columns: element before last `sep`, the `sep`, and the part after.
`.str.rsplit(pat, n=-1, expand=False)`	Return a Series (if `expand=False`) with a list of values split from the right side limited to `n` splits.
`.str.rstrip( to_strip=None)`	Return a series with rightstripped `to_strip` (whitespace default).
`.str.slice( start=None, stop=None, step=None)`	Return a series. Equivalent to `s[start:stop:step]`.
`.str.slice_replace( start=None, stop=None, repl=None)`	Return a series with slice replaced by the value of `repl`.
`.str.split(pat, n=-1, expand=False)`	Return a Series (if `expand=False`) with a list of values split by `sep` limited to `n` splits.
`.str.startswith(pat, na=np.nan)`	Return boolean array if value starts with `pat`.
`.str.strip(to_strip=None)`	Return a series with left and right stripped `to_strip` (whitespace default).
`.str.swapcase()`	Return swapcase series.
`.str.title()`	Return titlecase series.
`.str.translate( table)`	Return series using a dictionary `table` to replace characters. `table` maps code points to new code points (numbers not strings). Keys mapped to `None` are deleted.
`.str.upper()`	Return uppercase series.
`.str.wrap(width)`	Return a line wrapped series limited to `width`.
`.str.zfill(width)`	Return a series limited to `width` left padded with `'0'`.

13.11 Summary

The `object`, `'string'`, and `'category'` type series all can be used to store string data. They all have the `.str` accessor. You get much of the same functionality if you are familiar with Python strings. In addition, there is the ability to manipulate with regular expressions.

13.12 Exercises

With a dataset of your choice:

1. Using a string column, lowercase the values.
2. Using a string column, slice out the first character.
3. Using a string column, slice out the last three characters.
4. Create a series extracting the numeric values using a string column.
5. Using a string column, create a series extracting the non-ASCII values.
6. Using a string column, create a dataframe with the dummy columns for every character in the column.

Chapter 14

Date and Time Manipulation

Pandas allows you to create series with date and time information in them. In this chapter, we will explore everyday operations that you will need to perform with date data.

14.1 Date Theory

Let's talk about dates in brief. Coordinated Universal Time or universel temps coordonné (UTC) as our French friends who decide acronyms like to say is the time standard at 0 degrees longitude. It has an excellent property that it is monotonically increasing. I live in Salt Lake City, Utah, the *America/Denver* time zone is 6 or 7 hours offset of UTC, depending on the time of year.

In short, a time zone may contain one or more offsets (depending on if they observe daylight savings time or political whimsy). There is a standardized format, ISO 8601, for representing dates. It does not include the time zone but optionally an offset.

A note on time zone names. The public domain time zone database (also known as the Olsen database) from iana.org provides code and data regarding time zones and their history. From their documentation:

> Time zones are typically identified by continent or ocean and then by the name of the largest city within the region containing the clocks. For example, America/New_York represents most of the US eastern time zone; America/Phoenix represents most of Arizona, which uses mountain time without daylight saving time (DST); America/Detroit represents most of Michigan, which uses eastern time but with different DST rules in 1975; and other entries represent smaller regions like Starke County, Indiana, which switched from central to eastern time in 1991 and switched back in 2006.
>
> —https://data.iana.org/time-zones/tz-link.html

Getting the correct time zone name is essential and might be confusing or complicated. As I said, I live in Salt Lake City. If I search for "Time zone for Salt Lake City", I get "Mountain Daylight Time" or "GMT-6". Neither of which is a time zone. You might also see "US/Mountain", "MST", or "MDT". These are not time zones either. These are deprecated names or offsets. The correct time zone name is "America/Denver". However, many applications support erroneous names.

Remember that time zones are associated with a city!

I recommend prefacing your search with "IANA" (ie. "IANA Time zone for Salt Lake City") and then double checking your result in this Wikipedia article (which also shows deprecated names)[1].

It is important to have the offset information as well. Time zones with daylight savings time can have "ambiguous time" in the fall when the time goes back. For example, in Salt Lake on Nov 1, 2015, after 1:59 AM (MDT), the clock goes to 1:00 AM (MST). On that date, there are two 1:30 AMs. One at MDT and another an hour later at MST.

For this reason, if you are dealing with local times, you will want three things: the time, the time zone, and an offset. If you are only concerned with duration, you can use UTC time or seconds since the UNIX epoch.

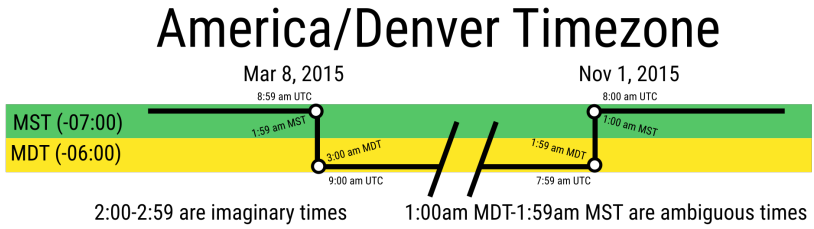


Figure 14.1: When daylight savings begins in the spring, it creates imaginary times. When daylight savings ends in the fall, there are ambiguous times (unless you include the offset).

Let's introduce a few more terms before jumping to an example. A time without a time zone or offset is called "naive" time. A time specified in local time is also called "civil time" or "wall time".

UTC time is unambiguous. It does not repeat. Also, to be pedantic, UTC is not a time zone.

Naive time is ambiguous. 2:37 PM happens multiple times per day for each time zone.

1:29 AM America/Denver might seem specific enough, but it is context-dependent. You also need offset information on the first Sunday in November

[1] https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

because it is ambiguous. There is 1:29 AM MDT, then after 1:59 AM MDT comes 1:00 AM MST, and another 1:29 AM for MST!

A general recommendation for programmers is to store dates in UTC times and convert them to local times as needed. The ISO 8601 format is insufficient to keep precise local dates as it supports offset but not time zone. If you need local times, I suggest you store one of these two options:

- UTC date and time zone
- Local date, offset, and time zone

Note

The pandas library can support dates stored in UTC values using the `datetime64[ns]` type. It also supports local times from a single time zone. It appears to (and by appear, I mean the operation goes without failure) support multiple time zones in a single series. However, the underlying datatype will be a `pd.Timestamp` object that does not support the `.dt` accessor.

If you have time data and you need to deal with multiple time zones, I would probably break up the data by time zone and put each time zone in its own dataframe or series.

14.2 Loading UTC Time Data

Here is a series of strings with UTC dates. Let's convert it to a date series. You need to remember to pass the `utc=True` parameter to `pd.to_datetime`:

```
>>> col = pd.Series(['2015-03-08 08:00:00+00:00',
...     '2015-03-08 08:30:00+00:00',
...     '2015-03-08 09:00:00+00:00',
...     '2015-03-08 09:30:00+00:00',
...     '2015-11-01 06:30:00+00:00',
...     '2015-11-01 07:00:00+00:00',
...     '2015-11-01 07:30:00+00:00',
...     '2015-11-01 08:00:00+00:00',
...     '2015-11-01 08:30:00+00:00',
...     '2015-11-01 08:00:00+00:00',
...     '2015-11-01 08:30:00+00:00',
...     '2015-11-01 09:00:00+00:00',
...     '2015-11-01 09:30:00+00:00',
...     '2015-11-01 10:00:00+00:00'])

>>> utc_s = pd.to_datetime(col, utc=True)
>>> utc_s

0    2015-03-08 08:00:00+00:00
```

```
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
                 ...
11   2015-11-01 09:00:00+00:00
12   2015-11-01 09:30:00+00:00
13   2015-11-01 10:00:00+00:00
Length: 14, dtype: datetime64[ns, UTC]
```

Notice the type of the result. It indicates that the dates are stored as UTC. Once you have converted a series into a `datetime64[ns]` object, you can leverage the `.dt` attribute.

Let's convert this series to the *America/Denver* time zone:

```
>>> utc_s.dt.tz_convert('America/Denver')
0    2015-03-08 01:00:00-07:00
1    2015-03-08 01:30:00-07:00
2    2015-03-08 03:00:00-06:00
                 ...
11   2015-11-01 02:00:00-07:00
12   2015-11-01 02:30:00-07:00
13   2015-11-01 03:00:00-07:00
Length: 14, dtype: datetime64[ns, America/Denver]
```

Note that if you have data with offsets that are not 00:00, you can still use the same code to load the data:

```
>>> s = pd.Series(['2015-03-08 01:00:00-07:00',
...  '2015-03-08 01:30:00-07:00',
...  '2015-03-08 03:00:00-06:00',
...  '2015-03-08 03:30:00-06:00',
...  '2015-11-01 00:30:00-06:00',
...  '2015-11-01 01:00:00-06:00',
...  '2015-11-01 01:30:00-06:00',
...  '2015-11-01 01:00:00-07:00',
...  '2015-11-01 01:30:00-07:00',
...  '2015-11-01 01:00:00-07:00',
...  '2015-11-01 01:30:00-07:00',
...  '2015-11-01 02:00:00-07:00',
...  '2015-11-01 02:30:00-07:00',
...  '2015-11-01 03:00:00-07:00'])

>>> pd.to_datetime(s, utc=True).dt.tz_convert('America/Denver')
0    2015-03-08 01:00:00-07:00
1    2015-03-08 01:30:00-07:00
2    2015-03-08 03:00:00-06:00
                 ...
11   2015-11-01 02:00:00-07:00
```

```
12   2015-11-01 02:30:00-07:00
13   2015-11-01 03:00:00-07:00
Length: 14, dtype: datetime64[ns, America/Denver]
```

14.3 Loading Local Time Data

If we want to load local date information, we need to have the date, the offset, and the time zone. Let's assume that we have local time information in one series and offset in another:

```
>>> time = pd.Series(['2015-03-08 01:00:00',
...  '2015-03-08 01:30:00',
...  '2015-03-08 02:00:00',
...  '2015-03-08 02:30:00',
...  '2015-03-08 03:00:00',
...  '2015-03-08 02:00:00',
...  '2015-03-08 02:30:00',
...  '2015-03-08 03:00:00',
...  '2015-03-08 03:30:00',
...  '2015-11-01 00:30:00',
...  '2015-11-01 01:00:00',
...  '2015-11-01 01:30:00',
...  '2015-11-01 02:00:00',
...  '2015-11-01 02:30:00',
...  '2015-11-01 01:00:00',
...  '2015-11-01 01:30:00',
...  '2015-11-01 02:00:00',
...  '2015-11-01 02:30:00',
...  '2015-11-01 03:00:00'])

>>> offset = pd.Series([-7, -7, -7, -7, -7, -6, -6,
...   -6, -6, -6, -6, -6, -6, -6, -7, -7, -7, -7, -7])
```

We want to apply the offset to the corresponding time. The mechanism in pandas is to use `.groupby` with `.transform`. (We will explain these in detail later in the grouping chapter.) The basic idea is that we group all dates from one offset together and call `.dt.tz_localize` on them. We repeat this for each offset. Calling the `.transform` method allows us to work on a group and then return a result in the original length of the grouped object (that has not been aggregated):

```
>>> (pd.to_datetime(time)
...     .groupby(offset)
...     .transform(lambda s: s.dt.tz_localize(s.name)
...                           .dt.tz_convert('America/Denver'))
... )
```

```
0    2015-03-07 18:00:07-07:00
1    2015-03-07 18:30:07-07:00
2    2015-03-07 19:00:07-07:00
                ...
16   2015-10-31 20:00:07-06:00
17   2015-10-31 20:30:07-06:00
18   2015-10-31 21:00:07-06:00
Length: 19, dtype: datetime64[ns, America/Denver]
```

Note that this operation did not error out and appeared to run successfully. However, if you look closely, the offsets were incorrect and moved the minute by 7 or 6 minutes instead of the hours. We need to use different offsets. We want them to be `'-07:00'` and `'-06:00'` respectively:

```
>>> offset = offset.replace({-7:'-07:00', -6:'-06:00'})
>>> local = (pd.to_datetime(time)
...     .groupby(offset)
...     .transform(lambda s: s.dt.tz_localize(s.name)
...                           .dt.tz_convert('America/Denver'))
... )

>>> local
0    2015-03-08 01:00:00-07:00
1    2015-03-08 01:30:00-07:00
2    2015-03-08 03:00:00-06:00
                ...
16   2015-11-01 02:00:00-07:00
17   2015-11-01 02:30:00-07:00
18   2015-11-01 03:00:00-07:00
Length: 19, dtype: datetime64[ns, America/Denver]
```

14.4 Converting Local time to UTC

If you have a series with local time information (stored as `datetime64[ns]` and not a string), you can use the `.dt.tz_convert` method to change it to UTC time:

```
>>> local.dt.tz_convert('UTC')
0    2015-03-08 08:00:00+00:00
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
                ...
16   2015-11-01 09:00:00+00:00
17   2015-11-01 09:30:00+00:00
18   2015-11-01 10:00:00+00:00
Length: 19, dtype: datetime64[ns, UTC]
```

14.5 Converting to Epochs

If you have a series with UTC or local time information, you can get the (nano) seconds past the UNIX epoch using this code:

```
>>> nano_secs = local.astype('int64[pyarrow]')
>>> nano_secs
0     1425801600000000000
1     1425803400000000000
2     1425805200000000000
             ...
16    1446368400000000000
17    1446370200000000000
18    1446372000000000000
Length: 19, dtype: int64[pyarrow]
```

To load epoch information into UTC use the following:

```
>>> (pd.to_datetime(nano_secs, unit='ns')
...   .dt.tz_localize('UTC'))
0    2015-03-08 08:00:00+00:00
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
                ...
16   2015-11-01 09:00:00+00:00
17   2015-11-01 09:30:00+00:00
18   2015-11-01 10:00:00+00:00
Length: 19, dtype: datetime64[ns, UTC]
```

If you get dates from the 1970s, you may have to use the `unit` keyword to adjust the granularity of the result. Below I divide by one million to get milliseconds but I'm trying to convert with nanoseconds (`'ns'`):

```
>>> (nano_secs
...  .truediv(1_000_000)
...  .pipe(pd.to_datetime, unit='ns')
...  .dt.tz_localize('UTC'))
0    1970-01-01 00:23:45.801600+00:00
1    1970-01-01 00:23:45.803400+00:00
2    1970-01-01 00:23:45.805200+00:00
                   ...
16   1970-01-01 00:24:06.368400+00:00
17   1970-01-01 00:24:06.370200+00:00
18   1970-01-01 00:24:06.372000+00:00
Length: 19, dtype: datetime64[ns, UTC]
```

If I use milliseconds (`'ms'`), I get the correct result:

```
>>> (nano_secs
...  .truediv(1_000_000)
...  .pipe(pd.to_datetime, unit='ms')
...  .dt.tz_localize('UTC'))
0    2015-03-08 08:00:00+00:00
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
                ...
16   2015-11-01 09:00:00+00:00
17   2015-11-01 09:30:00+00:00
18   2015-11-01 10:00:00+00:00
Length: 19, dtype: datetime64[ns, UTC]
```

14.6 Manipulating Dates

To further demo date manipulation, I am going to read in a dataset with snowfall levels from a local ski resort.

```
>>> url = 'https://github.com/mattharrison/datasets'+\
...     '/raw/master/data/alta-noaa-1980-2019.csv'
>>> alta_df = pd.read_csv(url)
```

I will show working with a series with date information in them. Then, we will look at a series with dates in the index. The date series will be pulled from the *DATE* column. Remember that when you read a CSV, it does not convert columns to dates by default. You can use the `parse_dates` parameter to try and convert to dates when reading a CSV, but the `to_datetime` function is more powerful. I generally recommend messing around with dates outside of the `read_csv` function:

```
>>> dates = (pd.to_datetime(alta_df.DATE)
...    .astype('timestamp[ns][pyarrow]'))
>>> dates
0        1980-01-01 00:00:00
1        1980-01-02 00:00:00
2        1980-01-03 00:00:00
                 ...
14157    2019-09-05 00:00:00
14158    2019-09-06 00:00:00
14159    2019-09-07 00:00:00
Name: DATE, Length: 14160, dtype: timestamp[ns][pyarrow]
```

A series with a date in it is a little boring. However, you will see dataframes with date columns in them. Remember that a column is just a series, and being able to manipulate that column as part of a dataframe will be helpful.

Note that the type of `date` is `datetime64[ns]`. This gives us some superpowers. It adds a `.dt` attribute to the series, allowing us to perform various date manipulations.

To get the weekdays in Spanish, I can specify the appropriate locale:

```
>>> dates.dt.day_name('es_ES')
0           martes
1        miércoles
2           jueves
           ...
14157       jueves
14158      viernes
14159       sábado
Name: DATE, Length: 14160, dtype: string[pyarrow]
```

Note

To get a list of locales on Linux, run the `locale` command from the terminal. My output looks like this:

```
$ locale -a
C
C.UTF-8
POSIX
en_US.utf8
es_ES
es_ES.iso88591
Spanish
```

Many of the attributes of the `.dt` attribute are properties and are not methods. Many ask me why they are properties and not methods. A property is not parameterizable. You get back the results. Also note that you do not put parentheses at the end of a property (i.e., you do not *call* it). If you do, you will get an error stating it is not callable.

The creators of the properties felt that there were no options for them. For example, `.is_month_end` tells you whether a day is the last of the month, so it is a property. However, `.strftime` requires that we parameterize it with a formatting string, so it is a method:

```
>>> dates.dt.is_month_end
0        False
1        False
2        False
         ...
14157    False
14158    False
```

```
14159    False
Name: DATE, Length: 14160, dtype: bool[pyarrow]
```

Here we format the date as a string:

```
>>> dates.dt.strftime('%d/%m/%y')
0        01/01/80
1        02/01/80
2        03/01/80
           ...
14157    05/09/19
14158    06/09/19
14159    07/09/19
Name: DATE, Length: 14160, dtype: string[pyarrow]
```

Table 14.1: Table of strftime codes

Code	Meaning	Sample
%y	Year (decimal)	14
%Y	Year (century)	2014
%m	Month (padded or unpadded)	08 or 8
%b	Month (Abbrev locale)	Aug
%B	Month	August
%d	Day (padded or unpadded)	04 or 4
%a	Weekday (Abbrev locale)	Mon
%A	Weekday (locale)	Monday
%H	Hour (24 padded)	22
%I	Hour (12 padded)	10
%M	Minutes (padded)	25
%S	Seconds (padded)	24
%p	AM/PM	PM
%-d	Day (unpadded unix*)	4
%e	Day (unpadded unix*)	4
%c	Locale representation	Mon Aug 4 22:25:24 2014
%x	Locale date	08/04/14
%X	Locale time	22:25:24
%W	Week num (Mon 1st)	31
%U	Week num (Sun 1st)	31
%j	Day of year (padded)	216
%z	UTC offset	+0000
%Z	Time Zone	MDT
%%	Percent sign	%

14.7 Date Math

Let's assume that you have a series with the starting dates for classes in a school. You also have a series with the ending dates for the classes. You want to know how long each class is. Here is the data:

```
>>> classes = ['cs106', 'cs150', 'hist205', 'hist206', 'hist207']
>>> start_dates = (pd.Series(['2015-03-08',
...  '2015-03-08',
...  '2015-03-09',
...  '2015-03-09',
...  '2015-03-11'], dtype='datetime64[ns]', index=classes)
...  .astype('timestamp[ns][pyarrow]'))
>>> start_dates
cs106      2015-03-08 00:00:00
cs150      2015-03-08 00:00:00
hist205    2015-03-09 00:00:00
hist206    2015-03-09 00:00:00
hist207    2015-03-11 00:00:00
dtype: timestamp[ns][pyarrow]

>>> classes = ['cs106', 'cs150', 'hist205', 'hist206', 'hist207']
>>> end_dates = (pd.Series(['2015-05-28 23:59:59',
...  '2015-06-01 3:00:00',
...  '2015-06-03',
...  '2015-06-02 14:20',
...  '2015-06-01'], dtype='datetime64[ns]', index=classes)
...  .astype('timestamp[ns][pyarrow]'))
>>> end_dates
cs106      2015-05-28 23:59:59
cs150      2015-06-01 03:00:00
hist205    2015-06-03 00:00:00
hist206    2015-06-02 14:20:00
hist207    2015-06-01 00:00:00
dtype: timestamp[ns][pyarrow]
```

To calculate the duration of each class, we can subtract the start date from the end date:

```
>>> duration = (end_dates - start_dates)
>>> duration
cs106      81 days 23:59:59
cs150      85 days 03:00:00
hist205    86 days 00:00:00
hist206    85 days 14:20:00
hist207    82 days 00:00:00
dtype: duration[ns][pyarrow]
```

```
>>> duration.astype('timedelta64[ns]')
cs106     81 days 23:59:59
cs150     85 days 03:00:00
hist205   86 days 00:00:00
hist206   85 days 14:20:00
hist207   82 days 00:00:00
dtype: timedelta64[ns]
```

`duration`

This gives us a `duration[ns][pyarrow]` series. This type is missing the attributes of legacy pandas 1.x `timedelta64[ns]` [1]. `timedelta64[ns]` is a special type of series that represents a duration of time. It is not a date, but it is not a number, either. It is a duration. It also has a `.dt` attribute that allows us to perform date-like manipulations on it:

The table below lists the duration attributes:

Table 14.2: Table of timedelta attributes

Attribute	Meaning
`.as_unit(unit)`	Change the units of the duration (supports `'s'`, `'ms'`, `'us'`, `'ns'`)
`.ceil(freq)`	Round up to specified units (supports `'D'`, `'H'`, `'T'` (`'min'`), `'s'`, `'ms'`, `'us'`, `'ns'`)
`.components`	Property with dataframe of duration components
`.days`	Property with number of days
`.floor(freq)`	Round down to specified units (supports `'D'`, `'H'`, `'T'` (`'min'`), `'s'`, `'ms'`, `'us'`, `'ns'`)
`.freq`	Frequency of the duration
`.microseconds`	Property with number of microseconds
`.nanoseconds`	Property with number of nanoseconds
`.round(freq)`	Round up to specified units (supports `'D'`, `'H'`, `'T'` (`'min'`), `'s'`, `'ms'`, `'us'`, `'ns'`)
`.seconds`	Property only with seconds unit
`.to_pytimedelta()`	Return NumPy array of `datetime.timedelta` objects
`.total_seconds()`	Return total seconds
`.unit`	Property to get the unit of the duration (use `.as_unit` to change)

A timedelta has many of the same attributes as a datetime. If we want to know the total number of seconds in the duration, we can use the `.total_seconds()` method:

[1] https://github.com/pandas-dev/pandas/issues/56269

```
>>> (duration
...    .astype('timedelta64[ns]')
...    .dt.total_seconds()
... )
cs106      7084799.0
cs150      7354800.0
hist205    7430400.0
hist206    7395600.0
hist207    7084800.0
dtype: float64
```

If we access the `.seconds` attribute, we get the number of seconds in the duration at the interval below a day (i.e., less than 24 hours or 24 hours * 60 minutes * 60 seconds = 86400 seconds):

```
>>> (duration
...    .astype('timedelta64[ns]')
...    .dt.seconds
... )
cs106      86399
cs150      10800
hist205        0
hist206    51600
hist207        0
dtype: int32
```

If we wanted to know the total number of days so we could determine how much of the year the class lasted, we can use the `.days` attribute:

```
>>> (duration
...    .astype('timedelta64[ns]')
...    .dt.days
... )
cs106      81
cs150      85
hist205    86
hist206    85
hist207    82
dtype: int64
```

Below is a table of `.dt` methods and properties.

Table 14.3: `.dt` methods and Properties

Method	Description
`.ceil(freq=None, ambiguous=None, nonexistent=None)`	Return ceiling according to offset alias in `freq`. The `nonexistent` parameter controls DST time issues.
`.date`	Property with a series of Python `datetime.date` objects.
`.day`	Property with a series of day of month.
`.day_name( locale='en_us')`	Return the string day of week.
`.dayofweek`	Property with a series of date of week as number (0 is Monday).
`.dayofyear`	Property with a series of day of the year.
`.days_in_month`	Property with a series of number of days in month.
`.daysinmonth`	Property with a series of number of days in month.
`.floor(freq=None, ambiguous=None, nonexistent=None)`	Return floor according to offset alias in `freq`. The `nonexistent` parameter controls DST time issues.
`.hour`	Property with a series of hour of date.
`.is_leap_year`	Property with a series of booleans if date is leap year.
`.is_month_end`	Property with a series of booleans if date is end of month.
`.is_month_start`	Property with a series of booleans if date is start of month.
`.is_quarter_end`	Property with a series of booleans if date is end of quarter.
`.is_quarter_start()`	Property with a series of booleans if date is start of quarter.
`.is_year_end`	Property with a series of booleans if date is end of year.
`.is_year_start`	Property with a series of booleans if date is start of year.
`.microsecond`	Property with a series of microseconds of date.
`.minute`	Property with a series of minutes of date.
`.month`	Property with a series of month of date (numeric).
`.month_name( locale='en_us')`	Return a series of month of date (string).
`.nanosecond`	Property with a series of nanoseconds of date.
`.normalize()`	Return a series of dates converted to midnight.
`.quarter`	Property with series of quarter of date (numeric 1-4).
`.round(freq=None, ambiguous=None, nonexistent=None)`	Return round according to fixed frequency (cannot be end like `'ME'`) in `freq`. The `nonexistent` parameter controls DST time issues.
`.second`	Property with a series of seconds of date (numeric).

Method	Description
`.strftime(date_format)`	Return a series with string dates. Formatted using strftime format codes.
`.time`	Property with a series of Python `datetime.time` objects.
`.timetz`	Property with a series of Python `datetime.time` objects with time zone information.
`.to_period(freq)`	Return a series with pandas `Period` objects.
`.to_pydatetime()`	Return a numpy array with `datetime.datetime` objects.
`.tz`	Property with time zone.
`.tz_convert(tz)`	Convert from one time zone aware series to another.
`.tz_localize(tz, ambiguous=None, nonexistent=None)`	Convert from naive to time zone aware.
`.week`	Property with a series of week of date (numeric 1-53).
`.weekday`	Property with a series of date of week as number 0 is Monday.
`.weekofyear`	Property with a series of week of date (numeric 1-53).
`.year`	Property with a series of year of date.

14.8 Summary

In the chapter, we explored converting series into time series. We discussed time zones, offsets, local time, and UTC time. If you have UTC time, you can convert it into a time zone. If you have local time, you will need offset information to convert it into a time zone (as many local times have ambiguous times). If you have a series with multiple time zone dates, I recommend leaving it as UTC because pandas will not allow you to work on the dates unless you split them out into one time zone.

14.9 Exercises

With a dataset of your choice:

1. Convert a column with date information to a date.
2. Convert a date column into UTC dates.
3. Convert a date column into local dates with a time zone.
4. Convert a date column into epoch values.
5. Convert an epoch number into UTC.

Chapter 15

Dates in the Index

If you have dates in the index, you can do some powerful manipulation and aggregation of your data.

We will shift gears and look at data with a date as an index. We will look at the amount of snow that fell each day at the ski resort. I will create a series with dates in the index and the amount of snow that fell that day in the values. It will be called `snow`:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets'+\
...     '/raw/master/data/alta-noaa-1980-2019.csv'
>>> alta_df = pd.read_csv(url, engine='pyarrow', dtype_backend='pyarrow')
>>> dates = pd.to_datetime(alta_df.DATE)

>>> snow = (alta_df
...     .SNOW
...     .rename(dates)
... )

>>> snow
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
              ...
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

15.1 Finding Missing Data

Let's look for missing data. A few methods help with dealing with missing data in time data. We can check if any values are missing using `.any`:

```
>>> snow.isna().any()
True
```

There is missing data. Let's look where it is:

```
>>> snow[snow.isna()]
1985-07-30    <NA>
1985-09-12    <NA>
1985-09-19    <NA>
              ...
2017-10-02    <NA>
2017-12-23    <NA>
2018-12-03    <NA>
Name: SNOW, Length: 365, dtype: double[pyarrow]
```

With a date index, we can provide partial date strings to the `.loc` indexing attribute. This will let us inspect the rows that surround the missing data and see if that gives us any insight into why it is missing:

```
>>> snow.loc['1985-09':'1985-09-20']
1985-09-01     0.0
1985-09-02     0.0
1985-09-03     0.0
              ...
1985-09-18     0.0
1985-09-19    <NA>
1985-09-20     0.0
Name: SNOW, Length: 20, dtype: double[pyarrow]
```

15.2 Filling In Missing Data

Often, we have time series data with missing values. For example, in the snow data, the value for the date *1985-09-19* is missing. (See previous code.)

This value looks like it could be filled in with zero (as this is the end of summer):

```
>>> (snow
...     .loc['1985-09':'1985-09-20']
...     .fillna(0)
... )
1985-09-01    0.0
1985-09-02    0.0
1985-09-03    0.0
              ...
1985-09-18    0.0
1985-09-19    0.0
```

```
1985-09-20    0.0
Name: SNOW, Length: 20, dtype: double[pyarrow]
```

However, these values might not be zero in January, the middle of the winter. (It is not clear to me why these values are missing. Did a sensor fail? Did someone forget to write down the amount? Was it really zero?) The best way to deal with missing data is to talk to a subject matter expert and determine why it is missing:

```
>>> snow.loc['1987-12-30':'1988-01-10']
1987-12-30     6.0
1987-12-31     5.0
1988-01-01    <NA>
              ...
1988-01-08     9.0
1988-01-09     5.0
1988-01-10     2.0
Name: SNOW, Length: 12, dtype: double[pyarrow]
```

Pandas has various tricks for dealing with missing data. Let's demonstrate them with the missing data from the end of December through January. Notice what happens to the January 1 value as we demo these.

We can do a forward fill or backfill using `.ffill` and `.bfill` respectively:

```
>>> (snow
...     .loc['1987-12-30':'1988-01-10']
...     .ffill()
... )
1987-12-30    6.0
1987-12-31    5.0
1988-01-01    5.0
              ...
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: double[pyarrow]

>>> (snow
...     .loc['1987-12-30':'1988-01-10']
...     .bfill()
... )
1987-12-30    6.0
1987-12-31    5.0
1988-01-01    0.0
              ...
1988-01-08    9.0
1988-01-09    5.0
```

```
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: double[pyarrow]
```

Be careful with backfill if you are planning on doing any machine learning. You don't want to use future data to predict the past. This is called data *leakage* and will cause your model to overfit.

15.3 Interpolation

We can also interpolate using `.interpolate`. By default, this does a linear interpolation for the missing values.

```
>>> (snow
...     .loc['1987-12-30':'1988-01-10']
...     .interpolate()
... )
1987-12-30    6.0
1987-12-31    5.0
1988-01-01    2.5
              ...
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: double[pyarrow]
```

Again, you will not want to use `.interpolate` if you are doing machine learning because it calculates the missing values based on future data.

We can use the code below to fill in the missing winter values (if the quarter is 1 or 4) with interpolated values and the other values with zero. (Because the index is a datetime, we can access `.dt` attributes directly on the index.)

This is a good example of the `.where` method. In the figure is a truth table for *winter* and *snow* values.

Snow/Winter Truth Table

	Snow Missing	Snow Not Missing
Winter	I	II
Not Winter	III	IV

Figure 15.1: Truth table for snow and winter options.

We will interpolate when it is winter, and we are missing snow values. This corresponds to the section I. When it is not winter and snow values are missing, we will fill in 0 (section III).

```
>>> winter = (snow.index.quarter == 1) | (snow.index.quarter== 4)
>>> (snow
...  .case_when([(winter & snow.isna(), snow.interpolate()),
...              (~winter & snow.isna(), 0)])
... )
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
              ...
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

Here is the `.where` version of the code. I find it very hard to read. Recall that the `.where` method keeps values where the first parameter is `True`, so we invert the conditions with `~`:

```
>>> (snow
...     .where(~(winter & snow.isna()), snow.interpolate())
...     .where(~(~winter & snow.isna()), 0)
... )
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
              ...
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

And we can validate some of the values to make sure that the interpolation worked. Here are some missing values during off winter and winter dates:

```
>>> (snow
...  .loc[['1985-09-19','1988-01-01']]
... )
1985-09-19    <NA>
1988-01-01    <NA>
Name: SNOW, dtype: double[pyarrow]
```

It looks like these values are corrected:

```
>>> (snow
...  .case_when([(winter & snow.isna(), snow.interpolate()),
...              (~winter & snow.isna(), 0)])
...  .loc[['1985-09-19','1988-01-01']]
... )
1985-09-19    0.0
1988-01-01    2.5
Name: SNOW, dtype: double[pyarrow]
```

15.4 Dropping Missing Values

We can also drop the missing data using the `.dropna` method:

```
>>> (snow
...     .loc['1987-12-30':'1988-01-10']
...     .dropna()
... )
1987-12-30    6.0
1987-12-31    5.0
1988-01-02    0.0
              ...
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 10, dtype: double[pyarrow]
```

Be careful with the method and only use it after talking to a subject matter expert who confirms that it is okay to drop the data. It can be hard to tell later if the data is missing. For example, if you plotted this data, you might not see that data was dropped unless you pay close attention.

15.5 Shifting Data

We can shift data up or down, which is helpful for sequence data like time series. This method works on any pandas series but comes in useful with time series when we want to compare to the previous or subsequent entry. Here is a forward and backward shift:

```
>>> snow.shift(1)
1980-01-01    <NA>
1980-01-02     2.0
1980-01-03     3.0
               ...
2019-09-05     0.0
2019-09-06     0.0
2019-09-07     0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

```
>>> snow.shift(-1)
1980-01-01     3.0
1980-01-02     1.0
1980-01-03     0.0
               ...
2019-09-05     0.0
2019-09-06     0.0
2019-09-07    <NA>
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

15.6 Rolling Average

To calculate the five-day moving average, we can leverage `.shift` and do the following:

```
>>> (snow
...    .add(snow.shift(1))
...    .add(snow.shift(2))
...    .add(snow.shift(3))
...    .add(snow.shift(4))
...    .div(5)
... )
1980-01-01    <NA>
1980-01-02    <NA>
1980-01-03    <NA>
              ...
2019-09-05     0.0
2019-09-06     0.0
2019-09-07     0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

That was a little tedious to write. Thankfully, pandas has a trick up its sleeve. The `.rolling` method allows us to specify a window size. This method returns a `Rolling` object so we can apply various aggregate methods to it. If we apply `.mean` to it, we get a very similar result to the above:

```
>>> (snow
...    .rolling(5)
...    .mean()
... )
1980-01-01    NaN
1980-01-02    NaN
1980-01-03    NaN
              ...
2019-09-05    0.0
2019-09-06    0.0
```

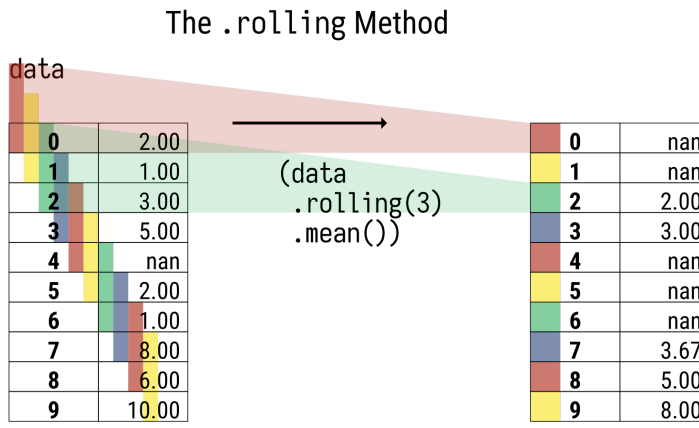


Figure 15.2: The `.rolling` method slides a window along the data, allowing you to call an aggregate function.

```
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64
```

Below are methods that work on a `Rolling` object:

Table 15.1: Rolling methods and properties

Method	Description
`r.agg(func=None, axis=0, *args, **kwargs)`	Returns a scalar if `func` is a single aggregation function. Returns a series if a list of aggregations are passed to `func`. (`aggregate` is a synonym.)
`r.apply(func, args=None, kwargs=None)`	Apply custom aggregation function to rolling group.
`r.corr(other, method='pearson')`	Returns correlation coefficient for `'pearson'`, `'spearman'`, `'kendall'`, or a callable.
`r.count(other, method='pearson')`	Returns count of non `NaN` values.
`r.cov(other, min_periods=None)`	Returns covariance.
`r.max(axis=None, skipna=None, level=None, numeric_only=None`	Returns maximum value.
`r.min(axis=None, skipna=None, level=None, numeric_only=None)`	Returns minimum value.

Method	Description
`r.mean(axis=None, skipna=None, level=None, numeric_only=None)`	Returns mean value.
`r.median(axis=None, skipna=None, level=None, numeric_only=None)`	Returns median value.
`r.quantile(q=.5, interpolation='linear')`	Returns 50% quantile by default. *Note* returns `Series` if `q` is a list.
`r.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased standard error of mean.
`r.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns sample standard deviation.
`r.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased variance.
`r.skew(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased skew.

15.7 Resampling

Because this series has dates as the index, it has more superpowers. The `.resample` method can aggregate values at different levels. At a high level, we group date entries by some interval (yearly, monthly, weekly) and then aggregate the values at that interval.

For example, to find the maximum snowfall by month, we can use this code:

```
>>> (snow
...     .resample('ME')
...     .max()
... )
1980-01-31    20.0
1980-02-29    25.0
1980-03-31    16.0
              ...
2019-07-31     0.0
2019-08-31     0.0
2019-09-30     0.0
Freq: ME, Name: SNOW, Length: 477, dtype: double[pyarrow]
```

The `'ME'` string in the `.resample` call is what pandas calls an *offset alias*. This is a string that specifies a grouping frequency. Using *ME* means group all values by the end of the month. If you look at the index for the result, you

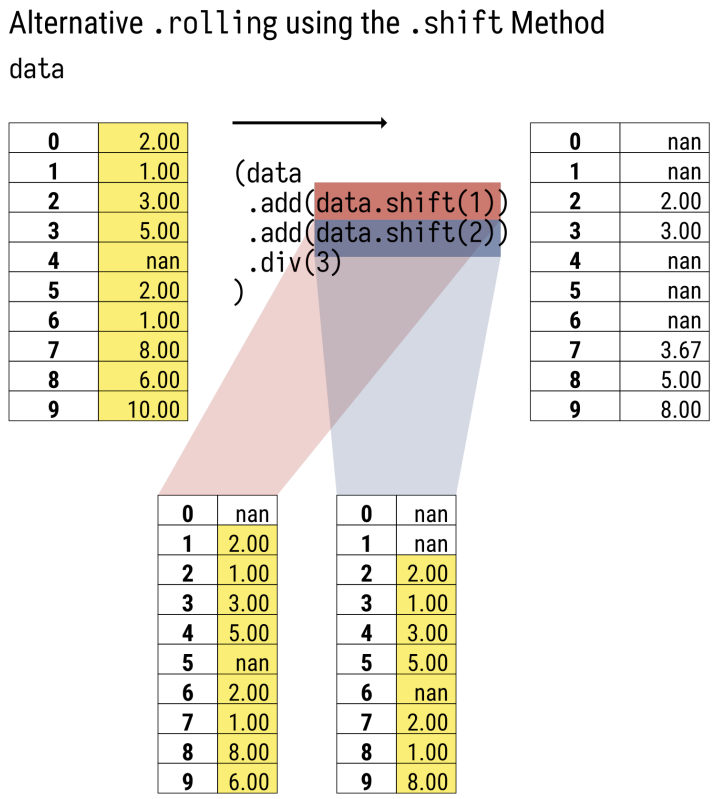


Figure 15.3: The `.rolling` method slide is similar to shifting the data for N-1 window size and then applying an aggregation.

will see that each date is the end of the month. If we want to aggregate at the end of every two months, we can use `'2ME'` as the offset alias:

```
>>> (snow
...     .resample('2ME')
...     .max()
... )
1980-01-31    20.0
1980-03-31    25.0
1980-05-31    10.0
              ...
2019-05-31    18.0
2019-07-31     0.0
```

```
2019-09-30     0.0
Freq: 2ME, Name: SNOW, Length: 239, dtype: double[pyarrow]
```

We could use the following code to aggregate the maximum value for each ski season, which normally ends in May. This offset alias, `'YE-MAY'`, indicates that we want a *year end* annual grouping (`'YE'`), but ending in May of each year:

```
>>> (snow
...     .resample('YE-MAY')
...     .max()
... )
1980-05-31    25.0
1981-05-31    26.0
1982-05-31    34.0
              ...
2018-05-31    21.8
2019-05-31    20.7
2020-05-31     0.0
Freq: YE-MAY, Name: SNOW, Length: 41, dtype: double[pyarrow]
```

Below is a table of the offset aliases.

Table 15.2: Offset aliases and date offset classes for `Grouper` and `.resample`

Offset Alias	Date Offset	Description
`None`	`DateOffset`	Default 1 day
`'A'`, `'YE'`	`YearEnd`	Calendar year end (Can specify `-MAY` to end year in May)
`'AS'`, `'YS'`	`YearBegin`	Calendar year start
`'BYE'`	`BYearEnd`	Business year end
`'BYS'`	`BYearBegin`	Business year start
`'RE'`	`FY5253`	Retail year end (52-53 week)
`'REQ'`	`FY5253Quarter`	Retail quarter end (52-53 week)
`'QE'`	`QuarterEnd`	Quarter end (Can specify `-JAN` to end quarter in January)
`'QS'`	`QuarterBegin`	Quarter start
`'BQE'`	`BQuarterEnd`	Business quarter end
`'BQS'`	`BQuarterBegin`	Business quarter start
`'ME'`	`MonthEnd`	Month end
`'MS'`	`MonthBegin`	Month start
`'BME'`	`BMonthEnd`	Business month end
`'BMS'`	`BMonthBegin`	Business month start
`'CBME'`	`CBMonthEnd`	Custom business month end
`'CBMS'`	`CBMonthBegin`	Custom business month start

Offset Alias	Date Offset	Description
'SME'	SemiMonthEnd	Semi-month end (15th and month end)
'SMS'	SemiMonthBegin	Semi-month start (15th and month start)
'W'	Week	Week (Can add -MON to end on Monday)
'WOM'	WeekOfMonth	Nth day of Mth week of month
'LWOM'	LastWeekOfMonth	Nth day of last week of month
'BH'	BusinessHour	Business hour
'CBH'	CustomBusinessHour	Custom business hour
'B'	BDay	Business day (weekday)
'C'	CDay	Custom business day
'D'	Day	Day
'H'	Hour	Hour
'T', 'min'	Minute	Minute
'S'	Second	Second
'L', 'ms'	Milli	Millisecond
'U', 'us'	Micro	Microsecond
'N'	Nano	Nanosecond

The result of calling `.resample` is a `DateTimeIndexResampler` object. It can perform many operations in addition to taking the maximum value (as shown in the examples). See the table in the next section.

15.8 Gathering Aggregate Values (But Keeping Index)

Below, instead of performing an aggregation with `.resample`, we leverage the `.transform` method, which works on aggregation groups but returns a series with the original index. This makes it easy to do things like calculating the percentage of quarterly snowfall that fell in a day:

```
>>> (snow
...     .div(snow
...          .resample('QE')
...          .transform('sum'))
...     .mul(100)
...     .fillna(0)
... )
1980-01-01    0.527009
1980-01-02    0.790514
1980-01-03    0.263505
                ...
2019-09-05       NaN
```

```
2019-09-06          NaN
2019-09-07          NaN
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

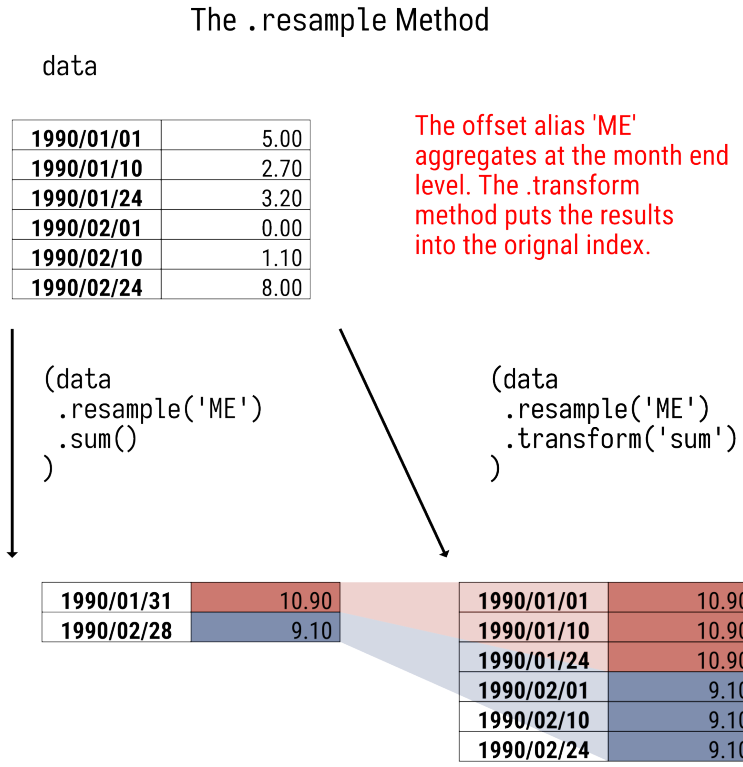


Figure 15.4: If you have dates in the index, you can use the `.resample` method to aggregate at date frequencies. The `.transform` method will take the resulting aggregates and place them back in the cell that contributed to the value (with the original index).

To compute the percentage of a season's snowfall that fell during each month, we could do the following:

```
>>> season2017 = snow.loc['2016-10':'2017-05']
>>> (season2017
...     .resample('ME')
...     .sum()
...     .div(season2017
```

```
...             .sum())
...     .mul(100)
... )
2016-10-31     2.153969
2016-11-30     9.772637
2016-12-31    15.715995
                 ...
2017-03-31     9.274033
2017-04-30    14.738732
2017-05-31     1.834862
Freq: ME, Name: SNOW, Length: 8, dtype: double[pyarrow]
```

Here is a table of the operations you can use on a resample object.

Table 15.3: Resampler Methods for a Series

Method	Description
`.agg(func, *args, **kwargs)`	Apply a function (to the group), string function name, list of functions, or dictionary (mapping column names to previous function/string/list). Returns a series if called with a single function, otherwise return a dateframe for multiple functions.
`.aggregate(func, *args, **kwargs)`	Same as `.agg`
`.apply(func, *args, **kwargs)`	Same as `.agg`
`.asfreq(fill_value=None)`	Return values at frequency (like `.reindex`)
`.backfill(limit=None)`	Backfill the missing values.
`.bfill(limit=None)`	Same as `.backfill`
`.count()`	Count of non-missing items in group.
`.ffill(limit=None)`	Forward fill the missing values.
`.fillna(method, limit=None)`	Method (`'ffill'`, `'bfill'`, or `'nearest'`) to use for filling in missing data for upsampling.
`.first()`	Return a series with the first value of each group.
`.get_group(name, obj=None)`	Return the series for grouping frequency of name.
`.interpolate( method='linear', axis=0, limit=None, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)`	Return a series with interpolated values.

Method	Description
`.last()`	Return a series with the final value from each group.
`.max()`	Return a series with maximum value from each group.
`.mean()`	Return a series with mean value from each group.
`.median()`	Return a series with median value from each group.
`.min()`	Return a series with minimum value from each group.
`.nearest(limit=None)`	Fill the missing values with nearest.
`.ngroups`	Property with number of groups in aggregation.
`.nunique()`	Return a series with the number of unique values from each group.
`.ohlc()`	Return a dataframe with columns for open, high, low, close.
`.pad(limit=None)`	Same as `.ffill`.
`.pipe(func, *args, **kwargs)`	Apply function to resampler object.
`.plot()`	Plot the groups.
`.prod()`	Return a series with the product of each group.
`.quantile(q=0.5)`	Return a series with the quantile. If `q` is a list, return a multi-index series.
`.sem()`	Return a series with the standard error of mean of each group.
`.size()`	Return a series with the size of each group (number of rows including missing values).
`.std()`	Return a series with the standard deviation of each group.
`.sum()`	Return a series with the sum of each group.
`.transform(function, *args, **kwargs)`	Return a series with the same index as the original (not grouped series). Function takes a group and returns a group with the same index.
`.var()`	Return a series with the variance of each group.

15.9 Groupby Operations

There is also a `.groupby` method that acts as a generic sort of `.resample`, and I use this more on dataframes than series. But here is an example of creating a function that will determine ski season by looking at the index with date information. It considers a season to be from October to September:

```
>>> def season(idx):
...     year = idx.year
...     month = idx.month
...     if month < 10:
...         return year
...     else:
...         return year + 1
...     return year.where((month < 10), year+ 1 )
```

We can now use this function with the `.groupby` method to aggregate all values for a season. Here we calculate the total snowfall for each season:

```
>>> (snow
...   .groupby(season)
...   .sum()
... )
1980    457.5
1981    503.0
1982    842.5
        ...
2017    524.0
2018    308.8
2019    504.5
Name: SNOW, Length: 40, dtype: double[pyarrow]
```

We can recreate the code from the previous section, where we calculated the percentage of snowfall that fell in each month for the 2017 season. But we can do it for all of the seasons in the dataset:

```
>>> def calc_pct(s):
...     return s.div(s.sum()).mul(100)

>>> (snow
...  .resample('ME')
...  .sum()
...  .groupby(season)
...  .apply(calc_pct)
... )
1980  1980-01-31     31.47541
      1980-02-29    24.590164
```

```
      1980-03-31    26.885246
                       ...
2019  2019-07-31          0.0
      2019-08-31          0.0
      2019-09-30          0.0
Name: SNOW, Length: 477, dtype: double[pyarrow]
```

This returns a series with a multi-index or hierarchical index. We will cover this in more detail in the grouping and reshaping chapters.

Note

Using an anchored offset alias, we could also do the above with `.resample`. The index would be a date instead of an integer:

```
>>> (snow
...   .resample('YE-SEP')
...   .sum()
... )

1980-09-30    457.5
1981-09-30    503.0
1982-09-30    842.5
              ...
2017-09-30    524.0
2018-09-30    308.8
2019-09-30    504.5
Freq: YE-SEP, Name: SNOW, Length: 40, dtype: double[pyarrow]
```

We will show more grouping operations like this when we dive into dataframes. Mastering these operations takes some time, but it has huge payoffs as it makes many calculations that would require creating a lot of declarative code easy.

15.10 Cumulative Operations

A handful of cumulative methods also work well with sequence data. These are `.cummin`, `.cummax`, `.cumprod`, and `.cumsum`. They return the cumulative minimum, maximum, product, and sum , respectively. To calculate the snowfall in a season, we can combine `.cumsum` with slicing:

```
>>> (snow
...     .loc['2016-10':'2017-09']
...     .cumsum()
... )
2016-10-01      0.0
2016-10-02      0.0
```

```
2016-10-03      4.9
                ...
2017-09-28    524.0
2017-09-29    524.0
2017-09-30    524.0
Name: SNOW, Length: 364, dtype: double[pyarrow]
```

Alternatively, if we wanted to do this calculation for every year, we can combine `.resample` with `.transform` and `'cumsum'`:

```
>>> (snow
...     .resample('YE-SEP')
...     .transform('cumsum')
... )
1980-01-01      2.0
1980-01-02      5.0
1980-01-03      6.0
                ...
2019-09-05    504.5
2019-09-06    504.5
2019-09-07    504.5
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

Table 15.4: Date Manipulation Methods

Method	Description
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, format=None, exact=True, unit='ns', infer_datetime_format=False, origin='unix', cache=True)`	Convert `arg` to date index, series, or timestamp for list, series, or scalar. Set `errors` to `'coerce'` to have invalid be `NaT`, `'ignore'` to leave. Specify strftime format with `format` or set `infer_datetime_format` to `True` if only one format type.
`.isna()`	Return boolean array (series) indicating where values are missing.
`.fillna(value=None, method=None, limit=None, downcast=None)`	Return series with missing values set to `value` (scalar, dict, series). Use `method` to fill additional holes (`'bfill'` or `'ffill'`) only `limit` times. Provide `downcast='infer'` to convert float to int if possible.

Method	Description
`.loc`	If the index is datetime, can use partial string indexing. `'2010'` to select all of 2010. `'2010-10'` to select Oct 2010. Stop index includes that stopping period. Indexing with `Timestamp` and `datetime` objects is not partial.
`.ffill(limit=None)`	Forward fill the missing values.
`.bfill(limit=None)`	Forward fill the missing values.

`.interpolate(`
`  method='linear', axis=0,`
`  limit=None, inplace=False,`
`  limit_direction='forward',`
`  limit_area=None,`
`  downcast=None, **kwargs,)`
	Return a series with interpolated values.

`.where(cond, other=nan,`
`  level=None,`
`  errors='raise',`
`  try_cast=False)`
	Return a series with values replaced with `other` where `cond` is `False`. `cond` can be boolean array or function (series passed in, return boolean array). `other` can be scalar, series, or function (series passed in, return scalar or series).

`.dropna()`	Return a series with missing values removed.
`.shift(periods=1, freq=None,` `  fill_value=None)`	Return a series with data shifted forward by `periods` (can be negative). If time series and `freq` is offset alias, index values are shifted to offset alias. Fill in empty values with `fill_value`.

`.rolling(window,`
`  min_periods=None,`
`  center=False,`
`  win_type=None,`
`  closed='right')`
	Return a `Window` or `Rolling` class to aggregate. `window` is number windows, offset alias (for time series), or `BaseIndexer`. Set `center=True` to label at the center of the window. To use a non-evenly weighted window, set `win_type` to string with Scipy window type.

`.resample(rule,`
`  closed='left',`
`  label='left',`
`  convention='start',`
`  kind=None, level=None,`
`  origin='start_day',`
`  offset=None)`
	Return `Resampler` object to aggregate on. Use `rule` to specify `DateOffset`, `TimeDelta`, or offset alias string.

Method	Description
`.transform(func)`	Return a series with the same index but with transformed values. Best when used on a `.groupby` or `.resample` result. `func` may be an aggregation function or string when called on groupby or resample.
`.groupby(by=None, level=None, sort=True, group_keys=True, observed=False, dropna=True)`	Return a groupby object to aggregate on. `by` may be a function (pass the index, return label), mapping (dict or series that maps index to label), or a sequence of labels. Use `observed=True` to limit combinatoric explosion with categorical series.
`.cummax(skipna=True)`	Return cumulative maximum of series
`.cummin(skipna=True)`	Return cumulative minimum of series
`.cumprod(skipna=True)`	Return cumulative product of series
`.cumsum(skipna=True)`	Return cumulative sum of series

15.11 Summary

This chapter explored many options for manipulating date information in pandas. Depending on whether you are manipulating dates in a series or dates in an index (time series), there are different options.

15.12 Exercises

With a dataset of your choice:

1. Convert a column with date information to a date.
2. Put the date information into the index for a numeric column.
3. Calculate the average value of the column for each month.
4. Calculate the average value of the column for every two months.
5. Calculate the percentage of the column out of the total for each month.
6. Calculate the average value of the column for a rolling window of size 7.
7. Using `.loc` pull out the first three months of a year.
8. Using `.loc` pull out the last four months of a year.

Chapter 16

Plotting with a Series

Inspecting statistical summaries and tables can reveal much about your data. Another technique to understand the data at a more intuitive level is to plot it. I am a massive fan of plotting, which has led to insights I do not believe I would have come across otherwise. I have used visualizations to debug and find errors in code. Mastering visualization will be a massive benefit to you.

In this chapter, we will explore how to create plots from series with pandas.

16.1 Plotting in Jupyter

Pandas has native integration with Matplotlib. When you create a plot, it should appear in Jupyter. In older versions of Jupyter, you needed to provide a cell magic to ensure the plots displayed in the browser:

```
%matplotlib inline
```

With modern Jupyter versions, this line is not required. If your plots don't appear, or you find yourself on an older version of Jupyter, you might want to include it.

16.2 The .plot Attribute

A series object has a `.plot` attribute. This attribute is interesting as you can call it directly to create plots or access sub-attributes of it. Let's load the snow data and make some plots:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/'\
...     'data/alta-noaa-1980-2019.csv'
>>> alta_df = pd.read_csv(url, dtype_backend='pyarrow')
>>> dates = pd.to_datetime(alta_df.DATE)
>>> snow = (alta_df
```

```
...     .SNOW
...     .rename(dates)
... )

>>> snow
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
              ...
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

The following plot attributes are available for plotting a series: `bar`, `barh`, `box`, `hist`, `kde`, and `line`. The next sections will explore them.

16.3 Histograms

If you have continuous numeric data, plotting a histogram can give you insight into how the data is distributed:

```
snow.plot.hist()
```

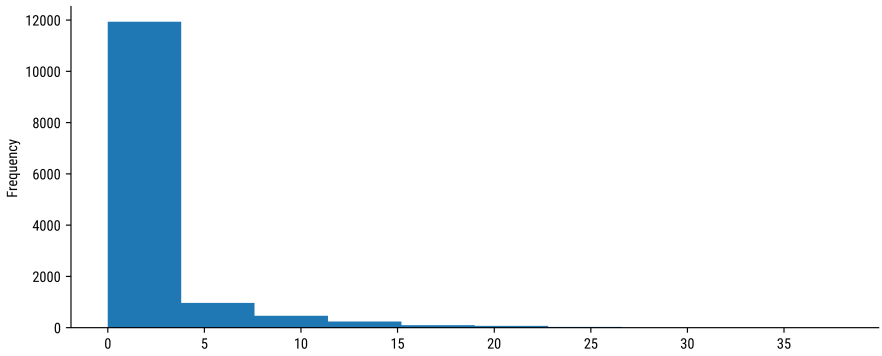


Figure 16.1: Basic histogram.

The snow data is heavily skewed. We might want to drop the zero entries and try again. We will also change the number of bins:

```
snow[snow>0].plot.hist(bins=20, title='Snowfall Histogram (in)')
```

You can also specify the bins directly with a list of numbers. The following code create bins from 0, 1-4, 5-9, 10-20, and 20+:

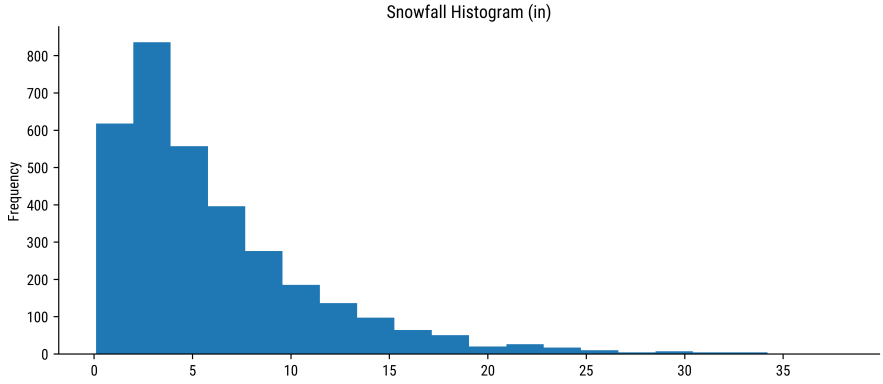


Figure 16.2: Histogram with zero values filtered out and 20 bins.

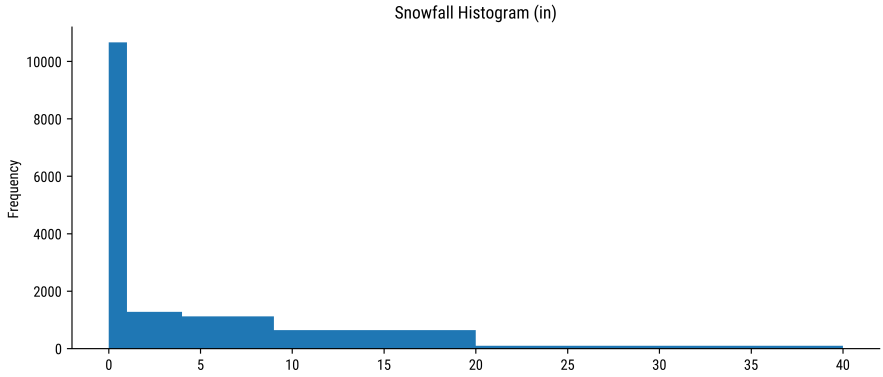


Figure 16.3: Histogram with custom bins.

16.4 Box Plot

You can also create a boxplot to view the distribution of the data. In this example, it does not look much like a box. This is because most of the time, it doesn't snow, so the plot shows that any time it snows is considered an outlier:

```
>>> snow.plot.box()
```

It looks boxier if we limit it to snow amounts during January (ignoring zero):

```
>>> (snow
...     [lambda s:(s.index.month == 1) & (s>0)]
```

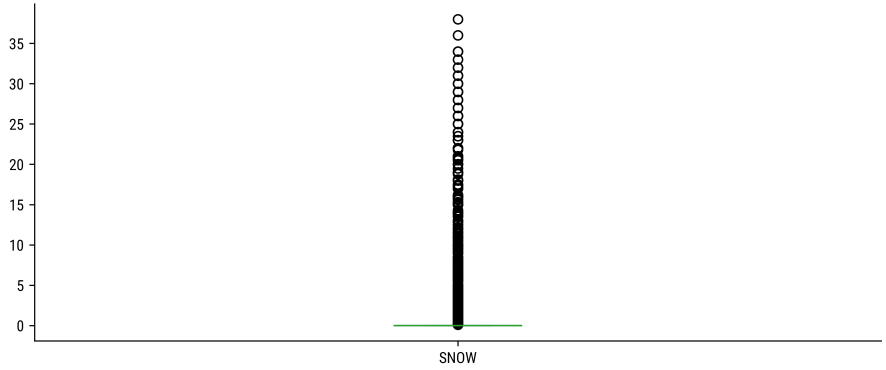


Figure 16.4: Basic boxplot.

```
...     .plot.box()
... )
```

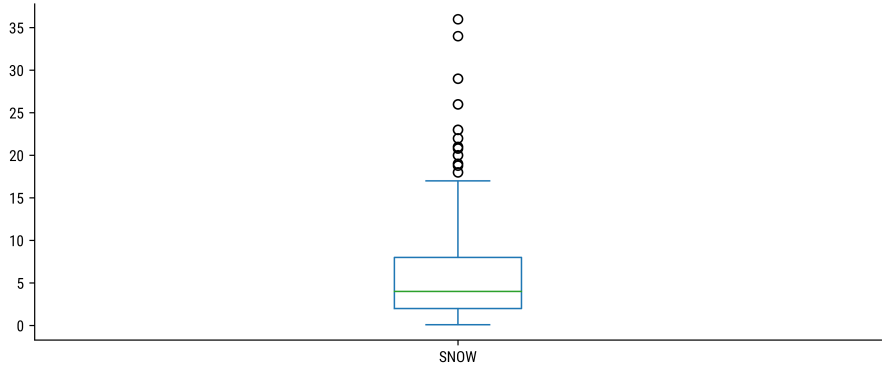


Figure 16.5: A better basic boxplot with snowfall levels for each January.

16.5 Kernel Density Estimation Plot

Another option is to view the kernel density estimation (KDE) plot. This is essentially a smoothed histogram:

```
>>> (snow
...     [lambda s:(s.index.month == 1) & (s>0)]
...     .plot.kde()
... )
```

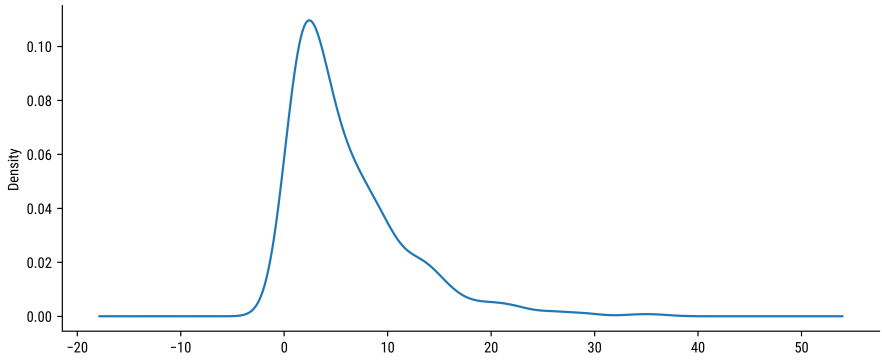


Figure 16.6: A basic kernel density estimate plot.

16.6 Line Plots

For numeric time series values, we can plot a line plot:

```
>>> snow.plot.line()
```

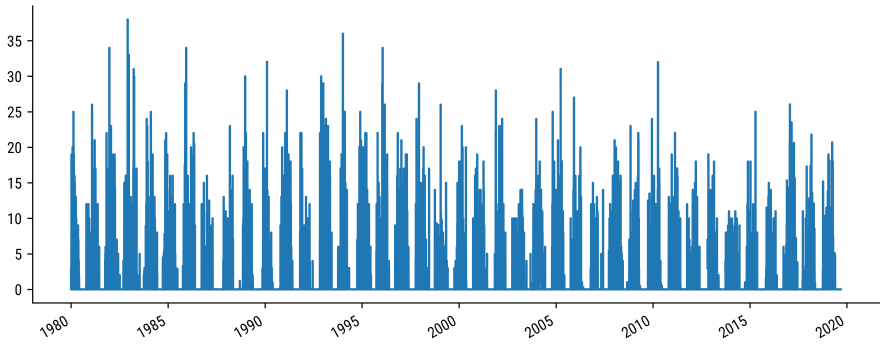


Figure 16.7: Basic line plot.

A line plot in pandas plots the values in the series on the y-axis and the index on the x-axis. This plot is a little crowded as we pack daily data for 40 years into the x-axis. We can slice off the last few years to zoom in or resample to view trends. Here, we pull off the last 300 values:

```
>>> (snow
...     .tail(300)
```

```
...     .plot.line()
... )
```

Note that by writing the code as above, I can easily comment out the line `.plot.line()` and inspect the series that will be plotted.

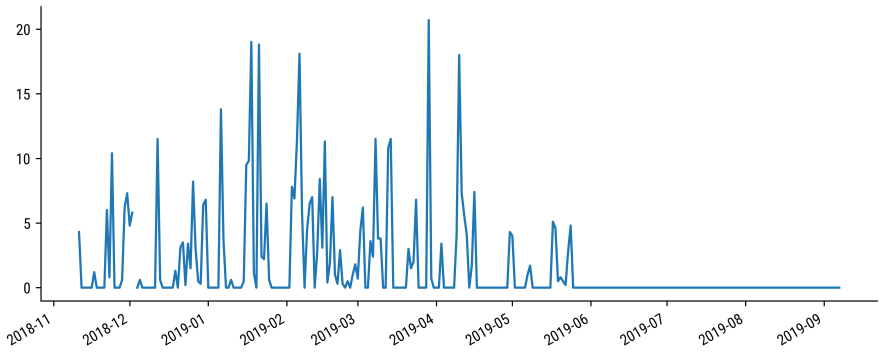


Figure 16.8: Last few values of basic line plot.

Here, I'm going to aggregate at the month end level and look at the mean snowfall using `.resample` with the `'ME'` offset alias and the `.mean` aggregation method:

```
>>> (snow
...  .resample('M')
...  .mean()
...  .plot.line()
... )
```

16.7 Line Plots with Multiple Aggregations

Plotting can be even more powerful with dataframes. To give you an idea, we will use the `.quantile` method to pull out the 50%, 90%, and 99% values at the quarter end level. This returns a series with multi-index (we will talk about those more later). If we chain the `.unstack` method, we can pull out the inner index (the one with the quantile names) into columns and create a dataframe that has a column for each quantile. If we plot this dataframe, each column will be its own line:

```
>>> (snow
...  .resample('QE')
...  .quantile([.5, .9, .99])
...  .unstack()
```

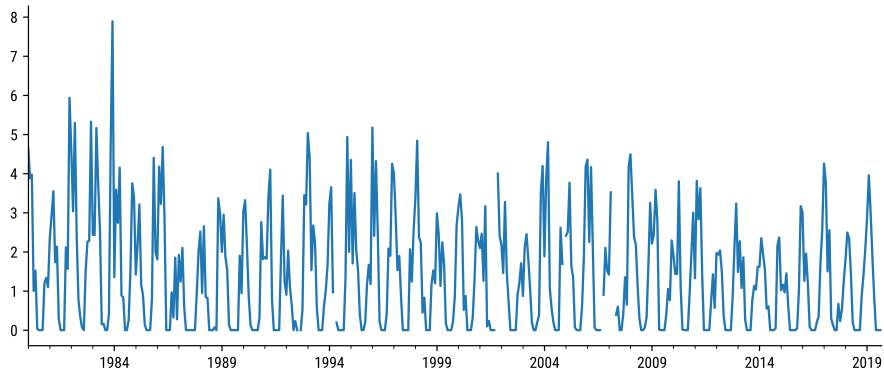


Figure 16.9: Resampled line plot.

```
...   .tail(100)
...   .plot.line()
... )
```

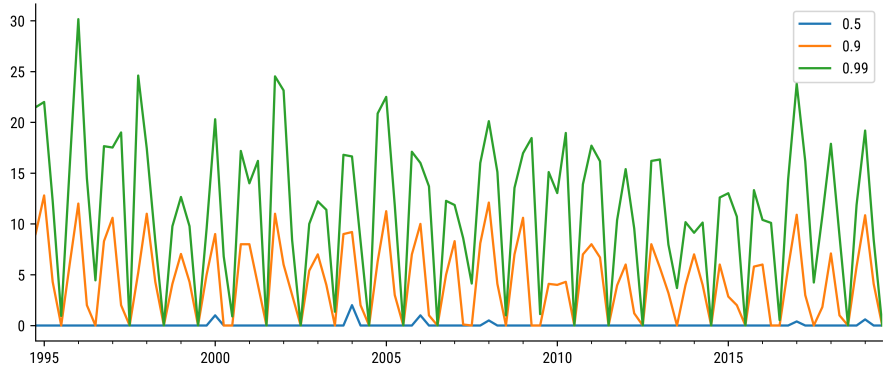


Figure 16.10: Resampled line plot from dataframe.

16.8 Bar Plots

You can also create bar plots. These are useful for comparing values. In the previous section, we looked at the percentage of snow that fell during each month:

```
>>> season2017 = (snow.loc['2016-10':'2017-05'])
>>> (season2017
```

```
...   .resample('ME')
...   .sum()
...   .div(season2017.sum())
...   .mul(100)
...   .rename(lambda idx: idx.month_name())
... )
October      2.153969
November     9.772637
December    15.715995
               ...
March        9.274033
April       14.738732
May          1.834862
Name: SNOW, Length: 8, dtype: double[pyarrow]
```

If you do a bar plot on a series, it will plot the index along the x-axis and draw a bar for each value. We will add a call to `.plot.bar` and set the title:

```
>>> (season2017
...   .resample('ME')
...   .sum()
...   .div(season2017.sum())
...   .mul(100)
...   .rename(lambda idx: idx.month_name())
...   .plot.bar(title='2017 Monthly Percent of Snowfall')
... )
```

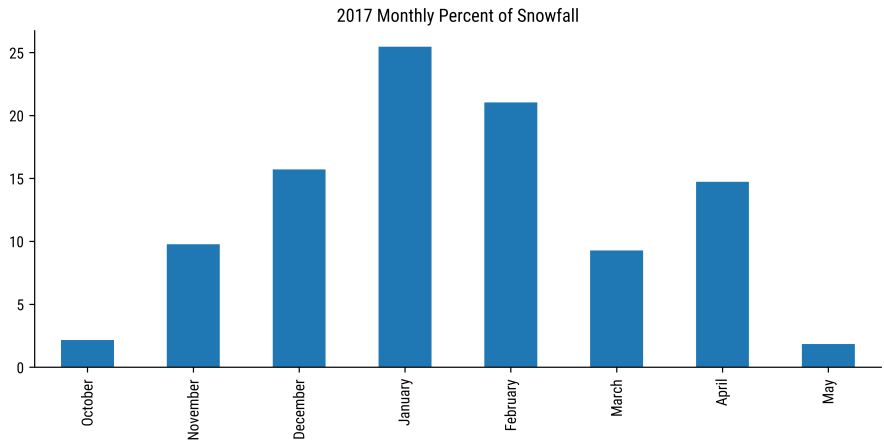


Figure 16.11: Basic series bar plot.

You can create a horizontal bar plot with the `.barh` method:

```
>>> (season2017
...    .resample('ME')
...    .sum()
...    .div(season2017.sum())
...    .mul(100)
...    .rename(lambda idx: idx.month_name())
...    .plot.barh(title='2017 Monthly Percent of Snowfall')
... )
```

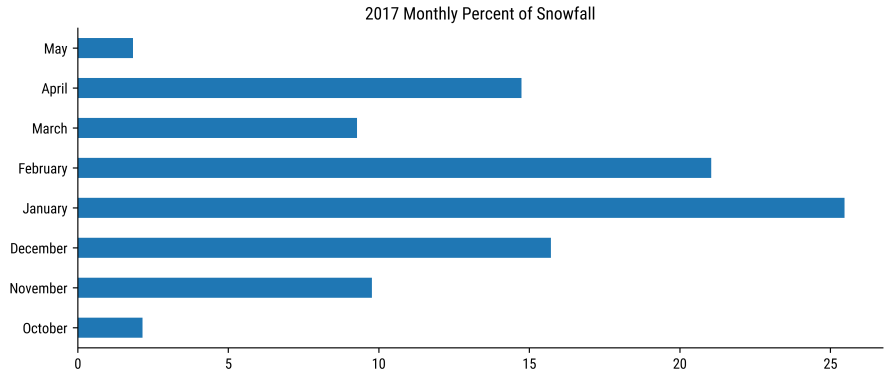


Figure 16.12: Basic series horizontal bar plot.

I like to use bar plots with categorical data. Let's pull in the makes of the auto data:

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       'vehicles.csv.zip'
>>> df = pd.read_csv(url, dtype_backend='pyarrow')
>>> make = df.make
```

The `.value_counts` method is my go-to tool for understanding the values in categorical data. It puts the categories in the index and counts as the values of the series:

```
>>> make.value_counts()
make
Chevrolet          4003
Ford               3371
Dodge              2583
                   ...
General Motors        1
Goldacre              1
```

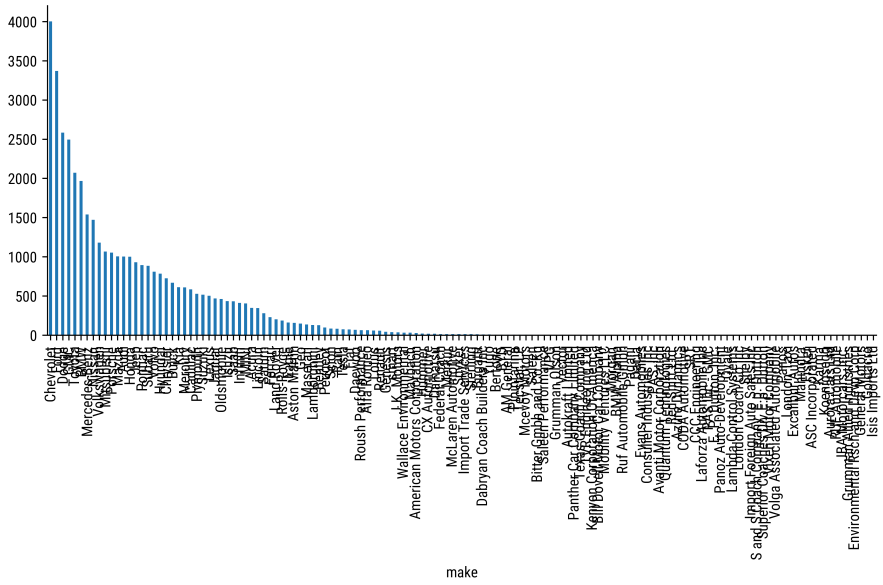


Figure 16.13: Crowded bar plot.

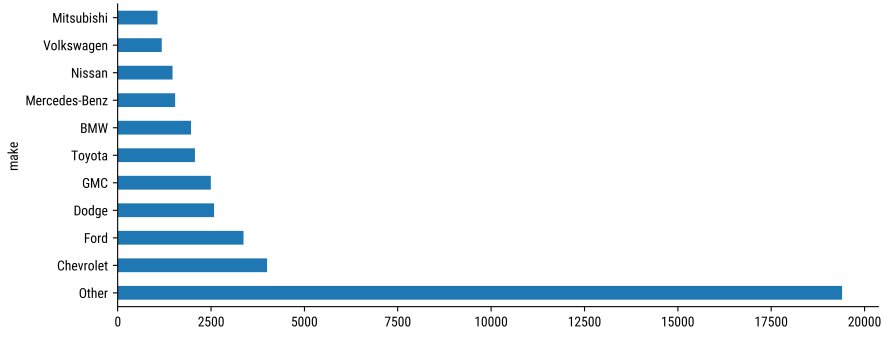


Figure 16.14: Grouping long-tail members together for legible bar plot.

```
Isis Imports Ltd       1
Name: count, Length: 136, dtype: int64[pyarrow]
```

It is also easy to visualize this by tacking on `.plot.bar`. This will plot the categories in the x-axis:

```
>>> (make
...     .value_counts()
...     .plot.bar()
... )
```

However, you can see that the plot is very crowded. As a rough rule of thumb, I don't like to create bar plots with more than 30 bars. Let's use some pandas code to limit this to 10 makes and plot it horizontally:

```
>>> top10 = make.value_counts().index[:10]
>>> (make
...     .where(make.isin(top10), 'Other')
...     .value_counts()
...     .plot.barh()
... )
```

16.9 Styling

You may notice that my plots don't look like the default plots of Matplotlib. I'm using the Seaborn library to set the font and color palette before plotting. Note that if you have a lot of text in the axis, you will need to use the `bbox_inches='tight'` option for the `.savefig` method or the export might be chopped off in the middle of the text. I'm also controlling the figure size with `plt.subplots` and passing in the resulting Matplotlib axes into the pandas `.plot` call. To do similar, you could use code like this:

```
import matplotlib
import seaborn as sns
with sns.plotting_context(rc=dict(font='Roboto', palette=color_palette)):
    fig, ax = plt.subplots(dpi=600, figsize=(10,4))
    snow.plot.hist()
    sns.despine()
    fig.savefig('snowhist.png', dpi=600, bbox_inches='tight')
```

Note that if your fonts are not showing up in your plots, you may need to delete the Matplotlib font cache. You can find the location of the cache with `matplotlib.get_cachedir()` and then delete the files in that directory. On my system, it is a JSON file located at `/home/matt/.matplotlib/fontlist-v300.json`.

Table 16.1: Series Plotting Methods

Method	Description
`s.plot(ax=None, style=None, logx=False, logy=False, xticks=None, yticks=None, xlim=None, ylim=None, xlabel=None, ylabel=None, rot=None, fontsize=None, colormap=None, table=False, **kwargs)`	Common plot parameters. Use `ax` to use existing Matplotlib axes, `style` for color and marker style (see `matplotlib.marker`), `_ticks` to specify tick locations, `_lim` to specify tick limits, `_label` to specify x/y label (default to index/column name), `rot` to rotate labels, `fontsize` for tick label size, `colormap` for coloring, `position`, `table` to create table with data. Additional arguments are passed to `plt.plot`
`s.plot.bar(position=.5, color=None)`	Create a bar plot. Use `position` to specify label alignment (0-left, 1-right). Use `color` (string, list) to specify line color.
`s.plot.barh(x=None, y=None, color=None)`	Create a horizontal bar plot. Use `position` to specify label alignment (0-left, 1-right). Use `color` (string, list) to specify line color.
`s.plot.hist(bins=10)`	Create a histogram. Use `bins` to change the number of bins. Can pass in a list of bin edges.
`s.plot.box()`	Create a boxplot.
`s.plot.kde( bw_method='scott', ind=None)`	Create a Kernel Density Estimate plot. Use `bw_method` to calculate estimator bandwidth (see `scipy.stats.gaussian_kde`). Use `ind` to specify evaluation points for PDF estimation (NumPy array of points, or integer with equally spaced points).
`s.plot.line(color=None)`	Create a line plot. Use `color` to specify line color.

16.10 Summary

In this chapter, we explored basic plotting functionality with series objects. We showed some of the functionality when plotting with a data frame. We will explore more of this later. Also, note that because the plotting functionality is built on top of Matplotlib, you can customize the plot using Matplotlib.

16.11 Exercises

With a dataset of your choice:

1. Create a histogram from a numeric column. Change the bin size.
2. Create a boxplot from a numeric column.
3. Create a Kernel Density Estimate plot from a numeric column.
4. Create a line from a numeric column.
5. Create a bar plot from the frequency count of a categorical column.

Chapter 17

Categorical Manipulation

So far, we have dealt with numeric and date data. Another common form of data is textual data, and a subset of textual data is categorical data. Categorical data is textual data that has repetitions. In this section, we will explore handling categorical data with pandas.

17.1 Categorical Data

Categories are labels that describe data. Generally, there are repeated values, and if they have an intrinsic order, they are referred to as *ordinal* values. One example is shirt sizes: small, medium, and large. Underordered values such as colors are called *nominal* values. In addition, you can convert numerical data to categories by binning them.

We will start by looking at the categorical values in the fuel economy data set. The *make* column has categorical information:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/' \
...       'data/vehicles.csv.zip'
>>> df = pd.read_csv(url, engine='pyarrow', dtype_backend='pyarrow')
>>> make = df.make
>>> make
0        Alfa Romeo
1           Ferrari
2             Dodge
              ...
41141         Subaru
41142         Subaru
41143         Subaru
Name: make, Length: 41144, dtype: string[pyarrow]
```

17.2 Frequency Counts

I like to use the `.value_counts` method to determine the *cardinality* of the values. The frequency of values will tell you if a column is categorical. If every value was unique or free-form text, it is not categorical:

```
>>> make.value_counts()
make
Chevrolet           4003
Ford                3371
Dodge               2583
                    ...
General Motors         1
Goldacre               1
Isis Imports Ltd       1
Name: count, Length: 136, dtype: int64[pyarrow]
```

We can also inspect the size and the number of unique items to infer the cardinality:

```
>>> make.shape, make.nunique()
((41144,), 136)
```

17.3 Benefits of Categories

The first benefit of categorical values is that they use less memory:

```
>>> cat_make = make.astype('category')
>>> make.memory_usage(deep=True)
425767

>>> cat_make.memory_usage(deep=True)
88701
```

Another benefit is that categorical computations can be faster for many operations. For example, we still have access to the `.str` attribute on categoricals. Let's compare creating uppercase results from a string type against a categorical type:

```
>>> %%timeit
>>> cat_make.str.upper()
357 µs ± 2.18 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
    each)

>>> %%timeit
>>> make.str.upper()
564 µs ± 1.59 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
    each)
```

This is one place where PyArrow operations have improved string performance. Compare this runtime to the runtime using the old legacy pandas 1.x string type:

```
>>> old_make = make.astype(str)

>>> %%timeit
>>> old_make.str.upper()
3.2 ms ± 16.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In this case, the same operation is ten times faster with the categorical data or using a PyArrow string. Note that the string operations do not return categorical series.

Also, remember that the binning functions that we showed previously, `pd.cut` and `pd.qcut`, create categorical results.

17.4 Conversion to Ordinal Categories

If you wanted to make an ordinal categorical (say alphabetic order) from the makes, and you want to specify the order (via the `categories` parameter), you could do the following:

```
>>> make_type = pd.CategoricalDtype(
...     categories=sorted(make.unique()), ordered=True)
>>> ordered_make = make.astype(make_type)
>>> ordered_make
0        Alfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: category
Categories (136, string[pyarrow]): [AM General < ASC Incorporated < Acura <
  Alfa Romeo ... Volvo < Wallace Environmental < Yugo < smart]
```

If you want to convert a categorical to an ordinal categorical and preserve the natural order, you can use the `.as_ordered` method off of the categorical accessor:

```
>>> (make
...  .astype('category')
```

```
...  .cat.as_ordered()
...  )
0        Alfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: category
Categories (136, string[pyarrow]): [AM General < ASC Incorporated < Acura <
  Alfa Romeo ... Volvo < Wallace Environmental < Yugo < smart]
```

A benefit of ordinal categoricals is that you can specify a lexical order to the items. If the items have an order, you can use reducing operations like maximum and minimum (where you can specify an order rather than using alphabetic order):

```
>>> ordered_make.max()
'smart'

>>> cat_make.max()
Traceback (most recent call last):
  ...
TypeError: Categorical is not ordered for operation max
you can use .as_ordered() to change the Categorical to an ordered one
```

You can also sort the series according to the order:

```
>>> ordered_make.sort_values()
20288    AM General
20289    AM General
369      AM General
358      AM General
19314    AM General
            ...
31289         smart
31290         smart
29605         smart
22974         smart
26882         smart
Name: make, Length: 41144, dtype: category
```

```
Categories (136, object): [AM General < ASC Incorporated < Acura <
    Alfa Romeo ... Volvo < Wallace Environmental < Yugo < smart]
```

17.5 The .cat Accessor

In addition, there are a few methods attached to the `.cat` attribute of categorical series. If you need to rename the categories, you can use the `.rename_categories` method. You need to pass in a list with the same length as the current categories or a dictionary mapping old values to new values. Here, we will lowercase the categories using both methods:

```
>>> cat_make.cat.rename_categories(
...     [c.lower() for c in cat_make.cat.categories])
0        alfa romeo
1           ferrari
2             dodge
3             dodge
4            subaru
            ...
41139        subaru
41140        subaru
41141        subaru
41142        subaru
41143        subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): [am general, asc incorporated, acura, alfa
    romeo, ..., volvo, wallace environmental, yugo, smart]

>>> ordered_make.cat.rename_categories(
...     {c:c.lower() for c in ordered_make.cat.categories})
0        alfa romeo
1           ferrari
2             dodge
3             dodge
4            subaru
            ...
41139        subaru
41140        subaru
41141        subaru
41142        subaru
41143        subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): [am general < asc incorporated < acura
     < alfa romeo ... volvo < wallace environmental < yugo < smart]
```

The `.cat` attribute also allows you to add or remove categories and change the order of nominal categories.

Here, we change the ordering. Previously *smart* was the maximum value because it was lowercase. Let's sort them ignoring case:

```
>>> ordered_make.cat.reorder_categories(
...     sorted(cat_make.cat.categories, key=str.lower))
0        Alfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
             ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): ['Acura' < 'Alfa Romeo' ...
    'Volvo' < 'VPG' < 'Wallace Environmental' < 'Yugo']
```

17.6 Category Gotchas

Here are a few oddities to be aware of with categorical data. Applying the `.value_counts` method or `.groupby` to categorical data uses all categories even if they have no values. In this example, we will look at the first hundred entries and count the frequency of entries. Note that this returns more than one hundred results because it includes every category!

```
>>> ordered_make.head(100).value_counts()
make
Dodge         17
Oldsmobile     8
Ford           8
              ..
Geo            0
Genesis        0
smart          0
Name: count, Length: 136, dtype: int64
```

Similarly, using the `.groupby` method will use all of the categories (this is even a bigger issue when we group by two categories with dataframes and get a combinatoric explosion):

```
>>> (cat_make
...  .head(100)
```

```
...  .groupby(cat_make.head(100), observed=False)
...  .first()
... )
make
AM General               <NA>
ASC Incorporated         <NA>
Acura                    <NA>
                         ...
Wallace Environmental    <NA>
Yugo                     <NA>
smart                    <NA>
Name: make, Length: 136, dtype: category
Categories (136, string[pyarrow]): [AM General, ASC Incorporated,
     Acura, Alfa Romeo, ..., Volvo, Wallace Environmental, Yugo, smart]
```

Compare this with just the result from the string series:

```
>>> (make
...  .head(100)
...  .groupby(make.head(100))
...  .first()
... )
make
Alfa Romeo    Alfa Romeo
Audi                Audi
BMW                  BMW
                   ...
Toyota            Toyota
Volkswagen    Volkswagen
Volvo              Volvo
Name: make, Length: 25, dtype: string[pyarrow]
```

There is an optional parameter, `observed`, for `.groupby` to tell it to only include results for which there are values:

```
>>> (cat_make
...  .head(100)
...  .groupby(cat_make.head(100), observed=True)
...  .first()
... )
make
Alfa Romeo    Alfa Romeo
Audi                Audi
BMW                  BMW
                   ...
Toyota            Toyota
```

```
Volkswagen    Volkswagen
Volvo              Volvo
Name: make, Length: 25, dtype: category
Categories (136, string[pyarrow]): [AM General, ASC Incorporated,
     Acura, Alfa Romeo, ..., Volvo, Wallace Environmental, Yugo, smart]
```

Also, note that pulling out a single value with `.iloc` will return a scalar, but if you pass in a list, it will return a categorical even if it is a single value:

```
>>> ordered_make.iloc[0]
'Alfa Romeo'

>>> ordered_make.iloc[[0]]
0    Alfa Romeo
Name: make, dtype: category
Categories (136, object): [AM General < ASC Incorporated < Acura
    < Alfa Romeo ... Volvo < Wallace Environmental < Yugo < smart]
```

17.7 Generalization

The manipulation methods chapter discussed generalizing categories when exploring the `.where` method. It is worth repeating similar code here since I find that I often want to limit the number of categorical values:

```
>>> def generalize_topn(ser, n=5, other='Other'):
...     topn = ser.value_counts().index[:n]
...     if isinstance(ser.dtype, pd.CategoricalDtype):
...         ser = ser.cat.set_categories(
...             topn.set_categories(list(topn)+[other]))
...     return ser.where(ser.isin(topn), other)

>>> cat_make.pipe(generalize_topn, n=20, other='NA')
0            NA
1            NA
2         Dodge
          ...
41141    Subaru
41142    Subaru
41143    Subaru
Name: make, Length: 41144, dtype: category
Categories (21, object): ['Chevrolet', 'Ford', 'Dodge', 'GMC', ...,
     'Volvo', 'Hyundai', 'Chrysler', 'NA']
```

Another generalization I like to make is hierarchical. Suppose I want country from make, but I only want US and German categories, and I want to label everything else as "Other":

```
>>> def generalize_mapping(ser, mapping, default):
...     seen = None
...     res = ser.astype(str)
...     for old, new in mapping.items():
...         mask = ser.str.contains(old)
...         if seen is None:
...             seen = mask
...         else:
...             seen |= mask
...         res = res.where(~mask, new)
...     res = res.where(seen, default)
...     return res.astype('category')

>>> generalize_mapping(cat_make, {'Ford': 'US', 'Tesla': 'US',
...     'Chevrolet': 'US', 'Dodge': 'US',
...     'Oldsmobile': 'US', 'Plymouth': 'US',
...     'BMW': 'German'}, 'Other')
0        Other
1        Other
2           US
         ...
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: category
Categories (3, object): ['German', 'Other', 'US']
```

Table 17.1: Category Attributes and Methods

Method	Description
`.astype(dtype)`	Return a series converted to categories. Set `dtype` to `'category'` for an unordered category, `CategoricalDType` for an ordered category.

`pd.CategoricalDtype(`
`  categories,`
`  ordered=False)`
	Create categorical type. Set `categories` to a list of categories.

`pd.cut(x, bins, right=True,`
`  labels=None,`
`  retbins=False,`
`  precision=3,`
`  include_lowest=False,`
`  duplicates='raise',`
`  ordered=True)`
	Bin values from `x` (a series). If `bins` is an integer, use equal-width bins. If `bins` is a list of numbers (defining minimum and maximum positions), use those for the edges. `right` defines whether the right edge is open or closed. `labels` allows us to specify bin names. Out of bounds values will be missing.

Method	Description
`pd.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')`	Bin values from `x` (a series) into `q` equal-sized bins (`10` for decile quantiles, `4` for quartile quantiles). Alternatively, we can pass in a list of quantile edges. Out of bounds values will be missing.
`.cat.add_categories(new_categories)`	Return a series with the new categories added. The new values are added at the end (highest) if it is ordinal.
`.cat.as_ordered()`	Convert categorical series to an ordered series. Use `.reorder_categories` or `CategoricalDtype` to specify the order.
`.cat.categories`	Property with the index of categories.
`.cat.codes`	Property with a series with category codes (index into a category).
`.cat.ordered`	Boolean property if series is ordered.
`.cat.remove_categories(removals)`	Return a series with the categories removed (replace with `NaN`).
`.cat.remove_unused_categories()`	Return a series with the categories removed that are being used.
`.cat.rename_categories(new_categories)`	Return a series with the categories replaced by a list (with new values) or a dictionary (mapping old to new values).
`.cat.reorder_categories(new_categories)`	Return a series with the categories replaced by a list.
`.cat.set_categories(new_categories, ordered=False, rename=False)`	Return a series with the categories replaced by a list.

17.8 Summary

If you are dealing with text data, it is worth considering whether converting the text data to categorical data makes sense. You can save a lot of memory and speed up many operations. A categorical series has a `.cat` attribute, allowing you to manipulate the categories.

17.9 Exercises

With a dataset of your choice:

1. Convert a text column into a categorical column. How much memory did you save?
2. Convert a numeric column into a categorical column by binning it (`pd.cut`). How much memory did you save?

3. Use the `generalize_topn` function to limit the amounts of categories in your column. How much memory did you save?

Chapter 18

Dataframes

In pandas, the two-dimensional counterpart to the one-dimensional `Series` is the `DataFrame`. If we want to understand this data structure, it helps to know how it is constructed. This chapter will introduce the dataframe.

18.1 Database and Spreadsheet Analogues

The interface will feel wrong if you think of a dataframe as row-oriented. Many tabular data structures are row-oriented. Perhaps this is due to spreadsheets and CSV files dealt with on a row-by-row basis. Perhaps it is due to the many OLTP[1] databases that are row-oriented out of the box. A `DataFrame` is often used for analytical purposes and is better understood when considered column-oriented, where each column is a `Series`.

Note

In practice, many highly optimized analytical databases (those used for OLAP cubes) are also column-oriented. Laying out the data in a columnar manner can improve performance and require fewer resources. Columns of a single type can be compressed easily. Performing analysis on a column requires loading only that column, whereas a row-oriented database would require reading the complete database to access an entire column.

18.2 A Simple Python Version

Below is a simple attempt to create a tabular Python data structure that is column-oriented. It has a 0-based integer index, but that is not required.

[1] *OLTP* (On-line Transaction Processing) characterizes databases that are meant for transactional data. Bank accounts are an example where data integrity is imperative, yet multiple users might need concurrent access. This contrasts with *OLAP* (Online Analytical Processing), which is optimized for complex querying and aggregation. Typically, reporting systems use these databases, which might store data in a denormalized form to speed up access.

The index could be string-based. Each column is similar to the Series-like structure developed previously:

```
>>> df = {
...   'index':[0,1,2],
...   'cols': [
...     { 'name':'growth',
...       'data':[.5, .7, 1.2] },
...     { 'name':'Name',
...       'data':['Paul', 'George', 'Ringo'] },
...   ]
... }
```

Rows are accessed via the index, and columns are accessible from the column name. Below are simple functions for accessing rows and columns:

```
>>> def get_row(df, idx):
...     results = []
...     value_idx = df['index'].index(idx)
...     for col in df['cols']:
...         results.append(col['data'][value_idx])
...     return results

>>> get_row(df, 1)
[0.7, 'George']

>>> def get_col(df, name):
...     for col in df['cols']:
...         if col['name'] == name:
...             return col['data']

>>> get_col(df, 'Name')
['Paul', 'George', 'Ringo']
```

18.3 Dataframes

Using the pandas `DataFrame` object, the previous data structure could be created like this:

```
>>> import pandas as pd
>>> df = pd.DataFrame({
...   'growth':[.5, .7, 1.2],
...   'Name':['Paul', 'George', 'Ringo'] })

>>> print(df)
   growth    Name
```

```
0     0.5    Paul
1     0.7  George
2     1.2   Ringo
```

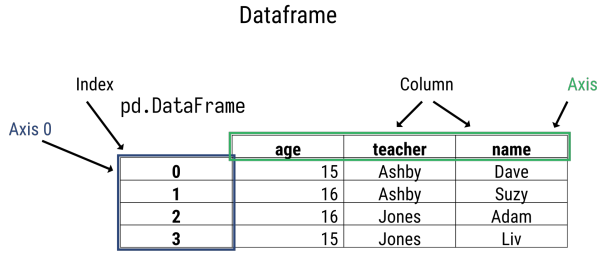


Figure 18.1: Figure showing column-oriented nature of Dataframe. (Note that a column can be pulled out as a `Series`

The index is the leftmost values, 0, 1, and 2. There are two columns, *growth* and *Name*. This data structure (like a series) has hundreds of attributes and methods. We will highlight many of the main features below.

One of the ways we can access a row is by location-indexing off of the `.iloc` attribute:

```
>>> df.iloc[2]
growth      1.2
Name      Ringo
Name: 2, dtype: object
```

Columns are also accessible via multiple methods. One is indexing the column name directly off of the object:

```
>>> df['Name']
0      Paul
1    George
2     Ringo
Name: Name, dtype: object
```

Note the type of column is a pandas `Series` instance. Any operation that can be done to a series can be applied to a column:

```
>>> type(df['Name'])
pandas.core.series.Series
```

```
>>> df['Name'].str.lower()
0      paul
1    george
2     ringo
Name: Name, dtype: object
```

Note

The `DataFrame` overrides `__getattr__` to allow access to columns as attributes. This tends to work ok but will fail if the column name conflicts with an existing method or attribute. It will also fail if the column has a non-valid attribute name (such as a column name with a space):

```
>>> df.Name
0      Paul
1    George
2     Ringo
Name: Name, dtype: object
```

You will find many who advise never to use attribute access to pull out a column, and they prefer using the index lookup. While the index lookup will work even with columns that do not have proper Python attribute names (alpha-numeric or underscore), I often use attribute access when using Jupyter! Why is that? Because tab completion works better when using attribute access. (I also tend to clean up my column names to non-conflicting Python attribute names.)

The above should explain why the `Series` was covered in such detail. When column operations are required, a series method is often involved. Also, the index behavior across both data structures is the same.

18.4 Construction

Dataframes can be created from many types of input:

- columns (dicts of lists)
- rows (list of dicts)
- CSV files (`pd.read_csv`)
- Parquet files
- NumPy ndarrays
- other: SQL, HDF5, arrow, etc

The previous creation of `df` illustrated making a dataframe from columns. Below is an example of creating a dataframe from rows:

```
>>> print(pd.DataFrame([
...   {'growth':.5, 'Name':'Paul'},
...   {'growth':.7, 'Name':'George'},
...   {'growth':1.2, 'Name':'Ringo'}]))
   growth    Name
0     0.5    Paul
1     0.7  George
2     1.2   Ringo
```

Similarly, here is an example of loading this data from a CSV file (I will mock out a file with `StringIO`):

```
>>> from io import StringIO
>>> csv_file = StringIO("""growth,Name
... .5,Paul
... .7,George
... 1.2,Ringo""")

>>> print(pd.read_csv(csv_file, dtype_backend='pyarrow', engine='pyarrow'))
   growth    Name
0     0.5    Paul
1     0.7  George
2     1.2   Ringo
```

The `pd.read_csv` function tries to be smart about its input. If you pass it a URL, it will download the file. If the extension ends in `.xz`, `.bz2`, or `.zip`, it will decompress the file automatically (you can provide a `compression='bz2'` parameter to explicitly force decompression of a file that has a different extension).

Note that if I want to use pyarrow types (and you do), you need to specify `dtype_backend='pyarrow'`. The `engine='pyarrow'` parameter uses pyarrow to speed up loading and parsing the CSV file.

After parsing the CSV file, pandas makes the best effort to give a type to each column. A "best-effort" means it will convert numerics to `int64[pyarrow]` if the column is whole numbers. Other numeric columns are converted to `float64[pyarrow]` (if they have decimals or are missing values). If there are non-numeric values, pandas will use the `pd.ArrowDtype(pa.string())` type.

One parameter to the `pd.read_csv` function is `dtypes`. It accepts a dictionary mapping column names to types. You can use the types listed below:

Table 18.1: types for `pd.read_csv` dtypes attribute

Type	Description
`float64[pyarrow]`	Floating point. Can specify different sizes, i.e., `float16`, `float32`, or `float64`.

Type	Description
`int64[pyarrow]`	Integer number. Can put `u` in front for unsigned. Can specify size, i.e., `int8`, `int16`, `int32`, or `int64`.
`datetime64[ns]`	Datetime number
`datetime64[ns, tz]`	Datetime number with timezone
`timedelta[ns]`	A difference between datetimes
`category`	Used to specify categorical columns
`object`	Used for other columns such as strings or Python objects
`pd.ArrowDtype(pa.string())`	Used for text data. Supports `<NA>` for missing values. Reported as `string[pyarrow]` in `dtype`

Note

Having said this, my experience with the `dtype` parameter is that converting many types after they are loaded into a dataframe is easier. I work on each column as a series and use the `.astype` method or one of the `to_*` functions at that point.

A dataframe can be instantiated from a NumPy array as well. The column names will need to be passed in as the `columns` parameter to the constructor:

```
>>> import numpy as np
>>> np.random.seed(42)
>>> print(pd.DataFrame(np.random.randn(10,3),
...     columns=['a', 'b', 'c']))
          a         b         c
0  0.496714 -0.138264  0.647689
1  1.523030 -0.234153 -0.234137
2  1.579213  0.767435 -0.469474
3  0.542560 -0.463418 -0.465730
4  0.241962 -1.913280 -1.724918
5 -0.562288 -1.012831  0.314247
6 -0.908024 -1.412304  1.465649
7 -0.225776  0.067528 -1.424748
8 -0.544383  0.110923 -1.150994
9  0.375698 -0.600639 -0.291694
```

18.5 Dataframe Axis

Unlike a series with one axis, there are two axes for a dataframe. They are commonly referred to as axis 0 and 1, or the `"index"` (or `'rows'`) axis and the `"columns"` axis, respectively:

```
>>> df.axes
[RangeIndex(start=0, stop=3, step=1),
 Index(['growth', 'Name'], dtype='object')]
```

For example, we can sum a dataframe down the index or along the columns using the labels 0 and 1:

```
>>> df.sum(axis=0)
growth               2.4
Name      PaulGeorgeRingo
dtype: object
```

Summing along the columns (`axis=1`) doesn't make much sense as it tries to add numbers to strings. In fact, pandas complains if we try to do it:

```
>>> df.sum(axis=1)
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

We can also spell out the axis by specifying a string. This is my preferred method because it is easier to read:

```
>>> df.sum(axis='index')
growth               2.4
Name      PaulGeorgeRingo
dtype: object
```

It still fails when we do it along the columns:

```
>>> df.sum(axis=1)
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

As many operations take an `axis` parameter, it is important to remember that 0 is the index and 1 is the columns:

```
>>> df.axes[0]
RangeIndex(start=0, stop=3, step=1)

>>> df.axes[1]
Index(['growth', 'Name'], dtype='object')
```

Note

Here is a clue to help you remember which axis is 0 and which is 1. Think back to a `Series`. It, like a `DataFrame`, has an index. *Axis 0 is along the index*.

A mnemonic to aid in remembering is that the 1 looks like a column (axis 1 is across columns):

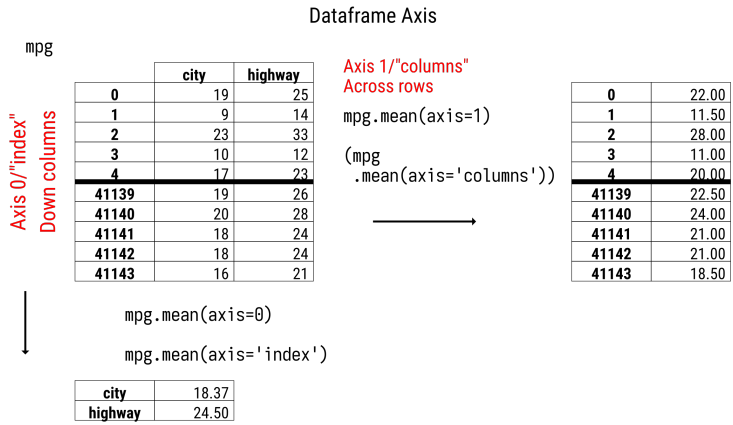
```
>>> df = pd.DataFrame({'Score1': [None, None],
...                    'Score2': [85, 90]})
>>> print(df)
  Score1  Score2
0   None      85
1   None      90
```

If we want to sum up each of the columns, then we sum down the index or row axis (`axis=0`):

```
>>> df.apply(np.sum, axis=0)
Score1      0
Score2    175
dtype: int64
```

To sum along every row, we sum across the columns axis (`axis=1`):

```
>>> df.apply(np.sum, axis='columns')
0    85
1    90
dtype: int64
```



	city	highway
0	19	25
1	9	14
2	23	33
3	10	12
4	17	23
41139	19	26
41140	20	28
41141	18	24
41142	18	24
41143	16	21

0	22.00
1	11.50
2	28.00
3	11.00
4	20.00
41139	22.50
41140	24.00
41141	21.00
41142	21.00
41143	18.50

city	18.37
highway	24.50

Figure 18.2: Figure showing the relation between axis 0 and axis 1. Note that when an operation is applied along axis 0, it is applied down the column. Likewise, operations along axis 1 operate across the values in the row.

Table 18.2: Dataframe creation

Code	Description
`pd.DataFrame(data=None, index=None, columns=None)`	Create a dataframe from scalar, sequence, dict, ndarray, or dataframe.
`.axes`	Tuple of index and columns.

18.6 Summary

This chapter introduced a Python data structure similar to how the pandas dataframe is implemented. It illustrated the index and the columnar nature of the dataframe. Then, we looked at the main components of the dataframe and how columns are just series objects. We saw various ways to construct dataframes. Finally, we looked at the two axes of the dataframe.

In future chapters, we will dig in more and see the dataframe in action.

18.7 Exercises

1. Create a dataframe with the names of your colleagues, their age (or an estimate), and their title.
2. Capitalize the values in the name column.
3. Sum up the values of the age column.

Chapter 19

Similarities with Series and DataFrames

We've spent a good portion of this book introducing the `Series` while mostly ignoring the other pandas class that you will use a lot, the `DataFrame`. Not to worry! Much of what we have discussed about a series object directly applies to a dataframe object.

In the following chapters, we will explore the similarities between the two classes before diving into the unique features of dataframes in the following chapters.

We will be exploring a dataset from a Siena College Poll in 2018. This data has rankings of United States Presidents in various attributes.

I was made aware of this dataset when one of my children pointed me to a visualization made from it. I will first pull the raw data and show how to recreate the visualization. Then, we will demonstrate more features of dataframes with the presidential data.

19.1 Getting the Data

Wikipedia has the data[1] from Siena College. I scraped the data using the following commands. (Given that Wikipedia can change anytime, there is no guarantee that this code will work for you.):

```
import pandas as pd
url = 'https://en.wikipedia.org/wiki/'\
      'Historical_rankings_of_presidents_of_the_United_States'
pres_dfs = pd.read_html(url, dtype_backend='pyarrow')
df = pres_dfs[2]
```

After I loaded the data, I removed some rows (the first and last), renamed the "Political Party" column to "Party", and then converted it to a categorical column type, using code like this:

[1] https://en.wikipedia.org/wiki/Historical_rankings_of_presidents_of_the_United_States

```
(df
     .iloc[1:-1]
     .rename(columns={'Political party': 'Party'})
     .assign(Party=lambda df_:df_
          .Party
          .str.replace(r'\[.*\]', '')
          .astype('category'))
)
```

Here are the column names with their associated explanation:

- Bg = Background
- Im = Imagination
- Int = Integrity
- IQ = Intelligence
- L = Luck
- WR = Willing to take risks
- AC = Ability to compromise
- EAb = Executive ability
- LA = Leadership ability
- CAb = Communication ability
- OA = Overall ability
- PL = Party leadership
- RC = Relations with Congress
- CAp = Court appointments
- HE = Handling of economy
- EAp = Executive appointments
- DA = Domestic accomplishments
- FPA = Foreign policy accomplishments
- AM = Avoid crucial mistakes
- EV = Experts' view
- O = Overall

At this point, I exported my data and saved it to a CSV (to avoid possible future changes at Wikipedia). You can load the data from my GitHub account:

```
>>> import pandas as pd

>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')

>>> print(df)
   Seq.          President                 Party  ...  AM  EV   O
1     1  George Washington           Independent  ...   1   2   1
2     2         John Adams            Federalist  ...  16  10  14
```

```
3     3   Thomas Jefferson  Democratic-Republican  ...   7   5   5
4     4      James Madison  Democratic-Republican  ...  11   8   7
..  ...                ...                    ...  ...  ..  ..  ..
41   42       Bill Clinton             Democratic  ...  30  14  15
42   43     George W. Bush             Republican  ...  36  34  33
43   44       Barack Obama             Democratic  ...  10  11  17
44   45       Donald Trump             Republican  ...  41  42  42

[44 rows x 24 columns]
```

Note that reading from CSV files is a risky proposition. We might lose fancy pandas types when we load from CSV, so I will double-check those:

```
>>> df.dtypes
Seq.         string[pyarrow]
President    string[pyarrow]
Party        string[pyarrow]
Bg            int64[pyarrow]
                  ...
FPA           int64[pyarrow]
AM            int64[pyarrow]
EV            int64[pyarrow]
O             int64[pyarrow]
Length: 24, dtype: object
```

Here is a function, `tweak_siena_pres`, to clean up this data:

```
>>> def tweak_siena_pres(df):
...     def int64_to_uint8(df_):
...         cols = df_.select_dtypes('int64')
...         return (df_
...                 .astype({col:'uint8[pyarrow]' for col in cols}))
...
...     return (df
...      .rename(columns={'Seq.':'Seq'})    # 1
...      .rename(columns={k:v.replace(' ', '_') for k,v in
...         {'Bg': 'Background',
...          'PL': 'Party leadership', 'CAb': 'Communication ability',
...          'RC': 'Relations with Congress', 'CAp': 'Court appointments',
...          'HE': 'Handling of economy', 'L': 'Luck',
...          'AC': 'Ability to compromise', 'WR': 'Willing to take risks',
...          'EAp': 'Executive appointments', 'OA': 'Overall ability',
...          'Im': 'Imagination', 'DA': 'Domestic accomplishments',
...          'Int': 'Integrity', 'EAb': 'Executive ability',
...          'FPA': 'Foreign policy accomplishments',
...          'LA': 'Leadership ability',
```

```
...         'IQ': 'Intelligence', 'AM': 'Avoid crucial mistakes',
...         'EV': "Experts' view", 'O': 'Overall'}.items()})
...       .astype({'Party':'category'})  # 2
...       .pipe(int64_to_uint8)  # 3
...       .assign(Average_rank=lambda df_:(df_.select_dtypes('uint8') # 4
...          .sum(axis=1).rank(method='dense').astype('uint8[pyarrow]')),
...               Quartile=lambda df_:pd.qcut(df_.Average_rank, 4,
...                     labels='1st 2nd 3rd 4th'.split())
...             )
...     )
```

Later chapters will detail the functionality exposed in the `tweak_siena_pres` function. I will briefly explain the chained operations.

The first call to `.rename` (#1) removes the period from the *Seq*.column. The next `.rename` call uses a dictionary comprehension to replace the shorted column names with the longer names but also replaces spaces with underscores. The call to `.astype` (#2) sets the type of the *Party* column to category. The resulting dataframe is passed to the `int64_to_uint8` function with the `.pipe` call (#3). This converts all the `int64` columns to unsigned 8-bit columns (since all the numeric data is below 44, we can store this information in a smaller type). The final call to `.assign` creates an *Average_rank* column by summing a row's numeric values and then taking the *dense* rank of the resulting values. It also makes a *Quartile* column by binning the *Average_rank* column into four bins.

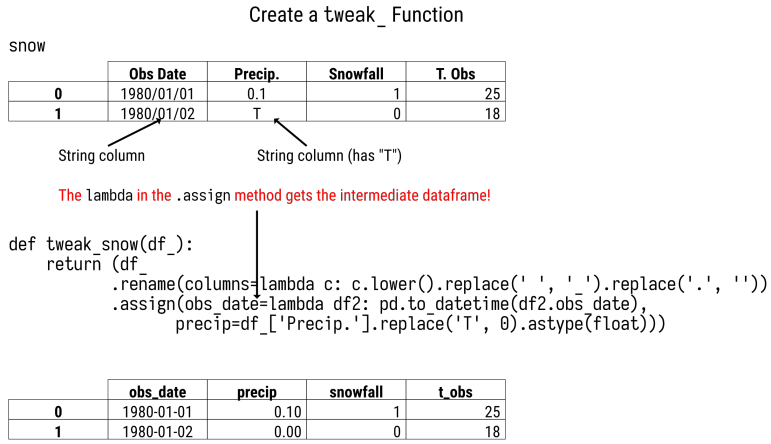


Figure 19.1: A tweak function is useful for maintaining order and sanity when working in Jupyter.

Note

You will see many examples of "tweak" functions later in this book. This is a pattern I like to follow. At the top of my Jupyter notebook, I will load the raw data into a dataframe. Then, in the cell below, I will make a tweak function (usually written with this chain style) that takes the raw data and returns a cleaned-up dataset.

This is advantageous for a few reasons. If you have used Jupyter for a while, you will know that your notebook may get unwieldy; it has many cells, and you may have executed them in an arbitrary order as you were working. When you return to your notebook, it can be hard to return to the state where your data is in the form you want it to be. Following this pattern makes it easy to open up a notebook, load the raw data, and then clean it up in the next cell.

Another advantage of writing this as a function is that you can pull this out and leverage it in production code.

I strongly recommend that you start adopting this practice in your notebooks and it will provide a significant improvement to your data workflow.

With this cleaned-up data, we can combine it with the Seaborn library to visualize the data. We will make a heatmap with Seaborn. Then, we will right-align the labels, rotate them, and add a title to the plot. Note that Seaborn is not particularly happy with pyarrow types [1] (as of Pandas 2.2, this is a Seaborn bug), so we will revert to NumPy types for the plotting:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> fig, ax = plt.subplots(figsize=(10,10), dpi=600)
>>> g = sns.heatmap((tweak_siena_pres(df)
...     .set_index('President')
...      .loc[:, 'Background':'Average_rank']
...      .astype('uint8')
...   ),annot=True, cmap='viridis', ax=ax)
>>> g.set_xticklabels(g.get_xticklabels(), rotation=45, fontsize=8,
...      ha='right')
>>> _ = plt.title('Presidential Ranking')
```

```
<Figure size 6000x6000 with 2 Axes>
```

The purpose of this chapter is not to look at visualizations but rather to see that most of what you can do with a series you can do with a dataframe. Let's start comparing.

[1] https://github.com/pandas-dev/pandas/issues/56270

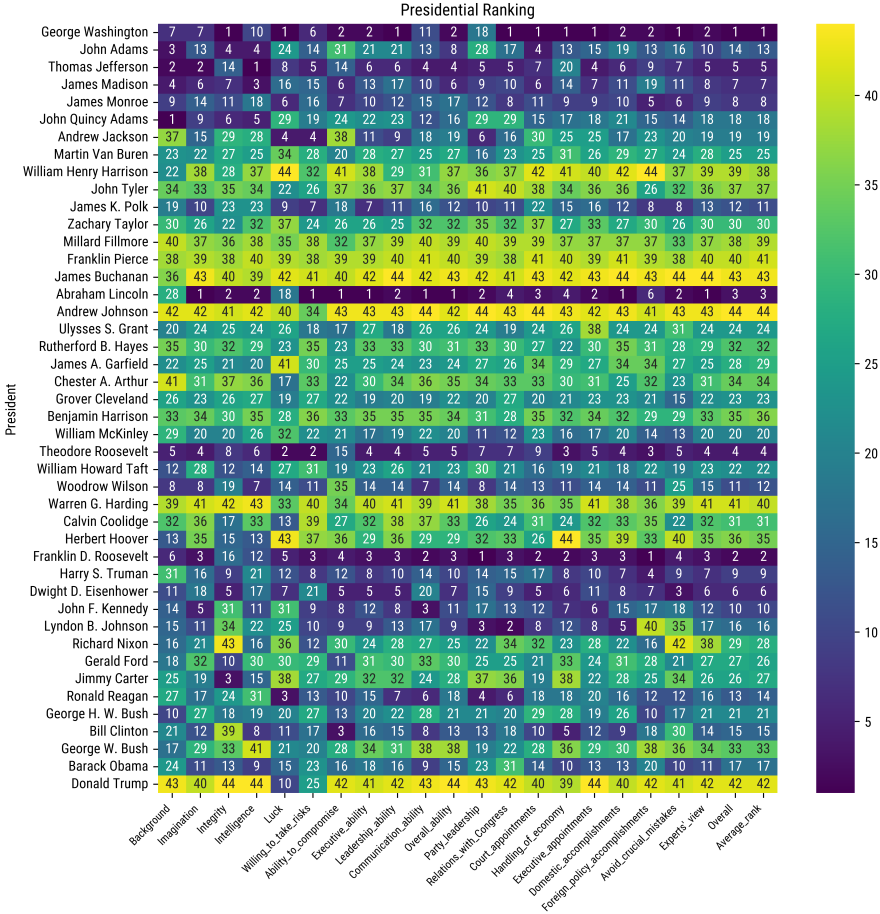


Figure 19.2: Visualization of United States presidential attributes.

19.2 Viewing Data

Dataframes have `.head` and `.tail` methods to view the data's first or last few rows. I also like to use `.sample`, as my experience is that the first few rows of data often do not represent the data as a whole. The rows at the top may be missing some entries or are test data:

```
>>> pres = tweak_siena_pres(df)
>>> print(pres.head(3))
  Seq          President                 Party  ...  Overall  \
1   1  George Washington            Independent  ...        1
2   2         John Adams             Federalist  ...       14
3   3   Thomas Jefferson  Democratic-Republican  ...        5

   Average_rank  Quartile
1             1       1st
2            13       2nd
3             5       1st

[3 rows x 26 columns]

>>> print(pres.sample(3))
   Seq          President       Party  ...  Overall  Average_rank  \
38  39       Jimmy Carter  Democratic  ...       26            27
27  28     Woodrow Wilson  Democratic  ...       11            12
35  36  Lyndon B. Johnson  Democratic  ...       16            16

    Quartile
38       3rd
27       2nd
35       2nd

[3 rows x 26 columns]
```

Table 19.1: Dataframe viewing Methods

Method	Description
`.head(n=5)`	Return a dataframe with the first `n` values.
`.tail(n=5)`	Return a dataframe with the last `n` values.
`s.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)`	Return a dataframe with `n` random entries. Can also specify a fraction with `frac` (if `frac` > 1, may specify `replace=True`).

19.3 Summary

This chapter demonstrated loading data from Wikipedia and then cleaning up the data, creating a "tweak" function. If you follow this pattern of making a function to clean up your data, it will make your life much easier when using pandas.

19.4 Exercises

With a tabular dataset of your choice:

1. Create a dataframe from the data.
2. View the first 20 rows of data.
3. Sample 30 rows from your data.

Chapter 20

Math Methods in DataFrames

We have seen that you can perform math operations on Series objects in pandas. This chapter will show that you can also perform math operations on dataframes.

We will begin by looking at the basic math operations. We will use a cleaned up version of the President data:

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')

>>> pres = tweak_siena_pres(df)
```

20.1 Index Alignment

We can perform math operations on the dataframe. There are the math methods like `.add` and `.div`, and we also have dunder methods that allow us to use the operators like `+`, `-`, `/`, and `*`.

Note that the index will *align* when we perform math. To demonstrate alignment, I will add the values from index values at rows 0-2 and column positions at index 0-3 and add them to the index values from rows 1-5 and 0-4:

```
>>> scores = (pres
...  .loc[:,'Background':'Average_rank']
... )
>>> print(scores)
   Background  Imagination  Integrity  Intelligence  ...  \
1           7            7          1            10  ...
2           3           13          4             4  ...
3           2            2         14             1  ...
4           4            6          7             3  ...
5           9           14         11            18  ...
```

```
..          ...          ...         ...           ...  ...
40           10           27          18            19  ...
41           21           12          39             8  ...
42           17           29          33            41  ...
43           24           11          13             9  ...
44           43           40          44            44  ...

    Avoid_crucial_mistakes  Experts'_view  Overall  Average_rank
1                        1              2        1             1
2                       16             10       14            13
3                        7              5        5             5
4                       11              8        7             7
5                        6              9        8             8
..                     ...            ...      ...           ...
40                      17             21       21            21
41                      30             14       15            15
42                      36             34       33            33
43                      10             11       17            17
44                      41             42       42            42

[44 rows x 22 columns]
```

We will pull out two sections of the data:

```
>>> s1 = scores.iloc[:3, :4]
>>> print(s1)
   Background  Imagination  Integrity  Intelligence
1           7            7          1            10
2           3           13          4             4
3           2            2         14             1

>>> s2 = scores.iloc[1:6, :5]
>>> print(s2)
   Background  Imagination  Integrity  Intelligence  Luck
2           3           13          4             4    24
3           2            2         14             1     8
4           4            6          7             3    16
5           9           14         11            18     6
6           1            9          6             5    29
```

Now let's add these together.

```
>>> print(s1 + s2)
   Background  Imagination  Integrity  Intelligence  Luck
1        <NA>         <NA>       <NA>          <NA>   NaN
2           6           26          8             8   NaN
3           4            4         28             2   NaN
```

```
4        <NA>        <NA>        <NA>        <NA>   NaN
5        <NA>        <NA>        <NA>        <NA>   NaN
6        <NA>        <NA>        <NA>        <NA>   NaN
```

Only the overlapping rows (rows 2 and 3) and columns (*Background* through *Intelligence*) get added together. The other values are missing!

20.2 Duplicate Index Entries

If you have duplicate index values, each index value in the left dataframe will match the index in the right dataframe. You should be aware if you have repeated index values before performing operations that align the index.

Let's add a dataframe to a dataframe that has duplicated values in the index (created by concatenating the dataframe with itself):

```
>>> print(scores.iloc[:3, :4] + pd.concat([scores.iloc[1:6, :5]]*2))
    Background  Imagination  Integrity  Intelligence  Luck
1         <NA>         <NA>       <NA>          <NA>   NaN
2            6           26          8             8   NaN
2            6           26          8             8   NaN
3            4            4         28             2   NaN
..         ...          ...        ...           ...   ...
5         <NA>         <NA>       <NA>          <NA>   NaN
5         <NA>         <NA>       <NA>          <NA>   NaN
6         <NA>         <NA>       <NA>          <NA>   NaN
6         <NA>         <NA>       <NA>          <NA>   NaN

[11 rows x 5 columns]
```

Notice that each index value in the left dataframe matches the index in the right dataframe. The values are added for every index and column combination that matches.

We can verify that the index is duplicated with the `.duplicated` method:

```
>>> pd.concat([scores.iloc[1:6, :5]]*2).index.duplicated().any()
True
```

20.3 More Math

Let's explore some common operations in data preparation for machine learning. We will look at normalization and standardization. Normalization is scaling the data in each column to values between 0 and 1. Standardization is scaling the data in each column to have a mean of 0 and a standard deviation of 1.

The formula for normalization is:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

The formula for standardization is:

$$x_{standardized} = \frac{x - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation.
Here's the code to normalize a single column:

```
>>> (scores.Imagination.sub(scores.Imagination.min())
...  .div(scores.Imagination.max() - scores.Imagination.min()))
1     0.142857
2     0.285714
3     0.023810
4     0.119048
        ...
41    0.261905
42    0.666667
43    0.238095
44    0.928571
Name: Imagination, Length: 44, dtype: double[pyarrow]
```

We can also write this with the math operations. This will make it easier to read. We will need to remember to wrap it with parentheses if we want to chain it with other methods:

```
>>> imag = scores.Imagination
>>> ((imag - imag.min()) / (imag.max() - imag.min()))
1     0.142857
2     0.285714
3     0.023810
4     0.119048
        ...
41    0.261905
42    0.666667
43    0.238095
44    0.928571
Name: Imagination, Length: 44, dtype: double[pyarrow]
```

We can use the `.describe` method to validate that the minimum and maximum values are 0 and 1, respectively:

```
>>> (scores.Imagination.sub(scores.Imagination.min())
...  .div(scores.Imagination.max() - scores.Imagination.min())
```

```
...  .describe())
count    44.000000
mean      0.494048
std       0.298095
min       0.000000
25%       0.238095
50%       0.488095
75%       0.744048
max       1.000000
Name: Imagination, dtype: double[pyarrow]
```

Let's see if we can normalize all of the numeric columns in the dataframe at once:

```
>>> nums = scores.select_dtypes('number')
>>> print(nums.sub(nums.min()).div(nums.max().sub(nums.min())))
    Background  Imagination  Integrity  Intelligence  ...  \
1     0.142857     0.142857   0.000000      0.209302  ...
2     0.047619     0.285714   0.069767      0.069767  ...
3     0.023810     0.023810   0.302326      0.000000  ...
4     0.071429     0.119048   0.139535      0.046512  ...
5     0.190476     0.309524   0.232558      0.395349  ...
..         ...          ...        ...           ...  ...
40    0.214286     0.619048   0.395349      0.418605  ...
41    0.476190     0.261905   0.883721      0.162791  ...
42    0.380952     0.666667   0.744186      0.930233  ...
43    0.547619     0.238095   0.279070      0.186047  ...
44    1.000000     0.928571   1.000000      1.000000  ...

    Avoid_crucial_mistakes  Experts'_view   Overall  Average_rank
1                 0.000000       0.023256  0.000000      0.000000
2                 0.348837       0.209302  0.302326      0.279070
3                 0.139535       0.093023  0.093023      0.093023
4                 0.232558       0.162791  0.139535      0.139535
5                 0.116279       0.186047  0.162791      0.162791
..                     ...            ...       ...           ...
40                0.372093       0.465116  0.465116      0.465116
41                0.674419       0.302326  0.325581      0.325581
42                0.813953       0.767442  0.744186      0.744186
43                0.209302       0.232558  0.372093      0.372093
44                0.930233       0.953488  0.953488      0.953488

[44 rows x 22 columns]
```

Let's try it with the operators:

```
>>> print(nums - nums.min()) / (nums.max() - nums.min())
    Background  Imagination  Integrity  ...  Experts'_view  Overall  \
1            6            6          0  ...              1        0
2            2           12          3  ...              9       13
3            1            1         13  ...              4        4
4            3            5          6  ...              7        6
..         ...          ...        ...  ...            ...      ...
41          20           11         38  ...             13       14
42          16           28         32  ...             33       32
43          23           10         12  ...             10       16
44          42           39         43  ...             41       41

    Average_rank
1              0
2             12
3              4
4              6
..           ...
41            14
42            32
43            16
44            41

[44 rows x 22 columns]

Background                <NA>
Imagination               <NA>
Integrity                 <NA>
Intelligence              <NA>
                          ...
Avoid_crucial_mistakes    <NA>
Experts'_view             <NA>
Overall                   <NA>
Average_rank              <NA>
Length: 22, dtype: uint8[pyarrow]
```

Again, we will use `.describe` to validate this.

```
>>> print(nums.sub(nums.min()).div(nums.max().sub(nums.min()))
...       .describe())
       Background  Imagination  Integrity  ...  Experts'_view  \
count   44.000000    44.000000  44.000000  ...      44.000000
mean     0.500000     0.494048   0.500000  ...       0.500000
std      0.295468     0.298095   0.298726  ...       0.298726
min      0.000000     0.000000   0.000000  ...       0.000000
25%      0.255952     0.238095   0.250000  ...       0.250000
```

```
50%      0.500000     0.488095   0.500000  ...       0.500000
75%      0.744048     0.744048   0.750000  ...       0.750000
max      1.000000     1.000000   1.000000  ...       1.000000

         Overall  Average_rank
count  44.000000     44.000000
mean    0.500000      0.500000
std     0.298726      0.298726
min     0.000000      0.000000
25%     0.250000      0.250000
50%     0.500000      0.500000
75%     0.750000      0.750000
max     1.000000      1.000000

[8 rows x 22 columns]
```

That looks like it worked.

If we wanted to put this logic into a scikit-learn transformer to use in machine learning pipelines, we need to keep track of the maximum and minimum during the `.fit` method and transform the data with the `.transform` method.

```
>>> from sklearn.base import BaseEstimator, TransformerMixin
>>> class NormalizerTransformer(BaseEstimator, TransformerMixin):
...     def __init__(self):
...         self.min = None
...         self.max = None
...
...     def fit(self, X, y=None):
...         self.min = X.min()
...         self.max = X.max()
...         return self
...
...     def transform(self, X):
...         return (X - self.min) / (self.max - self.min)

>>> nt = NormalizerTransformer()
>>> print(nt.fit_transform(nums))
    Background  Imagination  Integrity  ...  Experts'_view  \
1     0.142857     0.142857   0.000000  ...       0.023256
2     0.047619     0.285714   0.069767  ...       0.209302
3     0.023810     0.023810   0.302326  ...       0.093023
4     0.071429     0.119048   0.139535  ...       0.162791
..         ...          ...        ...  ...            ...
41    0.476190     0.261905   0.883721  ...       0.302326
42    0.380952     0.666667   0.744186  ...       0.767442
```

```
43    0.547619     0.238095   0.279070  ...       0.232558
44    1.000000     0.928571   1.000000  ...       0.953488

     Overall  Average_rank
1   0.000000      0.000000
2   0.302326      0.279070
3   0.093023      0.093023
4   0.139535      0.139535
..       ...           ...
41  0.325581      0.325581
42  0.744186      0.744186
43  0.372093      0.372093
44  0.953488      0.953488

[44 rows x 22 columns]
```

Let's write code to standardize data. We need to calculate the mean and standard deviation for each column. Then, we will subtract the mean and divide the result by the standard deviation.

```
>>> std = nums.std()
>>> print(nums.sub(nums.mean()).div(std))
    Background  Imagination  Integrity  ...  Experts'_view  \
1    -1.208734    -1.178117  -1.673773  ...      -1.595923
2    -1.531064    -0.698883  -1.440223  ...      -0.973124
3    -1.611646    -1.577478  -0.661724  ...      -1.362373
4    -1.450481    -1.257989  -1.206673  ...      -1.128823
..         ...          ...        ...  ...            ...
41   -0.080582    -0.778755   1.284523  ...      -0.661724
42   -0.402911     0.579074   0.817424  ...       0.895274
43    0.161165    -0.858627  -0.739574  ...      -0.895274
44    1.692228     1.457670   1.673773  ...       1.518073

     Overall  Average_rank
1  -1.673773     -1.673773
2  -0.661724     -0.739574
3  -1.362373     -1.362373
4  -1.206673     -1.206673
..       ...           ...
41 -0.583874     -0.583874
42  0.817424      0.817424
43 -0.428174     -0.428174
44  1.518073      1.518073

[44 rows x 22 columns]
```

Let's use `.describe` to validate that the mean is 0 and the standard deviation is 1:

```
>>> print(((nums - nums.mean()) / nums.std()).describe())
        Background   Imagination     Integrity  ...  Experts'_view  \
count  4.400000e+01  4.400000e+01  4.400000e+01  ...   4.400000e+01
mean  -3.532528e-17  1.009294e-17 -1.009294e-17  ...   2.018587e-17
std    1.000000e+00  1.000000e+00  1.000000e+00  ...   1.000000e+00
min   -1.692228e+00 -1.657350e+00 -1.673773e+00  ...  -1.673773e+00
25%   -8.259685e-01 -8.586273e-01 -8.368864e-01  ...  -8.368864e-01
50%    0.000000e+00 -1.996808e-02  0.000000e+00  ...   0.000000e+00
75%    8.259685e-01  8.386592e-01  8.368864e-01  ...   8.368864e-01
max    1.692228e+00  1.697287e+00  1.673773e+00  ...   1.673773e+00

         Overall  Average_rank
count  44.000000     44.000000
mean    0.000000      0.000000
std     1.000000      1.000000
min    -1.673773     -1.673773
25%    -0.836886     -0.836886
50%     0.000000      0.000000
75%     0.836886      0.836886
max     1.673773      1.673773

[8 rows x 22 columns]
```

It looks like it worked. The mean value isn't exactly 0, but that is because of rounding errors in the floating-point arithmetic. I won't convert this into a scikit-learn transformer, but you could practice doing that on your own. (Also note that scikit-learn has a `StandardScaler` transformer that does this for you.)

```
>>> from sklearn.preprocessing import StandardScaler

>>> ss = StandardScaler()
>>> results = (ss.fit_transform(nums))
>>> results[:2]
array([[-1.22270872, -1.19173682, -1.69312335, -0.98437404,
     -1.69312335,
        -1.33190648, -1.63027829, -1.63677071, -1.69312335,
     -0.90562412,
        -1.61437342, -0.35437465, -1.71690897, -1.69312335,
     -1.69312335,
        -1.69312335, -1.61437342, -1.61437342, -1.69312335,
     -1.61437342,
        -1.69312335, -1.69312335],
```

```
       [-1.54876438, -0.70696252, -1.45687358, -1.45687358,
     0.11812488,
        -0.65145499,  0.73362523, -0.09930968, -0.11812488,
     -0.74812427,
        -1.14187388,  0.43312458, -0.42417751, -1.45687358,
     -0.74812427,
        -0.59062442, -0.27562473, -0.74812427, -0.5118745 ,
     -0.98437404,
        -0.66937435, -0.74812427]])
```

By default, scikit-learn transformers will return NumPy arrays. If you want to return a dataframe, use the `return_df=True` parameter in the `.fit` method. You can also call the `.set_output` method so that the transformer will return a dataframe.

(Alternatively, you can use the `set_config(transform_output="pandas")` function in scikit-learn.)

```
>>> ss.set_output(transform='pandas')
>>> print(ss.fit_transform(nums))
    Background  Imagination  Integrity  ...  Experts'_view  \
1    -1.222709    -1.191737  -1.693123  ...      -1.614373
2    -1.548764    -0.706963  -1.456874  ...      -0.984374
3    -1.630278    -1.595715  -0.669374  ...      -1.378124
4    -1.467250    -1.272533  -1.220624  ...      -1.141874
..         ...          ...        ...  ...            ...
41   -0.081514    -0.787758   1.299374  ...      -0.669374
42   -0.407570     0.585769   0.826874  ...       0.905624
43    0.163028    -0.868554  -0.748124  ...      -0.905624
44    1.711792     1.474522   1.693123  ...       1.535624

     Overall  Average_rank
1  -1.693123    -1.693123
2  -0.669374    -0.748124
3  -1.378124    -1.378124
4  -1.220624    -1.220624
..       ...          ...
41 -0.590624    -0.590624
42  0.826874     0.826874
43 -0.433125    -0.433125
44  1.535624     1.535624

[44 rows x 22 columns]
```

20.4 PCA Calculation in Pandas

Let's do one more math example. We will perform principal component analysis (PCA) of the data. For those unfamiliar with principal components, it is a way to reduce the dimensionality of the data.

The key thing to realize for PCA is that it uses variance to represent the amount of information in the data. You probably think of variance as the spread of the data. A column with no variance (all the values are the same) has no information. You could remove that column and not lose any information from the data. In many finance, variance is used as a proxy for risk. A stock with a high variance is considered risky. In signal processing, variance is a measure of the power of a signal. In quality control, variance is used to measure stability. These properties hold stronger when the data is normally distributed. Non-normal data, outliers, and multi-modal data can cause problems with variance. We will ignore these issues for now and interpret variance as a measure of information.

At a high level, PCA uses variance to find the most important features in the data. It then creates new features that are a linear combination of the original features. The new features are called principal components. The first principal component is the linear combination of the original features with the most variance. The second principal component is the linear combination of the original features that has the second most variance, and so on. The principal components are orthogonal to each other (they are not correlated). The principal components are ordered by the amount of variance they explain. The first principal component explains the most variance. The second principal component explains the second most variance, and so on.

To calculate the principal components, we need to:

1. Center the data (subtract the mean from each column)
2. Calculate the covariance matrix
3. Calculate the eigenvectors and eigenvalues of the covariance matrix
4. Sort the eigenvectors by the eigenvalues
5. Multiply the centered data by the eigenvectors

Pandas doesn't have the math support to do this, so we will rely on NumPy. First, let's center the data:

```
>>> centered = nums - nums.mean()
>>> print(centered)
    Background  Imagination  Integrity  Intelligence  ...  \
1        -15.0       -14.75      -21.5         -12.5  ...
2        -19.0        -8.75      -18.5         -18.5  ...
3        -20.0       -19.75       -8.5         -21.5  ...
4        -18.0       -15.75      -15.5         -19.5  ...
5        -13.0        -7.75      -11.5          -4.5  ...
..         ...          ...        ...           ...  ...
```

```
40       -12.0         5.25       -4.5         -3.5  ...
41        -1.0        -9.75       16.5        -14.5  ...
42        -5.0         7.25       10.5         18.5  ...
43         2.0       -10.75       -9.5        -13.5  ...
44        21.0        18.25       21.5         21.5  ...

    Avoid_crucial_mistakes  Experts'_view  Overall  Average_rank
1                    -21.5          -20.5    -21.5         -21.5
2                     -6.5          -12.5     -8.5          -9.5
3                    -15.5          -17.5    -17.5         -17.5
4                    -11.5          -14.5    -15.5         -15.5
5                    -16.5          -13.5    -14.5         -14.5
..                     ...            ...      ...           ...
40                    -5.5           -1.5     -1.5          -1.5
41                     7.5           -8.5     -7.5          -7.5
42                    13.5           11.5     10.5          10.5
43                   -12.5          -11.5     -5.5          -5.5
44                    18.5           19.5     19.5          19.5

[44 rows x 22 columns]
```

Our next step (2) is to calculate the covariance matrix. A covariance matrix shows the *covariance* between each pair of columns. The covariance is a measure of how much two variables change together. If the covariance is positive, then the variables change together. (Note that the diagonal of the covariance matrix is the variance of each column.)

We will use the `.cov` method to calculate the covariance matrix:

```
>>> cov = centered.cov()
>>> print(cov)
                             Background  Imagination   Integrity  \
Background                   154.000000   105.976744  102.000000
Imagination                  105.976744   156.750000   94.290698
Integrity                    102.000000    94.290698  165.000000
Intelligence                 127.162791   136.895349  116.023256
Luck                          45.534884    93.406977   54.279070
...                                 ...          ...         ...
Foreign_policy_accomplishments  98.162791   130.337209  107.581395
Avoid_crucial_mistakes        82.511628   121.616279  123.023256
Experts'_view                104.697674   148.616279  119.418605
Overall                      111.651163   151.523256  113.976744
Average_rank                 113.116279   151.267442  114.116279

                             Intelligence  ...  \
Background                     127.162791  ...
Imagination                    136.895349  ...
```

```
Integrity                          116.023256  ...
Intelligence                       165.000000  ...
Luck                                56.302326  ...
...                                       ...  ...
Foreign_policy_accomplishments     121.232558  ...
Avoid_crucial_mistakes             102.302326  ...
Experts'_view                      133.209302  ...
Overall                            133.813953  ...
Average_rank                       134.279070  ...

                                 Avoid_crucial_mistakes  \
Background                                    82.511628
Imagination                                  121.616279
Integrity                                    123.023256
Intelligence                                 102.302326
Luck                                         115.162791
...                                                 ...
Foreign_policy_accomplishments               140.837209
Avoid_crucial_mistakes                       165.000000
Experts'_view                                148.302326
Overall                                      144.186047
Average_rank                                 144.069767

                                 Experts'_view     Overall  \
Background                           104.697674  111.651163
Imagination                          148.616279  151.523256
Integrity                            119.418605  113.976744
Intelligence                         133.209302  133.813953
Luck                                 113.441860  113.372093
...                                         ...         ...
Foreign_policy_accomplishments       143.279070  147.116279
Avoid_crucial_mistakes               148.302326  144.186047
Experts'_view                        165.000000  162.372093
Overall                              162.372093  165.000000
Average_rank                         162.116279  164.837209

                                 Average_rank
Background                         113.116279
Imagination                        151.267442
Integrity                          114.116279
Intelligence                       134.279070
Luck                               112.883721
...                                       ...
Foreign_policy_accomplishments     147.325581
Avoid_crucial_mistakes             144.069767
Experts'_view                      162.116279
```

```
Overall                          164.837209
Average_rank                     165.000000

[22 rows x 22 columns]
```

Now, we need to calculate the eigenvectors and eigenvalues of the covariance matrix (3). The eigenvectors determine the direction of a new space that maximizes the variance of the data. The eigenvalues represent the magnitude of the eigenvectors. Because the eigenvectors are the variance maximizing directions, the eigenvalues measure how much variance each eigenvector explains. You can calculate the percent of variance explained by each eigenvector by dividing the eigenvalue by the sum of the eigenvalues.

The eigenvectors represent the principal components. The eigenvalues represent the amount of variance explained by each principal component.

Pandas doesn't have a method to calculate the eigenvectors and eigenvalues, so we will use NumPy.

```
>>> import numpy as np
>>> vals, vecs = np.linalg.eig(cov)
```

Here are the eigenvectors, the magnitude of the variance in the corresponding direction:

```
>>> vals
array([2.88328405e+03, 2.00540034e+02, 1.24825903e+02, 7.72533366e+01,
       6.16970677e+01, 5.32032961e+01, 3.65440244e+01, 2.35890881e+01,
       2.01735967e+01, 1.78521221e+01, 1.32601928e+01, 1.27602591e+01,
       8.82620044e+00, 6.68501177e+00, 7.95455528e-02, 5.18161347e+00,
       9.17189176e-01, 1.47197226e+00, 1.77825675e+00, 2.07087071e+00,
       3.22018748e+00, 3.99812325e+00])
```

If we divide the eigenvalues by the sum of the eigenvalues, we get the percent of variance explained by each eigenvector. In this example, the first eigenvector explains 81% of the variance, and the second explains 5.6% of the variance:

```
>>> vals / vals.sum()
array([8.10090576e-01, 5.63439427e-02, 3.50712193e-02, 2.17051802e-02,
       1.73344742e-02, 1.49480550e-02, 1.02674482e-02, 6.62761546e-03,
       5.66799533e-03, 5.01575135e-03, 3.72559796e-03, 3.58513607e-03,
       2.47981873e-03, 1.87822807e-03, 2.23492037e-05, 1.45583167e-03,
       2.57694453e-04, 4.13566903e-04, 4.99620920e-04, 5.81834054e-04,
       9.04747322e-04, 1.12331699e-03])
```

Here are the first two eigenvectors in a NumPy array. This just looks like a bunch of numbers. The first row is the weights to multiply the first column, *Background*, by to get the contribution to each vector. When we sort

these columns in order of the eigenvalues, this becomes the weights for the principal components:

```
>>> vecs[:2].round(2)
array([[-0.16, -0.44,  0.19, -0.03,  0.57,  0.35,  0.34, -0.02,  0.13,
        -0.31,  0.11,  0.06,  0.1 , -0.12,  0.02,  0.01,  0.08,  0.06,
        -0.04, -0.08, -0.01, -0.  ],
       [-0.22, -0.03,  0.23,  0.14, -0.14, -0.04, -0.12, -0.02, -0.19,
        -0.17,  0.35, -0.14,  0.15,  0.14, -0.07,  0.19, -0.06,  0.35,
        -0.19,  0.34,  0.41,  0.33]])
```

Now, we will sort the eigenvectors by the eigenvalues (4). I will put them in a series and then we will use the `.argsort` method to get the indices that will sort the eigenvalues in descending order:

```
>>> idxs = pd.Series(vals).argsort()
>>> idxs
0     14
1     16
2     17
3     18
      ..
18     3
19     2
20     1
21     0
Length: 22, dtype: int64
```

Now, let's put the eigenvectors in a dataframe and order the columns by the eigenvalues. I will also label the columns and index to make understanding what is in the dataframe easier. (Sadly, pandas doesn't have a method to set the columns with a list of values, so I'm combining `.pipe` with mutating the `.columns` attribute.)

```
>>> def set_columns(df_):
...     df_.columns = [f'PC{i+1}' for i in range(len(df_.columns))]
...     return df_

>>> comps = (pd.DataFrame(vecs, index=nums.columns)
... .iloc[:, idxs[::-1]]
... .pipe(set_columns)
... )

>>> print(comps)
                               PC1       PC2       PC3  ...      PC20  \
Background           -0.163793 -0.438263  0.185140  ...  0.061550
Imagination          -0.221305 -0.033741  0.233670  ...  0.347679
```

```
Integrity              -0.163592 -0.462030 -0.492682  ...  0.123903
Intelligence           -0.197460 -0.417192  0.193787  ... -0.175805
...                          ...       ...       ...  ...       ...
Avoid_crucial_mistakes -0.208037  0.035619 -0.486695  ... -0.131391
Experts'_view          -0.235395 -0.014226 -0.110834  ...  0.148025
Overall                -0.238318  0.003287 -0.023624  ... -0.520106
Average_rank           -0.238427 -0.002321 -0.019445  ... -0.565484

                            PC21      PC22
Background              0.084777  0.017618
Imagination            -0.060294 -0.065134
Integrity              -0.022157 -0.012286
Intelligence           -0.074996  0.005522
...                          ...       ...
Avoid_crucial_mistakes -0.077041  0.039960
Experts'_view           0.611462 -0.013025
Overall                -0.121977  0.691586
Average_rank           -0.075944 -0.710245

[22 rows x 22 columns]
```

Each column represents the weights that we need to multiply the centered columns by to get the principal components.

Finally (5), we will take the dot product of the centered data and the eigenvectors to get the principal components. I will rename the columns to `PC1` and `PC2`. I'm using the `.dot` method to perform the dot product. I'm doing the dot product inside of I use the `.pipe` method to have the operands to the dot product in the correct order (the centered data needs to be the left operand and the vectors on the right). Because the *centered* dataframe index aligns with the *comps* columns, pandas will do the math correctly.

```
>>> print(centered.dot(comps))
          PC1        PC2        PC3  ...      PC20      PC21  \
1   87.474845   0.343354  15.136090  ... -0.957798  2.298340
2   36.726875  28.341533  -0.141936  ... -0.562758 -0.283342
3   74.211148   4.599394  -8.008068  ...  0.833371  0.835738
4   59.347214  14.357032  -1.319422  ...  1.967425  0.336845
..        ...        ...        ...  ...       ...       ...
41  37.511500 -11.828719 -17.302885  ... -0.503438 -0.282657
42 -38.920777 -10.401679  -7.763689  ... -1.326665 -0.785939
43  31.159052   6.487569   7.475127  ... -1.018540  0.339383
44 -82.841270 -22.261869   3.090102  ...  1.406558 -0.464754

        PC22
1   0.084040
2   0.234370
```

```
3   0.097331
4  -0.069161
..       ...
41  0.270175
42 -0.104504
43  0.259618
44 -0.311421

[44 rows x 22 columns]
```

Our final code looks like this. Note that I'm using `numpy.linalg.eig` which returns numpy arrays. I'm trying to stay in the pandas world, so I'm converting the arrays to dataframes.

```
import numpy as np
centered = nums - nums.mean() # 1
vals, vecs = np.linalg.eig(centered.cov()) # 2 & 3
idxs = pd.Series(vals).argsort() # 4

explained_variance = pd.Series(sorted(vals, reverse=True),
    index=[f'PC{i+1}' for i in range(len(nums.columns))])

def set_columns(df_):
    df_.columns = [f'PC{i+1}' for i in range(len(df_.columns))]
    return df_

comps = (pd.DataFrame(vecs, index=nums.columns)
 .iloc[:, idxs[::-1]]
 .pipe(set_columns)
)

pcas = (centered.dot(comps)) # 5
```

This example demonstrated doing a bit more math in pandas. If pandas doesn't have the math support you need, you can always use NumPy or SciPy.

In the real world, you would use scikit-learn to calculate the principal components. I'm showing you how to do it in pandas so that you can see the math behind the scenes. Here is the code to do it in scikit-learn. Note that the column names in the output of the scikit-learn code are wrong in my opinion. (*PCA* stands for principal components analysis; the first column is not an "analysis" but the first principal component. Also, it is numbered starting at 0, which is inconsistent with the ML community conventions.)

Also, note that the signs of the columns are different. This is because the signs of the eigenvectors are arbitrary. The signs of the eigenvectors can be flipped and still be valid. Don't worry too much about that, the key thing is the magnitude of the values are the same.

```
>>> from sklearn.decomposition import PCA
>>> from sklearn import set_config
>>> set_config(transform_output="pandas")
>>> pca = PCA()
>>> print(pca.fit_transform(nums).round(2))
     pca0   pca1   pca2   pca3  ...  pca18  pca19  pca20  pca21
1  -87.47   0.34 -15.14   0.32  ...   0.08   0.96  -2.30   0.08
2  -36.73  28.34   0.14 -12.80  ...   0.42   0.56   0.28   0.23
3  -74.21   4.60   8.01  -6.24  ...   1.22  -0.83  -0.84   0.10
4  -59.35  14.36   1.32   7.81  ...  -2.85  -1.97  -0.34  -0.07
5  -55.38  -3.72 -13.07   2.82  ...   0.94  -1.69   2.49   0.19
..    ...    ...    ...    ...  ...    ...    ...    ...    ...
40  -6.07   6.79 -10.23   2.98  ...   0.63   1.44  -0.30   0.04
41 -37.51 -11.83  17.30   9.90  ...   0.83   0.50   0.28   0.27
42  38.92 -10.40   7.76   6.25  ...   3.11   1.33   0.79  -0.10
43 -31.16   6.49  -7.48  -4.75  ...   1.69   1.02  -0.34   0.26
44  82.84 -22.26  -3.09 -14.13  ...   0.18  -1.41   0.46  -0.31

[44 rows x 22 columns]
```

The scikit-learn instance has a `.components_` attribute that contains the eigenvectors. This is transposed from the version we created.

```
>>> print(pca.components_[:2].round(3))
[[ 0.164  0.221  0.164  0.197  0.164  0.189  0.19   0.226  0.228
   0.224
  0.235  0.208  0.213  0.223  0.221  0.231  0.231  0.212  0.208
   0.235
  0.238  0.238]
 [-0.438 -0.034 -0.462 -0.417  0.455  0.18   0.062  0.104  0.136
  -0.062
 -0.068  0.245  0.155 -0.153  0.107 -0.051  0.123 -0.011  0.036
  -0.014
  0.003 -0.002]]
```

Again, that is just a bunch of numbers. Labeling in pandas helps to make sense of it. (I'll use the sklearn labels instead of my preferred naming and numbering.)

```
>>> print(pd.DataFrame(pca.components_,
...     index=[f'pca{i}' for i in range(nums.shape[1])],
...     columns=nums.columns))
      Background  Imagination  Integrity  Intelligence  ...  \
pc0     0.163793     0.221305   0.163592      0.197460  ...
pc1    -0.438263    -0.033741  -0.462030     -0.417192  ...
pc2    -0.185140    -0.233670   0.492682     -0.193787  ...
```

```
pc3    -0.029498     0.139844   0.087658      0.155311  ...
pc4    -0.574862     0.142825  -0.180732      0.206196  ...
...          ...          ...        ...           ...  ...
pc17   -0.077504     0.339750   0.037922      0.062553  ...
pc18    0.044865     0.187267   0.106134     -0.372864  ...
pc19   -0.061550    -0.347679  -0.123903      0.175805  ...
pc20   -0.084777     0.060294   0.022157      0.074996  ...
pc21    0.017618    -0.065134  -0.012286      0.005522  ...

      Avoid_crucial_mistakes  Experts'_view   Overall  Average_rank
pc0                 0.208037       0.235395  0.238318      0.238427
pc1                 0.035619      -0.014226  0.003287     -0.002321
pc2                 0.486695       0.110834  0.023624      0.019445
pc3                 0.057668       0.006368  0.008083      0.008353
pc4                 0.042191       0.091762 -0.011390     -0.022508
...                      ...            ...       ...           ...
pc17                0.008891      -0.479895 -0.013896     -0.018707
pc18               -0.160605      -0.228805  0.208768      0.207736
pc19                0.131391      -0.148025  0.520106      0.565484
pc20                0.077041      -0.611462  0.121977      0.075944
pc21                0.039960      -0.013025  0.691586     -0.710245

[22 rows x 22 columns]
```

The `.explained_variance_ratio_` attribute contains the percent of variance explained by each principal component.

```
>>> print(pd.Series(pca.explained_variance_ratio_,
...     index=[f'pca{i}' for i in range(nums.shape[1])]))
pca0     0.810091
pca1     0.056344
pca2     0.035071
pca3     0.021705
pca4     0.017334
           ...
pca17    0.000582
pca18    0.000500
pca19    0.000414
pca20    0.000258
pca21    0.000022
Length: 22, dtype: float64
```

Table 20.1: Dataframe Math Methods and Friends

Method	Description
`.add(other, axis='columns', level=None, fill_value=None)`	Add other to dataframe across `axis`. Unlike the operator, it can specify `fill_value`.
`.sub(other, axis='columns', level=None, fill_value=None)`	Subtract other from dataframe across `axis`. Unlike the operator, it can specify `fill_value`.
`.mul(other, axis='columns', level=None, fill_value=None)`	Multiply other with dataframe across `axis`. Unlike the operator, it can specify `fill_value`.
`.div(other, axis='columns', level=None, fill_value=None)`	Divide dataframe by other across `axis`. Unlike the operator, it can specify `fill_value`.
`.truediv(other, axis='columns', level=None, fill_value=None)`	Same as `.div`.
`.floordiv(other, axis='columns', level=None, fill_value=None)`	Integer divide dataframe by other across `axis`. Unlike the operator, it can specify `fill_value`.
`.mod(other, axis='columns', level=None, fill_value=None)`	Perform modulo operation with other across `axis`. Unlike the operator, it can specify `fill_value`.
`.pow(other, axis='columns', level=None, fill_value=None)`	Raise to other power across `axis`. Unlike the operator, it can specify `fill_value`.
`.dot(other)`	Matrix multiply dataframe by other.
`.min(axis=None, skipna=None, numeric_only=None)`	Return minimum value of each column.
`.max(axis=None, skipna=None, numeric_only=None)`	Return maximum value of each column.
`.mean(axis=None, skipna=None, numeric_only=None)`	Return mean value of each column.
`np.cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None)`	Calculate covariance matrix.

20.5 Summary

In this chapter, we demonstrated math operations on dataframes. I generally perform math operations on series, but it is nice to have the capability in dataframes. We also showed index alignment and advanced math operations like dot products and principal components analysis.

20.6 Exercises

With a tabular dataset of your choice:

1. Create a dataframe from the data and add it to itself.
2. Create a dataframe from the data and multiply it by two.
3. Are the results from the previous exercises equivalent?

Chapter 21

Looping and Aggregation

Often, we want to apply operations over items in a dataframe. We may want to use looping, the `.apply` method, or an aggregation method to do this.

21.1 For Loops

You can use a `for` loop with a dataframe, though you generally want to avoid for loops when doing numerical manipulation. When I see a for loop with pandas code, it means this is a slow operation, and you cannot take advantage of the vectorization that speeds up many operations. However, sometimes a for loop is appropriate (I use them when labeling plots).

If you need to loop over a dataframe, there are methods for doing it. The `.items` method gives you a tuple with the column name and the column (a series). The `.itertuples` method gives you a row represented as a named tuple (with the index in position 0):

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> # iteration over columns (col_name, series) tuple
>>> for col_name, col in pres.items():
...     print(col_name, type(col))
...     break
Seq <class 'pandas.core.series.Series'>
```

Note

Pandas 2.0 removed `.iteritems` in favor of `.items`.

```
>>> # iteration over rows as namedtuple (index as first item)
>>> for tup in pres.itertuples():
...     print(tup[0], tup.President)
...     break
1 George Washington
```

21.2 Aggregations

The aggregation methods available on a series are also available to a dataframe. Keep in mind that a dataframe has two dimensions. This means you can aggregate across both dimensions. So you can sum along axis 0 (the index) or axis 1 (the columns).

In this example, we will calculate the average of each row. We will isolate the numeric columns using `.loc`, then we will sum along the columns and divide the result by the length of the columns:

```
>>> scores = (pres
...  .loc[:,'Background':'Average_rank']
... )
>>> scores.sum(axis='columns') / len(scores.columns)
1      3.681818
2     14.454545
3      6.545455
4      9.636364
5     10.454545
         ...
40    20.818182
41    14.636364
42    30.363636
43    15.818182
44    39.772727
Length: 44, dtype: double[pyarrow]
```

(Note that we could also use `.mean(axis='columns')` to do the above.)

We can use multiple aggregations with the `.agg` method. Below, we will count the number of non-missing values for each column, the number of entries for each column (including the missing values), the sum of each column, and run a custom aggregation (that returns the value for index *1*):

```
>>> print(pres.select_dtypes('number').agg(
...     ['count', 'size', 'sum', lambda col: col.loc[1]]))
          Background  Imagination  Integrity  Intelligence  ...  \
count             44           44         44            44  ...
```

```
size              44          44         44          44  ...
sum              968         957        990         990  ...
<lambda>           7           7          1          10  ...

          Avoid_crucial_mistakes  Experts'_view  Overall  \
count                         44             44       44
size                          44             44       44
sum                          990            990      990
<lambda>                       1              2        1

          Average_rank
count               44
size                44
sum                990
<lambda>             1

[4 rows x 22 columns]
```

We can pass in a dictionary to perform multiple aggregations on a column:

```
>>> print(pres.agg({'Luck': ['count', 'size'],
...                 'Overall': ['count', 'max']}))
       Luck  Overall
count  44.0     44.0
size   44.0      NaN
max     NaN     44.0
```

You can use a keyword argument with a tuple to specify the index value of the resultant aggregation:

```
>>> print(pres.agg(Intelligence_count=('Intelligence', 'count'),
...          Intelligence_size=('Intelligence', 'size'))
... )
                    Intelligence
Intelligence_count            44
Intelligence_size             44
```

The `.describe` method is a meta-aggregation that returns a dataframe with summary statistics for each numeric column:

```
>>> print(pres.describe())
       Background  Imagination  Integrity  Intelligence  ...  \
count        44.0         44.0       44.0          44.0  ...
mean         22.0        21.75       22.5          22.5  ...
std     12.409674    12.519984  12.845233     12.845233  ...
min           1.0          1.0        1.0           1.0  ...
25%         11.75         11.0      11.75         11.75  ...
```

```
50%          22.0         21.5       22.5          22.5  ...
75%         32.25        32.25      33.25         33.25  ...
max          43.0         43.0       44.0          44.0  ...

       Avoid_crucial_mistakes  Experts'_view    Overall  \
count                    44.0           44.0       44.0
mean                     22.5           22.5       22.5
std                 12.845233      12.845233  12.845233
min                        1.0            1.0        1.0
25%                      11.75          11.75      11.75
50%                       22.5           22.5       22.5
75%                      33.25          33.25      33.25
max                       44.0           44.0       44.0

       Average_rank
count          44.0
mean           22.5
std       12.845233
min             1.0
25%            11.75
50%             22.5
75%            33.25
max             44.0

[8 rows x 22 columns]
```

Note

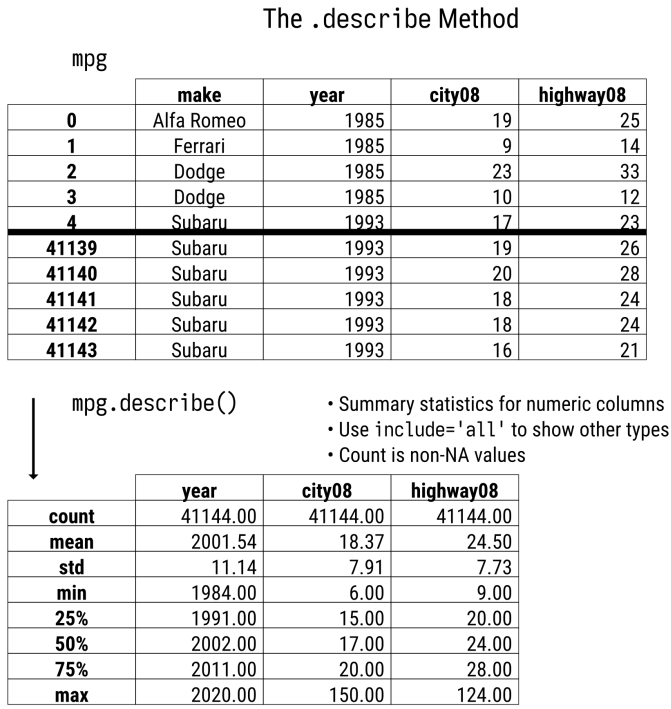
The *count* row in the summary statistics has a particular meaning in pandas. It is not the count of the rows. Instead, it is the count of the non-missing (not `na`) rows.

21.3 The .apply Method

Like the series, the dataframe has an `.apply` method. You should be wary of using the dataframe method, just like you should be wary of using the series method. More specifically, if you are dealing with numbers, you might want to see if you can operate in a vectorized way.

Also, keep in mind that a dataframe is two-dimensional. So rather than applying a function to a single value, when you call `.apply` on a dataframe, you work on a whole row or column. Because of that, I find that I rarely use this method.

Most of the `.apply` examples you find in the wild are silly examples that show how `.apply` works but also give a false impression that you should be everywhere, including using it for these silly examples.



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	year	city08	highway08
count	41144.00	41144.00	41144.00
mean	2001.54	18.37	24.50
std	11.14	7.91	7.73
min	1984.00	6.00	9.00
25%	1991.00	15.00	20.00
50%	2002.00	17.00	24.00
75%	2011.00	20.00	28.00
max	2020.00	150.00	124.00

Figure 21.1: The `.describe` method provides the count of non-missing values, the mean, standard deviation, minimum, maximum, and quartiles.

For example, if you wanted to calculate the range or spread of the presidential rankings for each row, I would do this:

```
>>> (pres
...     .select_dtypes('number')
...     .pipe(lambda df_:df_.max(axis='columns')
...                         - df_.min(axis='columns'))
...     .rename('range')
... )
1     17
2     28
3     19
4     16
5     13
      ..
40    19
```

```
41    36
42    24
43    22
44    34
Name: range, Length: 44, dtype: uint8[pyarrow]
```

The `.apply` version looks like this:

```
>>> (pres
...  .select_dtypes('number')
...  .apply(lambda row: row.max()-row.min(), axis='columns')
...  .rename('range')
... )

1     17
2     28
3     19
4     16
5     13
      ..
40    19
41    36
42    24
43    22
44    34
Name: range, Length: 44, dtype: int64
```

They look pretty similar, but the former does an optimized maximum and minimum calculation, while the latter does a separate calculation for each row.

Or you might see an example showing how to use `.apply` on the index axis. If you use `.apply` with `axis='index'`, it calls the function on each column. You might encounter silly examples like calculating the sum of each column:

```
>>> pres.select_dtypes('number').apply('sum')  # axis=0
Background                      968
Imagination                     957
Integrity                       990
Intelligence                    990
Luck                            990
                               ...
Foreign_policy_accomplishments  990
Avoid_crucial_mistakes          990
Experts'_view                   990
Overall                         990
Average_rank                    990
Length: 22, dtype: uint64[pyarrow]
```

In this case, it will calculate a sum on each column, but why not just do one call and get the same result?

```
>>> pres.select_dtypes('number').sum()  # axis=0
Background                          968
Imagination                         957
Integrity                           990
Intelligence                        990
Luck                                990
                                   ...
Foreign_policy_accomplishments      990
Avoid_crucial_mistakes              990
Experts'_view                       990
Overall                             990
Average_rank                        990
Length: 22, dtype: uint64[pyarrow]
```

21.4 Optimizing If Then

I have used `.apply` when replicating complicated logic from spreadsheets. Here is a snippet of sample data:

```
>>> import io
>>> billing_data = \
... '''cancel_date,period_start,start_date,end_date,rev,sum_payments
... 12/1/2019,1/1/2020,12/15/2019,5/15/2020,999,50
... ,1/1/2020,12/15/2019,5/15/2020,999,50
... ,1/1/2020,12/15/2019,5/15/2020,999,1950
... 1/20/2020,1/1/2020,12/15/2019,5/15/2020,499,0
... ,1/1/2020,12/24/2019,5/24/2020,699,100
... ,1/1/2020,11/29/2019,4/29/2020,799,250
... ,1/1/2020,1/15/2020,4/29/2020,799,250'''

>>> bill_df = pd.read_csv(io.StringIO(billing_data),
...     dtype_backend='pyarrow',
...     parse_dates=['cancel_date', 'period_start', 'start_date',
...                  'end_date'])

>>> print(bill_df)
  cancel_date period_start start_date   end_date  rev  sum_payments
0   12/1/2019   2020-01-01 2019-12-15 2020-05-15  999            50
1        <NA>   2020-01-01 2019-12-15 2020-05-15  999            50
2        <NA>   2020-01-01 2019-12-15 2020-05-15  999          1950
3   1/20/2020   2020-01-01 2019-12-15 2020-05-15  499             0
4        <NA>   2020-01-01 2019-12-24 2020-05-24  699           100
```

```
5        <NA>   2020-01-01 2019-11-29 2020-04-29  799           250
6        <NA>   2020-01-01 2020-01-15 2020-04-29  799           250
```

Pandas 2.2 doesn't convert *cancel_date* into a date using the `parse_dates` parameter. (In fact, it creates strings, `'<NA>'`.) [1] I need to force the conversion with `pd.to_datetime`.

```
def tweak_bill202(df_):
    return (df_
            .assign(cancel_date=pd.to_datetime(
                df_.cancel_date.replace('<NA>', ''), format='%m/%d/%Y')
                )
            )

bill_df = tweak_bill202(bill_df)
```

Here is some logic that is more involved. If the start and end dates bound the period start date, we calculate if the revenue is greater than the sum of the payments:

```
>>> import numpy as np
>>> def calc_unbilled_rec(vals):
...     cancel_date, period_start, start_date, end_date, rev, \
...          sum_payments = vals
...     if cancel_date < period_start:
...         return np.nan
...     if start_date < period_start and end_date > period_start:
...         if rev > sum_payments:
...             return rev - sum_payments
...         else:
...             return 0
```

We can use `.apply` to call this function with the values from each row. Note that to apply it to a row, we need to pass in `axis='columns'`:

```
>>> bill_df.apply(calc_unbilled_rec, axis='columns')
0      NaN
1    949.0
2      0.0
3    499.0
4    599.0
5    549.0
6      NaN
dtype: float64
```

[1] https://github.com/pandas-dev/pandas/issues/47950

Below is an attempt to vectorize this with `.case_when`. Sadly, this runs about seven times as slow on my machine on this small dataset. However, if the dataset has a hundred thousand rows, it runs about 50 times faster than the `.apply` version!

```
>>> (pd.Series(np.nan, dtype='float[pyarrow]', index=bill_df.index)
...   .case_when([(bill_df.cancel_date < bill_df.period_start,  # 1
...                np.nan),
...         (((bill_df.start_date < bill_df.period_start) &  # 2
...          (bill_df.end_date > bill_df.period_start) &
...          (bill_df.rev > bill_df.sum_payments)),
...                bill_df.rev - bill_df.sum_payments),
...         (((bill_df.start_date < bill_df.period_start) &  # 3
...          (bill_df.end_date > bill_df.period_start) &
...          (bill_df.rev <= bill_df.sum_payments)),
...                0)
...       ])
... )
0     <NA>
1    949.0
2      0.0
3    499.0
4    599.0
5    549.0
6     <NA>
dtype: double[pyarrow]
```

21.5 Optimizing .apply functions

In this section, we will revisit the logic from the previous section and benchmark it against a larger dataset. We will also look at how to optimize the `.apply` function using Cython.

Let's create a larger dataset with 100,000 rows:

```
bill_100k = bill_df.sample(100_000, replace=True)
```

Now, let's get some timing information for the `.apply` function:

```
>>> %%timeit
>>> bill_100k.apply(calc_unbilled_rec, axis='columns')
277 ms ± 2.44 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

On my machine, this takes around 280 milliseconds.

Let's compare this to the `.case_when` version:

```
>>> def calc_unbilled_case(bill_df):
...     return (pd.Series(np.nan, dtype='float[pyarrow]',
...                       index=bill_df.index)
...     .case_when([(bill_df.cancel_date < bill_df.period_start,  # 1
...                  np.nan),
...          (((bill_df.start_date < bill_df.period_start) &  # 2
...           (bill_df.end_date > bill_df.period_start) &
...           (bill_df.rev > bill_df.sum_payments)),
...                  bill_df.rev - bill_df.sum_payments),
...          (((bill_df.start_date < bill_df.period_start) &  # 3
...           (bill_df.end_date > bill_df.period_start) &
...           (bill_df.rev <= bill_df.sum_payments)),
...                  0)
...          ])
...     )

>>> %%timeit
>>> calc_unbilled_case(bill_100k)
4.27 ms ± 42.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops
     each)
```

The `.case_when` version is almost 65 times faster!

21.6 Optimizing .apply with Cython

Let's provide a Cython example to speed up this code.

First, we need to load the Cython extension (if we are using Jupyter):

```
%load_ext cython
```

Now, we will use our pure Python code but compile it with Cython.

```
%%cython

def calc_unbilled_rec_cy1(vals):
    cancel_date, period_start, start_date, end_date, rev, \
         sum_payments = vals
    if cancel_date < period_start:
        return float('nan')
    if start_date < period_start and end_date > period_start:
        if rev > sum_payments:
            return rev - sum_payments
        else:
            return 0
```

Let's see how fast that runs:

```
>>> %%timeit
>>> bill_100k.apply(calc_unbilled_rec_cy1, axis='columns')
272 ms ± 1.37 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Not particularly fast. Let's see if we can speed it up by using Cython code instead of pure Python.

This Cython code defines a function calc_unbilled_cy that takes six NumPy arrays of 64-bit integers as input and calculates unbilled amounts based on specified conditions. The function is optimized using Cython annotations to disable bounds checking and wraparound, improving performance. C data types are used for variables to enhance efficiency in the iteration process. The results are stored in a NumPy array and returned.

Note that this is using "Cython" syntax.

```
%%cython

cimport cython
import numpy as np
cimport numpy as np


@cython.wraparound(False)
cpdef calc_unbilled_cy(np.ndarray[np.int64_t] cancel_date,
                       np.ndarray[np.int64_t] period_start,
                       np.ndarray[np.int64_t] start_date,
                       np.ndarray[np.int64_t] end_date,
                       np.ndarray[np.int64_t] rev,
                       np.ndarray[np.int64_t] sum_payments):
    cdef np.ndarray[np.float64_t ] results = np.full(rev.shape[0], np.nan,
                                                     dtype=np.float64)
    cdef long cd, pd, sd, ed;
    for i in range(rev.shape[0]):
        cd = cancel_date[i]
        ps = period_start[i]
        sd = start_date[i]
        ed = end_date[i]
        if cd > 0 and cd < ps:
            continue
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

Let's time it:

```
>>> %%timeit
>>> calc_unbilled_cy(*(ser.astype(int).to_numpy()
...                    for name, ser in bill_100k.items()))
4.84 ms ± 84.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops
    each)
```

By using Cython constructs, we can speed up execution significantly. However, it is still slower than NumPy.

Cython 3 introduced the ability to use Python 3 type annotations instead of the above Cython syntax. Below, I've converted the code to use Python 3 type annotations.

```
%%cython

import cython
import numpy as np


@cython.wraparound(False)
def calc_unbilled_cy2(cancel_date: np.ndarray[np.int64_t],
        period_start: np.ndarray[np.int64_t],
        start_date: np.ndarray[np.int64_t],
        end_date: np.ndarray[np.int64_t],
        rev: np.ndarray[np.int64_t],
        sum_payments: np.ndarray[np.int64_t]) -> np.ndarray[np.float64_t]:
    results: np.ndarray[np.float64_t] = np.full(rev.shape[0], np.nan,
                                                dtype=np.float64)
    i: cython.int

    for i in range(rev.shape[0]):
        cd:cython.long = cancel_date[i]
        ps:cython.long = period_start[i]
        sd:cython.long = start_date[i]
        ed:cython.long = end_date[i]
        if cd > 0 and cd < ps:
            continue
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

This is much easier to write than Cython syntax. However, it is also slower. I filed a bug [1] to track the issue. If you are adept at C, you can run `%cython --annotate` to view the C generated code.

```
>>> %%timeit
>>> calc_unbilled_cy2(*(ser.astype(int).to_numpy()
...                         for name, ser in bill_100k.items()))
43.8 ms ± 484 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The bug I filed suggested the proper way to do this is with *memory view*. They allow efficient access to the memory buffer of the NumPy array.

```
%%cython

import cython
import numpy as np

@cython.wraparound(False)
def calc_unbilled_cy3(cancel_date: cython.long[:],
        period_start: cython.long[:],
        start_date: cython.long[:],
        end_date: cython.long[:],
        rev: cython.long[:],
        sum_payments: cython.long[:]) -> cython.double[:]:
    results: cython.double[:] = np.full(rev.shape[0], np.nan,
                                        dtype=np.float64)
    i: cython.int

    for i in range(rev.shape[0]):
        cd:cython.long = cancel_date[i]
        ps:cython.long = period_start[i]
        sd:cython.long = start_date[i]
        ed:cython.long = end_date[i]
        if cd > 0 and cd < ps:
            continue
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

```
>>> %%timeit
>>> calc_unbilled_cy3(*(ser.astype(int).to_numpy()
...                         for name, ser in bill_100k.items()))
```

[1] https://github.com/cython/cython/issues/5811

```
1.04 ms ± 5.71 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
    each)
```

This appears to run 40x faster! Insert brain explode emoji!

And if we turn off bounds checking (and add an `assert` to make sure the arguments are the correct size) we get another 20% improvement on this data size.

```
%%cython

import cython
import numpy as np

@cython.boundscheck(False)
@cython.wraparound(False)
def calc_unbilled_cy4(cancel_date: cython.long[:],
        period_start: cython.long[:],
        start_date: cython.long[:],
        end_date: cython.long[:],
        rev: cython.long[:],
        sum_payments: cython.long[:]) -> cython.double[:]:
    results: cython.double[:] = np.full(rev.shape[0], np.nan,
                                        dtype=np.float64)
    i: cython.int
    assert len(period_start) == len(start_date)

    for i in range(rev.shape[0]):
        cd:cython.long = cancel_date[i]
        ps:cython.long = period_start[i]
        sd:cython.long = start_date[i]
        ed:cython.long = end_date[i]
        if cd > 0 and cd < ps:
            continue
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

```
>>> %%timeit
>>> calc_unbilled_cy4(*(ser.astype(int).to_numpy()
...                     for name, ser in bill_100k.items()))
786 µs ± 24 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We can try to use `prange` to run the loop in parallel. It appears not to have an impact on this code and data combination.

```
%%cython

import cython
from cython.parallel import prange
import numpy as np

@cython.boundscheck(False)
@cython.wraparound(False)
def calc_unbilled_cy5(cancel_date: cython.long[:],
        period_start: cython.long[:],
        start_date: cython.long[:],
        end_date: cython.long[:],
        rev: cython.long[:],
        sum_payments: cython.long[:]) -> cython.double[:]:
    results: cython.double[:] = np.full(rev.shape[0], np.nan,
                                        dtype=np.float64)
    i: cython.int
    assert len(period_start) == len(start_date)

    for i in prange(rev.shape[0], nogil=True):
        cd:cython.long = cancel_date[i]
        ps:cython.long = period_start[i]
        sd:cython.long = start_date[i]
        ed:cython.long = end_date[i]
        if cd > 0 and cd < ps:
            continue
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

```
>>> %%timeit
>>> calc_unbilled_cy5(*(ser.astype(int).to_numpy()
...                     for name, ser in bill_100k.items()))
824 µs ± 10.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
     each)
```

21.7 Optimization with Numba

Another option to speed up code execution when you pass back from Pandas to running pure Python code is to use Numba.

Numba, is a just-in-time (JIT) compiler for Python that translates a Python function into machine code at runtime. I've found that it is one of the easiest

ways to optimize pure Python code. I stick the `@jit` decorator on my code. Numba does the rest. It is almost like magic.

```
import numpy as np
from numba import jit

@jit
def calc_unbilled_numba(cancel_date, period_start, start_date,
                        end_date, rev, sum_payments):
    results = np.full(rev.shape[0], np.nan, dtype=np.float64)
    for i in range(rev.shape[0]):
        cd = cancel_date[i]
        ps = period_start[i]
        sd = start_date[i]
        ed = end_date[i]
        if cd > 0 and cd < ps:
            results[i] = np.nan
        elif sd < ps < ed:
            if rev[i] > sum_payments[i]:
                results[i] = rev[i] - sum_payments[i]
            else:
                results[i] = 0
    return results
```

Let's time it. It is 4x faster than the `.case_when` solution and about as fast as the Cython memory view solution on my machine.

```
>>> %%timeit
>>> calc_unbilled_numba(*(ser.astype(int).to_numpy()
                          for name, ser in bill_100k.items()))
875 µs ± 47.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
    each)
```

Let's validate that my solutoins return the same thing:

```
>>> print(bill_df
...  .assign(cy=calc_unbilled_cy(*(ser.astype(int).to_numpy()
...                                for name, ser in bill_df.items())),
...          nb=calc_unbilled_numba(*(ser.astype(int).to_numpy()
...                                   for name, ser in bill_df.items())),
...          np=calc_unbilled_np(bill_df),
...          py=bill_df.apply(calc_unbilled_rec, axis='columns')
...         )
... )
  cancel_date period_start start_date   end_date  ...     cy     nb  \
0  2019-12-01   2020-01-01 2019-12-15 2020-05-15  ...    NaN    NaN
1         NaT   2020-01-01 2019-12-15 2020-05-15  ...  949.0  949.0
```

```
2         NaT   2020-01-01 2019-12-15 2020-05-15  ...    0.0    0.0
3  2020-01-20   2020-01-01 2019-12-15 2020-05-15  ...  499.0  499.0
4         NaT   2020-01-01 2019-12-24 2020-05-24  ...  599.0  599.0
5         NaT   2020-01-01 2019-11-29 2020-04-29  ...  549.0  549.0
6         NaT   2020-01-01 2020-01-15 2020-04-29  ...    NaN    NaN

      np     py
0    NaN    NaN
1  949.0  949.0
2    0.0    0.0
3  499.0  499.0
4  599.0  599.0
5  549.0  549.0
6    NaN    NaN

[7 rows x 10 columns]
```

Note

Be careful with your timing. It is not necessarily the case that code that is slower on small datasets is slower on larger datasets!

Table 21.1: Dataframe Looping Methods

Method	Description
`.items()`	Iterate over a tuple of column name and series.
`.itertuples(index=True, name="Pandas")`	Iterate over a namedtuples of rows. Include `index` by default. Use `name` to specify the classname of the namedtuple (or set it to `None` to return normal tuples).
`.sum(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return sum over axis. Default of empty sequence is 0. Set `min_count=1` to return `nan`.
`.min(axis=0, skipna=True, level=None, numeric_only=None`	Return minimum over the axis.
`.max(axis=0, skipna=True, level=None, numeric_only=None`	Return maximum over the axis.

Method	Description
`.idxmin(axis=0, skipna=True)`	Return the index of the first minimum value over the axis.
`.idxmax(axis=0, skipna=True)`	Return the index of the first maximum value over the axis.
`.agg(func=None, axis=0, *args, **kwargs)`	Aggregate using `func` over the axis. The `func` can be a function that collapses a column (or row), string, list of functions (or strings), dictionary of axis to function, list, or string.
`.describe(percentiles=[.25, .5, .75], include=None, exclude=None, datetime_is_numeric=False)`	Return summary statistics for dataframe.
`.apply(func=None, axis=0, raw=False, result_type=None, *args, **kwargs)`	Apply `func` over the axis. If `func` returns a sequence, then return a dataframe. If `func` returns a scalar, then return a series.
`s.case_when(caselist)`	Use `caselist` (list of tuples of (boolean array, result), to simulate if then statements

21.8 Summary

In this chapter, we demonstrated looping and aggregation methods of dataframes. We also showed the `.apply` method and optimized it with NumPy, Cython, and Numba.

21.9 Exercises

With a tabular dataset of your choice:

1. Loop over each row and calculate the maximum and minimum values.
2. Calculate each row and column's maximum and minimum value using the `.agg` method.
3. Calculate each row and column's maximum and minimum value using the `.apply` method.

Chapter 22

Columns Types, .assign, and Memory Usage

This chapter will explore updating, creating, and changing the types of columns. We will show how this impacts memory usage.

22.1 Conversion Methods

There are various methods and functions for changing the types of a series in pandas. We can use `.astype` to update column types. Or we can use the `.assign` method to return a new dataframe with the updated type.

The `.astype` method allows us to specify the types of each column with a dictionary.

The `.assign` method is a key method to master. You specify the name of a column with a keyword argument. If the argument name is an existing column, it will change the column's values. If the argument name is a new column, it creates a new column. One caveat is that this method returns a new dataframe. It does not mutate the existing dataframe.

You should also know that you can pass in a scalar value, a series, or a callable as the value for the keyword argument. The callable (a function or `lambda`) should accept the current state of the dataframe (this is important when chaining because each step returns a new dataframe), and should return a scalar or series.

We saw some examples of these methods in the `tweak_siena_pres` and `int64_to_uint8` functions. Here are the relevant snippets:

```
def tweak_siena_pres(df):
    def int64_to_uint8(df_):
        # ....
        return (df_
...
                .astype({col:'uint8' for col in cols}))
    return (df
     # ...
     .astype({'Party':'category'})
```

```
    .pipe(int64_to_uint8)
    .assign(Average_rank=lambda df_:(df_.select_dtypes('uint8')
                .sum(axis=1).rank(method='dense').astype('uint8')),
            )
)
```

22.2 Memory Usage

One thing to be aware of is memory usage. You can often shrink the memory usage of a dataframe by changing the type while not losing any data.

Here are the original column sizes of the presidential data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow',
...                  engine='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> df.memory_usage(deep=True)
Index        352
Seq.         258
President    843
Party        644
Bg           352
             ...
DA           352
FPA          352
AM           352
EV           352
O            352
Length: 25, dtype: int64
```

Here are the sizes of the columns where the numeric values have been optimized:

```
>>> pres.memory_usage(deep=True)
Index                    352
Seq                      258
President                843
Party                    307
Background                50
                         ...
Avoid_crucial_mistakes    50
Experts'_view             50
Overall                   50
```

```
Average_rank               50
Quartile                  456
Length: 27, dtype: int64
```

You can see that the ranking columns use less memory because they are stored as `uint8` values instead of `int64`.

Let's compare the total usage for both `df` and `pres`:

```
>>> df.memory_usage(deep=True).sum(), pres.memory_usage(deep=True).sum()
(9489, 3316)
```

Our tweak function allowed us to save 50% more memory.

If you are in a REPL and do not need to manipulate the results of the `.memory_usage`, an alternative is to call `.info`, which does not return a series, but prints the result to the screen:

```
>>> pres.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
Index: 44 entries, 1 to 44
Data columns (total 26 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Seq                             44 non-null     string[pyarrow]
 1   President                       44 non-null     string[pyarrow]
 2   Party                           44 non-null     category
 3   Background                      44 non-null     uint8[pyarrow]
 4   Imagination                     44 non-null     uint8[pyarrow]
 5   Integrity                       44 non-null     uint8[pyarrow]
 6   Intelligence                    44 non-null     uint8[pyarrow]
 7   Luck                            44 non-null     uint8[pyarrow]
 8   Willing_to_take_risks           44 non-null     uint8[pyarrow]
 9   Ability_to_compromise           44 non-null     uint8[pyarrow]
 10  Executive_ability               44 non-null     uint8[pyarrow]
 11  Leadership_ability              44 non-null     uint8[pyarrow]
 12  Communication_ability           44 non-null     uint8[pyarrow]
 13  Overall_ability                 44 non-null     uint8[pyarrow]
 14  Party_leadership                44 non-null     uint8[pyarrow]
 15  Relations_with_Congress         44 non-null     uint8[pyarrow]
 16  Court_appointments              44 non-null     uint8[pyarrow]
 17  Handling_of_economy             44 non-null     uint8[pyarrow]
 18  Executive_appointments          44 non-null     uint8[pyarrow]
 19  Domestic_accomplishments        44 non-null     uint8[pyarrow]
 20  Foreign_policy_accomplishments  44 non-null     uint8[pyarrow]
 21  Avoid_crucial_mistakes          44 non-null     uint8[pyarrow]
 22  Experts'_view                   44 non-null     uint8[pyarrow]
 23  Overall                         44 non-null     uint8[pyarrow]
```

```
 24  Average_rank                  44 non-null     uint8[pyarrow]
 25  Quartile                      44 non-null     category
dtypes: category(2), string[pyarrow](2), uint8[pyarrow](22)
memory usage: 3.2 KB
```

Because Arrow has native support for strings, we can save even more memory by using that as a backend.

Table 22.1: Dataframe Methods from this Chapter

Method	Description
`.astype(dtype, copy=True, errors='raise')`	Cast dataframe into `dtype`. (More common to use this on series.)
`.assign(**kwargs)`	Return a new dataframe with updated or new columns. `kwargs` maps column name to function, scalar, or series. If using a function, it is passed in the current state of the dataframe and should return a scalar or series. Subsequent columns may reference earlier columns in `kwargs` if you use a function.
`.memory_usage( index=True, deep=False)`	Return a series with the memory usage of each column in bytes. By default includes `index`. Use `deep=True` to show how much space `object` columns consume.
`.info( verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None)`	Print summary of dataframe to stdout. Use `memory_usage='deep'` to show object column memory usage.

22.3 Summary

The series chapters showed how to convert the type of a series from one type to another. With a dataframe, we want to optimize the types of each column. To create a dataframe with the newer columns, we use the `.assign` method. If you master this method, you will eliminate many bugs that pandas users encounter when they try to change columns using other methods.

22.4 Exercises

With a tabular dataset of your choice:

1. Find a numeric column and change its type. Did you save memory? Did you lose precision?

2. Find a string column and convert it to a category. What happened to memory usage? Time a few string operations. Are they faster on the categorical column or string column?

Chapter 23

Creating and Updating Columns

This chapter explores the "one true way" to create and update columns in pandas. This is potentially the most controversial subject of this book, probably because it is not talked about very often, and the syntax might be unclear at first.

23.1 Loading the Data

We will look at a dataset of Python users from JetBrains[1].

Let's load the data:

```
>>> import pandas as pd
>>> import numpy as np
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> print(jb)
      is.python.main other.lang.None  ...    age    country.live
0                Yes            None  ...  30-39             <NA>
1                Yes            None  ...  21-29            India
2                Yes            None  ...  30-39   United States
3                Yes            None  ...   <NA>             <NA>
...              ...             ...  ...    ...              ...
54458            Yes            None  ...   <NA>             <NA>
54459            Yes            None  ...  21-29  Russian Fed...
54460            Yes            None  ...  30-39            Spain
54461            Yes            None  ...  21-29          Algeria

[54462 rows x 264 columns]
```

This is a pretty good dataset. It has over 50,000 rows and 264 columns. However, we will need to clean it up to perform exploratory analysis.

[1] https://www.jetbrains.com/lp/python-developers-survey-2020/

Some of the columns have a dummy-like encoding. For example, the names of the columns that start with *database.* end with a database name. The database name is included in the values for those columns. Because a user might use multiple databases, that is a mechanism to encode this. However, it also creates many columns, one per database. I will filter out columns like the database columns to keep the book's data manageable.

Below is code that determines whether a feature can have multiple values (like a database) and removes those:

```
>>> import collections
>>> counter = collections.defaultdict(list)
>>> for col in sorted(jb.columns):
...     period_count = col.count('.')
...     if period_count >= 2:
...         part_end = 2
...     else:
...         part_end = 1
...     parts = col.split('.')[:part_end]
...     counter['.'.join(parts)].append(col)
>>> uniq_cols = []
>>> for cols in counter.values():
...     if len(cols) == 1:
...         uniq_cols.extend(cols)

>>> uniq_cols
['age',
 'are.you.datascientist',
 'company.size',
 'country.live',
 'employment.status',
 'first.learn.about.main.ide',
 'how.often.use.main.ide',
 'ide.main',
 'is.python.main',
 'job.team',
 'main.purposes',
 'missing.features.main.ide',
 'nps.main.ide',
 'python.years',
 'python2.version.most',
 'python3.version.most',
 'several.projects',
 'team.size',
 'use.python.most',
 'years.of.coding']
```

Note that these column names have a period in them. I'm going to replace those with an underscore, as it will allow us to access the names of the columns via attributes (with a period).

Let's look at the age column:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .value_counts(dropna=False)
... )
age
<NA>           29701
21–29           9710
30–39           7512
40–49           3010
18–20           2567
50–59           1374
60 or older      588
Name: count, dtype: int64[pyarrow]
```

I will pull out the first two characters from the *age* column and convert them to integer numbers:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .str.slice(0,2)
...  .replace('', np.nan)
...  .astype('int8[pyarrow]')
... )
0          30
1          21
2          30
3        <NA>
         ...
54458    <NA>
54459      21
54460      30
54461      21
Name: age, Length: 54462, dtype: int8[pyarrow]
```

Note

You can also write `.str.slice(0,2)` as `.str[0:2]`.

23.2 The .assign Method

Now that this column is cleaned up, let's put it in a dataframe. This is where `.assign` comes in. As a reminder, none of the operations we have looked at in this book mutate or update a series or dataframe. Instead, they return a new series or dataframe. This is what enables the chaining style we have seen throughout this book.

Sometimes (actually quite often), you will see the internet telling you to do something like this:

```
>>> jb2 = jb[uniq_cols]
>>> age_slice = jb.age.str.slice(0, 2)
>>> age_float = age_slice.astype(float)
>>> age_int = age_float.astype('Int64')
>>> jb2['age'] = age_int
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
  https://pandas.pydata.org/pandas-docs/
  stable/user_guide/indexing.html#returning-a-view-versus-a-copy

  jb2['age'] = age_int
```

Sometimes, the above code works, but you can see the infamous `SettingWithCopyWarning` warning telling you that it might not be working. However, if you use `.assign`, you sidestep this issue altogether.

Also, note that line `jb2['age'] = age_int` does not return anything. You cannot chain on it! The `.assign` method will let you update or add a column and will also return a dataframe for chaining:

```
>>> print(jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age
...     .str.slice(0,2)
...     .replace('', np.nan)
...     .astype('int8[pyarrow]'))
... )
        age are_you_datascientist  ... use_python_most  \
0        30            <NA>        ...            <NA>
```

```
1        21            Yes       ...  Software pr...
2        30             No       ...  DevOps / Sy...
3      <NA>           <NA>       ...  Web develop...
...     ...            ...       ...             ...
54458  <NA>             No       ...  Web develop...
54459    21           <NA>       ...  Web develop...
54460    30            Yes       ...   Data analysis
54461    21           <NA>       ...            <NA>

      years_of_coding
0           1–2 years
1           3–5 years
2           3–5 years
3           11+ years
...               ...
54458       1–2 years
54459      6–10 years
54460       3–5 years
54461       1–2 years

[54462 rows x 20 columns]
```

Note

When you call `.assign`, you generally pass in a keyword argument corresponding to the column name to create or update. You can assign the argument to a series, a scalar, or a function. You will see that many of my examples use `lambda` functions.

Using a function (it can be a regular function, but often we use a lambda to have the logic inline) has an unseen benefit. This function will accept the *current state* of the dataframe. If you have done any filtering or manipulation in the chain before calling `.assign`, it will be represented in this dataframe.

In the example above, my `lambda` looked like this:

```
.assign(age=lambda df_:df_.age
```

In this case, I could have gotten away without a `lambda` because the *age* column was not renamed. The code could have been this:

```
.assign(age=jb.age
```

Later on, we will see updating the *country.live* and *python.years* columns. Because we have a `.rename` in our chain, we will use a

`lambda` to refer to the new column names, *country_live* and *python_years*, respectively.

Another benefit of chaining is that this code reads like a step-by-step recipe. First, we pull out the columns we want, then rename the columns, and finally update the age column (with its own recipe).

Once you get used to this programming style, you will start thinking of making step-by-step changes to your data. This will make your code easier to read and understand.

Finally, some complain that working from the source data is slow, tedious, and repetitive. Maybe it is. But, in almost every data project I've been involved with, the boss has come around and asked for an explanation of the data. Using chaining makes stepping through the explanation easy. Using the style of pandas espoused by most of the internet makes this a considerable headache.

Ok, one more point. Chaining also will enable (future) query engine optimizers to speed up chained pandas code. Much like SQL optimizers can do predicate pushdown, one could envision optimizers (or a future tool that supports that pandas API) that work on chains. The use of chains would enable this.

I'll get off my `.assign` soapbox here. Many appear to have an almost allergic reaction to this coding style. Yet, they can't present anything better than the spaghetti code found everywhere else.

23.3 More Column Cleanup

The *are_you_datascientist* column can be converted to a boolean column with the `.replace` method in combination with `.astype`. Note that pyarrow types are strict and will not let us replace strings with booleans directly in the `.replace` call. So we need to follow this up with an `.astype` call:

```
>>> import numpy as np
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')
...         )
...  .are_you_datascientist
... )
0         <NA>
1         True
2        False
```

```
3         <NA>
          ...
54458    False
54459     <NA>
54460     True
54461     <NA>
Name: are_you_datascientist, Length: 54462, dtype: bool[pyarrow]
```

On to the next column. Let's look at *company_size*. I'll use the `.value_counts` method to see unique values:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')
...         )
...  .company_size
...  .value_counts(dropna=False)
... )
company_size
<NA>               35037
51–500              4608
More than 5,000     3635
11–50               3507
                   ...
1,001–5,000         1934
Just me             1492
501–1,000           1165
Not sure             526
Name: count, Length: 9, dtype: int64[pyarrow]
```

Then we practice cleaning it up.

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')
...         )
```

```
...   .company_size
...   .replace({'Just me': '1', '': pd.NA,
...     'Not sure': pd.NA, 'More than 5,000': '5000',
...     '2–10': '2', '11–50':'11','51–500': '51', '501–1,000':'501',
...     '1,001–5,000':'1001'}).astype('int64[pyarrow]')
...   .value_counts()
... )
company_size
51      4608
5000    3635
11      3507
2       2558
1001    1934
1       1492
501     1165
Name: count, dtype: int64[pyarrow]
```

After we have cleaned up the *company_size* column, we throw it back into the `.assign` method to update the column in the dataframe.

I'm going to do replacements here as well. Splitting or using a regular expression to pull out these values would be possible. I'm going to use the left value of the interval as a hint but use the `.replace` method. The code will look like this:

```
company_size=lambda df_:df_.company_size.replace(
    {'Just me': '1', '': pd.NA,
     'Not sure': pd.NA, 'More than 5,000': '5000',
     '2–10': '2', '11–50':'11','51–500': '51', '501–1,000':'501',
     '1,001–5,000':'1001'}).astype('int64[pyarrow]')
```

I'm not going to show the code for each column individually, but here is an overview of the steps I will take to the columns:

- *country_live* - Convert to categorical.
- *employment_status* - Fill missing values with `'Other'` and convert to categorical.
- *is_python_main* - Convert to categorical.
- *team_size* - Split on en-dash, pull out the first column, replace `'More than 40'` with 41, replace values where *company_size* is 1 with 1, and convert it to a float.
- *years_of_coding* - Replace `'Less than 1 year'` with .5, then pull out any numbers with a regular expression and convert them to floats.
- *python_years* - Replace `'_'` with `'.'`, then pull out any numbers with a regular expression and convert them to floats.
- *use_python_most* - Replace missing values with `'Unknown'`.

After the column manipulation, we will drop the *python2_version_most* column:

```
>>> jb2 = (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
...          company_size=lambda df_:df_.company_size.replace(
...            {'Just me': '1', '': pd.NA,
...             'Not sure': pd.NA, 'More than 5,000': '5000',
...             '2–10': '2', '11–50':'11','51–500': '51', '501–1,000':'501',
...             '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...          country_live=lambda df_:df_.country_live.astype('category'),
...          employment_status=lambda df_:df_.employment_status
...               .fillna('Other').astype('category'),
...          is_python_main=lambda df_:df_.is_python_main
...               .astype('category'),
...          team_size=lambda df_:df_.team_size
...              .str.split(r'-', n=1, expand=True)
...              .iloc[:,0].replace('More than 40 people', '41')
...              .where(df_.company_size!=1, '1')
...              .replace('', pd.NA)
...              .astype('int8[pyarrow]'),
...          years_of_coding=lambda df_:df_.years_of_coding
...              .replace('Less than 1 year', '.5')
...              .str.extract(r'(?P<years_of_coding>\.?\d+)')
...              .astype('float64[pyarrow]'),
...          python_years=lambda df_:df_.python_years
...              .replace('Less than 1 year', '.5')
...              .str.extract(r'(?P<python_years>\.?\d+)')
...              .astype('float64[pyarrow]'),
...          python3_ver=lambda df_:df_.python3_version_most
...               .str.replace('_', '.')
...               .str.extract(r'(?P<python3_ver>\d\.\d)'),
...               #.astype('float64[pyarrow]'),
...          use_python_most=lambda df_:df_.use_python_most
...               .fillna('Unknown')
...             )
...      .drop(columns=['python2_version_most'])
... )
```

The resulting dataframe has clean column names and data that is more amenable to analysis:

```
>>> print(jb2)
       age are_you_datascientist  ...  years_of_coding python3_ver
0       30                 <NA>  ...              1.0         3.7
1       21                 True  ...              3.0         3.6
2       30                False  ...              3.0         3.6
3     <NA>                 <NA>  ...             11.0         3.8
...    ...                  ...  ...              ...         ...
54458 <NA>                False  ...              1.0         3.7
54459   21                 <NA>  ...              6.0         3.7
54460   30                 True  ...              3.0         3.7
54461   21                 <NA>  ...              1.0         3.8

[54462 rows x 20 columns]
```

I also don't like to end chains with `.drop`. I prefer to use `.loc` to pull out the columns I want. To determine which columns I want to keep, I use the `.columns`, then I copy the results and put it into `.loc`.

```
>>> jb2.columns
Index(['age', 'are_you_datascientist', 'company_size',
       'country_live', 'employment_status',
       'first_learn_about_main_ide', 'how_often_use_main_ide',
       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
       'missing_features_main_ide', 'nps_main_ide', 'python_years',
       'python3_version_most', 'several_projects', 'team_size',
       'use_python_most', 'years_of_coding', 'python3_ver'],
      dtype='object')
```

```
>>> jb2 = (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
...          company_size=lambda df_:df_.company_size.replace(
...            {'Just me': '1', '': pd.NA,
...             'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...             '11–50':'11','51–500': '51', '501–1,000':'501',
...             '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...          country_live=lambda df_:df_.country_live.astype('category'),
...          employment_status=lambda df_:df_.employment_status
```

```
...               .fillna('Other').astype('category'),
...           is_python_main=lambda df_:df_.is_python_main
...               .astype('category'),
...           team_size=lambda df_:df_.team_size
...               .str.split(r'-', n=1, expand=True)
...               .iloc[:,0].replace('More than 40 people', '41')
...               .where(df_.company_size!=1, '1')
...               .replace('', pd.NA)
...               .astype('int8[pyarrow]'),
...           years_of_coding=lambda df_:df_.years_of_coding
...               .replace('Less than 1 year', '.5')
...               .str.extract(r'(?P<years_of_coding>\.?\d+)')
...               .astype('float64[pyarrow]'),
...           python_years=lambda df_:df_.python_years
...               .replace('Less than 1 year', '.5')
...               .str.extract(r'(?P<python_years>\.?\d+)')
...               .astype('float64[pyarrow]'),
...           python3_ver=lambda df_:df_.python3_version_most
...               .str.replace('_', '.')
...               .str.extract(r'(?P<python3_ver>\d\.\d)'),
...               #.astype('float64[pyarrow]'),
...           use_python_most=lambda df_:df_.use_python_most
...               .fillna('Unknown')
...             )
...     #.drop(columns=['python2_version_most'])
...   .loc[:, ['age', 'are_you_datascientist', 'company_size',
...       'country_live', 'employment_status',
...       'first_learn_about_main_ide', 'how_often_use_main_ide',
...       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
...       'missing_features_main_ide', 'nps_main_ide', 'python_years',
...       'python3_version_most', 'several_projects', 'team_size',
...       'use_python_most', 'years_of_coding', 'python3_ver']]
... )
```

Upon inspection, the *team_size* column still has several missing entries. It looks like there are over 5,000 respondents who are employed but neglected to enter a team size:

```
>>> def limit_index_length(df, max_length):
...     return df.rename(index=lambda x: x[:max_length])

>>> (jb2
...  .query('team_size.isna()')
...  .employment_status
...  .value_counts(dropna=False)
...  .pipe(limit_index_length, max_length=40)
```

```
... )
employment_status
Fully employed by a company / organizati    5279
Working student                              696
Partially employed by a company / organi     482
Self-employed (a person earning income d     430
Freelancer (a person pursuing a professi       0
Other                                          0
Retired                                        0
Student                                        0
Name: count, dtype: int64
```

I will use another call to `.assign` to use machine learning to predict the missing values for that column. I will leverage the CatBoost[1] library to do that. A nice feature of this library is that it will accept missing and string values (hence the name Category Boosting). Many machine learning libraries require that all data be numeric and that none of the values are missing.

Let's see what version of CatBoost I'm using:

```
>>> import catboost as cb
>>> cb.__version__
'1.2'
```

While CatBoost works with data from pandas dataframes, however, it is picky about the input that it supports. It doesn't like native pandas types (like `'Int64'` or `'category'`), so I'm going to make a function, `prep_for_ml`, that uses two dictionary comprehensions to change the column types when we make our predictions.

Note that this also uses `.assign` with **. You can pass in a dictionary to `.assign` or even another dataframe to merge the columns, but you need to use the ** operator to unpack the dictionary or dataframe. In this case, I'm passing in both a dataframe (the numbers and booleans) and a dictionary comprension for the categoricals. I could also have used this:

```
(df
 .select_types(['object', 'category', 'string'])
 .astype(str)
 .fillna('')
 )
```

instead of the dictionary comprehension:

```
{col:df[col].astype(str).fillna('')
 for col in df.select_dtypes(['object',
                              'category', 'string'])}
```

[1] https://catboost.ai

Since this is not a book about machine learning, I will not go deep into what is going on other than to say we are training the model on all the rows where *team_size* is not missing and using the trained model to predict the missing values. You may wish to use a more straightforward method, like `.fillna` to impute these missing values. (You can see I'm punting on the remaining missing values and just calling `.dropna` at the end. Also, note that summary statistics might be biased after filling in the values.)

```
>>> import catboost as cb
>>> import numpy as np

>>> def prep_for_ml(df):
...     # remove pandas types
...     return (df
...      .assign(**df.select_dtypes(['number', 'bool'])
...                  .astype('float[pyarrow]').astype(float),
...              **{col:df[col].astype(str).fillna('')
...                 for col in df.select_dtypes(['object',
...                                              'category', 'string'])})
...     )

>>> def predict_col(df, col):
...     df = prep_for_ml(df)
...     missing = df.query(f'~{col}.isna()')
...     cat_idx = [i for i,typ in enumerate(df.drop(columns=[col]).dtypes)
...                if str(typ) == 'object']
...     X = (missing
...          .drop(columns=[col])
...          .values
...         )
...     y = missing[col]
...     model = cb.CatBoostRegressor(iterations=20, cat_features=cat_idx)
...     model.fit(X,y, cat_features=cat_idx)
...     pred = model.predict(df.drop(columns=[col]))
...     return df[col].where(~df[col].isna(), pred)
```

I'm now going to calculate *team_size* with the `predict_col` function:

```
>>> jb2 = (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
```

```
...           company_size=lambda df_:df_.company_size.replace(
...             {'Just me': '1', '': pd.NA,
...             'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...             '11–50':'11','51–500': '51', '501–1,000':'501',
...             '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...           country_live=lambda df_:df_.country_live.astype('category'),
...           employment_status=lambda df_:df_.employment_status
...                .fillna('Other').astype('category'),
...           is_python_main=lambda df_:df_.is_python_main
...                .astype('category'),
...           team_size=lambda df_:df_.team_size
...               .str.split(r'-', n=1, expand=True)
...               .iloc[:,0].replace('More than 40 people', '41')
...               .where(df_.company_size!=1, '1')
...               .replace('', pd.NA)
...               .astype('int8[pyarrow]'),
...           years_of_coding=lambda df_:df_.years_of_coding
...               .replace('Less than 1 year', '.5')
...               .str.extract(r'(?P<years_of_coding>\.?\d+)')
...               .astype('float64[pyarrow]'),
...           python_years=lambda df_:df_.python_years
...               .replace('Less than 1 year', '.5')
...               .str.extract(r'(?P<python_years>\.?\d+)')
...               .astype('float64[pyarrow]'),
...           python3_ver=lambda df_:df_.python3_version_most
...                .str.replace('_', '.')
...                .str.extract(r'(?P<python3_ver>\d\.\d)'),
...           use_python_most=lambda df_:df_.use_python_most
...                    .fillna('Unknown'))
...       .assign(
...             team_size=lambda df_:predict_col(df_, 'team_size')
...             .astype(int))
...   .loc[:, ['age', 'are_you_datascientist', 'company_size',
...       'country_live', 'employment_status',
...       'first_learn_about_main_ide', 'how_often_use_main_ide',
...       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
...       'missing_features_main_ide', 'nps_main_ide', 'python_years',
...       'python3_version_most', 'several_projects', 'team_size',
...       'use_python_most', 'years_of_coding', 'python3_ver']]
... )

>>> print(jb2)

Learning rate set to 0.5
0:  learn: 2.9758568    total: 68.5ms   remaining: 1.3s
1:  learn: 2.8841040    total: 80.5ms   remaining: 724ms
```

```
2:  learn: 2.8443484    total: 92.7ms   remaining: 525ms
3:  learn: 2.8105584    total: 103ms    remaining: 414ms
4:  learn: 2.7922983    total: 112ms    remaining: 335ms
5:  learn: 2.7803329    total: 120ms    remaining: 280ms
6:  learn: 2.7756137    total: 129ms    remaining: 239ms
7:  learn: 2.7706510    total: 136ms    remaining: 205ms
8:  learn: 2.7571563    total: 145ms    remaining: 177ms
9:  learn: 2.7564631    total: 153ms    remaining: 153ms
10: learn: 2.7503591    total: 161ms    remaining: 132ms
11: learn: 2.7494745    total: 170ms    remaining: 113ms
12: learn: 2.7481258    total: 178ms    remaining: 95.6ms
13: learn: 2.7477180    total: 185ms    remaining: 79.3ms
14: learn: 2.7449738    total: 193ms    remaining: 64.4ms
15: learn: 2.7409940    total: 202ms    remaining: 50.6ms
16: learn: 2.7408640    total: 212ms    remaining: 37.4ms
17: learn: 2.7365108    total: 221ms    remaining: 24.6ms
18: learn: 2.7346780    total: 229ms    remaining: 12.1ms
19: learn: 2.7287662    total: 237ms    remaining: 0us
       age are_you_datascientist  ...  years_of_coding python3_ver
0       30              <NA>        ...             1.0         3.7
1       21              True        ...             3.0         3.6
2       30             False        ...             3.0         3.6
3     <NA>              <NA>        ...            11.0         3.8
...    ...               ...        ...             ...         ...
54458 <NA>             False        ...             1.0         3.7
54459   21              <NA>        ...             6.0         3.7
54460   30              True        ...             3.0         3.7
54461   21              <NA>        ...             1.0         3.8

[54462 rows x 20 columns]
```

I'm pretty satisfied with my chain (I would generally develop and debug this chain link by link using Jupyter). I like to create a function (I generally name it *tweak_**) and put it right at the top of my Jupyter notebook, in the cell below the cell where I load the raw data. This makes it easy to open up a notebook, load the raw data, and then run my tweak function to clean it up. After that, I'm off and running. If I need to modify my dataframe further, I will update the tweak function so all of my changes can be found in one place. It takes a little discipline to program pandas in this way, but you will reap benefits as your code will be easier to use, understand, and debug!

Note that the last line in my tweak function is to select the columns using `.loc`. Often, we are tempted to drop columns we don't want, but I prefer to choose the columns I want to keep. This makes it easier to see what columns are returned by the function. These are the columns that I want to use in my analysis. The columns that I drop are columns that I don't care about. My

code should focus on the columns I want to keep, not the columns I want to drop.

Here is what my cleaned-up code will look like:

```
>>> import catboost as cb
>>> import numpy as np
>>> import pandas as pd

>>> import collections

>>> def get_uniq_cols(jb):
...     counter = collections.defaultdict(list)
...     for col in sorted(jb.columns):
...         period_count = col.count('.')
...         if period_count >= 2:
...             part_end = 2
...         else:
...             part_end = 1
...         parts = col.split('.')[:part_end]
...         counter['.'.join(parts)].append(col)
...     uniq_cols = []
...     for cols in counter.values():
...         if len(cols) == 1:
...             uniq_cols.extend(cols)
...     return uniq_cols

>>> def prep_for_ml(df):
...     # remove pandas/pyarrow types
...     return (df
...      .assign(**df.select_dtypes(['number', 'bool'])
...                  .astype('float[pyarrow]').astype(float),
...              **{col:df[col].astype(str).fillna('')
...                 for col in df.select_dtypes(['object',
...                                              'category', 'string'])})
...     )

>>> def predict_col(df, col):
...     df = prep_for_ml(df)
...     missing = df.query(f'~{col}.isna()')
...     cat_idx = [i for i,typ in enumerate(df.drop(columns=[col]).dtypes)
...                if str(typ) == 'object']
...     X = (missing
...          .drop(columns=[col])
...          .values
...         )
...     y = missing[col]
```

```
...     model = cb.CatBoostRegressor(iterations=20, cat_features=cat_idx)
...     model.fit(X,y, cat_features=cat_idx)
...     pred = model.predict(df.drop(columns=[col]))
...     return df[col].where(~df[col].isna(), pred)

>>> def tweak_jb(jb):
...     uniq_cols = get_uniq_cols(jb)
...     return (jb
...      [uniq_cols]
...      .rename(columns=lambda c: c.replace('.', '_'))
...      .assign(age=lambda df_:df_.age.str.slice(0,2)
...                                .astype('int8[pyarrow]'),
...              are_you_datascientist=lambda df_: df_.are_you_datascientist
...                 .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...                 .astype('bool[pyarrow]'),
...              company_size=lambda df_:df_.company_size.replace(
...                {'Just me': '1', '': pd.NA,
...                'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...                '11–50':'11','51–500': '51', '501–1,000':'501',
...                '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...              employment_status=lambda df_:df_.employment_status
...                     .fillna('Other').astype('category'),
...              is_python_main=lambda df_:df_.is_python_main
...                     .astype('category'),
...              team_size=lambda df_:df_.team_size
...                    .str.split(r'-', n=1, expand=True)
...                    .iloc[:,0].replace('More than 40 people', '41')
...                    .where(df_.company_size!=1, '1')
...                    .replace('', pd.NA)
...                    .astype('int8[pyarrow]'),
...              years_of_coding=lambda df_:df_.years_of_coding
...                    .replace('Less than 1 year', '.5')
...                    .str.extract(r'(?P<years_of_coding>\.?\d+)')
...                    .astype('float64[pyarrow]'),
...              python_years=lambda df_:df_.python_years
...                    .replace('Less than 1 year', '.5')
...                    .str.extract(r'(?P<python_years>\.?\d+)')
...                    .astype('float64[pyarrow]'),
...              python3_ver=lambda df_:df_.python3_version_most
...                     .str.replace('_', '.')
...                     .str.extract(r'(?P<python3_ver>\d\.\d)'),
...              use_python_most=lambda df_:df_.use_python_most
...                     .fillna('Unknown'))
...       .assign(
...              team_size=lambda df_:predict_col(df_, 'team_size')
...                .astype(int))
```

```
...  .loc[:, ['age', 'are_you_datascientist', 'company_size',
...       'country_live', 'employment_status',
...       'first_learn_about_main_ide', 'how_often_use_main_ide',
...       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
...       'missing_features_main_ide', 'nps_main_ide', 'python_years',
...       'python3_version_most', 'several_projects', 'team_size',
...       'use_python_most', 'years_of_coding', 'python3_ver']]
... )

>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...    '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)
Learning rate set to 0.5
0:  learn: 2.9758568    total: 11ms remaining: 210ms
1:  learn: 2.8841040    total: 20.7ms   remaining: 186ms
2:  learn: 2.8443484    total: 30.8ms   remaining: 174ms
3:  learn: 2.8105584    total: 40.4ms   remaining: 161ms
4:  learn: 2.7922983    total: 49.3ms   remaining: 148ms
5:  learn: 2.7803329    total: 58.5ms   remaining: 136ms
6:  learn: 2.7756137    total: 66.8ms   remaining: 124ms
7:  learn: 2.7706510    total: 75.1ms   remaining: 113ms
8:  learn: 2.7571563    total: 83.7ms   remaining: 102ms
9:  learn: 2.7564631    total: 92.8ms   remaining: 92.8ms
10: learn: 2.7503591    total: 101ms    remaining: 82.6ms
11: learn: 2.7494745    total: 110ms    remaining: 73.3ms
12: learn: 2.7481258    total: 119ms    remaining: 63.8ms
13: learn: 2.7477180    total: 127ms    remaining: 54.4ms
14: learn: 2.7449738    total: 136ms    remaining: 45.3ms
15: learn: 2.7409940    total: 145ms    remaining: 36.2ms
16: learn: 2.7408640    total: 153ms    remaining: 27.1ms
17: learn: 2.7365108    total: 161ms    remaining: 17.9ms
18: learn: 2.7346780    total: 170ms    remaining: 8.94ms
19: learn: 2.7287662    total: 178ms    remaining: 0us
```

I use many `lambda` functions in my `.assign` calls. Because I have renamed the columns and want to refer to the columns by their new names, I need to use `lambda` functions. Another feature of `lambda` is that it allows me to refer to a column I just created. However, in this case, I only use it for the renaming.

If I wanted to use only one `lambda` function, I could have replaced:

```
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                           .astype('int8[pyarrow]'),
...       are_you_datascientist=lambda df_: df_.are_you_datascientist
...          .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...          .astype('bool[pyarrow]')  ,
```

```
...         company_size=lambda df_:df_.company_size.replace(
...           {'Just me': '1', '': pd.NA,
...            'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...            '11–50':'11','51–500': '51', '501–1,000':'501',
...            '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...         country_live=lambda df_:df_.country_live.astype('category'),
...         employment_status=lambda df_:df_.employment_status
...              .fillna('Other').astype('category'),
```

with code that uses a `.pipe` call with a single `lambda` function:

```
... .pipe(lambda df_: df_.assign(
...     age=df_.age.str.slice(0,2)
...            .astype('int8[pyarrow]'),
...     are_you_datascientist=df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
...     company_size=df_.company_size.replace(
...             {'Just me': '1', '': pd.NA,
...              'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...              '11–50':'11','51–500': '51', '501–1,000':'501',
...              '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...     country_live=df_.country_live.astype('category'),
...     employment_status=df_.employment_status
...             .fillna('Other').astype('category'),
```

This would pass in the renamed dataframe to the `.pipe` call and then I could use the `df_` variable without having to use `lambda` functions for every column.

Table 23.1: Dataframe Chapter Methods

Method	Description
`.rename( mapper=None, index=None, columns=None, axis=0, copy=True, level=None, errors='ignore')`	Change axis labels. Pass `columns` or `index` as a dictionary (mapping old values to new values) or a function (accepting the old value and returning the new value).
`.replace( to_replace=None, value=None, limit=None, regex=False, method='pad')`	Replace values from `to_replace` (string, regular expression, number, series or list of the previous, dictionary (mapping replacement if `value` is `None`), series, or `None`) with `value`. If `to_replace` is a list and there is no `value`, you can `bfill` or `ffill` with `method`.

Method	Description
`.drop(labels=None, axis=0, index=None, columns=None, level=None, errors='raise')`	Drop rows or columns with specified labels. Use `columns='age'` rather than `labels='age', axis='1'`.
`.dropna(axis=0, how='any', thresh=None, subset=None)`	Drop rows (`axis=0`) or columns (`axis=1`) with missing values. Require a certain amount missing with `thresh`. Limit columns with `subset`.
`.query(expr)`	Evaluate `expr` to filter the dataframe. Refer to variables by prefixing them with `@`. Use backticks around column names with spaces.
`.assign(**kwargs)`	Return a new dataframe with updated or new columns. `kwargs` maps column name to function, scalar, or series. If using a function, it is passed in the current state of the dataframe and should return a scalar or series. Subsequent columns may reference earlier columns in `kwargs` if you use a function.

23.4 Summary

If you need to update a column or add a new column, use the `.assign` method. If the `.assign` method is part of a chain, you may want to couple it with a function to have the current state of the dataframe you are working with. I generally will make a function to clean up my data and then put it right at the top of my notebook below where I load it so I can load the raw data and then clean it up in two steps.

23.5 Exercises

With a dataset of your choice:

1. Create a "tweak" function to clean up the data.
2. Explore the memory usage of the raw and tweaked data.

Chapter 24

Dealing with Missing and Duplicated Data

We have seen how to find missing and duplicated data with a series, and let's apply it to a dataframe. If you are doing analysis or creating machine learning models on your data, you will want to ensure that it is complete before you start to report on it. Also, many machine learning models will fail if you try to train them on dataframes with missing values.

We are going to jump back to the Presidential data for this chapter.

24.1 Missing Data

Determining where data is missing involves the same methods we saw in a series. We need to remember that a dataframe has an extra dimension. The dataframe has an `.isna` method that returns a dataframe with true and false values indicating whether values are missing:

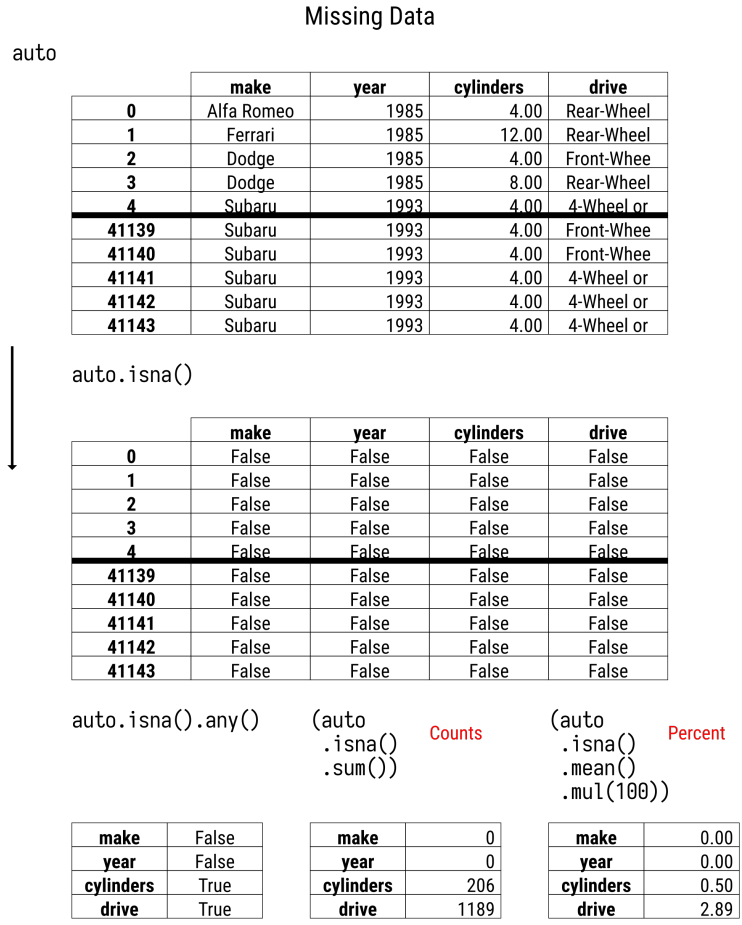
```
>>> def tweak_siena_pres(df):
...     def int64_to_uint8(df_):
...         cols = df_.select_dtypes('int64')
...         return (df_
...                 .astype({col:'uint8[pyarrow]' for col in cols}))
...
...     return (df
...      .rename(columns={'Seq.':'Seq'})    # 1
...      .rename(columns={k:v.replace(' ', '_') for k,v in
...         {'Bg': 'Background',
...          'PL': 'Party leadership', 'CAb': 'Communication ability',
...          'RC': 'Relations with Congress', 'CAp': 'Court appointments',
...          'HE': 'Handling of economy', 'L': 'Luck',
...          'AC': 'Ability to compromise', 'WR': 'Willing to take risks',
...          'EAp': 'Executive appointments', 'OA': 'Overall ability',
...          'Im': 'Imagination', 'DA': 'Domestic accomplishments',
...          'Int': 'Integrity', 'EAb': 'Executive ability',
...          'FPA': 'Foreign policy accomplishments',
```

```
...          'LA': 'Leadership ability',
...          'IQ': 'Intelligence', 'AM': 'Avoid crucial mistakes',
...          'EV': "Experts' view", 'O': 'Overall'}.items()})
...      .astype({'Party':'category'})  # 2
...      .pipe(int64_to_uint8)  # 3
...      .assign(Average_rank=lambda df_:(df_.select_dtypes('uint8') # 4
...         .sum(axis=1).rank(method='dense').astype('uint8[pyarrow]')),
...              Quartile=lambda df_:pd.qcut(df_.Average_rank, 4,
...                  labels='1st 2nd 3rd 4th'.split())
...             )
...     )

>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> print(pres.isna())
      Seq  President  ...  Average_rank  Quartile
1   False      False  ...         False     False
2   False      False  ...         False     False
3   False      False  ...         False     False
4   False      False  ...         False     False
..    ...        ...  ...           ...       ...
41  False      False  ...         False     False
42  False      False  ...         False     False
43  False      False  ...         False     False
44  False      False  ...         False     False

[44 rows x 26 columns]
```

Because each of these columns is a boolean array, you can use them to select rows where values are missing.

Let's look at rows where *Integrity* is missing:

```
>>> print(pres[pres.Integrity.isna()])
Empty DataFrame
Columns: [Seq, President, Party, Background, Imagination, Integrity,
     Intelligence, Luck, Willing_to_take_risks, Ability_to_compromise,
     Executive_ability, Leadership_ability, Communication_ability,
     Overall_ability, Party_leadership, Relations_with_Congress,
     Court_appointments, Handling_of_economy, Executive_appointments,
     Domestic_accomplishments, Foreign_policy_accomplishments,
     Avoid_crucial_mistakes, Experts'_view, Overall, Average_rank,
     Quartile]
```



	make	year	cylinders	drive
0	Alfa Romeo	1985	4.00	Rear-Wheel
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

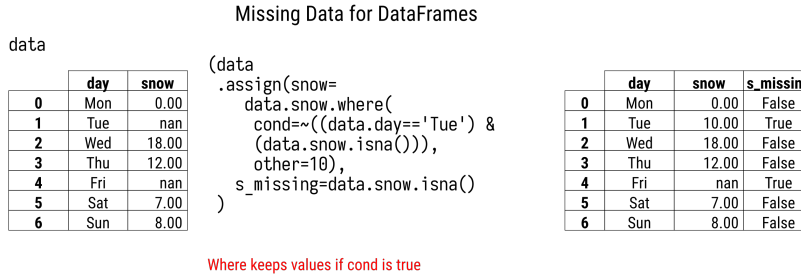
	make	year	cylinders	drive
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False
41139	False	False	False	False
41140	False	False	False	False
41141	False	False	False	False
41142	False	False	False	False
41143	False	False	False	False

make	False
year	False
cylinders	True
drive	True

make	0
year	0
cylinders	206
drive	1189

make	0.00
year	0.00
cylinders	0.50
drive	2.89

Figure 24.1: Using `.isna` to create a boolean array of missing values, counting them, or getting the percent of them.



	day	snow
0	Mon	0.00
1	Tue	nan
2	Wed	18.00
3	Thu	12.00
4	Fri	nan
5	Sat	7.00
6	Sun	8.00

	day	snow	s_missing
0	Mon	0.00	False
1	Tue	10.00	True
2	Wed	18.00	False
3	Thu	12.00	False
4	Fri	nan	True
5	Sat	7.00	False
6	Sun	8.00	False

Figure 24.2: A more complicated example of filling in missing values using `.where`.

```
Index: []

[0 rows x 26 columns]
```

It looks like there are no missing values for this column.
My current favorite way of doing this is to use the `.query` method:

```
>>> print(pres.query('Integrity.isna()'))
Empty DataFrame
Columns: [Seq, President, Party, Background, Imagination, Integrity,
     Intelligence, Luck, Willing_to_take_risks, Ability_to_compromise,
     Executive_ability, Leadership_ability, Communication_ability,
     Overall_ability, Party_leadership, Relations_with_Congress,
     Court_appointments, Handling_of_economy, Executive_appointments,
     Domestic_accomplishments, Foreign_policy_accomplishments,
     Avoid_crucial_mistakes, Experts'_view, Overall, Average_rank,
     Quartile]
Index: []

[0 rows x 26 columns]
```

We can sum the results to get the counts of columns with missing values:

```
>>> pres.isna().sum()
Seq              0
President        0
Party            0
Background       0
                ..
Experts'_view    0
Overall          0
```

```
Average_rank     0
Quartile         0
Length: 26, dtype: int64
```

We can take the mean of them to get the fraction missing. In this case, none of them are missing:

```
>>> pres.isna().mean()
Seq              0.0
President        0.0
Party            0.0
Background       0.0
                ...
Experts'_view    0.0
Overall          0.0
Average_rank     0.0
Quartile         0.0
Length: 26, dtype: float64
```

With these tools, you should be able to diagnose and locate missing data. Once you discover where the data is missing, you need to determine what actions to take. You can drop missing values with `.dropna`. There is a `.fillna` and an `.interpolate` method on the dataframe. But often, those are too rough of tools when dealing with multiple columns, as the columns represent different things. (I do find them helpful after grouping the data). I generally do that at the series level and then use `.assign` to update the column, filling in the missing values.

24.2 Duplicates

Like the series `.drop_duplicates` method, the same method is available to the dataframe. When called without parameters, it is often too blunt of a tool to use on a dataframe. However, the `subset` parameter allows you to specify which columns you want it to consider dropping:

```
>>> print(pres.drop_duplicates())
   Seq       President  ... Average_rank  Quartile
1    1  George Wash...  ...            1       1st
2    2      John Adams  ...           13       2nd
3    3  Thomas Jeff...  ...            5       1st
4    4   James Madison  ...            7       1st
..  ..             ...  ...          ...       ...
41  42    Bill Clinton  ...           15       2nd
42  43  George W. Bush  ...           33       3rd
43  44    Barack Obama  ...           17       2nd
44  45    Donald Trump  ...           42       4th
```

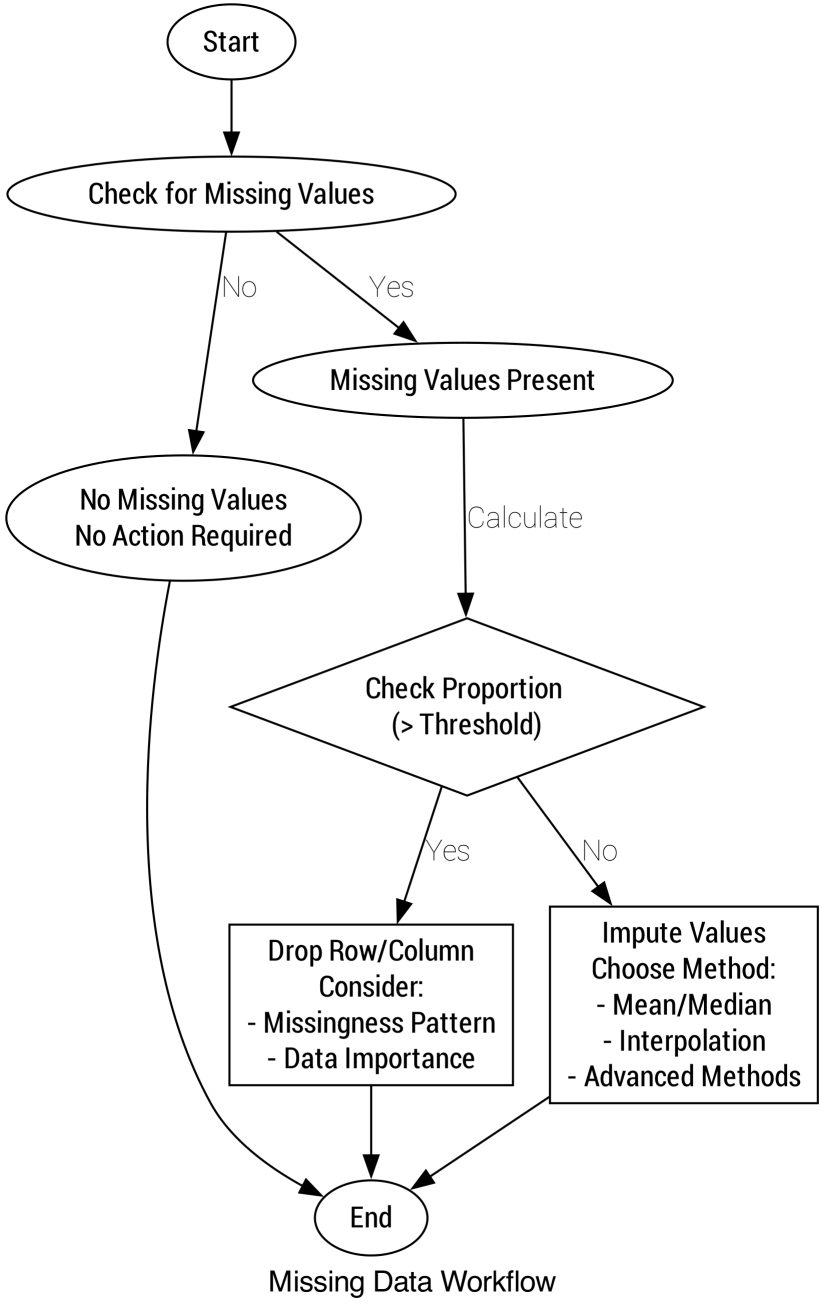


Figure 24.3: Flowchart for dealing with missing data. Remember to consult your local subject matter expert.

```
[44 rows x 26 columns]
```

The above call does nothing because none of the rows are complete copies. If we wanted to keep only the first president from each party, we could do the following:

```
>>> print(pres.drop_duplicates(subset='Party'))
   Seq       President  ... Average_rank  Quartile
1    1  George Wash...  ...            1       1st
2    2      John Adams  ...           13       2nd
3    3  Thomas Jeff...  ...            5       1st
7    7  Andrew Jackson  ...           19       2nd
9    9  William Hen...  ...           38       4th
16  16  Abraham Lin...  ...            3       1st

[6 rows x 26 columns]
```

You can use the `keep` parameter to specify how to drop values. The default value, `'first'` will keep the first value. You can use `'last'` or `False` to keep the last value or to drop all duplicates, respectively:

```
>>> print(pres.drop_duplicates(subset='Party', keep='last'))
   Seq       President  ... Average_rank  Quartile
2    2      John Adams  ...           13       2nd
6    6  John Quincy...  ...           18       2nd
10  10      John Tyler  ...           37       4th
13  13  Millard Fil...  ...           39       4th
43  44    Barack Obama  ...           17       2nd
44  45    Donald Trump  ...           42       4th

[6 rows x 26 columns]

>>> print(pres.drop_duplicates(subset='Party', keep=False))
  Seq   President  ... Average_rank  Quartile
2   2  John Adams  ...           13       2nd

[1 rows x 26 columns]
```

We need more logic to drop duplicates if only the previous row is a duplicate (rather than any row). We do this by creating a column that indicates whether it is not the same as the next value. This indicates whether it is the first entry in a sequence. Then we can combine this with a `lambda` function and `.loc`:

```
>>> print(pres
...  .assign(first_in_party_seq=lambda df_:
```

```
...             df_.Party != df_.Party.shift(1))
...  .query('first_in_party_seq')
... )
   Seq       President  ... Quartile  first_in_party_seq
1    1  George Wash...  ...      1st            True
2    2      John Adams  ...      2nd            True
3    3  Thomas Jeff...  ...      1st            True
7    7  Andrew Jackson  ...      2nd            True
..  ..             ...  ...      ...             ...
41  42    Bill Clinton  ...      2nd            True
42  43  George W. Bush  ...      3rd            True
43  44    Barack Obama  ...      2nd            True
44  45    Donald Trump  ...      4th            True

[26 rows x 27 columns]
```

Table 24.1: Dataframe Chapter Methods

Method	Description
`.isna()`	Return a boolean dataframe with the same dimensions with `True` values where cells are missing.
`.sum(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return sum over axis. The default of empty sequence is 0, set `min_count=1` t
`.mean(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return mean over axis.
`.drop_duplicates( subset=None, keep='first', ignore_index=False)`	Return dataframe that has duplicated rows removed. Indicate certain columns to consider with `subset`. `keep` can be `'first'`, `'last'`, or `False` (drop all dupes). Set `ignore_index=True` to reset the index.

24.3 Summary

In this chapter, we saw how you could diagnose how much data is missing in a dataframe. In a later chapter, we will see how to fill in the missing JetBrains survey data. The time series chapter will examine methods for dealing with missing data in sequential data sets.

24.4 Exercises

With a dataset of your choice:

1. Find out which columns have missing data.
2. Count the number of missing values for each column.
3. Find the percentage of missing values for each column.
4. Find the rows with missing data.
5. Find the rows that are duplicated.

Chapter 25

Sorting Columns and Indexes

In this chapter, we will explore sorting columns and index values.

25.1 Sorting Columns

The `.sort_values` method will allow you to sort the rows of a dataframe by arbitrary columns. In this example, we sort by the political party in alphabetic order:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow',
...                  engine='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> print(pres.sort_values(by='Party'))
      Seq       President  ... Average_rank  Quartile
                           ...
22  22/24  Grover Clev...  ...           23       3rd
31     32  Franklin D....  ...            2       1st
17     17  Andrew Johnson  ...           44       4th
32     33  Harry S. Tr...  ...            9       1st
..    ...             ...  ...          ...       ...
44     45    Donald Trump  ...           42       4th
13     13  Millard Fil...  ...           39       4th
12     12  Zachary Taylor  ...           30       3rd
9       9  William Hen...  ...           38       4th

[44 rows x 26 columns]
```

You can also sort by multiple columns and specify whether each column should be sorted in ascending (the default) or descending order. Here, we

sort by the *Party* column in ascending alphabetic order and *Average_rank* in descending order:

```
>>> print(pres
...   .sort_values(by=['Party', 'Average_rank'],
...       ascending=[True, False])
... )
   Seq       President  ... Average_rank  Quartile
                        ...
17  17  Andrew Johnson  ...           44       4th
15  15  James Buchanan  ...           43       4th
14  14  Franklin Pi...  ...           41       4th
38  39    Jimmy Carter  ...           27       3rd
..  ..             ...  ...          ...       ...
16  16  Abraham Lin...  ...            3       1st
13  13  Millard Fil...  ...           39       4th
9    9  William Hen...  ...           38       4th
12  12  Zachary Taylor  ...           30       3rd

[44 rows x 26 columns]
```

Like the built-in `sorted` function, you can supply a `key` function to the `.sort_values` method to determine how to sort the `by` column. Let's sort the rows by the last name of the president. We will use `.str.split` to separate the parts of the name:

```
>>> print(pres
...   .President
...   .str.split()
... )


1     ['George' '...
2     ['John' 'Ad...
3     ['Thomas' '...
4     ['James' 'M...
          ...
41    ['Bill' 'Cl...
42    ['George' '...
43    ['Barack' '...
44    ['Donald' '...
Name: President, Length: 44, dtype: list<item: string>[pyarrow]
```

This is a case where `.apply` might be appropriate (another hint is that we are manipulating strings that are not vectorized operations.) Each value is a Python list, and we need the last value:

```
>>> print(pres
...   .President
...   .str.split(' ')
...   .apply(lambda val: val[-1])
... )

1     Washington
2          Adams
3      Jefferson
4        Madison
         ...
41       Clinton
42          Bush
43         Obama
44         Trump
Name: President, Length: 44, dtype: object
```

However, if we use the pandas 1.x string type, the `.str` attribute supports slicing on it directly.

```
>>> print(pres
...   .President
...   .astype(str)
...   .str.split()
...   .str[-1]
... )

1     Washington
2          Adams
3      Jefferson
4        Madison
         ...
41       Clinton
42          Bush
43         Obama
44         Trump
Name: President, Length: 44, dtype: object
```

Awesome, we just need to put this logic into the `key` function:

```
>>> print(pres
...    .sort_values(by='President',
...       key=lambda name_ser: name_ser
...           .astype(str)
...           .str.split()
...           .str[-1])
```

```
... )

   Seq       President  ... Average_rank  Quartile
                        ...
2    2      John Adams  ...           13       2nd
6    6  John Quincy...  ...           18       2nd
21  21  Chester A. ...  ...           34       4th
15  15  James Buchanan  ...           43       4th
..  ..             ...  ...          ...       ...
44  45    Donald Trump  ...           42       4th
10  10      John Tyler  ...           37       4th
1    1  George Wash...  ...            1       1st
27  28  Woodrow Wilson  ...           12       2nd

[44 rows x 26 columns]
```

25.2 Sorting Column Order

If you want to sort the columns, you can use the `.sort_index` method and set the `axis` value appropriately:

```
>>> print(pres.sort_index(axis='columns'))
    Ability_to_compromise  Average_rank  ...  Seq  \
                                         ...
1                       2             1  ...    1
2                      31            13  ...    2
3                      14             5  ...    3
4                       6             7  ...    4
..                    ...           ...  ...  ...
41                      3            15  ...   42
42                     28            33  ...   43
43                     16            17  ...   44
44                     42            42  ...   45

    Willing_to_take_risks

1                       6
2                      14
3                       5
4                      15
..                    ...
41                     17
42                     20
43                     23
44                     25
```

```
[44 rows x 26 columns]
```

I don't find myself using this very often unless I have an index with string values (as we will see later).

25.3 Setting and Sorting the Index

You can stick a column into the index with `.set_index`. You may want to follow that up with sorting the index:

```
>>> print(pres
...   .set_index('President')
...   .sort_index()
... )
                Seq       Party  ...  Average_rank  Quartile
President                        ...
Abraham Lincoln  16  Republican  ...             3       1st
Andrew Jackson    7  Democratic  ...            19       2nd
Andrew Johnson   17  Democratic  ...            44       4th
Barack Obama     44  Democratic  ...            17       2nd
...              ..         ...  ...           ...       ...
William Howa...  27  Republican  ...            22       2nd
William McKi...  25  Republican  ...            20       2nd
Woodrow Wilson   28  Democratic  ...            12       2nd
Zachary Taylor   12        Whig  ...            30       3rd

[44 rows x 25 columns]
```

If you sort an index with duplicated string index values, then you can slice on the index. If you did not sort the index, you will get a `KeyError`:

```
>>> (pres
...     .set_index('Party')
...     .loc['Democratic':'Republican']
... )
Traceback (most recent call last):
  ...
KeyError: "Cannot get left slice bound for non-unique label: 'Democratic'"
```

Sorting the index allows us to slice the index by name:

```
>>> print(pres
...   .set_index('Party')
...   .sort_index()
...   .loc['Democratic':'Republican']
... )
```

```
              Seq       President  ...  Average_rank  Quartile
Party                                ...
Democratic  22/24  Grover Clev...  ...            23       3rd
Democratic     32  Franklin D....  ...             2       1st
Democratic     17  Andrew Johnson  ...            44       4th
Democratic     33  Harry S. Tr...  ...             9       1st
...           ...             ...  ...           ...       ...
Republican     25  William McK...  ...            20       2nd
Republican     23  Benjamin Ha...  ...            36       4th
Republican     18  Ulysses S. ...  ...            24       3rd
Republican     45    Donald Trump  ...            42       4th

[41 rows x 25 columns]
```

Table 25.1: Dataframe Sorting and Indexing Methods

Method	Description
`.sort_values(by, axis=0, ascending=True, kind='quicksort', na_position= 'last', ignore_index= False, key=None))`	Return dataframe with values sorted along the axis. Use `by` to specify a column (string) or a list of columns (for `axis=0`). You can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. A `key` function accepts a series and should return a series with the same index.
`.sort_index( axis=0, level=None, ascending= True, kind= 'quicksort', na_position= 'last', sort_remaining= True, ignore_index= False, key=None))`	Return dataframe with index (`axis=0`) or columns (`axis=1`) sorted. Can specify a single level or multiple levels with `levels`. Can specify the column (string) or a list of columns (for `axis=0`). Can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. You can reset the index with `ignore_index`. A `key` function accepts an index and should return an index. For multi-level indexes, each index is passed in independently to the function.

Method	Description
`.set_index( keys, drop=True, append=False, verify_integrity= False)`	Return dataframe with the new index. The `keys` argument can be a column name, a series (or numpy array) of labels for the index, or a list of column names or series. The `drop` parameter indicates whether to remove columns used for the index. The `append` parameter allows you to add additional index levels. You can check for duplicate index values by setting `verify_integrity=True`.
`.loc`	Attribute to index off of by index and column names. Slices use the closed interval (including start and end).

25.4 Summary

In this chapter, we showed how to sort both the index and the columns. If you want to sort based on arbitrary values, you can use the `key` parameter to determine how sorting occurs. You can also sort by various columns and control the sort's direction. Sorting the index is particularly useful if it contains strings because you can slice the string values (or substrings) after the index is sorted.

25.5 Exercises

With a dataset of your choice:

1. Sort the index.
2. Set the index to a string column, sort the index, and slice by a substring of index values.
3. Sort by a single column.
4. Sort by a single column in descending order.
5. Sort by two columns.
6. Sort by the last letter of a string column.

Chapter 26

Filtering and Indexing Operations

I like keeping my data in the columns, not the index. Occasionally, you will need to manipulate the index. This chapter will explore some of the operations to change the index and operations that result from that. Then, we will look at pulling data out based on index and column names and locations.

26.1 Renaming an Index

We will update the index values using the `.rename` method in this example. This method will accept a function that takes the current value and returns a new one. Here, we will use the first initial of the president:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> def name_to_initial(val):
...     names = val.split()
...     return ' '.join([f'{names[0][0]}.', *names[1:]])

>>> print(pres
...  .set_index('President')
...  .rename(name_to_initial)
... )
              Seq          Party  Background  ...  Overall  \
President                                     ...
G. Washington   1    Independent           7  ...        1
J. Adams        2     Federalist           3  ...       14
T. Jefferson    3  Democratic-...           2  ...        5
J. Madison      4  Democratic-...           4  ...        7
...            ..            ...         ...  ...      ...
```

```
B. Clinton     42      Democratic          21  ...       15
G. W. Bush     43      Republican          17  ...       33
B. Obama       44      Democratic          24  ...       17
D. Trump       45      Republican          43  ...       42

               Average_rank  Quartile
President
G. Washington             1       1st
J. Adams                 13       2nd
T. Jefferson              5       1st
J. Madison                7       1st
...                     ...       ...
B. Clinton               15       2nd
G. W. Bush               33       3rd
B. Obama                 17       2nd
D. Trump                 42       4th

[44 rows x 25 columns]
```

26.2 Resetting the Index

If you want a monotonically increasing integer index for a dataframe, use the `.reset_index` method:

```
>>> print(pres
...  .set_index('President')
...  .reset_index()
... )
         President Seq           Party  ...  Overall  Average_rank  \
0   George Wash...   1     Independent  ...        1             1
1       John Adams   2      Federalist  ...       14            13
2   Thomas Jeff...   3  Democratic-...  ...        5             5
3    James Madison   4  Democratic-...  ...        7             7
..             ...  ..             ...  ...      ...           ...
40    Bill Clinton  42      Democratic  ...       15            15
41  George W. Bush  43      Republican  ...       33            33
42    Barack Obama  44      Democratic  ...       17            17
43    Donald Trump  45      Republican  ...       42            42

    Quartile
0        1st
1        2nd
2        1st
3        1st
..       ...
40       2nd
```

```
41       3rd
42       2nd
43       4th

[44 rows x 26 columns]
```

26.3 Dataframe Indexing, Filtering, & Querying

We have already looked at how to use boolean arrays to index a series and limit what it returns. We can also do this with dataframes. Let's look at the presidents with an *Average_rank* below 10. First, we will make a boolean array where the column *Average_rank* is below 10. Then, we will index into the dataframe with this boolean array:

```
>>> lt10 = pres.Average_rank < 10
>>> print(pres[lt10])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
3    3  Thomas Jeff...  Democratic-...  ...        5             5
4    4   James Madison  Democratic-...  ...        7             7
5    5    James Monroe  Democratic-...  ...        8             8
..  ..             ...             ...  ...      ...           ...
25  26  Theodore Ro...      Republican  ...        4             4
31  32  Franklin D....      Democratic  ...        2             2
32  33  Harry S. Tr...      Democratic  ...        9             9
33  34  Dwight D. E...      Republican  ...        6             6

    Quartile
1        1st
3        1st
4        1st
5        1st
..       ...
25       1st
31       1st
32       1st
33       1st

[9 rows x 26 columns]
```

Let's add in another option if they are a Republican:

```
>>> print(pres[lt10 & (pres.Party == 'Republican')])
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
```

```
33  34  Dwight D. E...  Republican  ...        6            6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```

Note

Be careful when combining conditions in indexing operations. If we inline the above operation, we get a different result:

```
>>> pres[pres.Average_rank < 10 & pres.Party == 'Republican']
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for &: 'int' and 'Categorical'
```

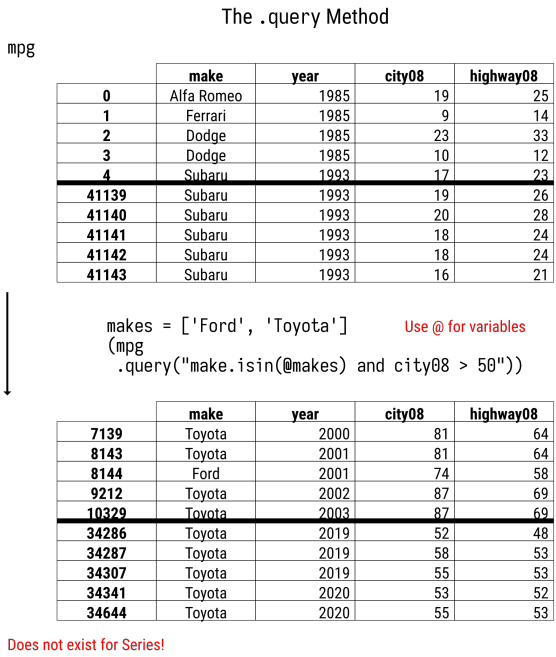
This is because the `&` operator has higher precedence than `>=`. So in effect the above is doing `pres.Average_rank < (10 & pres.Party == 'Republican')`. Let's look at what that does:

```
>>> 10 & pres.Party == 'Republican'
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for &: 'int' and 'Categorical'
```

Sometimes, you will get back an answer here (if you are not comparing to a categorical), but you might not get the answer you wanted due to precedence.

The takeaway here is that you should always put parentheses around multiple conditions in index operations if you inline them:

```
>>> print(pres[(pres.Average_rank < 10) & (pres.Party == 'Republican')])
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	make	year	city08	highway08
7139	Toyota	2000	81	64
8143	Toyota	2001	81	64
8144	Ford	2001	74	58
9212	Toyota	2002	87	69
10329	Toyota	2003	87	69
34286	Toyota	2019	52	48
34287	Toyota	2019	58	53
34307	Toyota	2019	55	53
34341	Toyota	2020	53	52
34644	Toyota	2020	55	53

Figure 26.1: The `.query` method allows you to call methods, include variables, and combine conditional expressions inside a string.

One method unique to the dataframe (not found on a series) is the `.query` method. Instead of creating boolean arrays, we create a string, similar to SQL, with the conditions we want:

```
>>> print(pres.query('Average_rank < 10 and Party == "Republican"'))
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```

In the case of `.query`, we can use `and` or `&`. In contrast, when we want to combine boolean arrays, we need to use `&` (likewise, we can use `or` and

not in .query). We also do not need to worry as much about precedence and parentheses.

If you have an existing variable and want to refer to it inside of the string, you can prefix the variable with a @:

```
>>> lt10 = pres.Average_rank < 10
>>> print(pres.query('@lt10 and Party == "Republican"'))

   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```

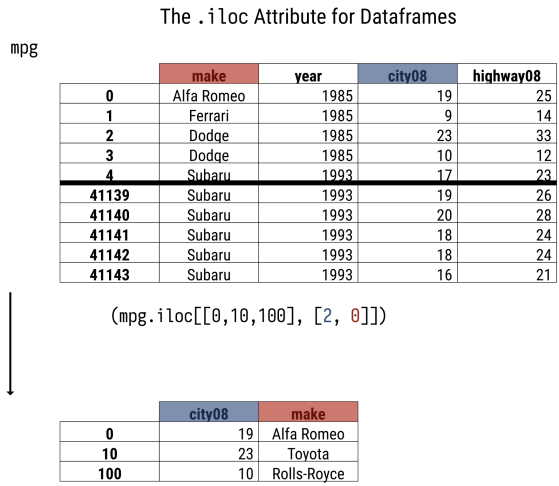
26.4 Indexing by Position

This section discusses .iloc. I have a pretty strong opinion that you should not use .iloc in your production code. I think it is much better to use .loc, .head, or .tail to get your desired data. It makes your code more readable. However, I will discuss this feature because you will see it, and in some cases, it can be handy when doing quick exploratory analysis.

The .iloc attribute allows us to pull out both rows and columns from a dataframe. Here, we pull out row position 1. Note that this returns the result as a series (even though it represents a row):

```
>>> pres.iloc[1]
Seq                     2
President        John Adams
Party            Federalist
Background                3
                        ...
Experts'_view            10
Overall                  14
Average_rank             13
Quartile                2nd
Name: 2, Length: 26, dtype: object
```

In the following example, instead of passing in the scalar position, we will pass in row position 1 in a list. Sometimes you will hear people say to use a "nested list". To be pedantic, this is not a nested list. It is an indexing



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	city08	make
0	19	Alfa Romeo
10	23	Toyota
100	10	Rolls-Royce

Figure 26.2: Using `.iloc` to select rows and columns by position. Note that Python is 0-based indexing, so 0 is the first entry, 1 is the second, etc.

operation (the outer brackets) with a list (the inner brackets). This does not return a series but a dataframe with a single row.

```
>>> print(pres.iloc[[1]])
  Seq   President       Party  ...  Overall  Average_rank  Quartile
2   2  John Adams  Federalist  ...       14            13       2nd

[1 rows x 26 columns]
```

We can also pass in slices and lists:

```
>>> print(pres.iloc[[0, 5, 10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]
```

```
>>> print(pres.iloc[0:11:5])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]
```

Finally, you can pass a function into the index operation. The function takes a dataframe and should return valid options for `.iloc`. The two following operations should give the same results:

```
>>> print(pres.iloc[[0, 5, 10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]

>>> print(pres.iloc[lambda df: [0,5,10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]
```

So far, this looks very similar to indexing on a series. But remember, a data frame is two-dimensional. We have been passing in a *row indexer*, but we can also pass in a *column indexer*. You put the column indexer after the row indexer following a comma.

Here, we will pull out the second column (index position 1). Because we are using a scalar for the column indexer, it will return a series:

```
>>> pres.iloc[[0, 5, 10], 1]
1     George Wash...
6     John Quincy...
11     James K. Polk
Name: President, dtype: string[pyarrow]
```

If we want to get a dataframe as a result (even if it only has one column), we need to pass in a list for the column indexer:

```
>>> print(pres.iloc[[0, 5, 10], [1]])
         President
1   George Wash...
6   John Quincy...
11   James K. Polk
```

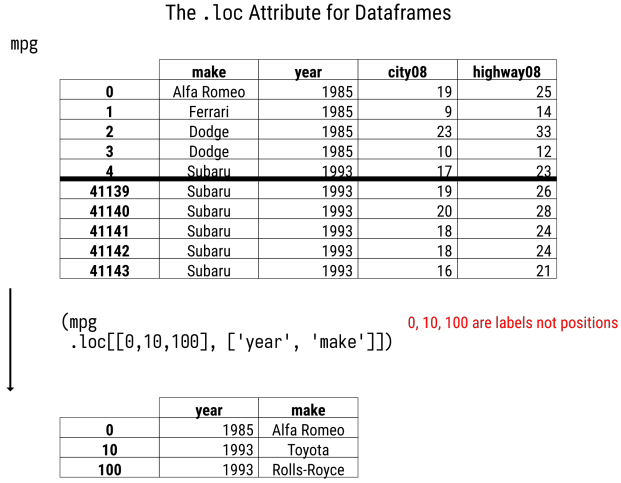
We can also pass a list of columns or a slice to the column indexer. If we want to include all rows but just filter columns, pass in `:` as the row indexer to select all rows:

```
>>> print(pres.iloc[:, [1, 2]])
         President           Party
1   George Wash...     Independent
2       John Adams      Federalist
3   Thomas Jeff...  Democratic-...
4    James Madison  Democratic-...
..             ...             ...
41    Bill Clinton      Democratic
42  George W. Bush      Republican
43    Barack Obama      Democratic
44    Donald Trump      Republican

[44 rows x 2 columns]

>>> print(pres.iloc[:, 1:3])
         President           Party
1   George Wash...     Independent
2       John Adams      Federalist
3   Thomas Jeff...  Democratic-...
4    James Madison  Democratic-...
..             ...             ...
41    Bill Clinton      Democratic
42  George W. Bush      Republican
43    Barack Obama      Democratic
44    Donald Trump      Republican
```

```
[44 rows x 2 columns]
```

26.5 Indexing by Name

Let's explore indexing by the name of index entries on a dataframe. This is done by indexing on `.loc`. If you are confused between `.loc` and `.iloc`, remember that `.iloc` indexes on position and that computer programs generally use the variable `i` to represent an index position.



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	year	make
0	1985	Alfa Romeo
10	1993	Toyota
100	1993	Rolls-Royce

Figure 26.3: Selecting rows and columns by name. You can pass in a list of index names and column names. Note that 0, 10, and 100 are the names, not the positions of the rows.

One thing to be aware of is the difference between `.iloc` and `.loc` when dealing with integer indexes. In particular, slicing has different behavior. Slicing with `.iloc` follows the half-open interval (includes the first index but not the last). Slicing with `.loc` follows the closed interval (consists of both the start and end index). (I know we mentioned this in the series chapter, but it bears repeating because it can be confusing).

In the following example, I will try to slice off index names from *1* through *5*. Because I'm using `.loc`, this will match the names. However, the index is not an integer index, so this fails (we set the *Seq* column to the index, and it had the entry "22/24", causing pandas to leave it as a string):

```
>>> pres.loc[1:5]
Traceback (most recent call last):
```

```
  ...
TypeError: cannot do slice indexing on Index with these
indexers [1] of type int
```

Let's try it again with strings:

```
>>> print(pres.loc['1':'5'])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
2    2      John Adams      Federalist  ...       14            13
3    3  Thomas Jeff...  Democratic-...  ...        5             5
4    4   James Madison  Democratic-...  ...        7             7
..  ..             ...             ...  ...      ...           ...
41  42    Bill Clinton      Democratic  ...       15            15
42  43  George W. Bush      Republican  ...       33            33
43  44    Barack Obama      Democratic  ...       17            17
44  45    Donald Trump      Republican  ...       42            42

    Quartile
1        1st
2        2nd
3        1st
4        1st
..       ...
41       2nd
42       3rd
43       2nd
44       4th

[44 rows x 26 columns]
```

Note that the slicing behavior with strings is doing a lexicographical comparison. Because all of the integers would be less than or equal to `'5'`, this returns all the rows. (I personally wish that pandas didn't allow indexing with strings when the index entries are numeric. I wish it threw an exception. However, this will be convenient when slicing date indexes.)

Note the difference when we use integers in the slice:

```
>>> print(pres.loc[1:5])
  Seq       President           Party  ...  Overall  Average_rank  \
1   1  George Wash...     Independent  ...        1             1
2   2      John Adams      Federalist  ...       14            13
3   3  Thomas Jeff...  Democratic-...  ...        5             5
4   4   James Madison  Democratic-...  ...        7             7
5   5    James Monroe  Democratic-...  ...        8             8
```

```
   Quartile
1       1st
2       2nd
3       1st
4       1st
5       1st

[5 rows x 26 columns]
```

Contrast this with positional slicing. This will return the four rows starting at the second position (by position and ignoring the names):

```
>>> print(pres.iloc[1:5])
  Seq       President           Party  ...  Overall  Average_rank  \
2   2      John Adams      Federalist  ...       14            13
3   3  Thomas Jeff...  Democratic-...  ...        5             5
4   4   James Madison  Democratic-...  ...        7             7
5   5    James Monroe  Democratic-...  ...        8             8

   Quartile
2       2nd
3       1st
4       1st
5       1st

[4 rows x 26 columns]
```

Let's shift gears for a bit and look at a dataframe with string entries in the columns. I'm going to stick the political party into the index and then pull out all of the Whig entries:

```
>>> print(pres
...   .set_index('Party')
...   .loc['Whig']
... )
      Seq       President  Background  ...  Overall  Average_rank  \
Party                                  ...
Whig    9  William Hen...          22  ...       39            38
Whig   12  Zachary Taylor          30  ...       30            30
Whig   13  Millard Fil...          40  ...       38            39

       Quartile
Party
Whig        4th
Whig        3rd
Whig        4th
```

```
[3 rows x 25 columns]
```

Note that this returns a dataframe, even though we used a scalar value for the index name. In fact, it returns the same result if we pass in a list:

```
>>> print(pres
...   .set_index('Party')
...   .loc[['Whig']]
... )
      Seq       President  Background  ...  Overall  Average_rank  \
Party                                  ...
Whig    9  William Hen...          22  ...       39            38
Whig   12  Zachary Taylor          30  ...       30            30
Whig   13  Millard Fil...          40  ...       38            39

       Quartile
Party
Whig        4th
Whig        3rd
Whig        4th

[3 rows x 25 columns]
```

This is because there are multiple entries for *Whig*. This is one of those areas to tread with caution. For example, the *Federalist* party only has one entry. So if you index with that name, you get back a series if you use a list:

```
>>> (pres
...   .set_index('Party')
...   .loc['Federalist']
... )
Seq                     2
President        John Adams
Background               3
Imagination             13
                       ...
Experts'_view           10
Overall                 14
Average_rank            13
Quartile               2nd
Name: Federalist, Length: 25, dtype: object

>>> print(pres
...   .set_index('Party')
...   .loc[['Federalist']]
```

```
... )
           Seq   President  Background  ...  Overall  Average_rank  \
Party                                    ...
Federalist   2  John Adams           3  ...       14            13

           Quartile
Party
Federalist      2nd

[1 rows x 25 columns]
```

One more thing is slicing with string indexes. Two things to remember:

- Sort the index if you want to slice it.
- You can slice with partial values.

If you don't sort the index before slicing it, you will get an error:

```
>>> (pres
...    .set_index('Party')
...    .loc['Democratic':'Independent']
... )
Traceback (most recent call last):
  ...
KeyError: "Cannot get left slice bound for non-unique label: 'Democratic'"
```

If you sort the index, you will get results:

```
>>> print(pres
...    .set_index('Party')
...    .sort_index()
...    .loc['Democratic':'Independent']
... )
                  Seq       President  Background  ...  Overall  \
Party                                                ...
Democratic      22/24  Grover Clev...          26  ...       23
Democratic         32  Franklin D....           6  ...        2
Democratic         17  Andrew Johnson          42  ...       44
Democratic         33  Harry S. Tr...          31  ...        9
...               ...             ...         ...  ...      ...
Democratic-R...     3  Thomas Jeff...           2  ...        5
Federalist          2      John Adams           3  ...       14
Independent         1  George Wash...           7  ...        1
Independent        10      John Tyler          34  ...       37

                Average_rank  Quartile
Party
```

```
Democratic                23       3rd
Democratic                 2       1st
Democratic                44       4th
Democratic                 9       1st
...                      ...       ...
Democratic-R...            5       1st
Federalist                13       2nd
Independent                1       1st
Independent               37       4th

[22 rows x 25 columns]
```

Note that you can also use partial strings on sorted indexes:

```
>>> print(pres
...   .set_index('President')
...   .sort_index()
...   .loc['C':'Thomas Jefferson', 'Party':'Integrity']
... )
                         Party  Background  Imagination  Integrity
President
Calvin Coolidge      Republican          32           36         17
Chester A. A...      Republican          41           31         37
Donald Trump         Republican          43           40         44
Dwight D. Ei...      Republican          11           18          5
...                         ...         ...          ...        ...
Ronald Reagan        Republican          27           17         24
Rutherford B...      Republican          35           30         32
Theodore Roo...      Republican           5            4          8
Thomas Jeffe...  Democratic-...           2            2         14

[31 rows x 4 columns]
```

You cannot use partial strings on categorical indexes:

```
>>> (pres
...    .set_index('Party')
...    .sort_index()
...    .loc['D':'J']
... )
Traceback (most recent call last):
  ...
KeyError: 'D'
```

If you convert the categorical index to a string index, then you can use partial strings:

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> print(pres
...   .assign(Party=pres.Party.astype(string_pa))
...   .set_index('Party')
...   .sort_index()
...   .loc['D':'J']
... )
                Seq       President  Background  ...  Overall  \
Party                                             ...
Democratic        7  Andrew Jackson          37  ...       19
Democratic        8  Martin Van ...          23  ...       25
Democratic       11   James K. Polk          19  ...       12
Democratic       14  Franklin Pi...          38  ...       40
...              ..             ...         ...  ...      ...
Democratic-R...   6  John Quincy...           1  ...       18
Federalist        2      John Adams           3  ...       14
Independent       1  George Wash...           7  ...        1
Independent      10      John Tyler          34  ...       37

                Average_rank  Quartile
Party
Democratic                19       2nd
Democratic                25       3rd
Democratic                11       1st
Democratic                41       4th
...                      ...       ...
Democratic-R...           18       2nd
Federalist                13       2nd
Independent                1       1st
Independent               37       4th

[22 rows x 25 columns]
```

You can also slice columns (if you sort the columns first):

```
>>> print(pres
...   .set_index('President')
...   .sort_index()
...   .sort_index(axis='columns')
...   .loc['C':'Thomas Jefferson', 'B':'D']
... )
                Background  Communication_ability  \
President
Calvin Coolidge         32                     37
Chester A. A...         41                     36
```

```
Donald Trump             43             43
Dwight D. Ei...          11             20
...                     ...            ...
Ronald Reagan            27              6
Rutherford B...          35             30
Theodore Roo...           5              5
Thomas Jeffe...           2              4

                Court_appointments
President
Calvin Coolidge                 31
Chester A. A...                 33
Donald Trump                    40
Dwight D. Ei...                  5
...                            ...
Ronald Reagan                   18
Rutherford B...                 27
Theodore Roo...                  9
Thomas Jeffe...                  7

[31 rows x 3 columns]
```

26.6 Filtering with Functions & .loc

You should know that you can pass in a boolean array and a function into `.loc`. Here, I select rows with *Average_rank* less than ten and the first three columns:

```
>>> print(pres
...  .loc[pres.Average_rank < 10, lambda df_: df_.columns[:3]]
... )
   Seq       President           Party
1    1  George Wash...     Independent
3    3  Thomas Jeff...  Democratic-...
4    4   James Madison  Democratic-...
5    5    James Monroe  Democratic-...
..  ..             ...             ...
25  26  Theodore Ro...      Republican
31  32  Franklin D....      Democratic
32  33  Harry S. Tr...      Democratic
33  34  Dwight D. E...      Republican

[9 rows x 3 columns]
```

An advantage of passing a function into `.loc` is that the function will receive the current state of the dataframe. If you have `.loc` in a chain of

operations, the column names or rows might have changed, so if you filter based on the original dataframe that began the chain, you might not be able to get the data you need.

26.7 .query vs .loc

There is often more than one way to do things in pandas. You may be wondering if you should use `.query` or `.loc`.

If you do a lot of chaining (which I recommend), `.query` has the advantage of working on the intermediate dataframe. One could argue that `.loc` does as well, but often when using boolean arrays with `.loc`, users insert a boolean array based on the original data, not the intermediate data. You need to use a function with `.loc` to get access to the original dataframe.

On the flipside, `.query` does not support column selection, but `.loc` does. I don't think this is a situation where you should only learn one of these constructs and neglect the other. Learn them both and figure out which one is appropriate, given your requirements.

Table 26.1: Dataframe Filtering and Indexing Methods

Method	Description
`.rename( mapper=None, index=None, columns=None, axis=0, copy=True, level=None, errors='ignore')`	Change axis labels. Pass the `columns` or `index` as a dictionary (mapping old values to new values) or a function (accepting the old value and returning the new value).
`.reset_index( level=None, drop=False, col_level=0, col_fill='')`	Return a dataframe with the new index (or new level). To remove a level, specify that with `level` (by position or name). Position 0 is the outermost level, and it goes up. Alternatively, -1 is the innermost level. Index values are moved to columns or dropped if `drop=True`. `col_level` determines where the index label goes with multiple column levels. Other levels will get the value of `col_fill`.

Method	Description
`.set_index(keys, drop=True, append=False, verify_integrity =False)`	Return a dataframe with a new index. The `keys` argument can be a column name, a series (or numpy array) of labels for the index, or a list of column names or series. The `drop` parameter indicates whether to remove columns used for the index. The `append` parameter allows you to add additional index levels. You can check for duplicate index values by setting `verify_integrity=True`.
`.sort_index( axis=0, level=None, ascending=True, kind='quicksort', na_position= 'last', sort_remaining= True, ignore_index= False, key=None))`	Return a dataframe with the index (`axis=0`) or columns (`axis=1`) sorted. Can specify a single level or multiple levels with `levels`. Can specify the direction of each level sort with `ascending`. Choose the axis (default is axis 0). Use `by` to specify a column (string) or a list of columns (for `axis=0`). Can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. Can reset the index with `ignore_index`. A `key` function accepts an index and should return an index. For multi-level indexes, each index is passed in independently to the function.
`.query(expr)`	Evaluate `expr` to filter the dataframe. Refer to variables by prefixing them with `@`. Use backticks around the column names with spaces.
`.iloc`	Attribute to index off of by index and column positions. Slices use the half-open interval (including start but not end).
`.loc`	Attribute to index off of by index and column names. Slices use the closed interval (including start and end).

26.8 Summary

In this chapter, we explored renaming the index. Then we saw how you can pull out rows and columns based on names or positions.

26.9 Exercises

With a dataset of your choice:

1. Pull out the first two rows by name.

2. Pull out the first two rows by position.
3. Pull out the last two columns by name.
4. Pull out the last two columns by position.

Chapter 27

Plotting with Dataframes

One feature I like about pandas is its integration with Matplotlib. This integration makes it easy to create various plots if you understand what type of plot you want. In this chapter, we will explore the built-in plotting capabilities of pandas.

27.1 Lines Plots

The dataframe has a `.plot` attribute that you can use to plot. Line plots are easy to create. Remember that pandas will plot the index on the x-axis, and each column will be its own line. Here is a default plot. It is a little hard to process, but along the x-axis is the president (from the first to the last). Each line represents what happens to the score from one president to the next president:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0)
>>> pres = tweak_siena_pres(df)

>>> pres.plot().legend(bbox_to_anchor=(1,1))

<matplotlib.legend.Legend at 0x107be0810>
<Figure size 640x480 with 1 Axes>
```

Let's make another line plot that is more involved. Each line will track the scores for a single president. If we want each line to be a president, then each column needs to represent the president's data.

I'll show you how I will build this up. Let's chain up the operations. We will need to put the president's name in the index:

```
>>> print(pres
...  .set_index('President')
```

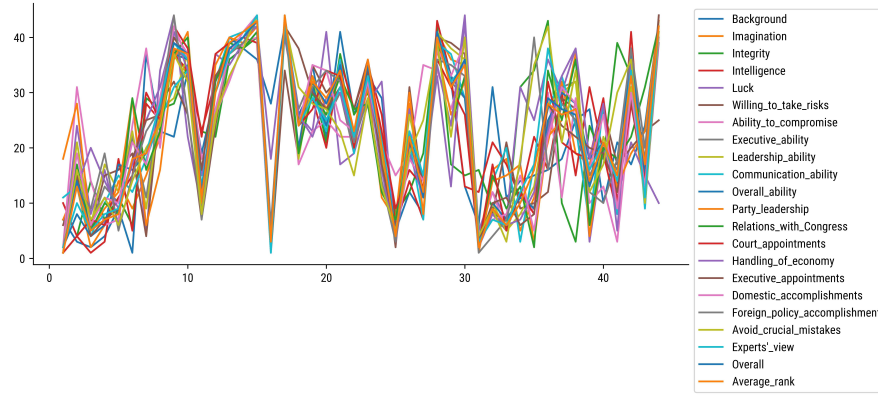


Figure 27.1: A line for each category, showing how it changed from president to president.

```
... )
                Seq           Party  Background  ...  Overall  \
President                                        ...
George Washi...   1     Independent           7  ...        1
John Adams        2      Federalist           3  ...       14
Thomas Jeffe...   3  Democratic-...           2  ...        5
James Madison     4  Democratic-...           4  ...        7
...              ..             ...         ...  ...      ...
Bill Clinton     42      Democratic          21  ...       15
George W. Bush   43      Republican          17  ...       33
Barack Obama     44      Democratic          24  ...       17
Donald Trump     45      Republican          43  ...       42

                Average_rank  Quartile
President
George Washi...            1       1st
John Adams                13       2nd
Thomas Jeffe...            5       1st
James Madison              7       1st
...                      ...       ...
Bill Clinton              15       2nd
George W. Bush            33       3rd
Barack Obama              17       2nd
Donald Trump              42       4th

[44 rows x 25 columns]
```

The `.plot` Method

`mpg`

	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

```
(mpg
 .groupby('year')
 .mean()
 .plot())
```

Plots each column against the index!

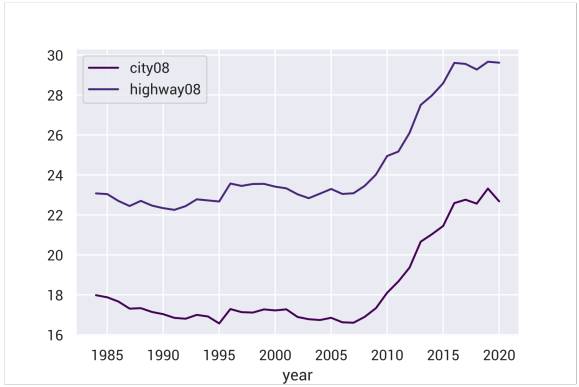


Figure 27.2: You can also call the `.plot` attribute. By default, it will create a line plot, plotting each numeric column against the index. The `kind` attribute specifies the type of plot. Rather than using `kind`, I recommend using the specific plot type attribute.

Next, we will filter out the columns we want (we will also remove every other president to give the plot some breathing room):

```
>>> print(pres
...   .set_index('President')
...   .loc[::2,'Background':'Overall']
... )
                Background  Imagination  Integrity  ...  \
President                                           ...
George Washi...          7            7          1  ...
Thomas Jeffe...          2            2         14  ...
James Monroe             9           14         11  ...
Andrew Jackson          37           15         29  ...
```

```
...                    ...          ...        ...  ...
Gerald Ford             18           32         10  ...
Ronald Reagan           27           17         24  ...
Bill Clinton            21           12         39  ...
Barack Obama            24           11         13  ...

               Avoid_crucial_mistakes  Experts'_view  Overall
President
George Washi...                     1              2        1
Thomas Jeffe...                     7              5        5
James Monroe                        6              9        8
Andrew Jackson                     20             19       19
...                               ...            ...      ...
Gerald Ford                        21             27       27
Ronald Reagan                      12             16       13
Bill Clinton                       30             14       15
Barack Obama                       10             11       17

[22 rows x 21 columns]
```

Next, let's transpose the result to flip the rows and columns with `.T`:

```
>>> print(pres
...   .set_index('President')
...   .loc[::2,'Background':'Overall']
...   .T
... )
President       George Washington  Thomas Jefferson  James Monroe  \
Background                      7                 2             9
Imagination                     7                 2            14
Integrity                       1                14            11
Intelligence                   10                 1            18
...                           ...               ...           ...
Foreign_poli...                 2                 9             5
Avoid_crucia...                 1                 7             6
Experts'_view                   2                 5             9
Overall                         1                 5             8

President       ...  Ronald Reagan  Bill Clinton  Barack Obama
Background      ...             27            21            24
Imagination     ...             17            12            11
Integrity       ...             24            39            13
Intelligence    ...             31             8             9
...             ...            ...           ...           ...
Foreign_poli... ...             12            18            20
Avoid_crucia... ...             12            30            10
```

```
Experts'_view    ...             16             14             11
Overall          ...             13             15             17

[21 rows x 22 columns]
```

This data looks good. Each column will be its own line. Let's plot it:

```
>>> (pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
...  .plot()
... )
<Axes: >
<Figure size 640x480 with 1 Axes>
```

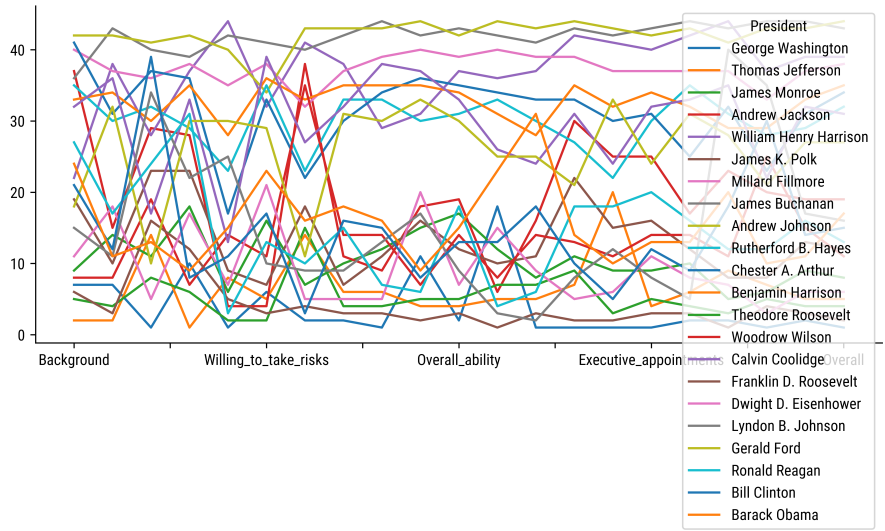


Figure 27.3: A basic line plot for each president.

This is a good start, but we can make it better. Let's clean the plot up. Because pandas leverages Matplotlib, I will use some of that library:

- Label every attribute
- Rotate the attribute labels
- Move the legend
- Add a label to the y-axis

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))

>>> (pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
...  .plot(ax=ax, rot=45).legend(bbox_to_anchor=(1,1))
... )
>>> ax.set_xticks(range(21))
>>> ax.set_xticklabels(pres
...   .loc[:,'Background':'Overall'].columns, ha='right')
>>> ax.set_ylabel('Rank')
Text(0, 0.5, 'Rank')
<Figure size 6000x2400 with 1 Axes>
```

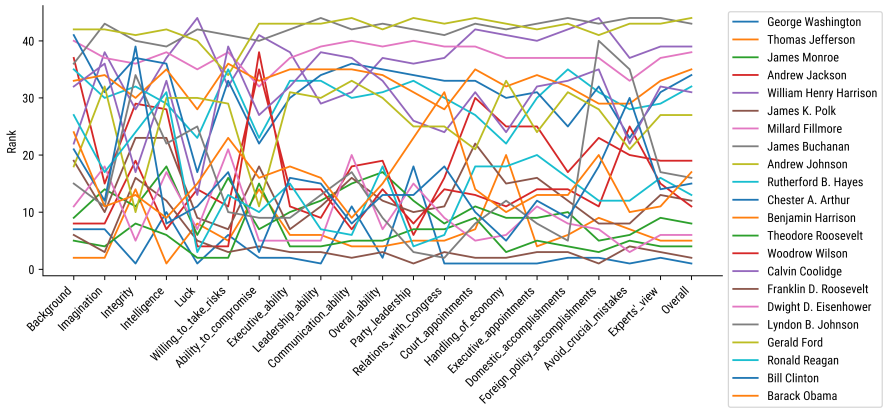


Figure 27.4: A cleaned-up line plot for each president.

This is still a little hard to read. Generally, we want to pull attention to a single line. Let's highlight Washington. A trick that visualization experts use is to mute the other colors. I will use the `.pipe` method to create a `colors` list to indicate the colors for each line:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))
>>> colors = []
>>> def set_colors(df):
...     for col in df.columns:
...         if 'George' in col:
...             colors.append('#990000')
...         else:
```

```
...             colors.append('#999999')
...     return df

>>> (pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
...  .pipe(set_colors)
...  .plot(ax=ax, rot=45, color=colors)
...  .legend(bbox_to_anchor=(1,1), ncols=2)
... )
>>> ax.set_xticks(range(21))
>>> ax.set_xticklabels(pres
...   .loc[:,'Background':'Overall'].columns, ha='right')
>>> ax.set_ylabel('Rank')
Text(0, 0.5, 'Rank')
<Figure size 6000x2400 with 1 Axes>
```

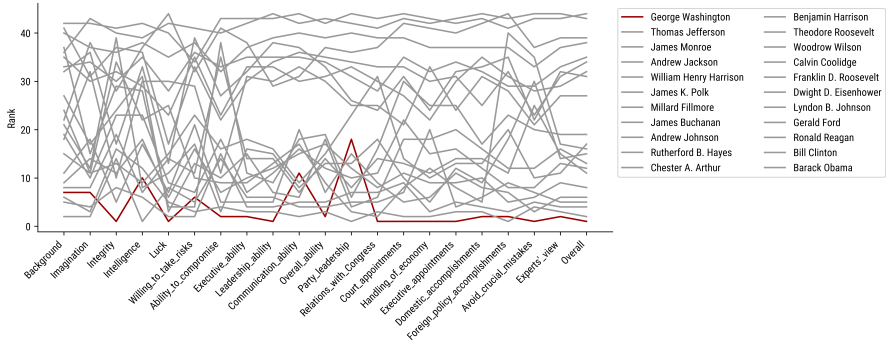


Figure 27.5: A cleaned-up line plot for each president highlighting George Washington.

27.2 Bar Plots

Let's make a bar plot comparing four attributes for each president. Again remember that pandas will plot the index on the x-axis. Here's the data:

```
>>> print(pres
...  .set_index('President')
...  .iloc[:, -5:-1]
... )
                Avoid_crucial_mistakes  Experts'_view  Overall  \
```

```
President
George Washi...                1                    2       1
John Adams                    16                   10      14
Thomas Jeffe...                7                    5       5
James Madison                 11                    8       7
...                           ...                  ...     ...
Bill Clinton                  30                   14      15
George W. Bush                36                   34      33
Barack Obama                  10                   11      17
Donald Trump                  41                   42      42

                Average_rank
President
George Washi...             1
John Adams                 13
Thomas Jeffe...             5
James Madison               7
...                       ...
Bill Clinton               15
George W. Bush             33
Barack Obama               17
Donald Trump               42

[44 rows x 4 columns]
```

Here's the plot. Each value will be its own bar above the president label:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))
>>> (pres
...  .set_index('President')
...  .iloc[:, -5:-1]
...  .plot.bar(rot=45, ax=ax)
... )
>>> ax.set_xticklabels(labels=ax.get_xticklabels(), ha='right')
>>> ax.legend(bbox_to_anchor=(1,1))
<matplotlib.legend.Legend at 0x157ed3f90>
<Figure size 6000x2400 with 1 Axes>
```

Often, it is easier to read a *horizontal bar plot*. We don't need to turn our heads sideways to read the labels. By changing `.bar` to `.barh` we create a horizontal bar plot:

```
>>> ax = (pres
...  .set_index('President')
...  .iloc[:, -5:]
...  .plot.barh(figsize=(4,12))
```

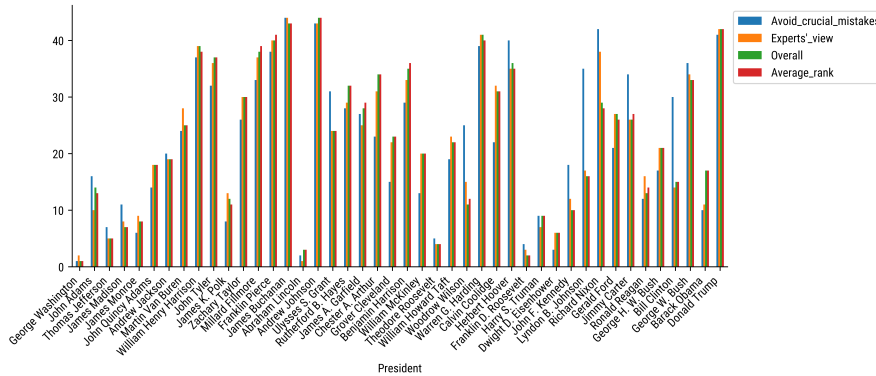


Figure 27.6: Bar plot for 4 attributes.

```
...  .legend(bbox_to_anchor=(1,1))
... )
<Figure size 400x1200 with 1 Axes>
```

27.3 Scatter Plots

A scatter plot helps determine the relationship between two numeric columns. We can evaluate what tends to happen to one value as the other value changes. I am going to use meteorological data from a ski resort named Alta.

```
>>> url = 'https://github.com/mattharrison/'\
...  'datasets/raw/master/data/alta-noaa-1980-2019.csv'
>>> alta = (pd.read_csv(url, parse_dates=['DATE'], dtype_backend='pyarrow')
...         .loc[:, ['DATE', 'PRCP', 'SNOW', 'SNWD', 'TMAX', 'TMIN']]
... )
```

```
>>> print(alta)
            DATE  PRCP  SNOW  SNWD  TMAX  TMIN
0     1980-01-01   0.1   2.0  29.0    38    25
1     1980-01-02  0.43   3.0  34.0    27    18
2     1980-01-03  0.09   1.0  30.0    27    12
3     1980-01-04   0.0   0.0  30.0    31    18
...          ...   ...   ...   ...   ...   ...
14156 2019-09-04   0.0   0.0   0.0    77    52
14157 2019-09-05   0.0   0.0   0.0    76    54
14158 2019-09-06  0.07   0.0   0.0    66    52
14159 2019-09-07   0.0   0.0   0.0    68    45

[14160 rows x 6 columns]
```

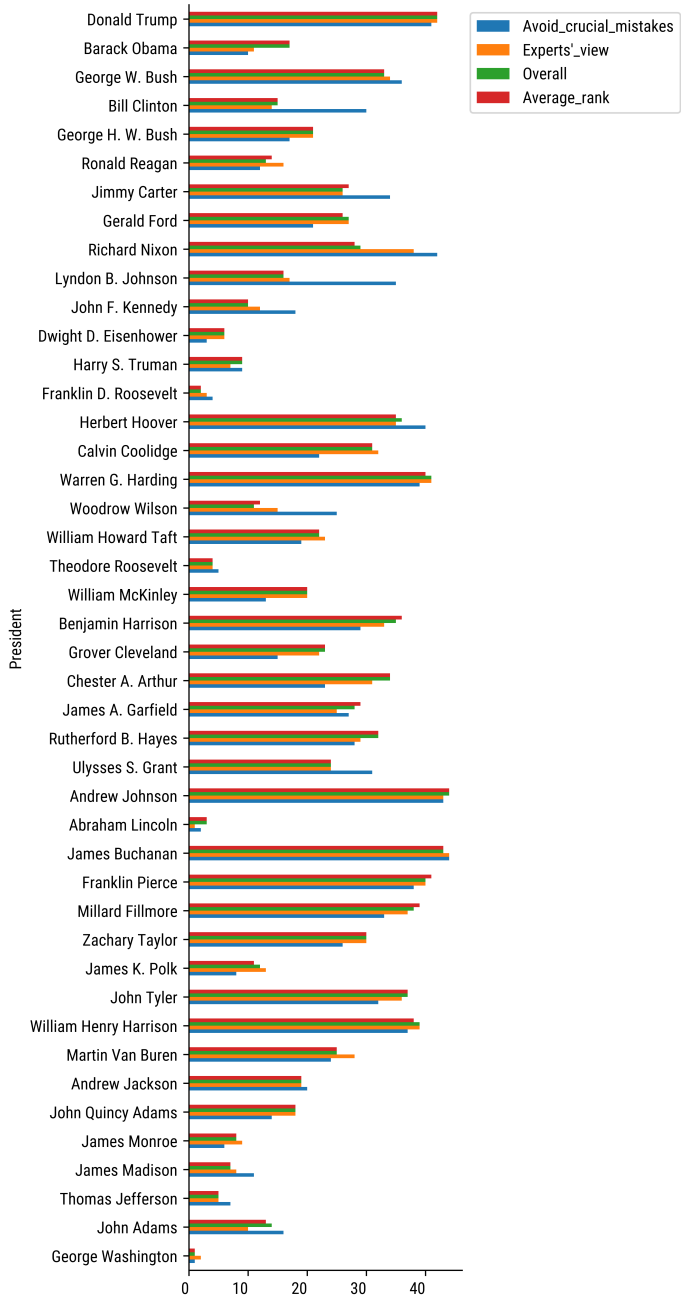


Figure 27.7: Horizontal bar plot for 4 attributes.

The `.plot.barh` Method

`mpg`

	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

Plots each column as a bar!

```
def topn(ser, n=5):
    vals = ser.value_counts().index[:n]
    return ser.where(ser.isin(vals), 'Other')

(mpg
 .make
 .pipe(topn)
 .value_counts()
 .plot.barh())
```

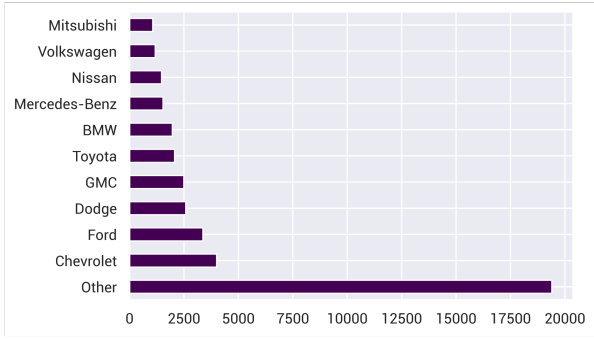


Figure 27.8: The `.plot.barh` method will plot each column as a bar plot. Because it is a horizontal bar plot, it will place the index in the y-axis.

Let's look at a plot to compare the precipitation and snowfall. Note that precipitation is the total inches of water that fell, and snowfall is the total inches of snow that fell. There is a high correlation between the two. When it rains, it tends to snow. However, the correlation is not strictly linear because one inch of fluffy snow has less water in it than an inch of wet, heavy snow.

```
>>> alta.SNOW.corr(alta.PRCP)
0.7639964347046551
```

Let's see what the scatter plot looks like:

```
>>> alta.plot.scatter(x='PRCP', y='SNOW')
<Axes: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 1 Axes>
```

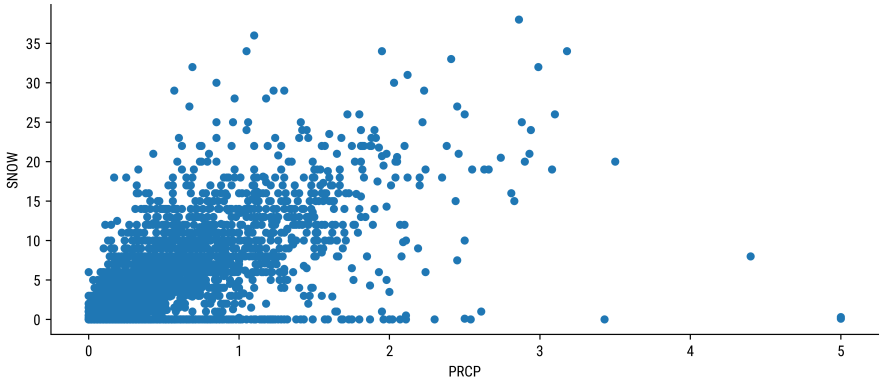


Figure 27.9: Scatter plot for precipiation and snowfall.

A couple of things to note. At the bottom, you see a line. This is when it rains but doesn't snow. During the summer months, it rains, but it is not cold enough to snow.

Also, you can see a high concentration of points in the lower left corner. I want to know where the values overlap, but because that opacity is completely opaque, I can't see the individual points. Also, I can see that values line up on a grid in both the x and y directions. In the real world, rain doesn't fall in whole-inch increments. This is an artifact of rounding the data to the nearest inch. This makes sense now that we look at the plot, but I probably wouldn't have realized that the data is rounded to the nearest inch.

I have a few techniques to deal with concentrated densities of data:

- Adjust the transparency (`alpha` attribute)
- Sample the data
- Change the scale of the axis to a log scale
- Change the size of the dots

My first lever to pull is to adjust the transparency. I will keep lowering the transparency until I can see a gradient from the lightest to the darkest. If I only see dark, then I need to keep reducing the transparency or use another technique.

Let's try lowering the transparency by setting the `alpha` attribute to `0.1`. Each point will be 10% opaque and 90% transparent. This will allow us to see the density of the data:

```
alta.plot.scatter(x='PRCP', y='SNOW', alpha=.1)
```

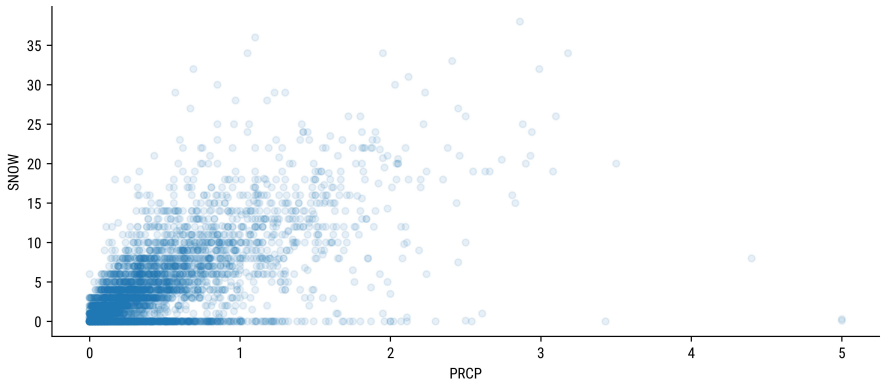


Figure 27.10: Reducing the transparency of the scatter plot

This lets us clearly see that there are many times when it rains less than 1 inch and does not snow. We can also see that there are many times when the precipitation is less than an inch but is cold enough that it falls in the form of snow.

Because we have skewed data, we could also try a log scale. This will spread out the data. Let's try it by passing `logx=True` to the `.plot.scatter` method:

```
>>> alta.plot.scatter(x='PRCP', y='SNOW', alpha=.3, logx=True, logy=True)
<Axes: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 1 Axes>
```

Because many of the values are 0 and the log of 0 is undefined, we will add 1 to each value. This will make the log of 0 equal to 0. We will also add 1 to the y-axis so that the log of 0 is defined for both axes. I will also color by the temperature. I use the *coolwarm* diverging colormap to distinguish between cold and warm temperatures.

```
>>> (alta
...    .assign(PRCP=lambda df: df.PRCP + 1,
...            SNOW=lambda df: df.SNOW + 1)
...    .plot.scatter(x='PRCP', y='SNOW', alpha=.7, logx=True, logy=True,
...                  c='TMAX', cmap='coolwarm')
... )
<Axes: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```

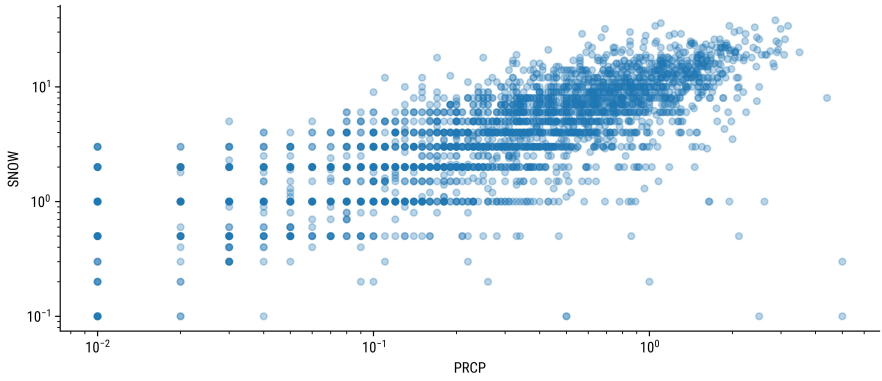


Figure 27.11: Broken plot with log-x and log-y axis.

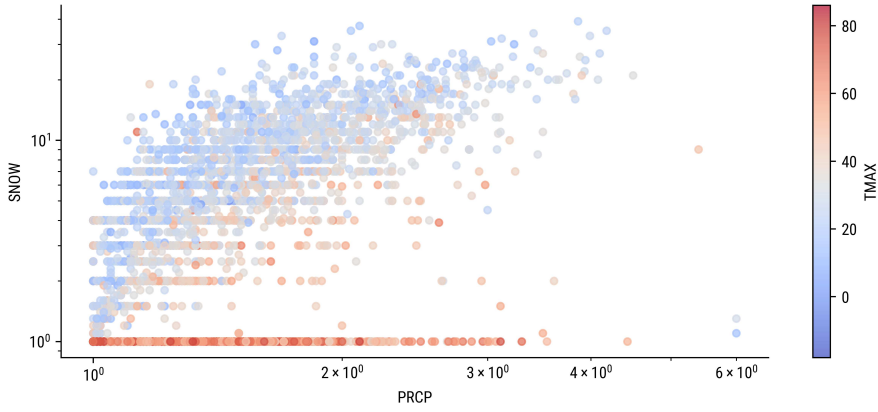


Figure 27.12: Plot showing relationship between temperature and snowfall on a log scale

I like this plot and think it tells a different story than the first scatter plot. It shows that when it is cold, it tends to snow more. It also shows that it rarely snows when the temperature is above 40 degrees. Finally, it shows that the snow is lighter or fluffier when it is colder. You can see this by noticing the left side of the points are blue. The blue color indicates that the temperature is cold and there is more snow per inch of water.

27.4 Jittering Data

When your scatter plots appear to fall into grids, you can add some random noise to the data. This is called *jittering*. It will spread out the data and make it easier to see the density of the data. I'll make the size of the points smaller to make it easier to see the jittering. I will also zoom in on the data to see the jittering.

Here's the before. Notice that many of the *SNOW* measurements are at the whole inch granularity, but some are at a smaller granularity.

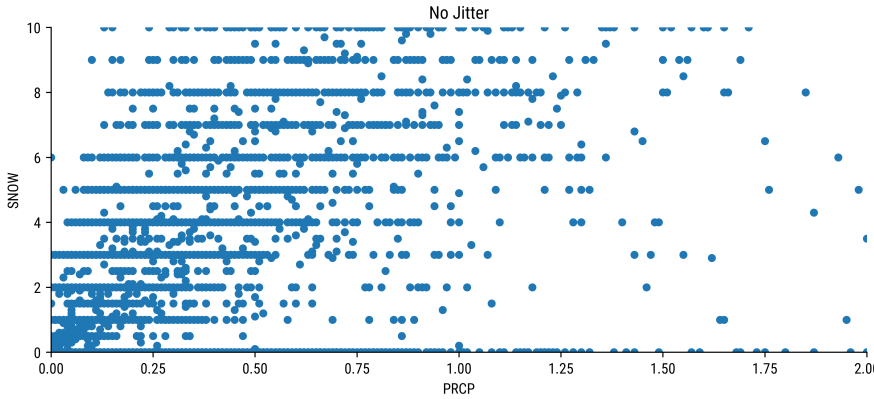


Figure 27.13: Scatter plot before jittering. Notice the horizontal lines.

Here's the code with the jittering.

```
import numpy as np
def jitter(df, column, scale=1):
    rands = np.random.random(len(df))
    return df[column] + (rands-.5) * scale
```

Now, I will apply the jittering to the *SNOW* column where it is not 0. This should eliminate the horizontal banding we saw earlier (except when it is raining).

```
(alta
    .assign(SNOW=lambda df: df
```

```
        .SNOW.where(df.SNOW == 0, jitter(df, 'SNOW').clip(lower=0)))
    .plot.scatter(x='PRCP', y='SNOW', alpha=1, title='Jitter', ax=ax)
)
```

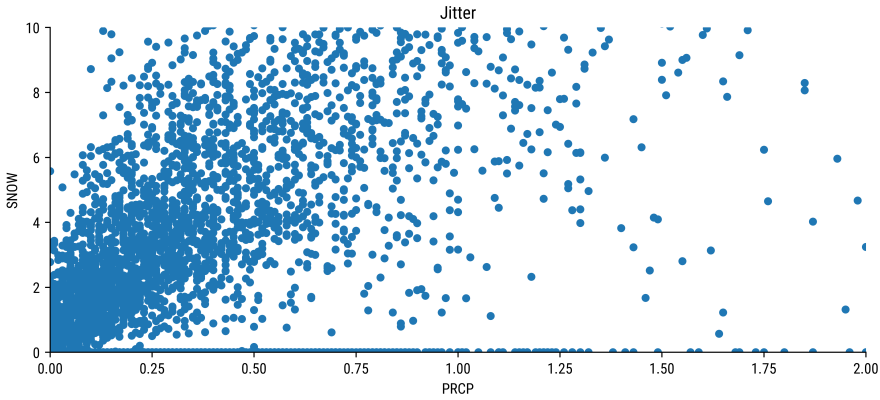


Figure 27.14: Scatter plot with jittering.

27.5 Correlation Heatmap

While not strictly a plot, a correlation heatmap is a great way to visualize the correlation between all the columns in a DataFrame. I combine this with a scatter plot to determine which columns to plot. I frequently see these in social media, which is a pet peeve of mine because they are often misused. The key to a heatmap is to see the correlation between columns quickly. If you color it incorrectly, it is hard to see the correlation. The key is to use the appropriate color map and to center the color map at 0. Centering the color map at 0 requires us to set the minimum and maximum values to -1 and 1.

Here's the correlation heatmap for the Alta data:

```
(alta
  .corr()
  .style
  .background_gradient(cmap='RdBu', vmin=-1, vmax=1)
)
```

The `.corr` method will compute the Pearson correlation coefficient by default. This is a measure of the linear relationship between two variables. It is a number between -1 and 1. A value of 1 indicates a perfect positive linear relationship. As one variable increases, the other variable increases. A value of -1 indicates a perfect negative linear relationship. As one variable

	DATE	PRCP	SNOW	SNWD	TMAX	TMIN
DATE	1.000000	-0.035043	-0.048535	-0.046148	0.050617	0.023996
PRCP	-0.035043	1.000000	0.763996	0.233244	-0.289254	-0.208226
SNOW	-0.048535	0.763996	1.000000	0.357771	-0.428575	-0.352145
SNWD	-0.046148	0.233244	0.357771	1.000000	-0.652343	-0.626040
TMAX	0.050617	-0.289254	-0.428575	-0.652343	1.000000	0.932398
TMIN	0.023996	-0.208226	-0.352145	-0.626040	0.932398	1.000000

Figure 27.15: A correlation heatmap that uses a color map appropriate for correlation. It is also centered at 0.

increases, the other variable decreases. A value of 0 indicates no linear relationship.

Often, we have a relationship that is not linear. For example, the relationship between age and height is not linear. As you get older, you tend to get taller, but there is a limit to how tall you can get. We can use the Spearman correlation coefficient (by calling `.corr(method='spearman')`) to measure the monotonic relationship between two variables. A monotonic relationship is one where as one variable increases, the other variable either increases or decreases. It is also a number between -1 and 1. A value of 1 indicates a perfect monotonic relationship. As one variable increases, the other variable increases. With the *alta* data, measuring the Spearman correlation coefficient is not too different than measuring the Pearson correlation coefficient.

I'll include an example of a bad correlation heatmap so that you can see it and do your part to prevent my eyes from seeing it again. The code is similar, it uses the `.background_gradient` method to color the cells. The difference is that it uses the *viridis* color map. This color map is a continuous color map that shows how values change. In a heatmap, we want to use a diverging color map. This color map has a neutral color in the middle and two contrasting colors on the ends. The *RdBu* color map is a good choice because it has a neutral white in the middle and a dark blue and red on the ends.

Using a diverging color map is not enough. We also need to center the color map at 0. This means that the color around zero should be the neutral color. The `.background_gradient` method allows us to set the minimum and maximum values for the color map. We will set the minimum to -1 and the maximum to 1. This will center the color map at 0.

A later chapter will cover the `.style` attribute in more detail.

```
>>> (alta
...  .corr()
...  .style
```

	DATE	PRCP	SNOW	SNWD	TMAX	TMIN
DATE	1.000000	-0.035043	-0.048535	-0.046148	0.050617	0.023996
PRCP	-0.035043	1.000000	0.763996	0.233244	-0.289254	-0.208226
SNOW	-0.048535	0.763996	1.000000	0.357771	-0.428575	-0.352145
SNWD	-0.046148	0.233244	0.357771	1.000000	-0.652343	-0.626040
TMAX	0.050617	-0.289254	-0.428575	-0.652343	1.000000	0.932398
TMIN	0.023996	-0.208226	-0.352145	-0.626040	0.932398	1.000000

Figure 27.16: A poor correlation heatmap that uses a color map inappropriate for correlation. It is also not centered at 0. Please don't use this!

```
...   .background_gradient(cmap='viridis')
... )
<pandas.io.formats.style.Styler at 0x17ef34a50>
```

27.6 Hexbin Plots

Another mechanism to visualize relationships between two continuous values as well as density (where the values overlap), is a hexbin plot. It would be best to choose an appropriate continuous colormap that increases from white to dark for this plot. I prefer to subsample the scatter plot or adjust the alpha value to see the density better before using a hexbin plot.

Here is a hexbin plot for *PRCP* and *SNOW*:

```
>>> (alta
...   .plot.hexbin(x='PRCP', y='SNOW',
...     cmap='Greens', gridsize=30)
... )
<AxesSubplot: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```

This plot is not particularly useful. Because there is such a concentration of values when *SNOW* and *PRCP* are both 0, we can't see the density of the other values. Let's try a different plot that only shows the density of values when *SNOW* is greater than 0:

```
>>> (alta
...   .query('SNOW > 0')
...   .plot.hexbin(x='PRCP', y='SNOW',
...     cmap='Greens', gridsize=30)
... )
<AxesSubplot: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```

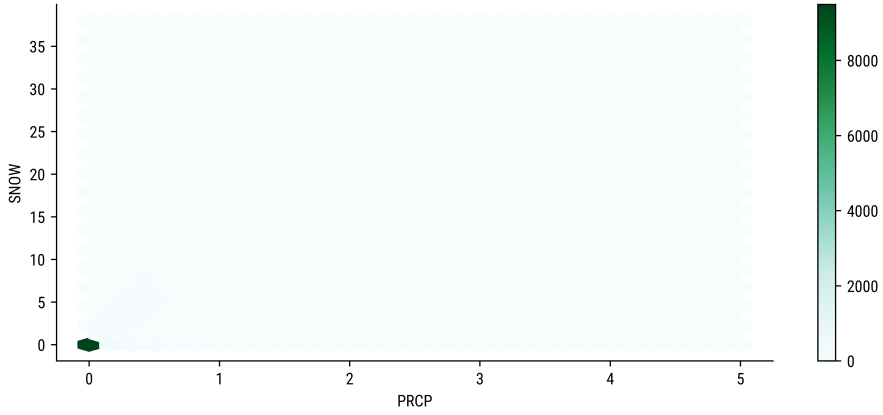


Figure 27.17: Hexbin plot for SNOW and PRCP, showing where the density of values occur.

This is a slight improvement. I rarely use a hexbin plot as I can generally get a scatter plot to tell the story in the data.

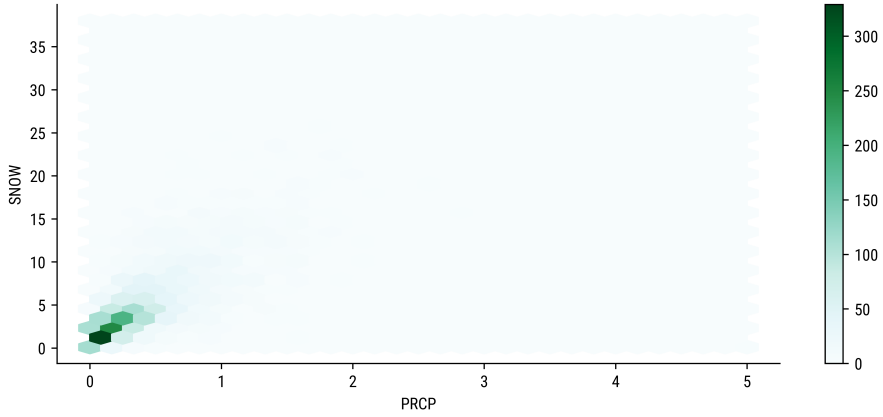


Figure 27.18: Hexbin plot for SNOW and PRCP, showing where the density of values occur. This plot ignores with the SNOW value is zero.

27.7 Area Plots and Stacked Bar Plots

A dataframe can create stacked area plots with the `.area` method. This plot is useful when you want to understand each column's relative contribution,

and the order of the data is essential. I prefer a stacked bar plot if there is no relationship and order between the values.

Below, I specify the numeric columns I want with the `y` parameter. After plotting, I adjust the number of ticks and labels:

```
>>> ax = (pres
...   .plot.area(x='President',
...     y='Background Imagination Integrity Intelligence Luck '\
...       'Willing_to_take_risks Ability_to_compromise'.split(),
...     rot=45)
... )
>>> ax.set_xticks(range(len(pres)))
>>> _ = ax.set_xticklabels(labels=pres.President, ha='right')
<Figure size 640x480 with 1 Axes>
```

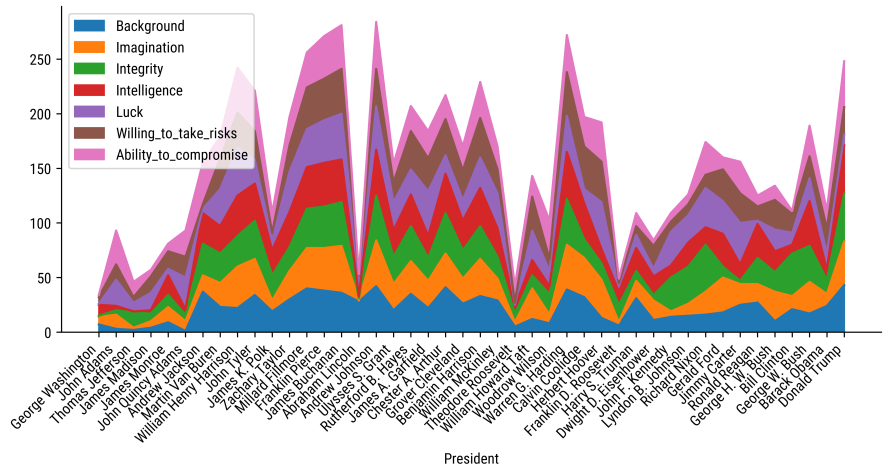


Figure 27.19: Stacked area plot.

In this case, a line plot indicates continuity from one president to the next. As presidential behavior should be somewhat independent of previous administrations, I prefer a stacked bar plot instead:

```
>>> ax = (pres
...   .plot.bar(x='President',
...     y='Background Imagination Integrity Intelligence Luck '\
...       'Willing_to_take_risks Ability_to_compromise'.split(),
...     rot=45, stacked=True, figsize=(10,4))
... )
>>> ax.set_xticks(range(len(pres)))
```

```
>>> _ = ax.set_xticklabels(labels=pres.President, ha='right')
<Figure size 1000x400 with 1 Axes>
```

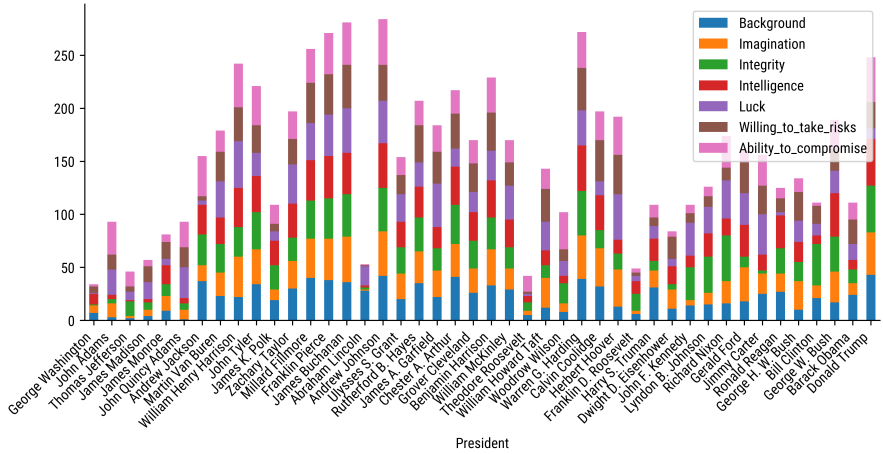


Figure 27.20: Stacked bar plot.

27.8 Column Distributions with KDEs, Histograms, and Boxplots

If you have numeric information in columns, you can run summary statistics on the columns with `.describe`. To visualize the distribution for each column, you can plot with `.hist` or `.density`.

I'm going to shuffle the presidential data around and put the president's name in the columns, with the numeric ratings in the index. I'm going to limit this to nine presidents:

```
>>> print(pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
... )
President       George Washington  John Adams  Thomas Jefferson  \
Background                      7           3                 2
Imagination                     7          13                 2
Integrity                       1           4                14
Intelligence                   10           4                 1
...                           ...         ...               ...
Avoid_crucia...                 1          16                 7
```

```
Experts'_view                2            10             5
Overall                      1            14             5
Average_rank                 1            13             5

President        ...  Andrew Jackson  Martin Van Buren  \
Background       ...              37              23
Imagination      ...              15              22
Integrity        ...              29              27
Intelligence     ...              28              25
...              ...             ...             ...
Avoid_crucia...  ...              20              24
Experts'_view    ...              19              28
Overall          ...              19              25
Average_rank     ...              19              25

President        William Henry Harrison
Background                       22
Imagination                      38
Integrity                        28
Intelligence                     37
...                             ...
Avoid_crucia...                  37
Experts'_view                    39
Overall                          39
Average_rank                     38

[22 rows x 9 columns]
```

The `.describe` method summarizes each column. In this case, the scores for each president:

```
>>> print(pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .describe()
... )
President  George Washington  John Adams  Thomas Jefferson  ...  \
count                   22.0        22.0              22.0  ...
mean                3.681818   14.454545          6.545455  ...
std                 4.444219    7.544959          4.404838  ...
min                      1.0         3.0               1.0  ...
25%                      1.0       10.75               4.25  ...
50%                      2.0        13.5               5.0  ...
75%                      5.0        18.5               7.0  ...
```

```
max                 18.0          31.0           20.0    ...

President  Andrew Jackson  Martin Van Buren  William Henry Harrison
count               22.0            22.0              22.0
mean           19.590909       25.681818         36.909091
std             9.465019        3.721064          5.485124
min                  4.0            16.0              22.0
25%                15.25           24.25             36.25
50%                 19.0            25.5              38.0
75%                 25.0           27.75             40.75
max                 38.0            34.0              44.0

[8 rows x 9 columns]
```

Let's visualize each president's scores with a Kernel Density Estimation (KDE). Remember that pandas will convert each column to a line in this plot. In this case, each line represents the values for a president's scores. The x-axis is the score, and the y-axis is the density of the values. A taller plot means the values are more concentrated around that score. A wider plot means the values are more spread out.

One thing that you need to be careful with is that the KDE plot has a wide x-axis. It stacks up a gaussian distribution for each value and sums the curves together. It shows negative values and values greater than 50. This is not possible for the presidential approval ratings.

```
>>> (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.density(figsize=(10, 4))
... )
<AxesSubplot: ylabel='Density'>
<Figure size 1000x400 with 1 Axes>
```

You can also create a histogram per column. This data does not create a very pretty histogram because there are not many scores:

```
>>> (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.hist(bins=20, figsize=(10,4))
... )
<AxesSubplot: ylabel='Frequency'>
<Figure size 1000x400 with 1 Axes>
```

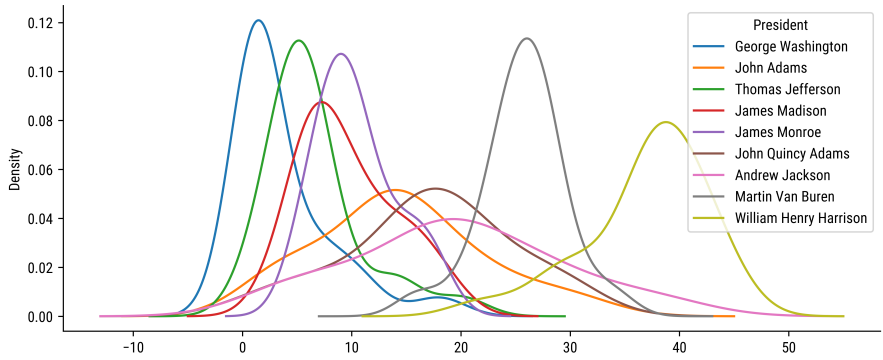


Figure 27.21: Kernel density estimation showing the distribution of scores for each president.

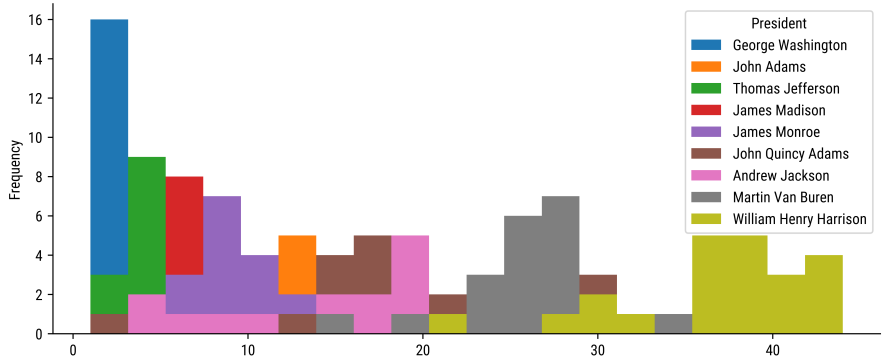


Figure 27.22: Histogram showing the distribution of scores for each president.

Finally, you can create boxplots to summarize the distributions of the columns:

```
>>> ax = (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.box(figsize=(10, 4), rot=45)
... )
>>> _ = ax.set_xticklabels(labels=(pres.President[:9]), ha='right')
<Figure size 1000x400 with 1 Axes>
```

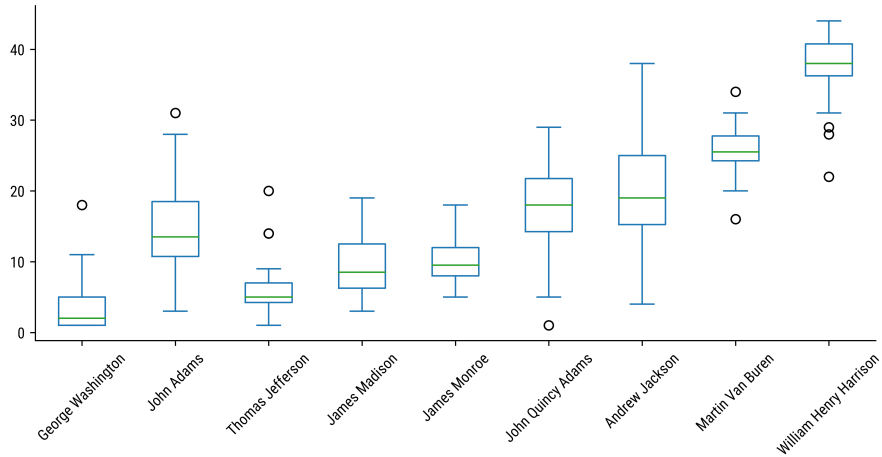


Figure 27.23: Boxplot showing the distribution of scores for each president.

Table 27.1: Dataframe Plotting Methods

Method	Description
`.plot(ax=None, style=None, subplots=False, logx=False, logy=False, xticks=None, yticks=None, xlim=None, ylim=None, xlabel=None, ylabel=None, rot=None, fontsize=None, colormap=None, table=False, **kwargs)`	Common plot parameters. Use `ax` to use existing Matplotlib axes, `style` for color and marker style (see `matplotlib.marker`), `subplots` to create a new plot for each column, `_ticks` to specify tick locations, `_lim` to specify tick limits, `_label` to specify x/y label (default to index/column name), `rot` to rotate labels, `fontsize` for tick label size, `colormap` for coloring, `position`, `table` to create a table with data. Additional arguments are passed to `plt.plot`. Ensure that values are greater than 0 if using `logx` or `logy`.
`.plot.area(x=None, y=None, stacked=True)`	Create a stacked area plot. Use column `x` for the x-axis. Plot each `y` (can be a list) column as a bar. Use `stack=False` to create an unstacked plot.
`.plot.bar(x=None, y=None, stacked=False)`	Create a bar plot. Use column `x` for the x-axis. Plot each `y` (can be a list) column as a bar. Use `stack=True` to stack bars for each x value.

Method	Description
`.plot.barh(x=None, y=None, stacked=False)`	Create a horizontal bar plot. Use column `y` for the x-axis. Plot each `x` (can be a list) column as a bar. Use `stack=True` to stack bars for each y value.
`.plot.kde( bw_method='scott', ind=None)`	Create a Kernel Density Estimate plot. Each column of the dataframe will get its own plot. Use `bw_method` to calculate estimator bandwidth (see `scipy.stats.gaussian_kde`). Use `ind` to specify evaluation points for PDF estimation (NumPy array of points or integer with equally spaced points).
`.plot.density()`	Synonym of `.plot.kde`.
`.plot.hist(bins=10)`	Create a histogram. Each column of the dataframe will get its own plot. Use `bins` to change the number of bins.
`.plot.box(by=None)`	Create boxplots for each column against the index.
`.plot.scatter(x=None, y=None, c=None, s=None, **kwargs)`	Create a scatter plot. `y` can only be a single column name, not a list. Can use `c` parameter to specify a column to color by. Can use the `s` parameter to specify a column to size points by.
`.plot.hexbin(x=None, y=None, C=None, reduce_C_function=None, gridsize=100)`	Create a hexagonal binning plot. `y` can only be a single column name, not a list. `C` can be a column containing an x,y point. `reduce_C_function` is a callable that reduces values in a bin (default `np.mean`. `gridsize` is the number of hexes in the x direction or (x,y) pair.
`.plot.line(x=None, y=None, color=None)`	Plot all columns against the index in the x-axis. Or specify a column for the x-axis with `x` and which column(s) you want to plot as line(s) with `y`. `color` can be a single string specifying a color, a list of colors to cycle over, or a dictionary mapping column to color.
`.plot.pie()`	A method you shouldn't use. (Use `.bar` instead.)

27.9 Summary

In this chapter, we explored basic plotting functionality with series objects. We showed some of the functionality when plotting with a data frame.

We will explore more of this later. Also, note that because the plotting functionality is built on top of Matplotlib, you can customize the plot using Matplotlib.

27.10 Exercises

With a dataset of your choice:

1. Create a histogram from a numeric column. Change the bin size.
2. Create a boxplot from a numeric column.
3. Create a Kernel Density Estimate plot from a numeric column.
4. Create a line from a numeric column.
5. Create a bar plot from the frequency count of a categorical column.

Chapter 28

Reshaping Dataframes with Dummies

This chapter will explore various options for manipulating and reshaping a dataframe. Various patterns will pop up when you start analyzing data, and we will give you the tools you need to deal with them.

28.1 Dummy Columns

Creating *dummy columns* is one way to convert a categorical column into numeric columns. The process is straightforward. If you have a column that has repeated string values, create a new column for each of those values and insert a one or a zero in each new column if it corresponds to the original value.

We will look at a concrete example using the JetBrains Python 2020 survey data. The job columns are almost in dummy format as is. But instead of having entries of one and zero, they have entries of the job title and `NaN`:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow')

>>> print(jb.filter(like='job.role'))
      job.role.DBA job.role.Architect  ... job.role.Systems analyst  \
0             <NA>               <NA>  ...                     <NA>
1             <NA>               <NA>  ...                     <NA>
2             <NA>               <NA>  ...                     <NA>
...            ...                ...  ...                      ...
54459         <NA>               <NA>  ...                     <NA>
54460         <NA>               <NA>  ...                     <NA>
54461         <NA>          Architect  ...                     <NA>

      job.role.Other
0               <NA>
```

```
1              <NA>
2              <NA>
...             ...
54459          <NA>
54460          <NA>
54461          <NA>

[54462 rows x 13 columns]
```

First, we will collapse these job columns into a single column and then I will show how to create proper dummy columns. I'm building up the chain to collapse them and walk through each link in the chain. After we have the job columns from above, we will use the `.where` method to insert 1 instead of the job name:

```
>>> print(jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
... )
      job.role.Architect job.role.Technical writer  ...  \
0                   <NA>                       <NA>  ...
1                   <NA>                       <NA>  ...
2                   <NA>                       <NA>  ...
...                  ...                        ...  ...
54459               <NA>                       <NA>  ...
54460               <NA>                       <NA>  ...
54461                  1                       <NA>  ...

      job.role.Systems analyst job.role.Other
0                         <NA>           <NA>
1                         <NA>           <NA>
2                         <NA>           <NA>
...                        ...            ...
54459                     <NA>           <NA>
54460                     <NA>           <NA>
54461                     <NA>           <NA>

[54462 rows x 8 columns]
```

Now, we will replace `NaN` with 0:

```
>>> print(jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
...  .fillna('0')
... )
```

```
      job.role.Architect job.role.Technical writer  ...  \
0                      0                         0  ...
1                      0                         0  ...
2                      0                         0  ...
...                  ...                       ...  ...
54459                  0                         0  ...
54460                  0                         0  ...
54461                  1                         0  ...

      job.role.Systems analyst job.role.Other
0                            0              0
1                            0              0
2                            0              0
...                        ...            ...
54459                        0              0
54460                        0              0
54461                        0              0

[54462 rows x 8 columns]
```

Now, we will convert the string values to numbers and use the `.idxmax` method. This method scans along an axis and reports the index (or column) where the maximum value is found. In our case, each row should have a single value corresponding to the column of the job:

```
>>> print(jb
...   .filter(regex=r'job.role.*t')
...   .where(jb.isna(), '1')
...   .fillna('0')
...   .astype('bool[pyarrow]')
...   .idxmax(axis='columns')
... )
0        job.role.Business ana...
1              job.role.Architect
2        job.role.Technical su...
                   ...
54459          job.role.Architect
54460       job.role.Data analyst
54461          job.role.Architect
Length: 54462, dtype: object
```

As of pandas 2.2, the `.idxmax` method returns legacy string columns [1]. I'll use `.astype` to convert the results to a pyarrow string column. Finally, I will remove the string `'job.role.'`:

[1] https://github.com/pandas-dev/pandas/issues/56272

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> job = (jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
...  .fillna('0')
...  .astype('bool[pyarrow]')
...  .idxmax(axis='columns')
...  .astype(string_pa)
...  .str.replace('job.role.', '', regex=False)
... )
>>> print(job)
0         Business analyst
1                Architect
2          Technical support
                  ...
54459            Architect
54460         Data analyst
54461            Architect
Length: 54462, dtype: string[pyarrow]
```

The `job` series now looks like a column with categorical data. This is the type of column we usually want to convert into dummy columns.

If you want to create dummy columns from a series (or a dataframe with multiple string columns), call the `pd.get_dummies` function.

```
>>> dum = pd.get_dummies(job)
>>> print(dum)
      Architect Business analyst  ... Technical support  \
0         False             True  ...             False
1          True            False  ...             False
2         False            False  ...              True
...         ...              ...  ...               ...
54459      True            False  ...             False
54460     False            False  ...             False
54461      True            False  ...             False

      Technical writer
0                False
1                False
2                False
...                ...
54459            False
54460            False
54461            False
```

```
[54462 rows x 8 columns]
```

28.2 Undoing Dummy Columns

To go from data arranged in dummy columns to a single column, we will use the `.idxmax` method. Note you will want to execute this on a dataframe that only has the dummy columns:

```
>>> print(dum.idxmax(axis='columns'))
0         Business analyst
1                Architect
2        Technical support
               ...
54459            Architect
54460         Data analyst
54461            Architect
Length: 54462, dtype: string[pyarrow]
```

Table 28.1: Dataframe Reshaping Methods

Method	Description
`.filter(items=None, like=None, regex=None, axis=1)`	Return a dataframe filtered by index axis labels. Use `items` to specify a list of names. Use `like` to specify a substring. Use `regex` to specify a regular expression.
`.where(cond, other=nan, axis=None, level=None, errors='raise', try_cast=None)`	Replace the values where `cond` (a boolean array) is `False`. Generally I use this on series.
`.fillna(value=None, method=None, axis=None, limit=None, downcast=None)`	Return a dataframe with missing values filled in. `value` can be a scalar, dictionary (mapping column to value), series (values for index) or dataframe. Use `method` for `'bfill'`, `'pad'`, or `'ffill'`. You can limit the replacements with `limit`. Use `downcast` to specify a dictionary mapping a column to a new type (ie from `float64` to `int64`).
`.idxmax(axis=0, skipna=True)`	Return the index of the first maximum value over an axis.

Method	Description
`pd.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, sparse=False, drop_first=False, dtype=None)`	Return a dataframe with string/categorical columns from `data` converted into dummy columns.
`np.where( condition, x=None, y=None)`	Return a numpy array where `condition` (boolean array) is `True` using value `x` (scalar, series) and `y` (scalar, series) otherwise.

28.3 Summary

Dummy columns are one way to encode categorical variables as numbers. Many will use this option to prepare data for machine learning because many machine learning algorithms do not support string data, only numeric.

28.4 Exercises

With a dataset of your choice:

1. Create dummy columns derived from a string column.
2. Undo the dummy columns.

Chapter 29

Reshaping By Pivoting and Grouping

This chapter will explore one of the most powerful options for data manipulation: pivot tables. Pandas provides multiple syntaxes for creating them. One uses the `.pivot_table` method, and the other common one leverages the `.groupby` method. You can also represent some of these operations with the `pd.crosstab` function.

We will explore all of these using the cleaned-up JetBrains survey data:

```
>>> print(jb2)
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                 True  ...              3.0         3.6
5       21                False  ...              3.0         3.8
10      21                False  ...              1.0         3.8
...    ...                  ...  ...              ...         ...
54442   50                 True  ...             11.0         3.6
54447   30                False  ...              3.0         3.6
54450   30                False  ...             11.0         3.8

[6980 rows x 20 columns]
```

29.1 A Basic Example

When your boss asks you to get numbers "by X column", that should be a hint to pivot (or group) your data. Assume your boss asked, "What is the average age by country for each employment status?" This is like one of those word problems that you had to learn how to do in math class, and you needed to translate the words into math operations. In this case, we must pick a pandas operation and map the problem into those operations.

I would translate this problem into:

- Put the country in the index
- Have a column for each employment status
- Put the average age in each cell

These map cleanly to the parameters of the `.pivot_table` method. One solution would look like this:

```
>>> print(jb2
...  .pivot_table(index='country_live', columns='employment_status',
...      values='age', aggfunc='mean')
... )
employment_status  Fully employed   Partially emplo  \
country_live
Algeria                      21.0             30.0
Argentina                30.291667             30.0
Armenia                      25.0             <NA>
...                           ...              ...
Uzbekistan                   21.0             21.0
Venezuela                26.428571             50.0
Viet Nam                 23.266667             19.5

employment_status  Self-employed (  Student  Working student
country_live
Algeria                       27.0     <NA>             21.0
Argentina                     32.2     <NA>            23.25
Armenia                       21.0     <NA>             <NA>
...                            ...      ...              ...
Uzbekistan                    <NA>     <NA>             21.0
Venezuela                     35.0     <NA>             30.0
Viet Nam                      40.5     <NA>             25.5

[76 rows x 5 columns]
```

It turns out that we can use the `pd.crosstab` function as well. Because this is a function, we need to provide the data as a series rather than the column names:

```
>>> print(pd.crosstab(index=jb2.country_live,
...  columns=jb2.employment_status, values=jb2.age, aggfunc='mean'))
employment_status  Fully employed   Partially emplo  \
country_live
Algeria                      21.0             30.0
Argentina                30.291667             30.0
Armenia                      25.0             <NA>
...                           ...              ...
Uzbekistan                   21.0             21.0
Venezuela                26.428571             50.0
Viet Nam                 23.266667             19.5

employment_status  Self-employed (  Student  Working student
```

Pivot Tables

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(auto.pivot_table(aggfunc="max",
  index="year",
  columns="make",
  values="city08")
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 29.1: The `.pivot_table` method allows you to pick column(s) for the index, column(s) for the column, and column(s) to aggregate. (If you specify multiple columns to aggregate, you will get hierarchical columns.)

```
country_live
Algeria                27.0      <NA>             21.0
Argentina              32.2      <NA>            23.25
Armenia                21.0      <NA>             <NA>
...                     ...       ...              ...
Uzbekistan             <NA>      <NA>             21.0
Venezuela              35.0      <NA>             30.0
Viet Nam               40.5      <NA>             25.5

[76 rows x 5 columns]
```

Finally, we can do this with a `.groupby` method call. The call to `.groupby` returns a `DataFrameGroupBy` object. It is a lazy object and does not perform any calculations until we indicate which aggregation to perform. We can also pull

Cross Tabulation

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(pd.crosstab(aggfunc="max",
  index=auto.year,
  columns=auto.make,
  values=auto.city08)
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 29.2: The `pd.crosstab` function allows you to pick column(s) for the index, column(s) for the column, and a column to aggregate. You cannot aggregate multiple columns (unlike `.pivot_table`).

off a column and then only perform an aggregation on that column instead of all of the non-grouped columns.

This operation is a little more involved. We pull off the *age* column and then calculate the mean for each *country_live* and *employment_status* group. Then we leverage `.unstack` to pull out the inner-most index and push it up into a column (we will dive into `.unstack` later). You can think of `.groupby` and subsequent methods as the low-level underpinnings of `.pivot_table` and `pd.crosstab`:

```
>>> print(jb2
...   .groupby(['country_live', 'employment_status'])
...   .age
...   .mean()
...   .unstack()
... )
```

```
employment_status  Freelancer (a p  Fully employed   ...  Student  \
country_live                                          ...
Algeria                       <NA>            21.0  ...     <NA>
Argentina                     <NA>       30.291667  ...     <NA>
Armenia                       <NA>            25.0  ...     <NA>
...                            ...             ...  ...      ...
Uzbekistan                    <NA>            21.0  ...     <NA>
Venezuela                     <NA>       26.428571  ...     <NA>
Viet Nam                      <NA>       23.266667  ...     <NA>

employment_status  Working student
country_live
Algeria                       21.0
Argentina                    23.25
Armenia                       <NA>
...                            ...
Uzbekistan                    21.0
Venezuela                     30.0
Viet Nam                      25.5

[76 rows x 8 columns]
```

Many programmers and SQL analysts find the `.groupby` syntax intuitive, while Excel junkies often feel more at home with the `.pivot_table` method. The `crosstab` function works in some situations but is less flexible. It makes sense to learn the different options. The `.groupby` method is the foundation of the other two, but a cross-tabulation may be more convenient.

29.2 Using a Custom Aggregation Function

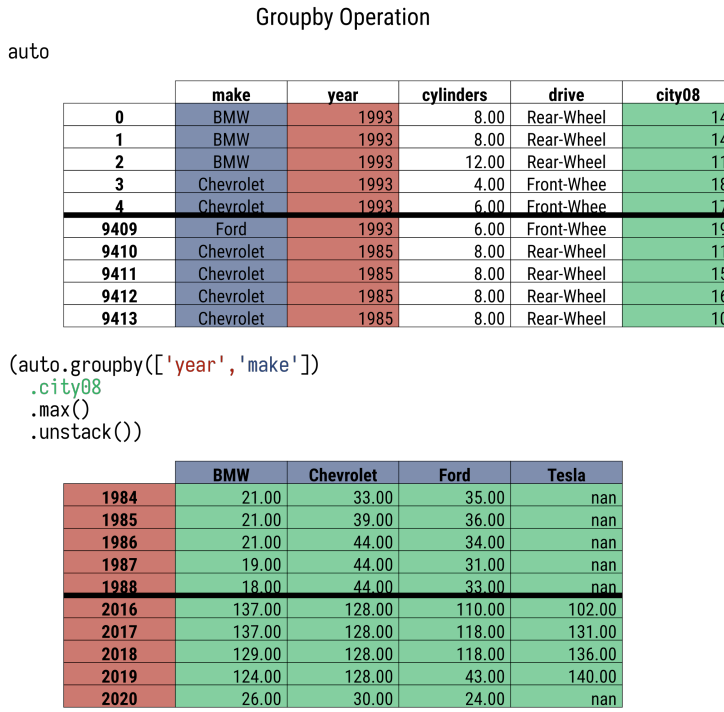
Your boss thanks you for providing insight on the age of employment status by country and says she has a more important question: "What is the percentage of Emacs users by country?"

We will need a function that takes a group (in this case, a series) of country respondents about IDE preference and returns the percentage that chose emacs:

```
>>> def per_emacs(ser):
...     return ser.str.contains('Emacs').sum() / len(ser) * 100
```

Note

When you need to calculate a percentage in pandas, you can use the `.mean` method. The following code is equivalent to the above:



	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

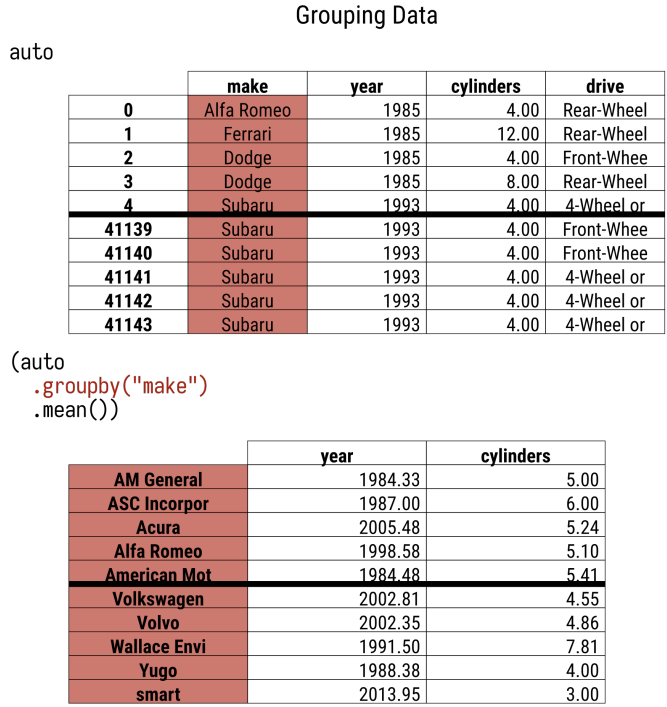
	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

Figure 29.3: The `.groupby` method allows you to pick a column(s) for the index and column(s) to aggregate. You can `.unstack` the inner column to simulate pivot tables and cross-tabulation.

```
>>> def per_emacs(ser):
...     return ser.str.contains('Emacs').mean() * 100
```

We are now ready to pivot. In this case, we still want the country in the index, but we only want a single column, the emacs percentage. So we don't provide a `columns` parameter:

```
>>> print(jb2
...  .pivot_table(index='country_live', values='ide_main',
...               aggfunc=per_emacs)
... )
             ide_main
country_live
Algeria             0
Argentina           0
```



	make	year	cylinders	drive
0	Alfa Romeo	1985	4.00	Rear-Wheel
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

	year	cylinders
AM General	1984.33	5.00
ASC Incorpor	1987.00	6.00
Acura	2005.48	5.24
Alfa Romeo	1998.58	5.10
American Mot	1984.48	5.41
Volkswagen	2002.81	4.55
Volvo	2002.35	4.86
Wallace Envi	1991.50	7.81
Yugo	1988.38	4.00
smart	2013.95	3.00

Figure 29.4: When your boss asks you to get the average values by make, you should recognize that you need to pull out `.groupby('make')`.

```
Armenia             0
...               ...
Uzbekistan          0
Venezuela           0
Viet Nam            0

[76 rows x 1 columns]
```

Using `pd.crosstab` is a little more complicated as it expects a "cross-tabulation" of two columns, one column going in the index and the other column going in the columns. To get a "column" for the cross tabulation, we will assign a column to a single scalar value (which will trick the cross-tabulation into creating just one column with the name of the scalar value):

```
>>> print(pd.crosstab(index=jb2.country_live,
...     columns=jb2.assign(iden='emacs_per').iden,
```

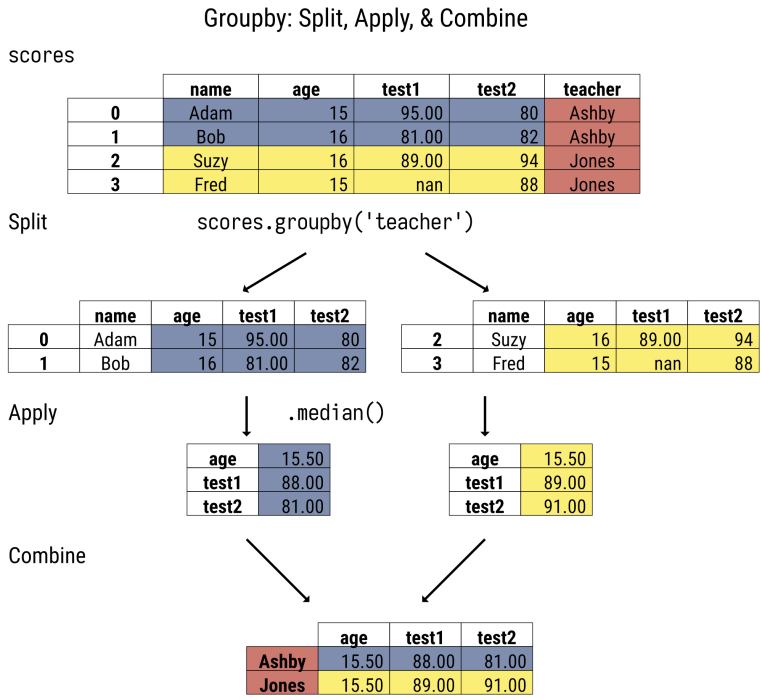


Figure 29.5: A groupby operation splits the data into groups. You can apply aggregate functions to the group. Then the results of the aggregates are combined. The column we are grouping by will be placed in the index.

```
...     values=jb2.ide_main, aggfunc=per_emacs))
iden          emacs_per
country_live
Algeria               0
Argentina             0
Armenia               0
...                 ...
Uzbekistan            0
Venezuela             0
Viet Nam              0

[76 rows x 1 columns]
```

Finally, here is the `.groupby` version. I find this one very clear. Group by the *country_live* column, pull out just the *ide_main* columns. Calculate the percentage of emacs users for each of those groups:

```
>>> print(jb2
...  .groupby('country_live')
...  [['ide_main']]
...  .agg(per_emacs)
... )
             ide_main
country_live
Algeria             0
Argentina           0
Armenia             0
...               ...
Uzbekistan          0
Venezuela           0
Viet Nam            0

[76 rows x 1 columns]
```

29.3 Multiple Aggregations

Assume your boss asked, "What is the minimum and maximum age for each country?" When you see "for each" or "by", your mind should think that whatever follows either of the terms should go in the index. This question is answered with a pivot table or using `.groupby`. (We can use a cross-tabulation, but you will need to add a column to do this, and it feels unnatural to me).

Here is the `.pivot_table` solution. The *country_live* column goes in the `index` parameter. *age* is what we want to aggregate, which goes in the `values` parameter. And we need to specify a sequence with `min` and `max` for the `aggfunc` parameter:

```
>>> print(jb2
...     .pivot_table(index='country_live', values='age',
...        aggfunc=('min', 'max'))
... )
              max  min
country_live
Algeria        30   21
Argentina      50   21
Armenia        60   18
...           ...  ...
Uzbekistan     21   21
Venezuela      50   21
Viet Nam       60   18
```

Grouping Data with Multiple Aggregations

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby('make')
 .agg(['min', 'max']))
```

Hierarchical columns!

	year	year	cylinders	cylinders
	min	max	min	max
Acura	1986	2020	4.00	6.00
Audi	1984	2020	4.00	12.00
BMW	1984	2020	2.00	12.00
BYD	2012	2019	nan	nan
Bentley	1998	2019	8.00	12.00
VPG	2011	2013	8.00	8.00
Vector	1992	1997	8.00	12.00
Volvo	1984	2019	4.00	8.00
Yugo	1986	1990	4.00	4.00
smart	2008	2019	3.00	3.00

Figure 29.6: You can leverage the `.agg` method with `.groupby` to perform multiple aggregations.

```
[76 rows x 2 columns]
```

When you look at this using the `.groupby` method, you first determine what you want in the index, *country_live*. Then we will pull off the *age* column from each group. Finally, we will apply two aggregate functions, `min` and `max`:

```
>>> print(jb2
...   .groupby('country_live')
...   .age
...   .agg(['min', 'max'])
... )
              min  max
country_live
```

```
Algeria        21   30
Argentina      21   50
Armenia        18   60
...           ...  ...
Uzbekistan     21   21
Venezuela      21   50
Viet Nam       18   60

[76 rows x 2 columns]
```

Here is the example for `pd.crosstab`. I don't recommend this, but I provide it to help explain how cross-tabulation works. Again, we want *country_live* in the index. With cross-tabulation, we must provide a series to spread out in the columns. We cannot use the *age* column as the `columns` parameter because we want to aggregate those numbers and hence need to set them as the `values` parameter. Instead, I will create a new column with a single scalar value, the string `'age'`. We can provide both of the aggregations we want to use to the `aggfunc` parameter. Below is my solution. Note that it has hierarchical columns:

```
>>> print(pd.crosstab(jb2.country_live, values=jb2.age,
...     aggfunc=('min', 'max'), columns=jb2.assign(val='age').val))
             max min
val          age age
country_live
Algeria       30  21
Argentina     50  21
Armenia       60  18
...           ..  ..
Uzbekistan    21  21
Venezuela     50  21
Viet Nam      60  18

[76 rows x 2 columns]
```

29.4 Per Column Aggregations

In the previous example, we looked at applying multiple aggregations to a single column. We can also apply various aggregations to many columns. Here, we get each numeric column's minimum and maximum value by country.

The default behavior is to aggregate every column. However, we have categorical columns that don't have an order, so this will fail:

```
>>> print(jb2
...   .pivot_table(index='country_live',
```

```
...                aggfunc=('min', 'max'))
... )
Traceback (most recent call last):
  ...
TypeError: Cannot perform min with non-ordered Categorical
```

We need to explicitly call out the numeric columns in the `values` parameter:

```
>>> print(jb2
...  .pivot_table(index='country_live', values=['age', 'company_size',
...                                             'nps_main_ide', 'python_years',
...                                             'team_size', 'years_of_coding'],
...                aggfunc=('min', 'max'))
... )
             age      ... years_of_coding
             max min  ...              max  min
country_live          ...
Algeria       30  21  ...              6.0  0.5
Argentina     50  21  ...             11.0  0.5
Armenia       60  18  ...             11.0  0.5
...           ..  ..  ...              ...  ...
Uzbekistan    21  21  ...              6.0  0.5
Venezuela     50  21  ...             11.0  0.5
Viet Nam      60  18  ...             11.0  0.5

[76 rows x 12 columns]
```

Here is the groupby version. Note that we specify the numeric columns to aggregate so we avoid the exception complaining that it cannot aggregate non-numeric values:

```
>>> print(jb2
...  .groupby('country_live')
...  [['age', 'company_size', 'nps_main_ide', 'python_years',
...    'team_size', 'years_of_coding']]
...  .agg(['min', 'max'])
... )
             age      ... years_of_coding
             min max  ...              min   max
country_live          ...
Algeria       21  30  ...              0.5   6.0
Argentina     21  50  ...              0.5  11.0
Armenia       18  60  ...              0.5  11.0
...           ..  ..  ...              ...   ...
Uzbekistan    21  21  ...              0.5   6.0
```

```
Venezuela     21  50  ...            0.5  11.0
Viet Nam      18  60  ...            0.5  11.0

[76 rows x 12 columns]
```

I'm not going to do this with `pd.crosstab`, and I recommend that you don't as well.

Sometimes, we want to specify aggregations per column. With both the `.pivot_table` and `.groupby` methods, we can provide a dictionary mapping a column to an aggregation function or a list of aggregation functions.

Assume your boss asked: "What are the minimum and maximum ages and the average team size for each country?". Here is the translation to a pivot table:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'team_size': 'mean'})
... )
             age     team_size
             max min      mean
country_live
Algeria       30  21       1.5
Argentina     50  21  4.155172
Armenia       60  18       5.2
...           ..  ..       ...
Uzbekistan    21  21  1.333333
Venezuela     50  21      3.25
Viet Nam      60  18  4.809524

[76 rows x 3 columns]
```

Here is the groupby version:

```
>>> print(jb2
...  .groupby('country_live')
...  .agg({'age': ['min', 'max'],
...        'team_size': 'mean'})
... )
             age     team_size
             min max      mean
country_live
Algeria       21  30       1.5
Argentina     21  50  4.155172
Armenia       18  60       5.2
...           ..  ..       ...
```

```
Uzbekistan    21  21  1.333333
Venezuela     21  50      3.25
Viet Nam      18  60  4.809524

[76 rows x 3 columns]
```

One nuisance of these results is that they have hierarchical columns. I generally find these types of columns annoying and confusing to work with. They do come in useful for stacking and unstacking, which we will explore in a later section. However, I like to remove them and will also show a general recipe for that later.

But I want to show one last feature that is specific to `.groupby` and may make you favor it as there is no equivalent functionality found in `.pivot_table`. That feature is called *named aggregations*. When calling the `.agg` method on a groupby object, you can use a keyword parameter to pass in a tuple of the column and aggregation function. The keyword parameter will be turned into a (flattened) column name.

We could re-write the previous example like this:

```
>>> print(jb2
...   .groupby('country_live')
...   .agg(age_min=('age', 'min'),
...        age_max=('age', 'max'),
...        team_size_mean=('team_size', 'mean')
...       )
... )
              age_min  age_max  team_size_mean
country_live
Algeria            21       30            1.5
Argentina          21       50       4.155172
Armenia            18       60            5.2
...               ...      ...            ...
Uzbekistan         21       21       1.333333
Venezuela          21       50           3.25
Viet Nam           18       60       4.809524

[76 rows x 3 columns]
```

Notice that the above result has flat columns.

29.5 Grouping by Hierarchy

I just mentioned how much hierarchical columns bothered me. I'll admit, they are sometimes helpful. Now, I'm going to show you how to create hierarchical indexes. Suppose your boss asked about minimum and maximum age by country and editor usage. We want to have both the country

and the editor in the index. We need to pass in a list of columns we want in the index:

```
>>> print(jb2.pivot_table(index=['country_live', 'ide_main'],
...   values='age', aggfunc=['min', 'max']))
                                    min max
                                    age age
country_live ide_main
Algeria      Atom                    21  21
             Jupyter Notebook        30  30
             PyCharm Community Edition  30  30
...                                  ..  ..
Viet Nam     PyCharm Professional E...  18  30
             VS Code                 18  21
             Vim                     30  40

[695 rows x 2 columns]
```

Here is the groupby version:

```
>>> print(jb2
...  .groupby(by=['country_live', 'ide_main'])
...  [['age']]
...  .agg(['min', 'max'])
... )
                              age
                              min   max
country_live ide_main
Algeria      Atom              21    21
             Eclipse + Pydev  <NA>  <NA>
             Emacs            <NA>  <NA>
...                            ...   ...
Viet Nam     Sublime Text     <NA>  <NA>
             VS Code            18    21
             Vim                30    40

[1216 rows x 2 columns]
```

Those paying careful attention will note that the results of apply multiple aggregations from `.groupby` and `.pivot_table` are not exactly the same. There are a few differences:

- The hierarchical column levels are swapped (*age* is inside of *min* and *max* when pivoting, but outside when grouping)
- The row count differs

Flattening Grouping Data by Multiple Columns

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby(['make', 'year'])
 .max()
 .reset_index())
```

	make	year	cylinders
0	Acura	1986	6.00
1	Acura	1987	6.00
2	Acura	1988	6.00
3	Acura	1989	6.00
4	Acura	1990	6.00
1345	smart	2015	3.00
1346	smart	2016	3.00
1347	smart	2017	3.00
1348	smart	2018	nan
1349	smart	2019	nan

Figure 29.7: Grouping with a list of columns will create a multi-index, an index with hierarchical levels.

I'm not sure why pandas swaps the levels. You could use the `.swaplevel` method to change that. However, I would personally use a named aggregation with a groupby for flat columns:

```
>>> print(jb2
...  .groupby(by=['country_live', 'ide_main'])
...  [['age']]
...  .agg(['min', 'max'])
...  .swaplevel(axis='columns')
... )
                              min   max
                              age   age
country_live ide_main
Algeria      Atom              21    21
             Eclipse + Pydev  <NA>  <NA>
             Emacs            <NA>  <NA>
```

```
...                              ...   ...
Viet Nam     Sublime Text       <NA>  <NA>
             VS Code              18    21
             Vim                  30    40

[1216 rows x 2 columns]
```

```
>>> print(jb2
...   .groupby(by=['country_live', 'ide_main'])
...   .agg(age_min=('age', 'min'), age_max=('age', 'max'))
... )
                               age_min  age_max
country_live ide_main
Algeria      Atom                 21       21
             Eclipse + Pydev     <NA>     <NA>
             Emacs               <NA>     <NA>
...                               ...      ...
Viet Nam     Sublime Text        <NA>     <NA>
             VS Code               18       21
             Vim                   30       40

[1216 rows x 2 columns]
```

The reason the row count is different is a little more nuanced. I have set the *country_live* and *ide_main* columns to be categorical. When you perform a groupby with categorical columns, pandas will create the cartesian product of those columns even if there is no corresponding value. You can see above a few rows with both values of `<NA>`. The pivot table version (at the start of the section) did not have the missing values.

Note

Be careful when grouping with multiple categorical columns with high cardinality. You can generate a very large (and sparse) result!

You could always call `.dropna` after the fact, but I prefer to use the `observed` parameter instead (pandas 3 will set this to true by default):

```
>>> print(jb2
...   .groupby(by=['country_live', 'ide_main'], observed=True)
...   .agg(age_min=('age', 'min'), age_max=('age', 'max'))
... )
                                        age_min  age_max
country_live ide_main
Algeria      Atom                          21       21
             Jupyter Notebook              30       30
             PyCharm Community Edition     30       30
```

```
...                                          ...      ...
Viet Nam     PyCharm Professional E...        18       30
             VS Code                          18       21
             Vim                              30       40

[695 rows x 2 columns]
```

That's looking better!

29.6 Grouping with Functions

Until now, we have been grouping by various values found in columns. Sometimes, I want to group by something other than an existing column, and I have a few options.

Often, I will create a particular column containing the values I want to group by. In addition, both pivot tables and groupby operations support passing in a function instead of a column name. This function accepts a single index label and should return a value to group on. In the example below, we group based on whether the index value is even or odd. We then calculate the size of each group. Here is the grouper function and the `.pivot_table` implementation:

```
>>> def even_grouper(idx):
...     return 'odd' if idx % 2 else 'even'

>>> jb2.pivot_table(index=even_grouper, aggfunc='size')
even    3515
odd     3465
dtype: int64
```

And here is the `.groupby` version:

```
>>> (jb2
...  .groupby(even_grouper)
...  .size()
... )
even    3515
odd     3465
dtype: int64
```

When we look at time series manipulation later, we will see that pandas provides a handy `pd.Grouper` class to allow us to easily group by time attributes.

Table 29.1: Dataframe Pivoting and Grouping Methods

Method	Description
`pd.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)`	Create a cross-tabulation (counts by default) from an `index` (series or list of series) and `columns` (series or list of series). Can specify a column (series) to aggregate `values` along with a function, `aggfunc`. Using `margins=True` will add subtotals. Using `dropna=False` will keep columns that have no values. Can normalize over `'all'` values, the rows (`'index'`), or the `'columns'`.
`.pivot_table( values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, margins_name='All', dropna=True, observed=False, sort=True)`	Create a pivot table. Use `index` (series, column name, `pd.Grouper`, or list of previous) to specify index entries. Use `columns` (series, column name, `pd.Grouper`, or list of previous) to specify column entries. The `aggfunc` (function, list of functions, dictionary (column name to function or list of functions) specifies a function to aggregate values. Missing values are replaced with `fill_value`. Set `margins=True` to add subtotals/totals. Using `dropna=False` will keep columns that have no values. Use `observed=True` to only show values that appeared for categorical groupers.
`.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, observed=False, dropna=True)`	Return a grouper object, grouped using `by` (column name, function (accepts each index value, returns group name/id), series, `pd.Grouper`, or list of column names). Use `as_index=False` to leave grouping keys as columns. Common plot parameters. Use `observed=True` to only show values that appeared for categorical groupers. Using `dropna=False` will keep columns that have no values.
`.stack(level=-1, dropna=True)`	Push column level into the index level. Can specify a column `level` (-1 is innermost).
`.unstack(level=-1, dropna=True)`	Push index level into the column level. Can specify an index `level` (-1 is innermost).

Table 29.2: Groupby Methods and Operations

Method	Description
Column access	Access a column by attribute or index operation.
`g.agg(func=None, *args, engine=None, engine_kwargs= None, **kwargs)`	Apply an aggregate `func` to groups. `func` can be string, function (accepting a column and returning a reduction), a list of the previous, or a dictionary mapping column name to string, function, or list of strings and/or functions.
`g.aggregate`	Same as `g.agg`.
`g.all(skipna= True)`	Collapse each group to `True` if all the values are truthy.
`g.any(skipna= True)`	Collapse each group to `True` if any of the values are truthy.
`g.apply(func, *args, **kwargs)`	Apply a function to each group. The function should accept the group (as a dataframe) and return scalar, series, or dataframe. These return a series, dataframe (with each series as a row), and a dataframe (with the index as an inner index of the result), respectively.
`g.count()`	Count of non-missing values for each group.
`g.ewm(com=None, span=None, halflife=None)`	Return an Exponentially Weighted grouper. Can specify the center of mass (`com`), decay `span`, or `halflife`. Will need to apply further aggregation to this.
`g.expanding( min_periods=1, center=False, axis=0, method= 'single')`	Return an expanding Window object. Can specify the minimum number of observations per period (`min_periods`), set label at `center` of the window, and whether to execute over `'single'` column or the whole group (`'table'`). Will need to apply further aggregation to this.
`g.filter(func, dropna=True, *args, **kwargs)`	Return the original dataframe but with filtered groups removed. `func` is a predicate function that accepts a group and returns `True` to keep values from the group. If `dropna=False`, groups that evaluate to `False` are filled with `NaN`.
`g.first( numeric_only= False, min_count=-1)`	Return the first row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.

Method	Description
`g.get_group( name, obj=None)`	Return a dataframe with named group.
`g.groups`	Property with dictionary mapping group name to list of index values. (See `.indices`.)
`g.head(n=5)`	Return the first `n` rows of each group. Uses the original index.
`g.idxmax( axis=0, skipna=True)`	Return an index label of the maximum value for each group.
`g.idxmin( axis=0, skipna=True)`	Return an index label of minimum value for each group.
`g.indices`	Property with a dictionary mapping group name to `np.array` of index values. (See `.groups`.)
`g.last( numeric_only= False, min_count=-1)`	Return the last row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.max( numeric_only= False, min_count=-1)`	Return the maximum row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.mean( numeric_only= True)`	Return the mean of each group.
`g.min( numeric_only= False, min_count=-1)`	Return the minimum row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.ndim`	Property with the number of dimensions of the result.
`g.ngroup( ascending= True)`	Return a series with the original index and values for each group number.
`g.ngroups`	Property with the number of groups.
`g.nth(n, dropna=None)`	Take the nth row from each group.
`g.nunique( dropna=True)`	Return a dataframe with unique counts for each group.
`g.ohlc()`	Return a dataframe with open, high, low, and close values for each group.
`g.pipe(func, *args, **kwargs)`	Apply the `func` to each group.
`g.prod( numeric_only= True, min_count=0)`	Return a dataframe with product of each group.
`g.quantile( q=.5, interpolation= 'linear')`	Return a dataframe with quantile for each group. Can pass a list for `q` and get the inner index for each value.

Method	Description
`g.rank( method='average', na_option= 'keep', ascending= True, pct=False, axis=0)`	Return a dataframe with numerical ranks for each group. `method` allows to specify tie handling. `'average'`, `'min'`, `'max'`, `'first'` (uses order they appear in series), `'dense'` (like `'min'`, but rank only increases by one after a tie). `na_option` allows you to specify `NaN` handling. `'keep'` (stay at `NaN`), `'top'` (move to smallest), `'bottom'` (move to largest).
`g.resample( rule, *args, **kwargs)`	Create a resample objectnp.where(dum with offset alias frequency specified by `rule`. Will need to apply further aggregation to this.
`g.rolling( window_size)`	Create a rolling grouper. Will need to apply further aggregation to this.
`g.sample( n=None, frac=None, replace=False, weights=None, random_state= None)`	Return a dataframe with sample from each group. Uses the original index.
`g.sem(ddof=1)`	Return the mean of the standard error of the mean of each group. Can specify degrees of freedom (`ddof`).
`g.shift( periods=1, freq=None, axis=0, fill_value=None`	Create a shifted values for each group. Uses the original index.
`g.size()`	Return a series with the size of each group.
`g.skew(axis=0, skipna=True, level=None, numeric_only= False)`	Return a series with numeric columns inserted as the inner level of a grouped index with unbiased skew.
`g.std(ddof=1)`	Return the standard deviation of each group. Can specify degrees of freedom (`ddof`).
`g.sum( numeric_only= True, min_count=0)`	Return a dataframe with the sum of each group.
`g.tail(n=5)`	Return the last `n` rows of each group. Uses the original index.
`g.take(indices, axis=0)`	Return a dataframe with the index positions (`indices`) from each group. Positions are relative to the group.
`g.transform( func, *args, **kwargs)`	Return a dataframe with the original index. The function will get passed a group and should return a dataframe with the same dimensions as the group.

Method	Description
`g.var(ddof=1)`	Return the variance of each group. Can specify degrees of freedom (`ddof`).

29.7 Summary

Grouping is one of the most powerful tools that pandas provides. It is the underpinning of the `.pivot_table` method, which in turn implements the `pd.crosstab` function. These constructs can be hard to learn because of the inherent complexity of the operation, the hierarchical nature of the result, and the syntax. If you are using `.groupby` remember to write out your chains and step through them one step at a time. That will help you understand what is going on. You will also need to practice these. Once you learn the syntax, practicing will help you master these concepts.

29.8 Exercises

With a dataset of your choice:

1. Group by a categorical column and take the mean of the numeric columns.
2. Group by a categorical column and take the mean and max of the numeric columns.
3. Group by a categorical column and apply a custom aggregation function that calculates the mode of the numeric columns.
4. Group by two categorical columns and take the mean of the numeric columns.
5. Group by binned numeric column and take the mean of the numeric columns.

Chapter 30

More Aggregations

The previous chapter introduced grouping and pandas' related pivoting and cross-tabulation functionality. We will dive in a little deeper and explore the `.transform` method and the `.filter` method of a groupby object.

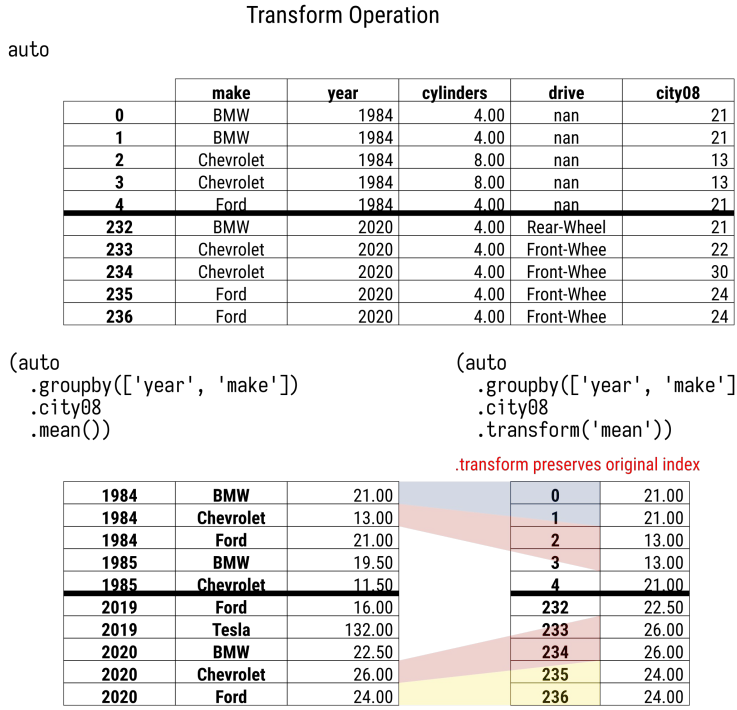
30.1 Aggregations while Keeping Rows

Let's assume we are still looking at the JetBrains dataset and want to add a new column, the count of responses from a country. One way would be to create a pivot table (or groupby) of the count of responses for each country and then merge that data back into the original dataframe. However, if we use the `.transform` method following `.groupby`, we get the aggregation, but they are not collapsed. The result is in terms of the original index.

This is one of the reasons I gravitate towards `.groupby` instead of `.pivot_table`, the flexibility. (Coming from a software backward and familiarity with SQL probably doesn't hurt either).

Here is the count of the country for each original row. We can provide our own function to the `.transform` method or take advantage of existing functions. We want to use the `'size'` function to get new counts. However, we just want to apply it to a single column, it doesn't matter which column we choose, so I will use *age*:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)

>>> (jb2
...  .groupby('country_live', observed=True)
...  .age
...  .transform('size')
... )
```



	make	year	cylinders	drive	city08
0	BMW	1984	4.00	nan	21
1	BMW	1984	4.00	nan	21
2	Chevrolet	1984	8.00	nan	13
3	Chevrolet	1984	8.00	nan	13
4	Ford	1984	4.00	nan	21
232	BMW	2020	4.00	Rear-Wheel	21
233	Chevrolet	2020	4.00	Front-Whee	22
234	Chevrolet	2020	4.00	Front-Whee	30
235	Ford	2020	4.00	Front-Whee	24
236	Ford	2020	4.00	Front-Whee	24

1984	BMW	21.00
1984	Chevrolet	13.00
1984	Ford	21.00
1985	BMW	19.50
1985	Chevrolet	11.50
2019	Ford	16.00
2019	Tesla	132.00
2020	BMW	22.50
2020	Chevrolet	26.00
2020	Ford	24.00

0	21.00
1	21.00
2	13.00
3	13.00
4	21.00
232	22.50
233	26.00
234	26.00
235	24.00
236	24.00

Figure 30.1: The `.transform` method allows us to perform aggregations on groups but returns the resulting aggregations in terms of the original index.

```
1         687
5         300
10        156
         ...
54442     389
54447     517
54450    1322
Name: age, Length: 6980, dtype: int64[pyarrow]
```

Here is the code to create a new column *country_responses*:

```
>>> print(jb2
...   .assign(country_responses=(jb2
...       .groupby('country_live', observed=True)
...       .age
...       .transform('size')))
```

```
... )
       age are_you_datascientist  ...  python3_ver country_responses
1       21                  True  ...          3.6               687
5       21                 False  ...          3.8               300
10      21                 False  ...          3.8               156
...    ...                   ...  ...          ...               ...
54442   50                  True  ...          3.6               389
54447   30                 False  ...          3.6               517
54450   30                 False  ...          3.8              1322

[6980 rows x 21 columns]
```

You can pass in a function to the `.transform` method. This function should accept a DataFrame if you are working with a `DataFrameGroupBy` object or a series if you are working with a `SeriesGroupBy` object. It should return a scalar value.

Below is a table with the strings that `.transform` accepts (you can find these in `pd.core.groupby.generic.base.transform_kernel_allowlist`). Those that return a series are marked with (S).

Table 30.1: Groupby Transform String

String	Description
`'all'`	Returns `True` for every value if every value is truthy.
`'any'`	Returns `True` for every value if any value is truthy.
`'backfill'`	Backfills values for the group.
`'bfill'`	Backfills values for the group.
`'count'`	Count of non-NA values for the group.
`'cumcount'`	Number of each item in the group starting at 0 (S).
`'cummax'`	Cumulative maximum for each group.
`'cummin'`	Cumulative minimum for each group.
`'cumprod'`	Cumulative product for each group.
`'cumsum'`	Cumulative sum for each group.
`'diff'`	Subtract the previous row from each row. The group needs to be numeric.
`'ffill'`	Forward fill each group.
`'fillna'`	Fill in missing values for each group. Must specify `method` (`'ffill'` or `'bfill'`) or `value` parameter.
`'first'`	First row for each group.
`'idxmax'`	Index of the maximum value for each group.
`'idxmin'`	Index of the minimum value for each group.
`'last'`	Last row for each group.
`'mad'`	Mean absolute deviation for each group.
`'max'`	Maximum value for each group.
`'mean'`	Mean value for each group.
`'median'`	Mean value for each group.

String	Description
`'min'`	Minimum value for each group.
`'nth'`	Nth value for each group. Must specify the `n` parameter.
`'nunique'`	Number of unique values for each group.
`'pad'`	Synonym for `'ffill'`.
`'pct_change'`	Percent change from current row and previous for each group. The group needs to be numeric.
`'prod'`	Product of each group.
`'quantile'`	Median of each group. Specify `q` (0-1) to change the quantile. The group needs to be numeric.
`'rank'`	Rank of each group.
`'sem'`	Unbiased standard error of each group.
`'shift'`	Shift each group row down. Can specify `periods` (default 1) or `freq` with date index.
`'size'`	Size of each group. Only works for a group with a single column (not a dataframe).
`'skew'`	Skew of each group.
`'std'`	Standard deviation of each group.
`'sum'`	Sum of each group. (Will add strings!)
`'var'`	Variance of each group.

30.2 Filtering Parts of Groups

Our treatment of grouping operations has shown us how to aggregate by specific columns. In the previous section, we explored the `.transform` method of a groupby object and saw that we could calculate aggregations on groups but retain the original index. In this section, we will explore how to filter parts of groups by an aggregation but return the result with the original index.

Using the cleaned-up JetBrains data, let's remove any row where the size of the country is less than the median size of countries. It looks like the median value is 33:

```
>>> (jb2
...  .country_live
...  .value_counts()
...  .median())
33.0
```

With our existing pandas knowledge, we could calculate the median size and then filter out countries below those sizes:

```
>>> countries_to_remove = (jb2
...   .country_live
...   .value_counts()
...   .lt(330)
```

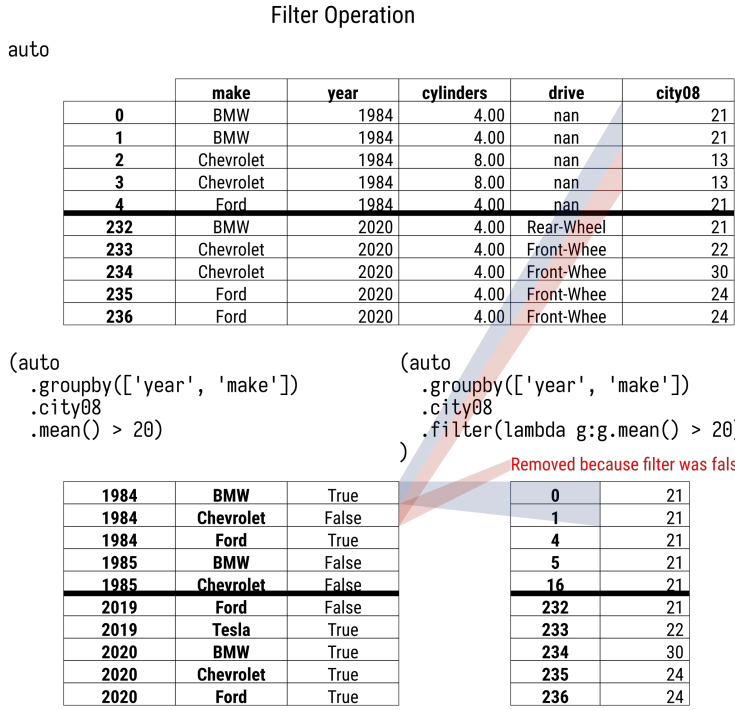
```
...   .pipe(lambda ser: ser[ser])
...   .index)
```

Here is the result. Note that the index values are skipping, hinting that some filtering is going on:

```
>>> print(jb2
...   .query('~country_live.isin(@countries_to_remove)')
... )
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...             3.0         3.6
11      21                  True  ...             3.0         3.9
15      50                 False  ...            11.0         3.6
...    ...                   ...  ...             ...         ...
54442   50                  True  ...            11.0         3.6
54447   30                 False  ...             3.0         3.6
54450   30                 False  ...            11.0         3.8

[2915 rows x 20 columns]
```

The `.filter` method of the groupby object makes the previous few lines a single operation. The `.filter` method accepts a function that takes the current group. If the function returns `True` (it must return a scalar, not a series or dataframe), the rows are kept for the result:

```
>>> print(jb2
...   .groupby('country_live')
...   .filter(lambda g: g.country_live.size >=330)
... )
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...             3.0         3.6
11      21                  True  ...             3.0         3.9
15      50                 False  ...            11.0         3.6
...    ...                   ...  ...             ...         ...
54442   50                  True  ...            11.0         3.6
54447   30                 False  ...             3.0         3.6
54450   30                 False  ...            11.0         3.8

[2915 rows x 20 columns]
```



	make	year	cylinders	drive	city08
0	BMW	1984	4.00	nan	21
1	BMW	1984	4.00	nan	21
2	Chevrolet	1984	8.00	nan	13
3	Chevrolet	1984	8.00	nan	13
4	Ford	1984	4.00	nan	21
232	BMW	2020	4.00	Rear-Wheel	21
233	Chevrolet	2020	4.00	Front-Whee	22
234	Chevrolet	2020	4.00	Front-Whee	30
235	Ford	2020	4.00	Front-Whee	24
236	Ford	2020	4.00	Front-Whee	24

1984	BMW	True
1984	Chevrolet	False
1984	Ford	True
1985	BMW	False
1985	Chevrolet	False
2019	Ford	False
2019	Tesla	True
2020	BMW	True
2020	Chevrolet	True
2020	Ford	True

0	21
1	21
4	21
5	21
16	21
232	21
233	22
234	30
235	24
236	24

Figure 30.2: The `.filter` method allows us to filter in terms of the original data based on aggregations on groups.

Table 30.2: Chapter Groupby Methods

Method	Description
`g.filter(func, dropna=True, *args, **kwargs)`	Return the original dataframe but with filtered groups removed. `func` is a predicate function that accepts a group and returns `True` to keep values from the group. If `dropna=False`, groups that evaluate to `False` are filled with `NaN`.
`g.transform( func, *args, **kwargs)`	Return a dataframe with the original index. The function will get passed a group and should return a dataframe with the same dimensions as the group.

30.3 Summary

You often group and aggregate but want to get the result in terms of the original index, not the aggregated index. The `.transform` method will allow you to preserve the original index. If you want to filter based on aggregated data but keep the original index (sans filtered rows), use the `.filter` method on the groupby object.

30.4 Exercises

With a dataset of your choice:

1. Add a new column, the sum of a numeric column grouped by a string column.
2. Filter out the rows with less than three entries when grouped by a string column.

Chapter 31

Cross-tabulation Deep Dive

You can emulate some groupby and pivot table actions with the `crosstab` function. (In fact, if you look at the source code for `crosstab`, you will see that it calls `.pivot_table` under the covers. And `.pivot_table` calls `.groupby` under the covers!)

Let's explore some more of the cross-tabulation functionality using the Presidential data.

31.1 Cross-tabulation Summaries

Using the JetBrains dataset, let us summarize the count of respondents by country and age:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)

>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age))
age           18  21  ...  50  60
country_live          ...
Algeria        0   4  ...   0   0
Argentina      0  18  ...   3   0
Armenia        1   8  ...   0   1
...           ..  ..  ...  ..  ..
Uzbekistan     0   3  ...   0   0
Venezuela      0   5  ...   1   0
Viet Nam       2  14  ...   0   1

[76 rows x 6 columns]
```

31.2 Adding Margins

Both `.pivot_table` and `crosstab` have a `margins` parameter that will put in a column and row at the right and bottom, respectively, that summarize the data:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     margins=True))
age            18    21  ...  60   All
country_live              ...
Algeria         0     4  ...   0     8
Argentina       0    18  ...   0    58
Armenia         1     8  ...   1    10
...           ...   ...  ...  ..   ...
Venezuela       0     5  ...   0    12
Viet Nam        2    14  ...   1    21
All           203  2925  ...  77  6980

[77 rows x 7 columns]
```

31.3 Normalizing Results

The `crosstab` function has another parameter, `normalize`, that will calculate the percentage of each cell:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize=True))
age                18        21  ...        50        60
country_live                     ...
Algeria       0.000000  0.000573  ...  0.000000  0.000000
Argentina     0.000000  0.002579  ...  0.000430  0.000000
Armenia       0.000143  0.001146  ...  0.000000  0.000143
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  0.000430  ...  0.000000  0.000000
Venezuela     0.000000  0.000716  ...  0.000143  0.000000
Viet Nam      0.000287  0.002006  ...  0.000000  0.000143

[76 rows x 6 columns]
```

You can also normalize down the columns or across the rows. (This seems backward compared to most `axis` operations to me as specifying `'columns'` normally means to apply the operation across the columns axis.) Here, we normalize each column to sum to one:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize='columns'))
```

```
age                18        21  ...        50        60
country_live                     ...
Algeria       0.000000  0.001368  ...  0.000000  0.000000
Argentina     0.000000  0.006154  ...  0.009404  0.000000
Armenia       0.004926  0.002735  ...  0.000000  0.012987
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  0.001026  ...  0.000000  0.000000
Venezuela     0.000000  0.001709  ...  0.003135  0.000000
Viet Nam      0.009852  0.004786  ...  0.000000  0.012987

[76 rows x 6 columns]
```

If you normalize by `'index'`, every row will sum up to 1.0:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize='index'))
age                18        21  ...        50        60
country_live                     ...
Algeria       0.000000  0.500000  ...  0.000000  0.000000
Argentina     0.000000  0.310345  ...  0.051724  0.000000
Armenia       0.100000  0.800000  ...  0.000000  0.100000
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  1.000000  ...  0.000000  0.000000
Venezuela     0.000000  0.416667  ...  0.083333  0.000000
Viet Nam      0.095238  0.666667  ...  0.000000  0.047619

[76 rows x 6 columns]
```

31.4 Hierarchical Columns with Cross Tabulations

In addition, we can create hierarchical indices and columns with `crosstab`. Let's look at the breakdown of country and age by where people use Python and Python versions and then focus on the United States:

```
>>> print(pd.crosstab(index=[jb2.country_live, jb2.age],
...     columns=[jb2.use_python_most, jb2.python3_version_most])
...  .loc[['United States']]
... )
use_python_most      Computer graphics             ...  \
python3_version_most        Python 3_6 Python 3_7  ...
country_live  age                                  ...
United States 18                     0         0  ...
              21                     0         0  ...
              30                     0         0  ...
              40                     0         0  ...
              50                     0         0  ...
```

```
              60                    0          0  ...

use_python_most      Web development
python3_version_most      Python 3_8 Python 3_9
country_live  age
United States 18                   1          0
              21                  33          3
              30                  79          6
              40                  29          5
              50                   8          0
              60                   2          1

[6 rows x 81 columns]
```

Let's dive in a little more and look at data analysis and web development:

```
>>> print(pd.crosstab(index=[jb2.country_live, jb2.age],
...     columns=[jb2.use_python_most, jb2.python3_version_most])
...  .loc[['United States'], ['Data analysis', 'Web development']]
... )
use_python_most            Data analysis             ...  \
python3_version_most Python 3_5 or lower Python 3_6  ...
country_live  age                                    ...
United States 18                       0          0  ...
              21                       1         12  ...
              30                       1         12  ...
              40                       0         10  ...
              50                       2          3  ...
              60                       0          1  ...

use_python_most      Web development
python3_version_most      Python 3_8 Python 3_9
country_live  age
United States 18                   1          0
              21                  33          3
              30                  79          6
              40                  29          5
              50                   8          0
              60                   2          1

[6 rows x 10 columns]
```

31.5 Heatmaps

Let me show you one more trick. Remember how I said humans aren't optimized for pulling out the parts that stand out? I like to add some

visualizations to make this pop. I'm going to color the background (this works great in Jupyter, if I needed to generate a plot, I would use Seaborn's `heatmap` function). I will use the `.style` attribute to change the background gradient:

```
(pd.crosstab(index=[jb2.country_live, jb2.age],
    columns=[jb2.use_python_most, jb2.python3_version_most])
 .loc[['United States'], ['Data analysis', 'Web development']]
 .style.background_gradient(cmap='viridis', axis=None)
)
```

This makes it clear that in this data, Python 3.8 is the most popular, as is age 30.

```
(pd.crosstab([jb2.country_live, jb2.age],  [jb2.use_python_most, jb2.python3_version_most])
 .loc[['United States'], ['Data analysis', 'Web development']]
 .style.background_gradient(cmap='viridis', axis=None)
)
```

	use_python_most	Data analysis					Web development			
	python3_version_most	3.5	3.6	3.7	3.8	3.9	3.5	3.6	3.7	3.8	3.9
country_live	age										
United States	18.0	0	0	1	5	0	0	0	1	4	0
	21.0	1	18	48	64	11	3	28	54	81	4
	30.0	3	29	66	90	12	3	60	77	129	14
	40.0	0	14	30	45	3	1	14	26	66	8
	50.0	2	6	26	36	1	0	10	12	14	2
	60.0	0	1	11	11	1	0	3	5	5	1

Figure 31.1: Jupyter showing a view of dataframe with a heatmap. This pulls attention to versions and ages that are most common.

Table 31.1: Chapter Methods

Method	Description
`pd.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)`	Create a cross-tabulation (counts by default) from `index` (series or list of series) and `columns` (series or list of series). Can specify a column (series) to aggregate `values` along with a function, `aggfunc`. Using `margins=True` will add subtotals. Using `dropna=False` will keep columns that have no values. Can normalize over `'all'` values, the rows (`'index'`), or the `'columns'`.
`.style .background_gradient( cmap='PuBu', low=0, high=0, axis=0, subset=None, text_color_threshold= 0.408, vmin=None, vmax=None, gmap=None)`	Color a dataframe in Jupyter with Matplotlib colormap (`cmap`). Specify the ends of the color map with `vmin` and `cmax`. If `axis=None` apply to the whole dataframe.

31.6 Summary

We could live in a world without the `pd.crosstab` function. However, for certain operations, it is much more convenient than `.groupby` or `.pivot_table`. If you master this function, you can quickly summarize categoricals.

31.7 Exercises

With a dataset of your choice:

1. Summarize the count of one categorical column against another.
2. Summarize the count of one categorical column against another, adding margins.
3. Summarize the count of one categorical column against another in a heatmap.

Chapter 32

Melting, Transposing, and Stacking Data

We have shown a lot of ways to manipulate a dataframe. But we are not done yet. This chapter will show some of the more complicated operations you can do to a dataframe to bend the data to your will. You probably will not use these operations often but will be grateful they are around when needed.

32.1 Melting Data

Another transformation we can do to data is "melt" it. Before looking at the method to melt data, let's discuss the structure of data. Two ways to organize the same data are "wide" (also called *stacked* or *record* form) and "long" (sometimes called *tidy* form) data. (Note that this differs from "big data", which refers to the amount of data.)

An OLAP database is an analytical database optimized for reporting. In OLAP terms, there is a notion of a *fact* and a *dimension*. A fact is a value that is measured and reported on, and a dimension is a value that describes the conditions of the fact. There are often multiple dimensions for a fact. In a sales scenario, typical facts would be the number of sales of an item and the cost. The dimensions might include the store where the item was sold, the date, and the customer.

The dimensions can then be *sliced* to explore the data. We might want to view sales by store. A dimension may be hierarchical; a store could have a region, zip code, or state, and we could view sales by any of those dimensions.

Here is data that tracks students' ages and scores. The test columns are fact columns, and the other columns are dimensions:

name	age	test1	test2	teacher
Adam	15	95	80	Ashby
Bob	16	81	82	Ashby
Dave Fred	16 15	89	84 88	Jones Jones

The scores data is in a *wide format*. This is in contrast to a *long format*, where each row contains a single fact (with perhaps other variables describing the dimensions). If we consider test scores to be a fact, this wide-format has more than one fact in a row. Hence, it is wide.

Often, tools require that data be stored in a long format and only have one fact per row. This format is *denormalized* and repeats many of the dimensions but may make analysis easier.

One long version of our scores looks like this (note that we dropped teacher information):

name	age	test	score
Adam	15	test1	95
Bob	16	test1	81
Dave	16	test1	89
Fred	15	test1	NaN
Adam	15	test2	80
Bob	16	test2	82
Dave	16	test2	84
Fred	15	test2	88

Melting Data

```
scores
```

	name	age	test1	test2	teacher
0	Adam	15	95.00	80	Ashby
1	Bob	16	81.00	82	Ashby
2	Suzy	16	89.00	94	Jones
3	Fred	15	nan	88	Jones

```
pd.melt(scores, id_vars=['name', 'age'],
        value_vars=['test1', 'test2'])
```

	name	age	variable	value
0	Adam	15	test1	95.00
1	Bob	16	test1	81.00
2	Suzy	16	test1	89.00
3	Fred	15	test1	nan
4	Adam	15	test2	80.00
5	Bob	16	test2	82.00
6	Suzy	16	test2	94.00
7	Fred	15	test2	88.00

Figure 32.1: Melting data with pandas. Melting allows you to stack columns on top of each other.

Let's show how to convert wide data to long data. We will start by creating a dataframe with scores:

```
>>> import io
>>> data = '''name,age,test1,test2,teacher
... Adam,15,95.0,80,Ashby
... Bob,16,81.0,82,Ashby
... Dave,16,89.0,84,Jones
... Fred,15,,88,Jones'''
>>> scores = pd.read_csv(io.StringIO(data), dtype_backend='pyarrow')

>>> print(scores)
   name  age  test1  test2 teacher
0  Adam   15   95.0     80   Ashby
1   Bob   16   81.0     82   Ashby
2  Dave   16   89.0     84   Jones
3  Fred   15   <NA>     88   Jones
```

Right now, the score for each test is in its own column. If we wanted to calculate the average of all of the tests, it would require some work to pull out all of the test score columns, stack them, and calculate the mean. Let's melt the data and put it into long form. Below, we keep the name and age as dimensions, and pull out the test scores as facts:

```
>>> print(scores.melt(id_vars=['name', 'age'],
...         value_vars=['test1', 'test2']))
   name  age variable  value
0  Adam   15    test1   95.0
1   Bob   16    test1   81.0
2  Dave   16    test1   89.0
3  Fred   15    test1   <NA>
4  Adam   15    test2   80.0
5   Bob   16    test2   82.0
6  Dave   16    test2   84.0
7  Fred   15    test2   88.0
```

Using techniques that we have learned, we can accomplish this by building up a chain. But the `.melt` method is a friendly convenience method. Here is the hand-rolled non-melt version:

```
>>> print(scores
...   .groupby(['name', 'age'])
...   .apply(lambda g: pd.concat([
...     g[['test1']].rename(columns={'test1':'val'}).assign(var='test1'),
...     g[['test2']].rename(columns={'test2':'val'}).assign(var='test2')]))
...   .reset_index()
...   .drop(columns='level_2')
```

```
... )
   name  age   val    var
0  Adam   15  95.0  test1
1  Adam   15  80.0  test2
2   Bob   16  81.0  test1
3   Bob   16  82.0  test2
4  Dave   16  89.0  test1
5  Dave   16  84.0  test2
6  Fred   15  <NA>  test1
7  Fred   15  88.0  test2
```

As you can see, the melt version is much easier to create.

If we want to change the description of the fact column to a more descriptive name, pass that as the `var_name` parameter. We can change the name of the value of the column (it defaults to `value`) by providing a `value_name` parameter. Here, we change the description to *test* and the value to *score*:

```
>>> print(scores.melt(id_vars=['name', 'age'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score'))
   name  age   test  score
0  Adam   15  test1   95.0
1   Bob   16  test1   81.0
2  Dave   16  test1   89.0
3  Fred   15  test1   <NA>
4  Adam   15  test2   80.0
5   Bob   16  test2   82.0
6  Dave   16  test2   84.0
7  Fred   15  test2   88.0
```

If we want to preserve the teacher information, we would need to include it in the `id_vars` parameter:

```
>>> print(scores.melt(id_vars=['name', 'age', 'teacher'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score'))
   name  age teacher   test  score
0  Adam   15   Ashby  test1   95.0
1   Bob   16   Ashby  test1   81.0
2  Dave   16   Jones  test1   89.0
3  Fred   15   Jones  test1   <NA>
4  Adam   15   Ashby  test2   80.0
5   Bob   16   Ashby  test2   82.0
6  Dave   16   Jones  test2   84.0
7  Fred   15   Jones  test2   88.0
```

Note

Long data is also referred to as *tidy* data. See the Tidy Data paper[a] by Hadley Wickham.

[a] http://vita.had.co.nz/papers/tidy-data.html

32.2 Un-melting Data

We can go from a long to a wide format using a pivot table. Here is our melted data from the previous section:

```
>>> melted = scores.melt(id_vars=['name', 'age', 'teacher'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score')
>>> print(melted)
   name  age teacher   test  score
0  Adam   15   Ashby  test1   95.0
1   Bob   16   Ashby  test1   81.0
2  Dave   16   Jones  test1   89.0
3  Fred   15   Jones  test1   <NA>
4  Adam   15   Ashby  test2   80.0
5   Bob   16   Ashby  test2   82.0
6  Dave   16   Jones  test2   84.0
7  Fred   15   Jones  test2   88.0
```

It is a little more involved going in the reverse direction because we will put the id variables that we kept from the original data in a hierarchical index. I generally flatten hierarchical indices with the `.reset_index` method. You can use `.pivot_table` or `.groupby` to do this:

```
>>> print(melted
...   .pivot_table(index=['name','age', 'teacher'],
...                columns='test', values='score')
...   .reset_index())
test  name  age teacher  test1  test2
0     Adam   15   Ashby   95.0   80.0
1      Bob   16   Ashby   81.0   82.0
2     Dave   16   Jones   89.0   84.0
3     Fred   15   Jones   <NA>   88.0

>>> print(melted
...   .groupby(['name', 'age', 'teacher', 'test'])
...   .score
...   .mean()
...   .unstack()
...   .reset_index()
```

```
... )
test  name  age teacher  test1  test2
0     Adam   15   Ashby   95.0   80.0
1      Bob   16   Ashby   81.0   82.0
2     Dave   16   Jones   89.0   84.0
3     Fred   15   Jones   <NA>   88.0
```

Undoing Melting

```
melted
```

	name	age	variable	value
0	Adam	15	test1	95.00
1	Bob	16	test1	81.00
2	Suzy	16	test1	89.00
3	Fred	15	test1	nan
4	Adam	15	test2	80.00
5	Bob	16	test2	82.00
6	Suzy	16	test2	94.00
7	Fred	15	test2	88.00

```
(melted
 .pivot_table(index=['name', 'age'],
              columns='variable', values='value')
 .reset_index()
)
```

	name	age	test1	test2
0	Adam	15	95.00	80.00
1	Bob	16	81.00	82.00
2	Fred	15	nan	88.00
3	Suzy	16	89.00	94.00

Figure 32.2: Unmelting data with pandas. By pivoting the data, you can specify the label column (`columns`) for the stacked columns (`values`).

32.3 Pulling Out Categorical Values into Columns

Suppose we wanted to create a dataframe where every column was the scores for a teacher. This might be useful if we wanted to plot a histogram of the scores for each teacher. We can use the `.pivot` method to do this. The `.pivot` method differs from the `.pivot_table` method in that it requires the index and column values to be unique. The `.pivot_table` method is more flexible and can handle aggregation of duplicate values.

Here is the scores dataframe.

```
>>> print(scores)
   name  age  test1  test2 teacher
```

```
0  Adam   15   95.0     80   Ashby
1   Bob   16   81.0     82   Ashby
2  Dave   16   89.0     84   Jones
3  Fred   15   <NA>     88   Jones
```

We can create a new dataframe where the columns are the teachers, and the values are the scores. We will use the `.pivot` method and pass the `teacher` column as the columns and the test score columns as the values:

```
>>> print(scores
...  .pivot(columns='teacher', values=['test1', 'test2'])
... )
        test1       test2
teacher Ashby Jones Ashby Jones
0        95.0   NaN    80   NaN
1        81.0   NaN    82   NaN
2         NaN  89.0   NaN    84
3         NaN  <NA>   NaN    88
```

Note that this returns a sparse dataframe. Also note that as of pandas 2.2, this returns `object` columns [1]. We will address this using the `.apply` method to pack each column and convert the types.

```
>>> print(scores
...  .pivot(columns='teacher', values=['test1', 'test2'])
...  .apply(lambda ser: ser
...             [~ser.isna()]
...             .reset_index(drop=True)
...             .astype('int64[pyarrow]')
...         )
... )
        test1       test2
teacher Ashby Jones Ashby Jones
0          95    89    80    84
1          81  <NA>    82    88
```

If we want to combine all of the scores for each teacher, we can use the same technique with the pivoted data.

```
>>> def pack_as_int(ser):
...     return (ser[~ser.isna()].reset_index(drop=True)
...             .astype('int64[pyarrow]'))

>>> print(melted
...  .pivot(columns='teacher', values='score')
```

[1] https://github.com/pandas-dev/pandas/issues/43547

```
...  .apply(pack_as_int)
... )

teacher  Ashby  Jones
0           95     89
1           81     84
2           80     88
3           82   <NA>
```

32.4 Transposing Data

We have been exploring reshaping data. We have already seen and used a common method to reshape data, the `.transpose` method or the `.T` property. Remember, this flips rows and columns.

I find that I use transposition mostly in two places:

- Viewing more data in Jupyter
- Swapping axis for plotting

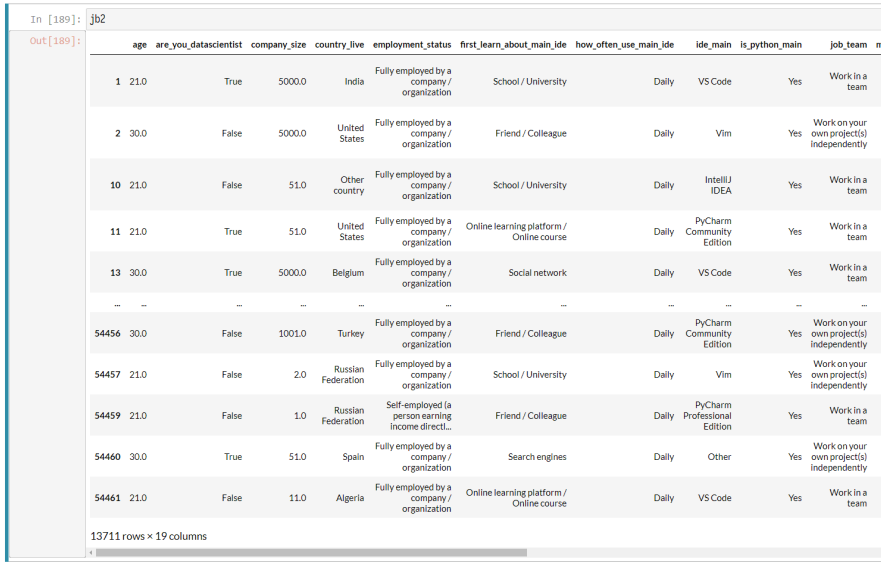
Transposition often works for viewing more data because pandas uses numeric index values by default. When the numeric index goes into the column, it takes up less horizontal space, and you can see more data without having to scroll around.

I have some thoughts on viewing data. Often, when I teach, a student will ask how to turn off the default behavior of pandas in Jupyter to only show a limited number of rows and columns. (You can change `pd.options.display.max_columns` and `pd.options.display.min_rows` to modify these if you really want to.) I generally try to dissuade them from changing these settings.

However, if you change these settings to view more data and find yourself scrolling through a million rows of data, your spidey sense should go off, telling you that you are doing things incorrectly. Humans are not made to look for interesting data by scrolling through rows of data. It is better to use a computer (which is optimized to search through data) to find rows you might be interested in. My two favorite methods of leveraging a computer to search for us are visualization and filtering the data.

On that note, if you use the `.transpose` method to view more data on your screen, you might not want to transpose your whole data set. Remember that pandas stores and optimizes data by column types. If you make a row that contains different data types (strings, dates, numbers) into a column that can be a slow and memory-loving operation. It is better to pull off the head and tail or take a sample of the data and then transpose it.

When we explored line plots in the plotting section, we showed an example of transposing the data. We had a presidential data set with the president's names in the index and ratings for various skills in the columns. When we did a line plot of this data, each characteristic was its own line.



In [189]: jb2

Out[189]:

	age	are_you_datascientist	company_size	country_live	employment_status	first_learn_about_main_ide	how_often_use_main_ide	ide_main	is_python_main	job_team	mai
1	21.0	True	5000.0	India	Fully employed by a company / organization	School / University	Daily	VS Code	Yes	Work in a team	Bc a
2	30.0	False	5000.0	United States	Fully employed by a company / organization	Friend / Colleague	Daily	Vim	Yes	Work on your own project(s) independently	Bc a
10	21.0	False	51.0	Other country	Fully employed by a company / organization	School / University	Daily	IntelliJ IDEA	Yes	Work in a team	Bc a
11	21.0	True	51.0	United States	Fully employed by a company / organization	Online learning platform / Online course	Daily	PyCharm Community Edition	Yes	Work in a team	Bc a
13	30.0	True	5000.0	Belgium	Fully employed by a company / organization	Social network	Daily	VS Code	Yes	Work in a team	Bc a
...	...	...	...	...	...	...	...	...	...	...	...
54456	30.0	False	1001.0	Turkey	Fully employed by a company / organization	Friend / Colleague	Daily	PyCharm Community Edition	Yes	Work on your own project(s) independently	Bc a
54457	21.0	False	2.0	Russian Federation	Fully employed by a company / organization	School / University	Daily	Vim	Yes	Work on your own project(s) independently	Bc a
54459	21.0	False	1.0	Russian Federation	Self-employed (a person earning income directl...	Friend / Colleague	Daily	PyCharm Professional Edition	Yes	Work in a team	Bc a
54460	30.0	True	51.0	Spain	Fully employed by a company / organization	Search engines	Daily	Other	Yes	Work on your own project(s) independently	Bc a
54461	21.0	False	11.0	Algeria	Fully employed by a company / organization	Online learning platform / Online course	Daily	VS Code	Yes	Work in a team	Bc a

13711 rows × 19 columns

Figure 32.3: Jupyter showing default view of dataframe. We have ten rows but need to scroll to see all of the data.

Instead, we wanted each president to be its own line, so we transposed the data.

32.5 Stacking & Unstacking

I have previously used the `.unstack` method but have not discussed it. It (along with its complement, `.stack`) is a powerful method for reshaping your data.

At a high level, `.unstack` moves an index level into the columns. Usually, we use this operation on multi-index data, moving one of the indices into the columns (creating hierarchical columns). The `.stack` method does the reverse, moving a multi-level column into the index.

Let's look at an example using the JetBrains data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)

>>> print(jb2)
       age are_you_datascientist  ...  years_of_coding python3_ver
```

```
In [190]: (jb2
           .head(10)
           .T
          )
```

Out[190]:

	1	2	10	11	13	14	15	17	22	25
age	21.0	30.0	21.0	21.0	30.0	30.0	50.0	30.0	40.0	50.0
are_you_datascientist	True	False	False	True	True	True	False	True	False	True
company_size	5000.0	5000.0	51.0	51.0	5000.0	501.0	1001.0	2.0	51.0	11.0
country_live	India	United States	Other country	United States	Belgium	Ecuador	Germany	Chile	Australia	United States
employment_status	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization
first_learn_about_main_ide	School / University	Friend / Colleague	School / University	Online learning platform / Online course	Social network	Other	Friend / Colleague	Social network	Technical review / Forum / Blog	Search engines
how_often_use_main_ide	Daily	Daily	Daily	Daily	Daily	Weekly	Daily	Daily	Daily	Daily
ide_main	VS Code	Vim	IntelliJ IDEA	PyCharm Community Edition	VS Code	VS Code	Vim	VS Code	VS Code	PyCharm Professional Edition
is_python_main	Yes	Yes	Yes	Yes	Yes	Yes	No, I use Python as a secondary language	Yes	No, I use Python as a secondary language	Yes
job_team	Work in a team	Work on your own project(s) independently	Work in a team	Work in a team	Work in a team	Work on your own project(s) independently	Work in a team	Work on your own project(s) independently	Work in a team	Work on your own project(s) independently
main_purposes	Both for work and personal	Both for work and personal	Both for work and personal	Both for work and personal	Both for work and personal	For work	For work	Both for work and personal	Both for work and personal	Both for work and personal
missing_features_main_ide	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	Yes - Please list:	No, it has all the features I need	No, it has all the features I need	Yes - Please list:
nps_main_ide	8.0	10.0	10.0	9.0	10.0	10.0	5.0	10.0	10.0	9.0
python_years	3.0	3.0	1.0	3.0	6.0	3.0	1.0	1.0	6.0	11.0
python3_version_most	3.6	3.6	3.8	3.9	3.7	3.8	3.6	3.8	3.7	3.8
several_projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects
team_size	2	5	2	2	2	5	2	0	2	2
use_python_most	Software prototyping	DevOps / System administration / Writing autom...	Web development	Data analysis	Data analysis	Programming of web parsers / scrapers / crawlers	Web development	Machine learning	Software prototyping	Data analysis
years_of_coding	3.0	3.0	1.0	3.0	3.0	3.0	11.0	1.0	11.0	11.0

Figure 32.4: Jupyter showing a transposed view of dataframe. Notice that we see ten complete samples of data showing on the screen without scrolling.

```
1       21                True  ...          3.0          3.6
5       21               False  ...          3.0          3.8
10      21               False  ...          1.0          3.8
11      21                True  ...          3.0          3.9
13      30                True  ...          3.0          3.7
...    ...                 ...  ...          ...          ...
54432   21                True  ...          0.5          3.8
54433   30               False  ...         11.0          3.6
54442   50                True  ...         11.0          3.6
54447   30               False  ...          3.0          3.6
54450   30               False  ...         11.0          3.8

[6980 rows x 20 columns]
```

We will create a hierarchical or multi-index by grouping with multiple columns. Let's take the size of responses to *are_you_datascientist* column by country:

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
... )
country_live  are_you_datascientist
Algeria       False                   5
              True                    3
Argentina     False                  48
              True                   10
Armenia       False                   8
                                     ..
Uzbekistan    True                    1
Venezuela     False                   8
              True                    4
Viet Nam      False                  13
              True                    8
Length: 152, dtype: int64
```

Notice that the result is a series with a multi-index. This result is useful but a little hard to scan through. It would be easier if we had countries in the index and each of the responses to *are_you_datascientist* as their own column. We can do that by unstacking the inner index into a column (note that you could also do this operation with `pd.crosstab`):

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack()
... )
are_you_datascientist  False  True
country_live
Algeria                    5      3
Argentina                 48     10
Armenia                    8      2
Australia                104     35
Austria                   42     19
...                      ...    ...
United States            976    346
Uruguay                    8      6
Uzbekistan                 2      1
Venezuela                  8      4
Viet Nam                  13      8

[76 rows x 2 columns]
```

By default, `.unstack` moves the inner index up to the columns. Because this operation was performed on a series, it is changed to a dataframe. (If

we perform `.unstack` on a dataframe, we will get a dataframe with nested columns.)

If we wanted to pull up the country index (the outer index), we could specify it by name or position. The position is 0 for the outer index, *country_live*, and 1 for *are_you_datascientist*:

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack(0)
... )
country_live           Algeria  Argentina  ...  Venezuela  Viet Nam
are_you_datascientist                      ...
False                        5         48  ...          8        13
True                         3         10  ...          4         8

[2 rows x 76 columns]
```

I would prefer to use the index name (rather than the index position) in this case as it is easier to understand (and one less thing you need to memorize):

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack('country_live')
... )
country_live           Algeria  Argentina  ...  Venezuela  Viet Nam
are_you_datascientist                      ...
False                        5         48  ...          8        13
True                         3         10  ...          4         8

[2 rows x 76 columns]
```

32.6 Stacking

Let's look at stacking. Previously, we saw that we could specify multiple aggregation functions with the `.pivot_table` method. The result is a dataframe with hierarchical columns:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'company_size': ['min', 'max']})
... )
              age      company_size
```

```
              max min          max min
country_live
Algeria        30  21           11   1
Argentina      50  21         5000   1
Armenia        60  18         5000   1
Australia      60  18         5000   1
Austria        50  21         5000   1
...            ..  ..          ...  ..
United States  60  18         5000   1
Uruguay        30  21         5000   2
Uzbekistan     21  21           51   1
Venezuela      50  21           51   1
Viet Nam       60  18         1001   1

[76 rows x 4 columns]
```

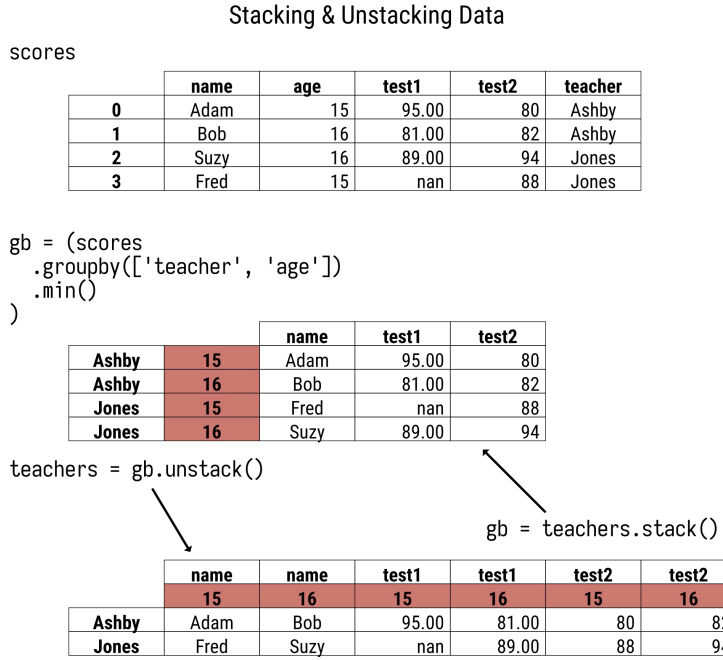


Figure 32.5: Stacking and unstacking data with pandas. Stacking puts column labels into the index. Unstacking moves index labels into columns.

In a previous example, we saw that we could unstack the index by the name of the index (the name of the column before it was put in the index) or by the position. In this example, we want to stack one of the hierarchical columns into the index. The columns do not have names, so we must use the position. The outermost column level is 0. Stacking by this level will move *age* and *company_size* into the index:

```
>>> print(jb2
...   .pivot_table(index='country_live',
...                aggfunc={'age': ['min', 'max'],
...                         'company_size': ['min', 'max']})
...   .stack(0)
... )
                           max  min
country_live
Algeria      age            30   21
             company_size   11    1
Argentina    age            50   21
             company_size  5000    1
Armenia      age            60   18
...                        ...  ...
Uzbekistan   company_size   51    1
Venezuela    age            50   21
             company_size   51    1
Viet Nam     age            60   18
             company_size  1001    1

[152 rows x 2 columns]
```

If we want to move the inner columns, *max* and *min*, into the index, this is the default behavior. Alternatively, we can specify level 1 as an argument for `.stack`:

```
>>> print(jb2
...   .pivot_table(index='country_live',
...                aggfunc={'age': ['min', 'max'],
...                         'company_size': ['min', 'max']})
...   .stack(1)
... )
                  age  company_size
country_live
Algeria      max   30            11
             min   21             1
Argentina    max   50          5000
             min   21             1
Armenia      max   60          5000
```

```
...                ...           ...
Uzbekistan   min   21              1
Venezuela    max   50             51
             min   21              1
Viet Nam     max   60           1001
             min   18              1

[152 rows x 2 columns]
```

Finally, if you want to change the order of the levels in a hierarchical index or columns, you can use the `.swaplevel` method:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'company_size': ['min', 'max']})
...  .stack(1)
...  .swaplevel()
... )
                  age  company_size
    country_live
max Algeria        30            11
min Algeria        21             1
max Argentina      50          5000
min Argentina      21             1
max Armenia        60          5000
...               ...           ...
min Uzbekistan     21             1
max Venezuela      50            51
min Venezuela      21             1
max Viet Nam       60          1001
min Viet Nam       18             1

[152 rows x 2 columns]
```

32.7 Flattening Hierarchical Indexes and Columns

When you start applying grouping operations, you can end up with a hierarchical index or columns. In practice, I find these nested structures challenging to deal with and often want to remove (or flatten them).

Let's start by discussing removing the hierarchical index, which is simple. We use the `.reset_index` method. Here is a dataframe with a hierarchical index:

```
>>> print(jb2
...  .groupby(['country_live', 'age'])
```

```
...  .mean(numeric_only=True)
... )
                  company_size  nps_main_ide  python_years  \
country_live age
Algeria      18           <NA>          <NA>          <NA>
             21           3.75          7.25           1.0
             30            1.5          10.0          2.75
             40           <NA>          <NA>          <NA>
             50           <NA>          <NA>          <NA>
...                        ...           ...           ...
Viet Nam     21     133.142857      8.928571           2.5
             30       7.666667      9.333333      2.333333
             40           51.0           9.0           3.0
             50           <NA>          <NA>          <NA>
             60            1.0           8.0           3.0

                  team_size  years_of_coding
country_live age
Algeria      18        <NA>             <NA>
             21         1.5            0.875
             30         1.5            2.625
             40        <NA>             <NA>
             50        <NA>             <NA>
...                     ...              ...
Viet Nam     21    5.571429         1.464286
             30    1.666667         4.166667
             40         2.0              6.0
             50        <NA>             <NA>
             60         1.0              1.0

[456 rows x 5 columns]
```

We can use `.reset_index` to push each index level into a column:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
...  .mean(numeric_only=True)
...  .reset_index()
... )
    country_live  age  ...  team_size  years_of_coding
0        Algeria   21  ...        1.5            0.875
1        Algeria   30  ...        1.5            2.625
2      Argentina   21  ...   5.333333         2.916667
3      Argentina   30  ...   3.814815              5.0
4      Argentina   40  ...        3.6             8.15
..           ...  ...  ...        ...              ...
```

```
308     Viet Nam   18  ...        7.5            0.75
309     Viet Nam   21  ...   5.571429        1.464286
310     Viet Nam   30  ...   1.666667        4.166667
311     Viet Nam   40  ...        2.0             6.0
312     Viet Nam   60  ...        1.0             1.0

[313 rows x 7 columns]
```

Alternatively, when using `.groupby`, you can set the `as_index` parameter to `False`, and the result does not insert the grouping columns in the index, they will stay as columns:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], as_index=False, observed=True)
...  .mean(numeric_only=True)
... )
    country_live  age  ...  team_size  years_of_coding
0         Algeria   21  ...        1.5            0.875
1         Algeria   30  ...        1.5            2.625
2       Argentina   21  ...   5.333333        2.916667
3       Argentina   30  ...   3.814815             5.0
4       Argentina   40  ...        3.6             8.15
..            ...  ...  ...        ...              ...
308      Viet Nam   18  ...        7.5            0.75
309      Viet Nam   21  ...   5.571429        1.464286
310      Viet Nam   30  ...   1.666667        4.166667
311      Viet Nam   40  ...        2.0             6.0
312      Viet Nam   60  ...        1.0             1.0

[313 rows x 7 columns]
```

Now, let's explore flattening hierarchical columns. Sadly, the `.reset_index` method won't work for the column names. Generally, we don't want to push the column names into a row, but we want to combine them into a single level of column names. And there is no convenient method to do that in pandas.

Here is an example of data with a hierarchical column. For every country we have the mean values for each numeric column broken down by age:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
...  .mean(numeric_only=True)
...  .unstack()
... )
             company_size                ... years_of_coding  \
age                    18          21    ...             50
country_live                             ...
```

```
Algeria              <NA>        3.75  ...           <NA>
Argentina            <NA>  566.055556  ...           11.0
Armenia              11.0       761.0  ...           <NA>
Australia             6.0  752.862069  ...      10.583333
Austria              <NA>  222.294118  ...           11.0
...                   ...         ...  ...            ...
United States       693.0 1745.711688  ...      10.557692
Uruguay              <NA>        21.0  ...           <NA>
Uzbekistan           <NA>   17.666667  ...           <NA>
Venezuela            <NA>        35.0  ...            6.0
Viet Nam             51.0  133.142857  ...           <NA>


age                    60
country_live
Algeria              <NA>
Argentina            <NA>
Armenia               0.5
Australia       9.333333
Austria              <NA>
...                   ...
United States  10.381579
Uruguay              <NA>
Uzbekistan           <NA>
Venezuela            <NA>
Viet Nam              1.0

[76 rows x 30 columns]
```

In addition to the lack of a convenient method to flatten columns being a gaping hole in the pandas API, to add insult to injury, you have to mutate the dataframe to update the columns. Remember, mutation generally throws a wrench in our chaining operations.

To get around this, I make a function that will flatten columns. The function joins each level of columns with an underscore. Then, I combine that function with the `.pipe` method. This lets me do a column flattening operation in a chain:

```
>>> def flatten_cols(df):
...     cols = ['_'.join(map(str, vals))
...             for vals in df.columns.to_flat_index()]
...     df.columns = cols
...     return df

>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
```

Flattening Grouping Data with Multiple Aggregations

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
def flatten(df_):
    cols = ['_'.join(cs) for cs in df_.columns.to_flat_index()]
    df_.columns = cols
    return df_
(auto
 .groupby('make')
 .agg(['min', 'max'])
 .pipe(flatten))
```

	year_min	year_max	cylinders_min	cylinders_max
Acura	1986	2020	4.00	6.00
Audi	1984	2020	4.00	12.00
BMW	1984	2020	2.00	12.00
BYD	2012	2019	nan	nan
Bentley	1998	2019	8.00	12.00
VPG	2011	2013	8.00	8.00
Vector	1992	1997	8.00	12.00
Volvo	1984	2019	4.00	8.00
Yugo	1986	1990	4.00	4.00
smart	2008	2019	3.00	3.00

Figure 32.6: Grouping and then flattening hierarchical columns.

```
...  .mean(numeric_only=True)
...  .unstack()
...  .pipe(flatten_cols)
... )
              company_size_18  company_size_21  ...  \
country_live                                    ...
Algeria                  <NA>             3.75  ...
Argentina                <NA>       566.055556  ...
Armenia                  11.0            761.0  ...
Australia                 6.0       752.862069  ...
Austria                  <NA>       222.294118  ...
...                       ...              ...  ...
United States           693.0      1745.711688  ...
```

```
Uruguay                   <NA>            21.0  ...
Uzbekistan                <NA>       17.666667  ...
Venezuela                 <NA>            35.0  ...
Viet Nam                  51.0      133.142857  ...

              years_of_coding_50  years_of_coding_60
country_live
Algeria                      <NA>                <NA>
Argentina                    11.0                <NA>
Armenia                      <NA>                 0.5
Australia               10.583333            9.333333
Austria                      11.0                <NA>
...                           ...                 ...
United States           10.557692           10.381579
Uruguay                      <NA>                <NA>
Uzbekistan                   <NA>                <NA>
Venezuela                     6.0                <NA>
Viet Nam                     <NA>                 1.0

[76 rows x 30 columns]
```

Table 32.3: Chapter Methods

Method	Description
`.melt( id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None, ignore_index=True)`	Return an unpivoted dataframe. With each column in `value_vars` stack on top of each other. Keep the `id_vars` columns.
`g.transform(func, *args, **kwargs)`	Return a dataframe with original index. The function will get passed a group and should return a dataframe with same dimensions as group.
`pd.options .display.max_columns`	Property to set to configure pandas to show at most this amount of columns.
`pd.options .display.min_rows`	Property to set to configure pandas to show at most this amount of row.
`.stack(level=-1, dropna=True)`	Push a column level into an index level. Can specify the column `level` (-1 is innermost).
`.unstack(level=-1, dropna=True)`	Push an index level into a column level. Can specify an index `level` (-1 is innermost).

Method	Description
`.swaplevel(i=-2, j=-1, axis=0)`	Swap the levels of multi-indexed object (0 is outermost, -1 (or length of multi-index) is innermost). Can specify the name for `i` and `j`.
`.reset_index( level=None, drop=False, col_level=0, col_fill='')`	Return a dataframe with a new index (or new level). To remove a level, specify that with `level` (by position or name). Position 0 is the outermost level, and it goes up. Alternatively, -1 is the innermost level. Index values are moved to columns or dropped if `drop=True`. `col_level` determines where the index label goes with multiple column levels. Other levels will get the value of `col_fill`.
`.pipe(func, *args, **kwargs)`	Apply a function to a dataframe. Return the result of the function.

32.8 Summary

In this chapter, we showed how to melt and unmelt data. If you use the Seaborn library for plotting, you might need to transform your data so that you can plot with this library. We also explored stacking and unstacking data. Finally, we showed how to remove nested columns and indexes.

32.9 Exercises

With a dataset of your choice:

1. Melt two numeric column values into a single column. Add a new column to indicate what the values mean.
2. Un-melt the above.
3. Group by two columns, take the mean, and unstack the result.
4. Group by two columns, take the mean, unstack the result, and flatten the columns.

```
h
```

Chapter 33

Working with Time Series

This chapter will explore how to manipulate and work with time-series data. One thing to note, when we say "time-series", we are not talking about the pandas `Series` object, but rather data that has a date component. Often we will have that date component in the index of a pandas series or dataframe because that allows us to do time aggregations easily.

33.1 Loading the Data

For this section, I'm going to explore a dataset from the US Geologic Survey that deals with river flow of a river in Utah called the Dirty Devil river[1].

This data is a tab-delimited ASCII file in detail described here[2].

The columns are:

- *agency_cd* - Agency collecting data
- *site_no* - USGS identification number of site
- *datetime* - Date
- *tz_cd* - Timezone
- *144166_00060* - Discharge (cubic feet per second)
- *144166_00060_cd* - Status of discharge. "A" (approved), "P" (provisional), "e" (estimate).
- *144167_00065* - Gage height (feet)
- *144167_00065_cd* - Status of gage_height. "A" (approved), "P" (provisional), "e" (estimate).

Here is my code to load the data. I have also included a tweak function that converts the date information to actual dates and renames some columns.

[1] `https://nwis.waterdata.usgs.gov/usa/nwis/uv/?cb_00060=on&cb_00065=on-&format=rdb&site_no=09333500&period=&begin_date-=2000-01-01&end_date=2020-09-28`

[2] `https://help.waterdata.usgs.gov/faq/about-tab-delimited-output` Also, see this link for a description of the spelling of "gage" `https://www.usgs.gov/faqs/why-does-usgs-use-spelling-gage-instead-gauge`

Note that the file is not a CSV file, but we can specify a tab as a separator. Also, we need to skip a few of the rows:

This data provided by the US government is not really a CSV file. It has many lines at the top that we want to skip. Then, it has the columns. Then, we want to skip one more line and keep the rest. If you use the pandas engine for parsing the CSV, you use this parameter to accomplish that:

```
skiprows=lambda num: num <34 or num == 35
```

The pyarrow library in pandas 2.2 doesn't accept anything other than an integer for the `skiprows` parameter [1], so I'm going to write a function to remove the lines for me:

```
import urllib.request

def download_and_modify_url(url, local_filename):
    # Download the file from the URL
    urllib.request.urlretrieve(url, local_filename)
    with open(local_filename, 'r') as file:
        lines = file.readlines()

    with open(local_filename, 'w') as file:
        for i, line in enumerate(lines):
            if i <34 or i == 35:
                continue
            file.write(line)

url = 'https://github.com/mattharrison/datasets/raw/master'\
    '/data/dirtydevil.txt'
local_filename = 'data/devilclean.txt'
download_and_modify_url(url, local_filename)
```

Now, I'm going to load the cleaned up file:

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def tweak_river(df_):
...     return (df_
...   .assign(datetime=pd.to_datetime(df_.datetime))
...   .rename(columns={'144166_00060': 'cfs',
...                    '144167_00065': 'gage_height'})
...   .set_index('datetime')
...   .loc[:, ['agency_cd', 'site_no', 'tz_cd', 'cfs', 'gage_height'] ]
```

[1] https://github.com/pandas-dev/pandas/issues/38872

```
... )

>>> dd = tweak_river(df)
>>> print(dd)
                    agency_cd  site_no tz_cd   cfs  gage_height
datetime
2001-05-07 01:00:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:15:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:30:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:45:00      USGS  9333500   MDT  70.0         <NA>
2001-05-07 02:00:00      USGS  9333500   MDT  70.0         <NA>
...                       ...      ...   ...   ...          ...
2020-09-28 08:30:00      USGS  9333500   MDT  9.53         6.16
2020-09-28 08:45:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:00:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:15:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:30:00      USGS  9333500   MDT   9.2         6.15

[539305 rows x 5 columns]
```

33.2 Adding Timezone Information

Many times the date column is missing timezone information. In the Dirty Devil dataset, the *tz_cd* column has offset abbreviations:

```
>>> dd.tz_cd
datetime
2001-05-07 01:00:00    MDT
2001-05-07 01:15:00    MDT
2001-05-07 01:30:00    MDT
2001-05-07 01:45:00    MDT
2001-05-07 02:00:00    MDT
                      ...
2020-09-28 08:30:00    MDT
2020-09-28 08:45:00    MDT
2020-09-28 09:00:00    MDT
2020-09-28 09:15:00    MDT
2020-09-28 09:30:00    MDT
Name: tz_cd, Length: 539305, dtype: string[pyarrow]
```

I ignored it above and have "naive" time data. Getting timezone information into a date column can be slow, buggy, or frustrating. I spent a few hours trying to add timezone information to this dataset.

My takeaway is that although the documentation and API make it appear that `pd.to_datetime` should handle timezone data, I would not go down that

path. Generally, you should use `pd.to_datetime` to get a naive time and then convert the naive times to timezones with `.dt.tz_localize`.

I tried concatenating the *datetime* and *tz_cd* columns together and passing that into `pd.to_datetime`. That worked but took two minutes, whereas code to convert into a naive date column in a fraction of that time (54 ms). I tried using format strings, replacing the timezones with alternate spellings, and using offsets with `pd.to_datetime`[1] in an attempt to speed up the conversion. They silently failed or errored out.

With the help of the pandas core developers, I was able to get that 2 minutes down to 15 seconds with this code. The key points below are using numeric date offsets (not timezone abbreviations) and `utc=True`:

```
>>> def tweak_river(df_):
...     return (df_
...       .assign(datetime=lambda df_:
...           pd.to_datetime(df_.datetime + " " +
...               df_.tz_cd.str.replace('MST', '-0700')
...                  .str.replace('MDT', '-0600'),
...               format='%Y-%m-%d %H:%M %z', utc=True))
...       .rename(columns={'144166_00060': 'cfs',
...                        '144167_00065': 'gage_height'})
...       .set_index('datetime')
...     )
```

However, I was able to get the runtime down to 1 second. The code is more involved, but this is 15-120x faster than the other code.

For my dataset, I wrote the function, `to_america_denver_time`, to get my date parsing with timezone information down from 2 minutes to 2 seconds. I group by the offset column and then use the grouping name (the offset name) to call `.dt.tz_localize`. This creates a date with local times. However, they are using offsets and not timezones.

Note that this uses dictionary unpacking (**). Because the column name is passed in as a variable, `tz_col`, we can't use the variable as the named parameter for the column name in `.assign`, or it would make a new variable named *tz_col*. To get around that, we create a dictionary using the variable name as a key, and then unpack that dictionary in the call to `.assign`.

To add timezone, you need to use `.dt.tz_convert` after creating the local time:

```
>>> def to_america_denver_time(df_, time_col, tz_col):
...     return (df_
...             .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...             .groupby(tz_col)
...             [time_col]
```

[1] https://github.com/pandas-dev/pandas/issues/43140

```
...             .transform(lambda s: pd.to_datetime(s)
...                 .dt.tz_localize(s.name, ambiguous=True)
...                 .dt.tz_convert('America/Denver'))
...             )

>>> def tweak_river(df_):
...     return (df_
...       .assign(datetime=to_america_denver_time(df_, 'datetime',
...               'tz_cd'))
...       .rename(columns={'144166_00060': 'cfs',
...                        '144167_00065': 'gage_height'})
...       .set_index('datetime')
...       .loc[:, ['agency_cd', 'site_no', 'tz_cd', 'cfs', 'gage_height'] ]
...      )

>>> dd = tweak_river(df)
```

Here is the resulting data:

```
>>> print(dd)
                          agency_cd  site_no tz_cd   cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                              ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                          gage_height
datetime
2001-05-07 01:00:00-06:00        <NA>
2001-05-07 01:15:00-06:00        <NA>
2001-05-07 01:30:00-06:00        <NA>
2001-05-07 01:45:00-06:00        <NA>
2001-05-07 02:00:00-06:00        <NA>
...                               ...
2020-09-28 08:30:00-06:00        6.16
2020-09-28 08:45:00-06:00        6.15
2020-09-28 09:00:00-06:00        6.15
2020-09-28 09:15:00-06:00        6.15
```

```
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

Note

One thing that bit me was I was trying to use `'MST'` and `'MDT'` as offset names. The underlying `pytz` library that handles timezone information didn't like them. (For a list of valid names, inspect `pytz.all_timezones`.) The timezone for this data is *America/Denver*.

33.3 Exploring the Data

I'm going to visualize the flow (cfs) of the river over time:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(dpi=600)
dd.cfs.plot(ax=ax)
```

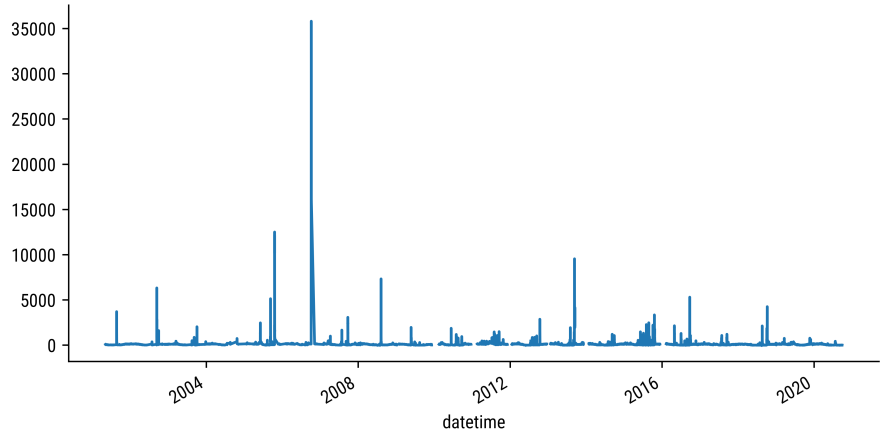


Figure 33.1: Visualization of flow of Dirty Devil river.

From this visualization, it looks like there are some pretty big outliers. (Looking at a histogram or calling `.describe` would also confirm this.):

```
>>> dd.cfs.describe()
count    493124.0
mean     104.460537
std      477.341329
min           0.0
```

```
25%            34.7
50%            81.0
75%           115.0
max         35800.0
Name: cfs, dtype: double[pyarrow]
```

33.4 Slicing Time Series

We get some special slicing abilities because the dataframe has datetime data in the index. We can slice with strings representing dates (or parts of dates). Below, we will slice out the rows from 2018 onward:

```
>>> (dd
...  .cfs
...  .loc['2018':]
... )
datetime
2018-01-01 00:00:00-07:00    92.8
2018-01-01 00:15:00-07:00    88.3
2018-01-01 00:30:00-07:00    90.5
2018-01-01 00:45:00-07:00    90.5
2018-01-01 01:00:00-07:00    94.0
                             ...
2020-09-28 08:30:00-06:00    9.53
2020-09-28 08:45:00-06:00     9.2
2020-09-28 09:00:00-06:00     9.2
2020-09-28 09:15:00-06:00     9.2
2020-09-28 09:30:00-06:00     9.2
Name: cfs, Length: 95886, dtype: double[pyarrow]
```

We can specify a slice including the month as well. Make sure the dates are sorted. When you specify just the month on an end slice, it includes all entries from that month on both the start and end slices (note that this is different behavior than both partial string slicing with `.loc` and position slicing with `.iloc`):

```
>>> (dd
...  .cfs
...  .sort_index()
...  .loc['2018-03-05':'2019/05']
... )
datetime
2018-03-05 00:00:00-07:00    104.0
2018-03-05 00:15:00-07:00    103.0
2018-03-05 00:30:00-07:00    103.0
2018-03-05 00:45:00-07:00    105.0
```

```
2018-03-05 01:00:00-07:00    106.0
                             ...
2019-05-31 22:45:00-06:00    121.0
2019-05-31 23:00:00-06:00    123.0
2019-05-31 23:15:00-06:00    123.0
2019-05-31 23:30:00-06:00    125.0
2019-05-31 23:45:00-06:00    123.0
Name: cfs, Length: 43478, dtype: double[pyarrow]
```

Let's visualize what that slice of data looks like:

```
(dd
  .sort_index()
  .cfs
  .loc['2018/3':'2019/5']
  .plot()
)
```

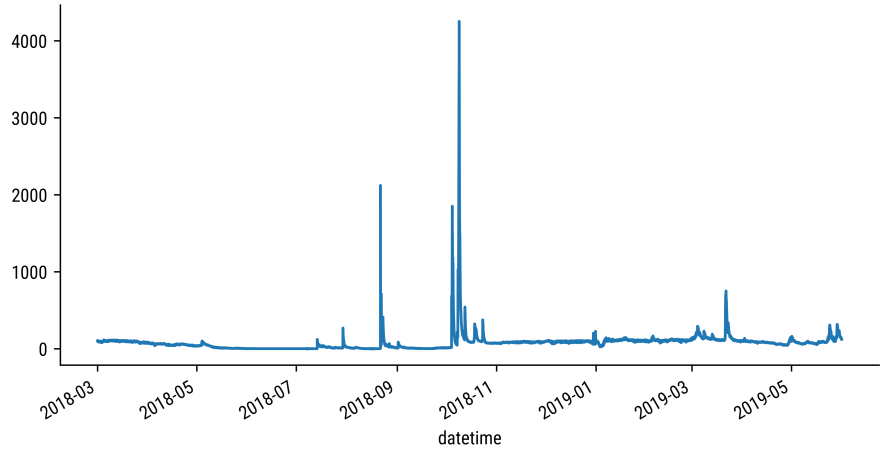


Figure 33.2: Visualization of the flow of Dirty Devil River from March 2018 through May 2019.

I'm going to clip the visualization and limit the upper value to 400 and try the visualization again:

```
(dd
  .sort_index()
  .cfs
  .loc['2018/3':'2019/5']
  .clip(upper=400)
```

```
  .plot()
)
```

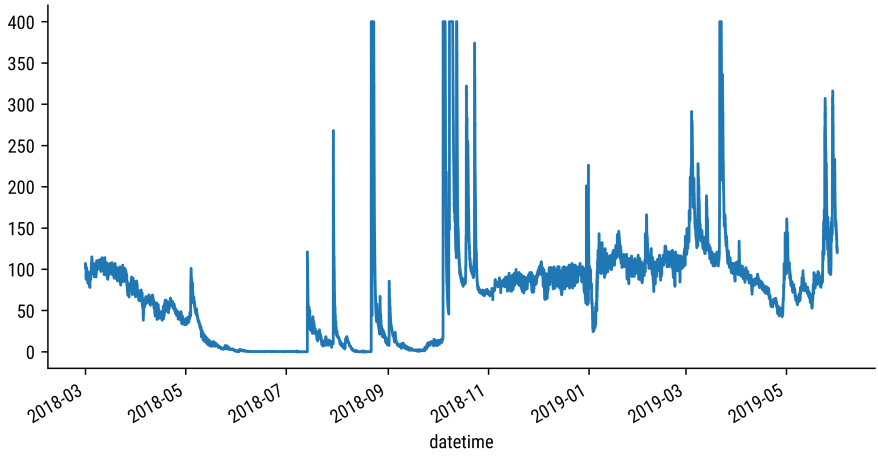


Figure 33.3: Visualization of the flow of Dirty Devil River from March 2018 through May 2019 with values clipped at 400.

Because the index is a time series, we can leverage the ability to resample. A typical operation these days is to plot rolling 7-day average data on top of daily data. The `.rolling` method accepts a moving window size, `window`, and like a grouping operation, you generally aggregate the result. Let's do it:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(8,4))
>>> dd2018 = (dd
...   .sort_index()
...   .cfs
...   .loc['2018/3':'2019/5']
...   .clip(upper=400))

>>> (dd2018
...   .resample('D')
...   .mean()
...   .plot(figsize=(10,4), alpha=.5, linewidth=1, label='Daily')
... )

>>> ax = (dd2018
...   .resample('D')
...   .mean()
...   .rolling(7)
```

```
...   .mean()
...   .plot(figsize=(10,4), ax=ax, label='7-day Rolling')
... )
>>> ax.legend()
>>> ax.set_title('Dirty Devil Flow 2018 (cfs)')
>>> sns.despine()
>>> fig.savefig('img/pandas2/dd4.png', dpi=600, bbox_inches='tight')
<Figure size 6000x2400 with 1 Axes>
```

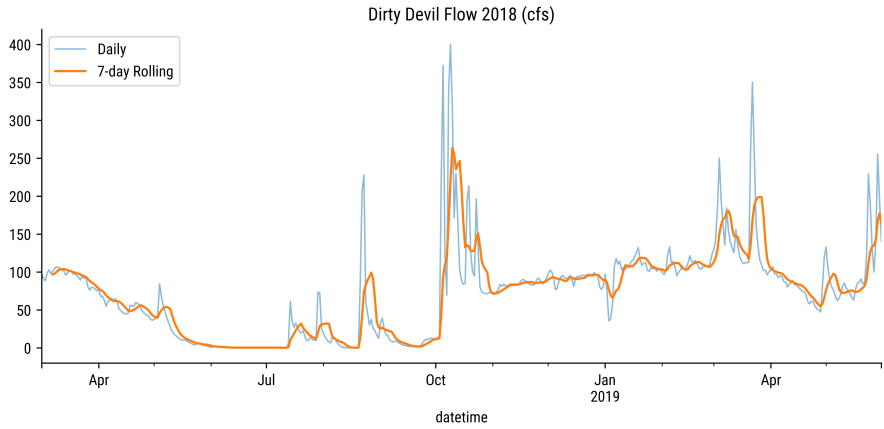


Figure 33.4: Visualization of the flow of daily and weekly levels of Dirty Devil River from March 2018 through May 2019, with values clipped at 400

33.5 Missing Timeseries Data

Let's look at dealing with missing data in timeseries data. First, we will search for it using `.isna`. One of the nice features of the `.query` method is that you can call other methods in the string that you pass into it. Here we use `.query` and `.isna` to find missing values from the *cfs* column:

```
>>> print(dd
...   .sort_index()
...   [['cfs']]
...   .loc['2018/3':'2019/5']
...   .query('cfs.isna()')
... )
                           cfs
datetime
2018-07-07 13:15:00-06:00  <NA>
```

```
2018-07-07 13:30:00-06:00  <NA>
2018-07-07 13:45:00-06:00  <NA>
2018-07-07 14:00:00-06:00  <NA>
2018-07-07 14:15:00-06:00  <NA>
...                         ...
2018-08-18 08:15:00-06:00  <NA>
2018-08-18 08:30:00-06:00  <NA>
2018-08-18 08:45:00-06:00  <NA>
2018-08-18 09:15:00-06:00  <NA>
2018-08-18 10:30:00-06:00  <NA>

[337 rows x 1 columns]
```

Here is code to visualize the missing data from July 7-8. This will help us understand how the various methods work to deal with these missing values:

```
(dd
   .sort_index()
   [['cfs']]
   .loc['2018/7/7':'2018/7/8']
   .plot()
)
```

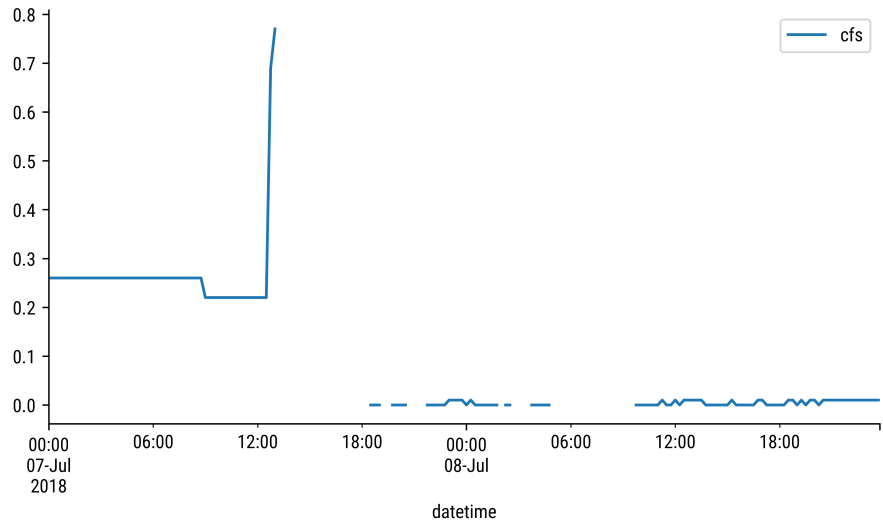


Figure 33.5: Visualization of missing data from flow of Dirty Devil river.

The series chapter discussed various methods for filling in missing data. Let's visualize those below. I'm adding an offset to each line so you can see the behavior:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(6,4))
>>> dd_july = (dd
...  .sort_index()
...  ['cfs']
...  .loc['2018/7/7 11:00':'2018/7/7 20:00']
... )

>>> dd_july.plot(ax=ax, label='original', linewidth=2)
>>> (dd_july
...  .bfill()
...  .add(.05)
...  .plot(label='bfill', ax=ax, linewidth=.5))

>>> (dd_july
...  .ffill()
...  .add(.1)
...  .plot(label='ffill', ax=ax, linewidth=.5))

>>> (dd_july
...  .astype(float)
...  .interpolate(method='polynomial', order=3)
...  .add(.15)
...  .plot(label='interpolate poly (order 3)', ax=ax, linewidth=.5))

>>> (dd_july
...  .astype(float)
...  .interpolate()
...  .add(.2)
...  .plot(label='interpolate default', ax=ax, linewidth=.5))

>>> (dd_july
...  .astype(float)
...  .interpolate(method='nearest')
...  .add(.25)
...  .plot(label='interpolate nearest', ax=ax, linewidth=.5))

>>> (dd_july
...  .fillna(1)
...  .add(.3)
...  .plot(label='fillna 1', ax=ax, linewidth=.5))

>>> ax.legend()
```

```
>>> ax.set_title('Missing Values Demo')
>>> sns.despine()
>>> fig.savefig('img/pandas2/dd6-na.png', dpi=600, bbox_inches='tight')
<Figure size 3600x2400 with 1 Axes>
```

Note that as of pandas 2.0.2, `.interpolate` doesn't like to run with pyarrow data.

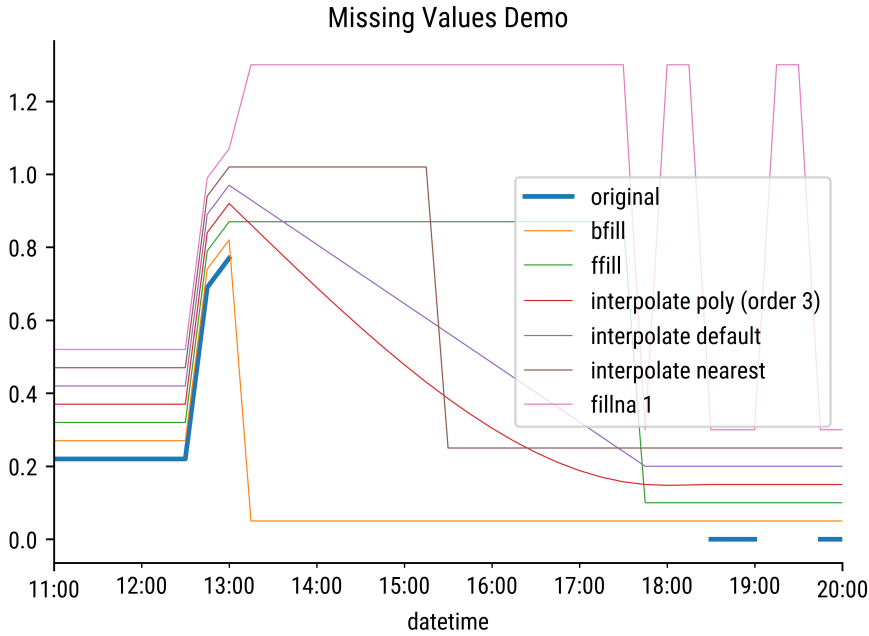


Figure 33.6: Visualization of filling in missing data from flow of Dirty Devil river.

33.6 Exploring Seasonality

Time series data may have a seasonal component to it. Let's examine how to explore this with pandas (and related tools). We will explore the Dirty Devil dataset's cubic feet per second column (*cfs*). We can summarize monthly behavior in this column by combining `.groupby` and `.describe`. Note that we already have an index with date information, so one might suppose we could use `.resample` with `'M'` as an offset alias. However, a `.resample` operation will put the end date of each month in the index, while a `.grouby` on the month number will have only twelve entries in the index:

```
>>> print(dd
...   .groupby(dd.index.month)
```

```
...  .cfs
...  .describe()
... )
            count        mean  ...    75%      max
datetime                       ...
1         26011.0  117.268802  ...  132.0    265.0
2         41309.0  125.890293  ...  141.0    303.0
3         51807.0  127.037609  ...  136.0    750.0
4         50669.0   82.786214  ...   97.8   2140.0
5         49507.0   63.007851  ...   78.5   1960.0
...           ...         ...  ...    ...      ...
8         37584.0   74.676246  ...   59.1   7320.0
9         42272.0  128.309332  ...   55.9   9540.0
10        44647.0  196.285529  ...   80.9  35800.0
11        42165.0   97.194344  ...  105.0    766.0
12        28685.0  100.042608  ...  113.0    407.0

[12 rows x 8 columns]
```

We can also visualize these components by plotting. Here is a chain to plot the mean for each month as a bar plot:

```
(dd
  .groupby(dd.index.month)
  ['cfs']
  .describe()
  ['mean']
  .plot.bar()
)
```

We can also plot a line plot of each of the quantiles (I'm not showing the maximum value because it has so many outliers, it blows out the y-axis):

```
(dd
  .groupby(dd.index.month)
  ['cfs']
  .describe()
  .loc[:, 'min':'75%']
  .plot.bar()
)
```

To get much fancier, we could leverage the pandas `.boxplot` method, but at that point, I would prefer using Seaborn[1], which is built on top of Matplotlib and pandas provide a lot of power. I'm going to use the Seaborn `boxplot` function, and pass in clipped measurements to the `data` parameter. We must

[1] https://seaborn.pydata.org/

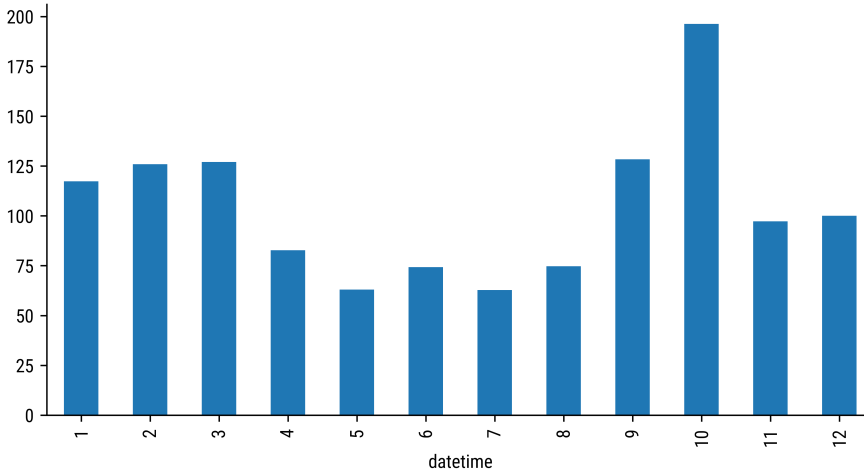


Figure 33.7: Visualization of monthly average of flow of Dirty Devil river.

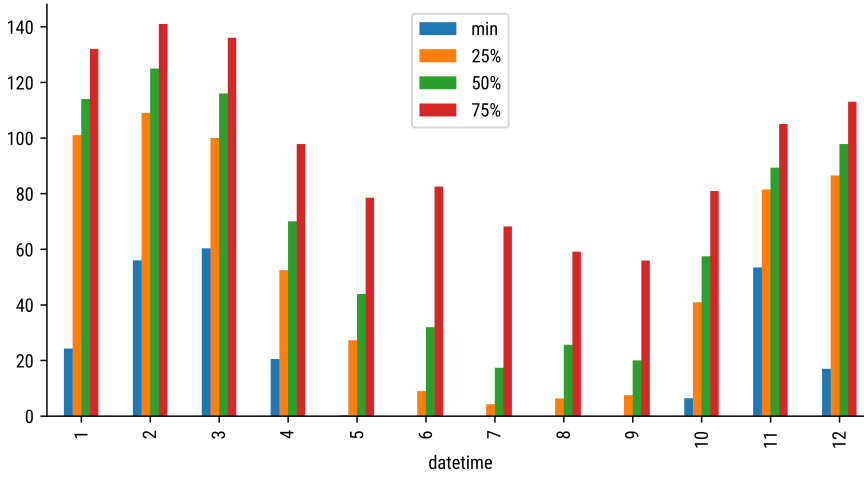


Figure 33.8: Visualization of monthly quantiles of flow of Dirty Devil river.

also specify what we plot in the x and y axes. I create a column from the index with the month data (and rename it from *datetime* to *Month*) and the *cfs* column for x and y, respectively:

```
import seaborn as sns
sns.boxplot(data=dd.assign(cfs=dd.cfs.clip(upper=400)),
    x=dd.index.month.rename('Month'), y='cfs')
```

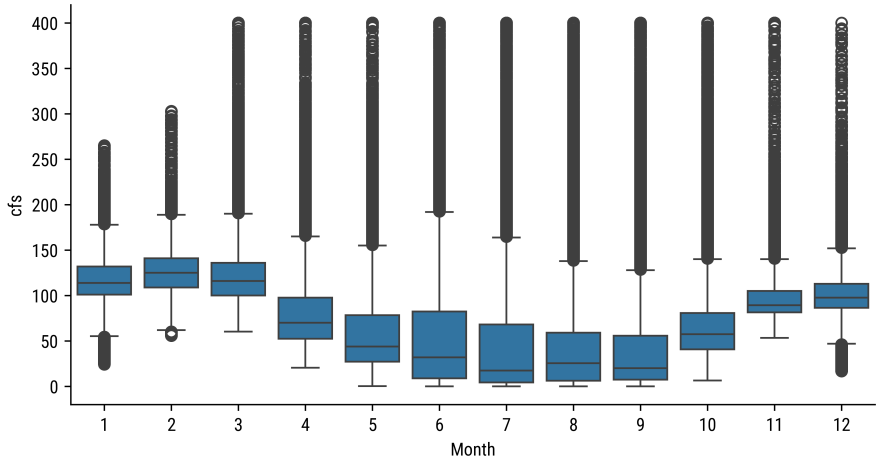


Figure 33.9: Boxplot of monthly quantiles of the flow of Dirty Devil River.

Plots such as these can give us an understanding of the monthly patterns we see in the data.

33.7 Resampling Data

We explored resampling in the series section, but I want to show some of the power you get by using offset aliases. We will use the flow data from the Dirty Devil dataset to dive into resampling. This data has information sampled to 15-minute intervals:

```
>>> dd.cfs
datetime
2001-05-07 01:00:00-06:00    71.0
2001-05-07 01:15:00-06:00    71.0
2001-05-07 01:30:00-06:00    71.0
2001-05-07 01:45:00-06:00    70.0
2001-05-07 02:00:00-06:00    70.0
                             ...
```

```
2020-09-28 08:30:00-06:00    9.53
2020-09-28 08:45:00-06:00     9.2
2020-09-28 09:00:00-06:00     9.2
2020-09-28 09:15:00-06:00     9.2
2020-09-28 09:30:00-06:00     9.2
Name: cfs, Length: 539305, dtype: double[pyarrow]
```

Let's aggregate this information from a 15-minute interval to a daily interval. Because the index has date information in it, we can use `.resample` in combination with `'D'` (daily) as the offset alias. I am going to use `.median` as the aggregation method because the flow data is heavily skewed:

```
>>> print(dd
...    .resample('D')
...    .median(numeric_only=True)
... )
                            site_no   cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0  71.5         <NA>
2001-05-08 00:00:00-06:00  9333500.0  69.0         <NA>
2001-05-09 00:00:00-06:00  9333500.0  63.5         <NA>
2001-05-10 00:00:00-06:00  9333500.0  55.0         <NA>
2001-05-11 00:00:00-06:00  9333500.0  55.0         <NA>
...                              ...   ...          ...
2020-09-24 00:00:00-06:00  9333500.0  9.53         6.16
2020-09-25 00:00:00-06:00  9333500.0  10.2         6.18
2020-09-26 00:00:00-06:00  9333500.0  10.9          6.2
2020-09-27 00:00:00-06:00  9333500.0  10.2         6.18
2020-09-28 00:00:00-06:00  9333500.0  9.53         6.16

[7085 rows x 3 columns]
```

33.8 Rules with Offset Aliases

We could also provide a numeric *rule* before the alias if we wanted to combine multiple days. You can insert a number before the offset alias. In this example, we will aggregate every two days using the offset alias `'2D'`. Pay attention to the index of the result:

```
>>> print(dd
...    .resample('2D')
...    .median(numeric_only=True)
... )
                            site_no   cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0  69.0         <NA>
```

```
2001-05-09 00:00:00-06:00  9333500.0  56.0         <NA>
2001-05-11 00:00:00-06:00  9333500.0  54.0         <NA>
2001-05-13 00:00:00-06:00  9333500.0  47.0         <NA>
2001-05-15 00:00:00-06:00  9333500.0  54.0         <NA>
...                              ...   ...          ...
2020-09-20 00:00:00-06:00  9333500.0  6.83         6.07
2020-09-22 00:00:00-06:00  9333500.0  7.68          6.1
2020-09-24 00:00:00-06:00  9333500.0  9.86         6.17
2020-09-26 00:00:00-06:00  9333500.0  10.5         6.19
2020-09-28 00:00:00-06:00  9333500.0  9.53         6.16

[3543 rows x 3 columns]
```

33.9 Combining Offset Aliases

We can also combine offset aliases. If we want to aggregate at the three-day, 2-hour, and 10-minute intervals, we can combine all of these rules with the offset aliases into a single string:

```
>>> print(dd
...    .resample('3D2h10min')
...    .median(numeric_only=True)
... )
                           site_no    cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0   67.0         <NA>
2001-05-10 02:10:00-06:00  9333500.0   55.0         <NA>
2001-05-13 04:20:00-06:00  9333500.0   49.0         <NA>
2001-05-16 06:30:00-06:00  9333500.0   50.0         <NA>
2001-05-19 08:40:00-06:00  9333500.0   46.0         <NA>
...                              ...    ...          ...
2020-09-14 13:20:00-06:00  9333500.0   5.79         6.03
2020-09-17 15:30:00-06:00  9333500.0   6.04         6.04
2020-09-20 17:40:00-06:00  9333500.0   7.11         6.08
2020-09-23 19:50:00-06:00  9333500.0  10.03        6.175
2020-09-26 22:00:00-06:00  9333500.0   9.86         6.17

[2293 rows x 3 columns]
```

33.10 Anchored Offset Aliases

Some of the frequencies in offset aliases allow you to modify when the window for the frequency ends. You can use this operation on a weekly, quarterly, or yearly frequency. Note that the default quarter ends in March, June, September, and December:

```
>>> print(dd
...   .resample('QE')
...   .median(numeric_only=True)
... )
                            site_no    cfs  gage_height
datetime
2001-06-30 00:00:00-06:00  9333500.0   44.0         <NA>
2001-09-30 00:00:00-06:00  9333500.0   27.0         <NA>
2001-12-31 00:00:00-07:00  9333500.0   85.0         <NA>
2002-03-31 00:00:00-07:00  9333500.0  122.0         <NA>
2002-06-30 00:00:00-06:00  9333500.0   46.0         <NA>
...                              ...    ...          ...
2019-09-30 00:00:00-06:00  9333500.0   13.3         6.21
2019-12-31 00:00:00-07:00  9333500.0   92.1         6.75
2020-03-31 00:00:00-06:00  9333500.0  126.0         6.99
2020-06-30 00:00:00-06:00  9333500.0   23.2         6.55
2020-09-30 00:00:00-06:00  9333500.0   5.79         5.96

[78 rows x 3 columns]
```

We can tack on `-JAN` to force the quarters to end in January, April, July, and October:

```
>>> print(dd
...   .resample('QE-JAN')
...   .median(numeric_only=True)
... )
                            site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0         <NA>
2001-10-31 00:00:00-07:00  9333500.0   39.0         <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0         <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0         <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0         <NA>
...                              ...    ...          ...
2019-10-31 00:00:00-06:00  9333500.0   12.8         6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0         6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0         6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9         6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5         5.49

[78 rows x 3 columns]
```

For annual and quarterly offset aliases, you can change the anchoring by using `-JAN`, `-FEB`, … `-DEC`. For weekly offset aliases, you can change the anchoring by using `-SUN`, `-MON`, … `-SAT`.

33.11 Resampling to Finer-grain Frequency

Remember, this river flow data is at the 15-minute frequency. If we wanted to have it at a two-minute frequency, we could do the following:

```
>>> print(dd
...    .resample('2min')
...    .median(numeric_only=True)
... )
                           site_no   cfs  gage_height
datetime
2001-05-07 01:00:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:02:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:04:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:06:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:08:00-06:00       <NA>  <NA>         <NA>
...                              ...   ...          ...
2020-09-28 09:22:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:24:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:26:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:28:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:30:00-06:00  9333500.0   9.2         6.15

[5100736 rows x 3 columns]
```

You will notice that there is now a bunch of missing data. You will probably want to refer to the missing data section and adopt an appropriate option to deal with it. Below, we interpolate the missing values using a forward fill:

```
>>> print(dd
...    .resample('2min')
...    .median(numeric_only=True)
...    .ffill()
... )
                           site_no   cfs  gage_height
datetime
2001-05-07 01:00:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:02:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:04:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:06:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:08:00-06:00  9333500.0  71.0         <NA>
...                              ...   ...          ...
2020-09-28 09:22:00-06:00  9333500.0   9.2         6.15
2020-09-28 09:24:00-06:00  9333500.0   9.2         6.15
2020-09-28 09:26:00-06:00  9333500.0   9.2         6.15
```

```
2020-09-28 09:28:00-06:00  9333500.0   9.2         6.15
2020-09-28 09:30:00-06:00  9333500.0   9.2         6.15

[5100736 rows x 3 columns]
```

33.12 Grouping a Date Column with pd.Grouper

The `.resample` method is a powerful way to aggregate data with dates in the index. But what if you want to aggregate dataframes by a column with date information? Enter the `pd.Grouper` class.

Here is an anchored offset alias using `.resample` on the Dirty Devil data. It aggregates on quarters that end in January:

```
>>> print(dd
...   .resample('QE-JAN')
...   .median(numeric_only=True)
... )
                           site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0         <NA>
2001-10-31 00:00:00-07:00  9333500.0   39.0         <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0         <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0         <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0         <NA>
...                              ...    ...          ...
2019-10-31 00:00:00-06:00  9333500.0   12.8         6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0         6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0         6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9         6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5         5.49

[78 rows x 3 columns]
```

Assuming that we have a date column that we want to aggregate on (I'm going to move the index into a column, *datetime*), we could perform the same aggregation using `pd.Grouper`. The `key` parameter specifies the column to group on, and the `freq` parameter specifies the offset alias:

```
>>> print(dd
...   .reset_index()
...   .groupby(pd.Grouper(key='datetime', freq='QE-JAN'))
...   .median(numeric_only=True)
... )
                           site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0         <NA>
```

```
2001-10-31 00:00:00-07:00  9333500.0   39.0         <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0         <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0         <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0         <NA>
...                              ...    ...          ...
2019-10-31 00:00:00-06:00  9333500.0   12.8         6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0         6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0         6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9         6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5         5.49

[78 rows x 3 columns]
```

Table 33.1: Chapter Methods

Method	Description
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=False, format=None, exact=True, unit=None, infer_datetime_format False, origin='unix', cache=True)`	Convert an `arg` to a datetime. Not guaranteed to return a `datetime64` type. Use `utc=True` to convert from naive to UTC (tz-aware) time. Specify strftime string with `format`. When parsing time since epoch, set `unit='s'` for seconds.
`s.dt.tz_localize(tz, ambiguous='raise', nonexistent='raise')`	Return a date converted to a timezone. Set `tz=None` to convert to naive time. For ambiguous times (when clocks move back for daylight savings) set to `'infer'` to base on order, array of `True`/`False` for DST, non-DST time, `'NaT'` to leave empty. For nonexistent times (when the clock moves forward) set `nonexistent` to `'shift_forward'`, `'shift_backward'`, `'NaT'`, or timedelta object.
`s.dt.tz_convert(tz)`	Convert from an existing timezone to another timezone. Set `tz=None` to convert to UTC time.
`df.loc`	If you have a dataframe/series with a datetime index, you can slice on partial date strings.

Method	Description
`df.resample(rule, axis=0, closed=None, label=None, convention='start', kind=None, on=None, level=None, origin='start_day')`	Return a resampled dataframe (with a date in the index, or specify the date column with `on`). Set closed to `'right'` to include the right side of the interval (default is `'right'` for `M/A/Q/BM/BQ/W`). Set the label to `'right'` to use the right label for the bucket. Can specify the timestamp to start `origin`.
`df.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0, closed=None, method='single')`	Return a window object to perform aggregations on.
`s.bfill(axis=0, limit=None, downcast=None)`	Backward fill the missing values. Alternate syntax for `s.fillna(method='bfill')`
`s.ffill(axis=0, limit=None, downcast=None)`	Forward fill the missing values. Alternate syntax for `s.fillna(method='ffill')`
`s.interpolate( method='linear', axis=0, limit=None, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)`	Return a series with interpolated values.
`s.fillna(value=None, method=None, axis=0, limit=None, downcast=None)`	Use the `value` (scalar, dict, series) or `method` (`'ffill'`, `'bfill'`, or `'nearest'`) for filling in missing data.
`pd.Grouper(key=None, level=None, freq=None, axis=0, sort=False, closed=None, label=None, convention=None, origin='start_day', offset=None, dropna=True)`	Return a groupby object based on the column (`key`) or date index (`key=None`) and offset alias (`freq`).

33.13 Summary

There are many tools to manipulate time-series data in pandas. I recommend combining liberal amounts of visualizations when manipulating the data to validate the results.

33.14 Exercises

With a dataset of your choice:

1. Convert a date column from a string to a valid date.
2. Group the data by month names and look at the mean values.
3. Group the data by each month of every year and look at the mean values.
4. Insert the date column in the index and slice out a portion of the rows by date.

Chapter 34

Combining and Joining Data

Dataframes hold tabular data. Databases hold tabular data. You can perform many of the same operations on dataframes that you do to database tables. In this section, we will look at the theory for joining dataframes. Then, we will look at a real-world example of joining.

34.1 Data for Joining

Here are the two tables we will be using for examples: hs

Table 34.1: Table of favorite colors

Index	Color	Name
0	Blue	John
1	Blue	George
2	Purple	Ringo

Table 34.2: Table of car colors

Index	Car Color	Name
3	Red	Paul
1	Blue	George
2		Ringo

34.2 Adding Rows to Dataframes

Let's assume we have two dataframes that we want to combine into a single dataframe, with rows from both. The simplest way to do this is with the `concat` function. Below, we create the dataframes:

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

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> df1 = (pd.DataFrame({'name': ['John', 'George', 'Ringo'],
...                      'color': ['Blue', 'Blue', 'Purple']})
...        .astype(string_pa))
>>> df2 = (pd.DataFrame({'name': ['Paul', 'George', 'Ringo'],
...                      'carcolor': ['Red', 'Blue', np.nan]},
...                     index=[3, 1, 2])
...        .astype(string_pa))

>>> print(df1)
     name   color
0    John    Blue
1  George    Blue
2   Ringo  Purple

>>> print(df2)
     name carcolor
3    Paul      Red
1  George     Blue
2   Ringo     <NA>
```

The concat function in the pandas library accepts a list of dataframes to combine. This function is useful when combining multiple files into one dataframe. By default (`axis=0`), it will stack in the vertical direction. It will find any columns that have the same name and stack them into a single column. In this case, *name* is common to both dataframes:

```
>>> print(pd.concat([df1, df2]))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     <NA>
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
1  George    <NA>     Blue
2   Ringo    <NA>     <NA>
```

Note that `.concat` preserves index values, so the resulting dataframe has duplicate index values. If you would prefer an error when duplicates appear, you can pass the `verify_integrity=True` parameter setting:

```
>>> pd.concat([df1, df2], verify_integrity=True)
Traceback (most recent call last):
  ...
ValueError: Indexes have overlapping values:
  Int64Index([1, 2], dtype='int64')
```

Alternatively, if you would prefer that pandas create new index values for you, pass in `ignore_index=True` as a parameter:

```
>>> print(pd.concat([df1, df2], ignore_index=True))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     <NA>
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
4  George    <NA>     Blue
5   Ringo    <NA>     <NA>
```

34.3 Adding Columns to Dataframes

The `concat` function also can align dataframes based on the index values, rather than using the columns. If you set `axis=1` (`axis='columns'`), we get this behavior. I do not use this operation often. Rather, I use `.assign` to create columns. However, here is an example of `concat` along the columns axis:

```
>>> print(pd.concat([df1, df2], axis=1))
     name   color    name carcolor
0    John    Blue    <NA>     <NA>
1  George    Blue  George     Blue
2   Ringo  Purple   Ringo     <NA>
3    <NA>    <NA>    Paul      Red
```

Note that this repeats the `name` column. Using SQL, we can *join* two database tables together based on common columns. If we want to perform a join similar to a database join on a dataframe, we need to use the `.merge` method. We will cover that in the next section.

34.4 Joins

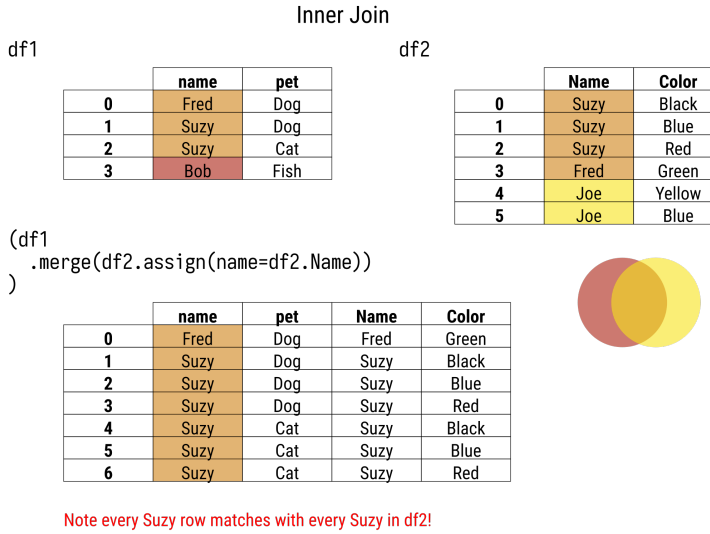
Databases have different types of joins. The four common ones include inner, outer, left, and right. However, there is also a cross-join and an anti-join. We will cover those as well. The dataframe has two methods to support these operations, `.join` and `.merge`. I prefer the `.merge` method.

Note

The `.join` method is meant for joining based on the index rather than columns. In practice, I join based on columns instead of index values.

If you want the `.join` method to join based on column values, you need to set that column as the index first:

```
>>> print(df1.set_index('name').join(df2.set_index('name')))
         color carcolor
```



	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red

Figure 34.1: The `.merge` method performs an inner join by default. The resulting dataframe will only have rows where the merge column value exists in both dataframes.

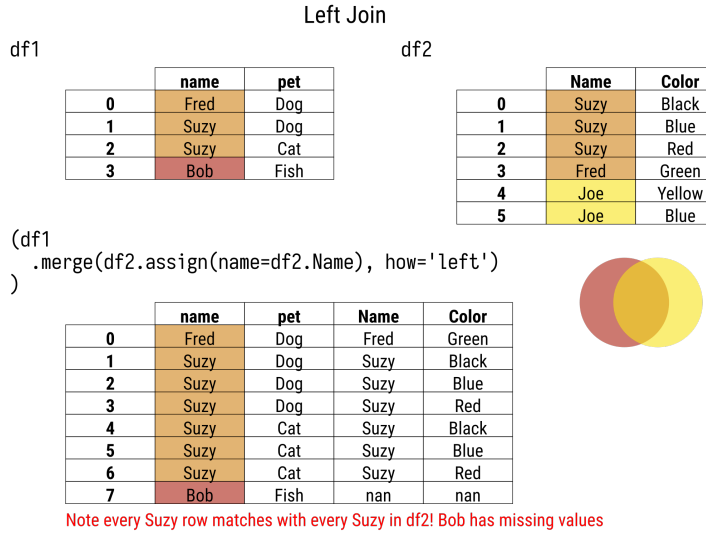
```
name
John      Blue     <NA>
George    Blue     Blue
Ringo   Purple     <NA>
```

It is easier to just use the `.merge` method.

The default join type for the `.merge` method is an *inner join*. The `.merge` method looks for common column names in the dataframes it is going to join. The method aligns the values in those columns. If both columns have values that are the same, they are kept along with the remaining columns from both data frames. Rows with values in the aligned columns that only appear in one data frame are discarded:

```
>>> print(df1.merge(df2))  # inner join
     name   color carcolor
0  George    Blue     Blue
1   Ringo  Purple     <NA>
```

When the `how='outer'` parameter setting is passed in, an *outer join* is performed. Again, the method looks for common column names. It aligns



	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red
7	Bob	Fish	nan	nan

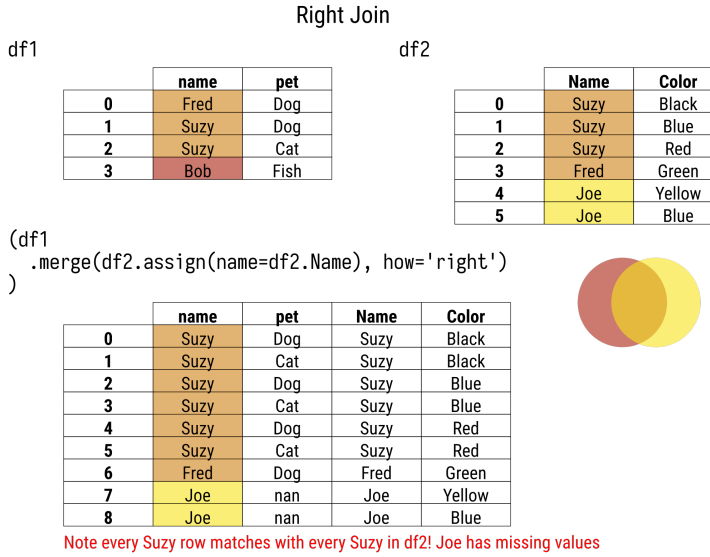
Figure 34.2: A left join keeps all values from the left merge column (orange and red). The values that are unique to the right dataframe (yellow) are dropped. Note the combinatoric explosion for *Suzy* because each left value is matched with all the values in the right.

the values for those columns and adds the values from the other columns of both data frames. If either dataframe had a value in the field that we joined on that was absent from the other, the new columns are filled with `<NA>`:

```
>>> print(df1.merge(df2, how='outer'))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     Blue
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
```

To perform a *left join*, pass the `how='left'` parameter setting. A left join keeps only the values from the columns in the dataframe that the `.merge` method is called on. If the other dataframe is missing aligned values, `<NA>` is used to fill in their values:

```
>>> print(df1.merge(df2, how='left'))
     name   color carcolor
0    John    Blue     <NA>
```



	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

	name	pet	Name	Color
0	Suzy	Dog	Suzy	Black
1	Suzy	Cat	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Cat	Suzy	Blue
4	Suzy	Dog	Suzy	Red
5	Suzy	Cat	Suzy	Red
6	Fred	Dog	Fred	Green
7	Joe	nan	Joe	Yellow
8	Joe	nan	Joe	Blue

Figure 34.3: A right join keeps all values from the right merge column (orange and yellow). The values that are unique to the left dataframe (red) are dropped.

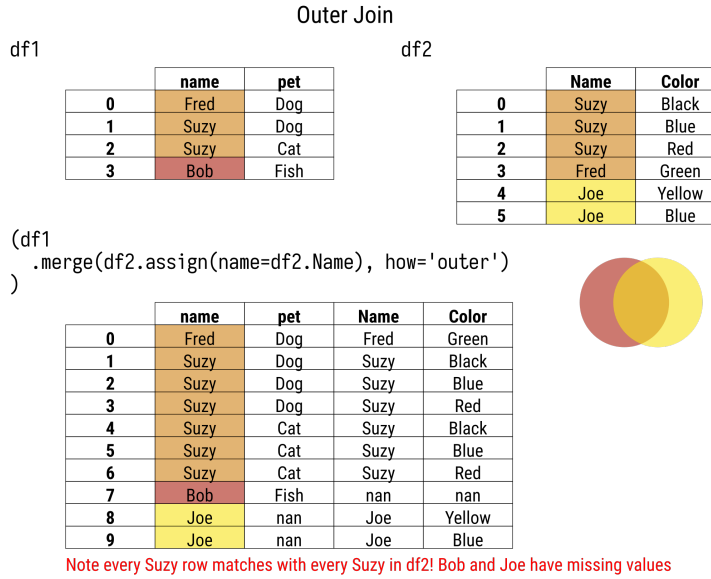
```
1  George    Blue     Blue
2   Ringo  Purple     <NA>
```

Finally, there is support for a *right join* as well. A right join keeps the values from the dataframe that are passed in as the first parameter of the `.merge` method. If the dataframe that `.merge` was called on has aligned values, they are kept, otherwise `<NA>` is used to fill in the missing values:

```
>>> print(df1.merge(df2, how='right'))
     name   color carcolor
0    Paul    <NA>      Red
1  George    Blue     Blue
2   Ringo  Purple     <NA>
```

If we want to join on columns that don't have the same name, we can use the `left_on` and `right_on` parameters. We can also specify a subset of columns if we don't want to merge on all of the common columns:

```
>>> print(df1.merge(df2, how='right', left_on='color',
...     right_on='carcolor'))
   name_x color  name_y carcolor
```



	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red
7	Bob	Fish	nan	nan
8	Joe	nan	Joe	Yellow
9	Joe	nan	Joe	Blue

Figure 34.4: An outer join keeps all values from the left and right merge columns.

```
0    <NA>  <NA>    Paul      Red
1    John  Blue  George     Blue
2  George  Blue  George     Blue
3    <NA>  <NA>   Ringo     <NA>
```

Next, we have a *cross-join*. A cross join is a join where every row from the first dataframe is matched with every row from the second dataframe. This is also known as a *Cartesian product*. Pass in `how='cross'` to perform a cross-join:

```
>>> print(df1.merge(df2, how='cross'))
   name_x   color  name_y carcolor
0    John    Blue    Paul      Red
1    John    Blue  George     Blue
2    John    Blue   Ringo     <NA>
3  George    Blue    Paul      Red
4  George    Blue  George     Blue
5  George    Blue   Ringo     <NA>
6   Ringo  Purple    Paul      Red
7   Ringo  Purple  George     Blue
8   Ringo  Purple   Ringo     <NA>
```

Imagine you worked in a restaurant that had three types of curries and four types of protein. You could use a cross-join to generate all the possible combinations of curry and protein.

The `.merge` method has a few other parameters that are useful in practice. The table below lists them:

Table 34.3: Table for `.merge` method parameters.

Parameter	Meaning
`on`	Column names to join on. String or list. (Default is the intersection of names).
`left_on`	Column names for left dataframe. String or list. Used when names don't overlap.
`right_on`	Column names for right dataframe. String or list. Used when names don't overlap.
`left_index`	Join based on left dataframe index. Boolean
`right_index`	Join based on right dataframe index. Boolean

After the next section, I'll discuss one more type of join, the anti-join.

34.5 Join Indicators

The `.merge` method can add a column that indicates where the data in the row can come from. If you include the `indicator=True` parameter, pandas will create a column called *_merge*. The `indicator` parameter can also be a string, in which the new column will be the name of the string rather than *_merge*.

The *_merge* column will have the values of `left_only`, `right_only`, or `both` to indicate the row came from the dataframe `.merge` was called on, the data frame passed in, or both of them, respectively:

```
>>> print(df1.merge(df2, how='outer',
...     indicator=True))
     name   color carcolor       _merge
0    John    Blue     <NA>   left_only
1  George    Blue     Blue        both
2   Ringo  Purple     <NA>        both
3    Paul    <NA>      Red  right_only
```

34.6 Anti Joins

An *anti-join* is a join where we keep only the rows that don't match between two dataframes. This is useful during data cleaning and validation. You can inspect the rows that don't match and decide what to do with them.

We can perform an anti-join by using the `.merge` method and passing in `how='outer'` and `indicator=True`. Then, we can filter out the rows that have a

value of both in the *_merge* column. The values that are *left_only* are the rows in the left dataframe but not the right dataframe. The values that are *right_only* are the rows that were in the right dataframe but not the left dataframe:

```
>>> print(df1.merge(df2, on='name', how='outer', indicator=True)
...   .query('_merge != "both"')
... )
   name color carcolor      _merge
0  John  Blue     <NA>   left_only
3  Paul  <NA>      Red  right_only
```

34.7 Merge Validation

The `.merge` method recently added a useful option, the `validate` parameter. It will raise a `MergeError` if the join validates a constraint. The constraint can be `'1:1'`, `'1:m'`, or `'m:1'` to ensure that the join keys are indeed one-to-one, one-to-many, or many-to-one. You can also specify `'m:m'` for many to many, but that constraint is always ignored.

In the following example, the left key is *color*, which has non-unique values (many), and the right key is *carcolor*, which is unique (one), so the constraint should be `'m:1'`. If we pass in a wrong constraint, like a one-to-many constraint, the `MergeError` is raised:

```
>>> df1.merge(df2, how='right', left_on='color',
...     right_on='carcolor', validate='1:m')
Traceback (most recent call last):
    ...
pandas.errors.MergeError: Merge keys are not
    unique in left dataset; not a one-to-many merge
```

This parameter is helpful to check that your data looks like you think it should. I recommend validating your data after merges.

34.8 Dirty Devil Flow and Weather Data

In the previous section, we discussed the theory behind joining data. In this section, we will look at a concrete example.

Most of the data we have looked at in the book has been delivered in a single CSV file. Sometimes, we have data from multiple sources, and we need to combine them. This section will explore joining a real-world dataset.

In this section, we will revisit the Dirty Devil data. Let's load the flow and gage height data. In this case, we will leave the *datetime* column as a column and not use it for the index:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
```

```
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...     )

>>> dd = tweak_river(df)
>>> print(dd)
                      datetime agency_cd  ...   cfs gage_height
0      2001-05-07 01:00:00-06:00      USGS  ...  71.0        <NA>
1      2001-05-07 01:15:00-06:00      USGS  ...  71.0        <NA>
2      2001-05-07 01:30:00-06:00      USGS  ...  71.0        <NA>
3      2001-05-07 01:45:00-06:00      USGS  ...  70.0        <NA>
4      2001-05-07 02:00:00-06:00      USGS  ...  70.0        <NA>
...                          ...       ...  ...   ...         ...
539300 2020-09-28 08:30:00-06:00      USGS  ...  9.53        6.16
539301 2020-09-28 08:45:00-06:00      USGS  ...   9.2        6.15
539302 2020-09-28 09:00:00-06:00      USGS  ...   9.2        6.15
539303 2020-09-28 09:15:00-06:00      USGS  ...   9.2        6.15
539304 2020-09-28 09:30:00-06:00      USGS  ...   9.2        6.15

[539305 rows x 6 columns]
```

I'm also going to load some meteorological data[1] from Hanksville, Utah, a city near the river. We will then combine both datasets to have flow data and temperature and precipitation information in the same dataset.

Some of the interesting columns are:

- *DATE* - Date

[1] https://www.ncdc.noaa.gov/cdo-web/

- *PRCP* - Precipitation in inches
- *TMIN* - Minimum temperature (F) for day
- *TMAX* - Maximum temperature (F) for day
- *TOBS* - Observed temperature (F) when measurement made

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       'hanksville.csv'

>>> temp_df = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> def tweak_temp(df_):
...     return (df_
...             .assign(DATE=pd.to_datetime(df_.DATE)
...                  .dt.tz_localize('America/Denver', ambiguous=False))
...             .loc[:,['DATE', 'PRCP', 'TMIN', 'TMAX', 'TOBS']]
...     )

>>> temp_df = tweak_temp(temp_df)
>>> print(temp_df)
                       DATE  PRCP  TMIN  TMAX  TOBS
0    2000-01-01 00:00:00-07:00  0.02    21    43    28
1    2000-01-02 00:00:00-07:00  0.03    24    39    24
2    2000-01-03 00:00:00-07:00   0.0     7    39    18
3    2000-01-04 00:00:00-07:00   0.0     5    39    25
4    2000-01-05 00:00:00-07:00   0.0    10    44    22
...                        ...   ...   ...   ...   ...
6843 2020-09-20 00:00:00-06:00   0.0    46    92    83
6844 2020-09-21 00:00:00-06:00   0.0    47    92    84
6845 2020-09-22 00:00:00-06:00   0.0    54    84    77
6846 2020-09-23 00:00:00-06:00   0.0    47    91    87
6847 2020-09-24 00:00:00-06:00   0.0    43    94    88

[6848 rows x 5 columns]
```

34.9 Joining Data

The pandas API provides a function for merging data, `pd.merge`. It also has two methods for joining data, `.join` and `.merge`, that wrap that function. I will use the `.merge` method.

Let's try to use `.merge` and merge by date. This method will try to merge columns that have the same name. The `dd` dataframe has a *datetime* column, and `temp_df` has a *DATE* column. We can use the `left_on` and `right_on` parameters to help pandas align the data. The `.merge` method tries to do an *inner join* by default. That means that rows with values that are the same in the merge columns will be joined together:

```
>>> print(dd
...  .merge(temp_df, left_on='datetime', right_on='DATE')
... )
                     datetime agency_cd  ...  TMAX TOBS
0    2001-05-08 00:00:00-06:00      USGS  ...    85   58
1    2001-05-09 00:00:00-06:00      USGS  ...    92   64
2    2001-05-10 00:00:00-06:00      USGS  ...    92   67
3    2001-05-11 00:00:00-06:00      USGS  ...    87   60
4    2001-05-12 00:00:00-06:00      USGS  ...    93   72
...                        ...       ...  ...   ...  ...
4968 2020-09-20 00:00:00-06:00      USGS  ...    92   83
4969 2020-09-21 00:00:00-06:00      USGS  ...    92   84
4970 2020-09-22 00:00:00-06:00      USGS  ...    84   77
4971 2020-09-23 00:00:00-06:00      USGS  ...    91   87
4972 2020-09-24 00:00:00-06:00      USGS  ...    94   88

[4973 rows x 11 columns]
```

This appears to have worked but is somewhat problematic. Remember that the `dd` dataset has a 15-minute frequency, but `temp_df` only has daily data, so we only use the value from midnight. We should probably use our resampling skills to calculate the median flow value for each date and then merge. In that case, we will want to use the index of the grouped data to merge, so we specify `left_index=True`:

```
>>> print(dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE')
... )
        site_no   cfs  ...  TMAX TOBS
492   9333500.0  71.5  ...    82   55
493   9333500.0  69.0  ...    85   58
494   9333500.0  63.5  ...    92   64
495   9333500.0  55.0  ...    92   67
496   9333500.0  55.0  ...    87   60
...         ...   ...  ...   ...  ...
6843  9333500.0  6.83  ...    92   83
6844  9333500.0  6.83  ...    92   84
6845  9333500.0  7.39  ...    84   77
6846  9333500.0  7.97  ...    91   87
6847  9333500.0  9.53  ...    94   88

[6356 rows x 8 columns]
```

That looks better (and gives us a few more rows of data).

34.10 Validating Joined Data

Let's validate that we had a one-to-one join, i.e., each date from the flow data matched up with a single date from the temperature data. We can use the `validate` parameter to do this:

```
>>> print(dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...         validate='1:1')
... )
        site_no   cfs  ...  TMAX TOBS
492   9333500.0  71.5  ...    82   55
493   9333500.0  69.0  ...    85   58
494   9333500.0  63.5  ...    92   64
495   9333500.0  55.0  ...    92   67
496   9333500.0  55.0  ...    87   60
...         ...   ...  ...   ...  ...
6843  9333500.0  6.83  ...    92   83
6844  9333500.0  6.83  ...    92   84
6845  9333500.0  7.39  ...    84   77
6846  9333500.0  7.97  ...    91   87
6847  9333500.0  9.53  ...    94   88

[6356 rows x 8 columns]
```

Because this did not raise a `MergeError`, we know that our data had non-repeating date fields.

34.11 Visualization of Merged Data

You know that I'm a big fan of visualization. Let's visualize the merged time series. We will add it to our merge chain, stick the date in the index, pull out the years from 2014 forward, use the *cfs*, *gage_height*, *PRCP*, and *TOBS* columns, interpolate the missing values, do a rolling 15-day average and plot the result in their own subplot:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(dpi=600)
>>> (dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...         validate='1:1')
...  .set_index('DATE')
```

```
...   .loc['2014':,['cfs', 'gage_height', 'PRCP', 'TOBS']]
...   .rolling(15)
...   .mean()
...   .plot(subplots=True, figsize=(10,8), ax=ax, sharex=True)
... )
>>> fig.suptitle('Dirty Devil Metrics (15 day average)')

<Figure size 3840x2880 with 4 Axes>
```

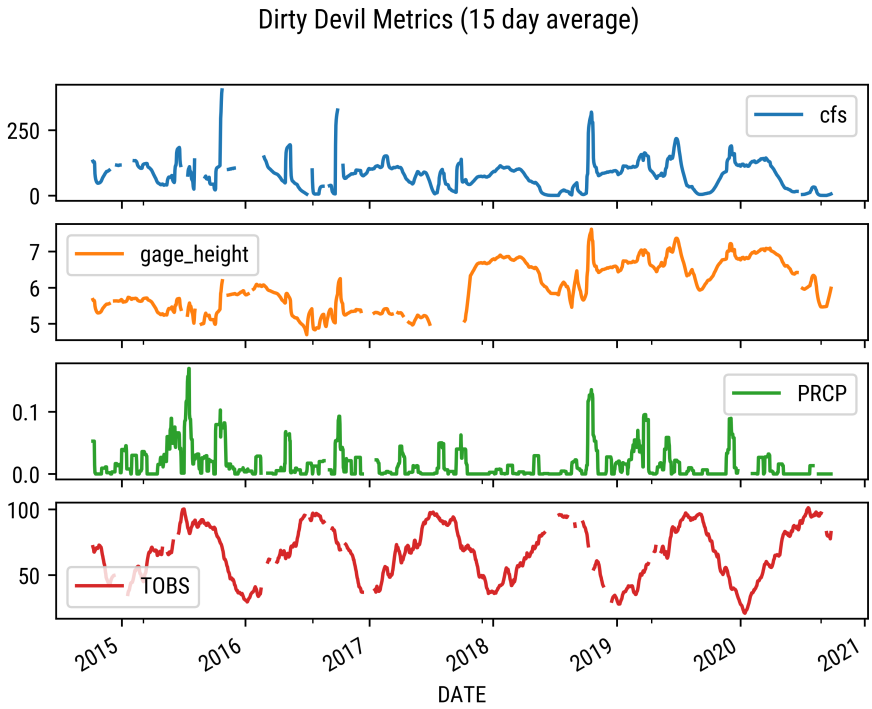


Figure 34.5: Visualization of 15-day average for the Dirty Devil River metrics.

hs Here's a scatterplot of temperature against river flow. I'm coloring this by month of the year:

```
>>> fig, ax = plt.subplots(dpi=600)
>>> dd2 = (dd
...   .groupby(pd.Grouper(key='datetime', freq='D'))
...   .median(numeric_only=True)
...   .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...          validate='1:1')
```

```
...  .query('cfs < 400')
... )
>>> (dd2
...  .plot.scatter(x='cfs', y='TOBS', c=dd2.DATE.dt.month,
...                ax=ax, cmap='hsv', alpha=.5)
... )
>>> ax.set_title('Observation Temperature (TOBS) '
...    'vs River Flow (cubic feet per sec)\nColored by Month')
>>> fig.savefig('img/pandas2/dd-scat.png', dpi=600, bbox_inches='tight')
<Figure size 3840x2880 with 2 Axes>
```

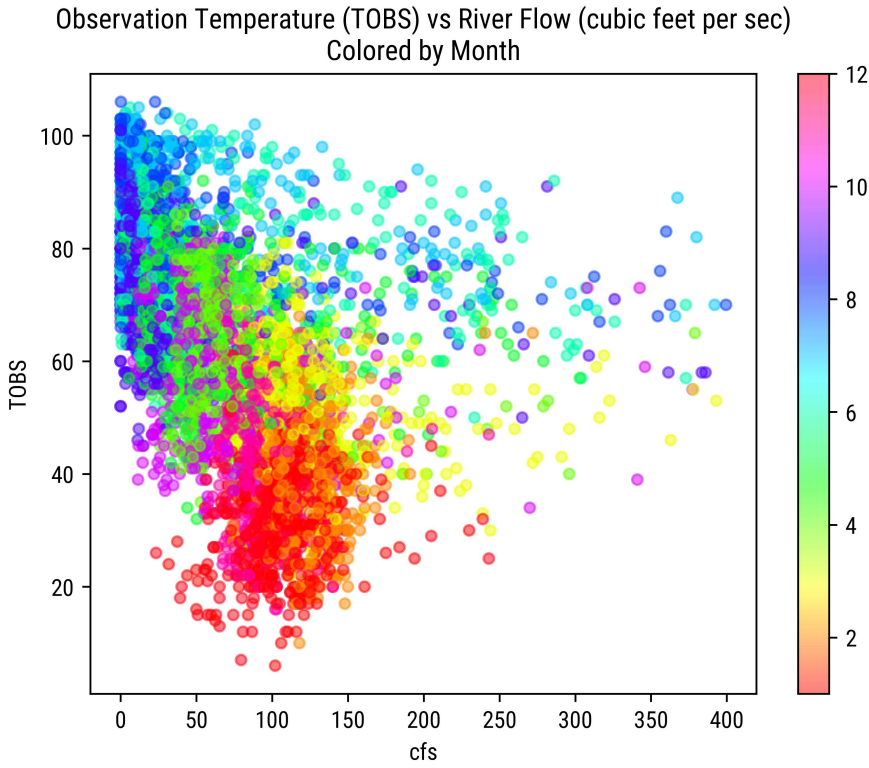


Figure 34.6: Scatterplot of temperature against river flow, colored by month.

34.12 Summary

Data can often have more utility if we combine it with other data. In the '70s, *relational algebra* was invented to describe various joins among tabular data.

The `.merge` method of the `DataFrame` lets us apply these operations to tabular data in the pandas world. This chapter described concatenation and the four basic joins that are possible via `.merge`.

34.13 Exercises

1. Create a dataframe for employees. It should have:

Index	name	company
0	Fred	AMZN
1	John	GOOG
2	Sally	GOOG
3	Annie	NFLX

Create a dataframe for location. It should have:

Index	ticker	location
0	AMZN	Seattle
1	GOOG	SF

2. What type of join do we need to do to get the location of each employee?
3. How would you validate the join?

Chapter 35

Exporting Data

This book has dealt with exploring, tweaking, and visualizing data. In addition, you may need to share data with others. This chapter will explore some of the mechanisms for exporting data.

35.1 Dirty Devil Data

In this section, we will revisit the Dirty Devil data. Let's load the flow and gage height data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...      .set_index('datetime')
```

```
...     )

>>> dd = tweak_river(df)
>>> print(dd)
                          agency_cd  site_no tz_cd   cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                              ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

35.2 Reading and Writing

There are a bunch of functions in pandas that deal with ingesting data. They all begin with `read_`. Similarly, there are analogous exporting methods on the dataframe object. These exporting methods start with `.to_`. We will talk about the common methods for exporting in this chapter.

35.3 Creating CSV Files

The Comma Separated Value (CSV) file is ubiquitous. It has been around since the early 70s. This format has the benefit of being human-readable, and

that is about where the benefits end. For a long time, there was no standard for CSV files. In 2005, a standard was released [1], but the damage was already done. As such, escaping mechanisms, encoding, header handling, and data types all suffer. You can see the pandas developers' attempts to deal with these issues when you look at the interface for the `pd.read_csv` function. It has over 40 parameters!

We can use the `.to_csv` method to write our data to a file. One thing to be aware of is that by default, pandas will write the index values in a CSV, but when reading a CSV, it will create a new index unless we specify a column for the index:

```
>>> dd.to_csv('/tmp/dd.csv')
```

Note

If you don't provide a filename, `.to_csv` will return the string content that would go into the file rather than writing the file. We will take advantage of that in this book to examine what the export looks like.

Let's look at what the first five lines of the export look like:

```
>>> print(dd.head(5).to_csv())
datetime,agency_cd,site_no,tz_cd,cfs,gage_height
2001-05-07 01:00:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:15:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:30:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:45:00-06:00,USGS,9333500,MDT,70.0,
2001-05-07 02:00:00-06:00,USGS,9333500,MDT,70.0,
```

If we wanted to read this and stick *datetime* in the index, we could use this code:

```
>>> dd2 = pd.read_csv('/tmp/dd.csv', index_col='datetime',
...     dtype_backend='pyarrow', engine='pyarrow')
```

Note that CSV files don't do much type conversion other than trying to convert strings to numbers. You can use the `parse_dates` parameter to attempt to convert the index into proper dates, but I would recommend creating a tweak function and revisiting the section on dealing with timezones to properly handle this (hint: it will look much like the `tweak_river` function from above).

There are a bunch of optional parameters for exporting CSV files, but I usually don't adjust them.

[1] https://www.ietf.org/rfc/rfc4180.txt

35.4 Reading ZIP Files with CSVs

If a single CSV is zipped up, pandas can read it directly with the `pd.read_csv` function. However, if multiple files are in the zip file, we need to use the `zipfile` module to extract the file and then read it. You could use a function like this:

```
import zipfile

def extract_csv_from_zip(zip_file, csv_file):
    with zipfile.ZipFile(zip_file) as z:
        z.extract(csv_file)

extract_csv_from_zip('/tmp/dd.zip', 'dd.csv')
pd.read_csv('dd.csv')
```

The `pd.read_csv` function can also read from a URL. Other `read_` functions will probably not support reading from a URL or a ZIP file.

35.5 Exporting to Excel

Another commonly used option is exporting the data frame to an Excel spreadsheet. The benefits of this method are that the world revolves around Excel. Everyone was taught how to use it in Kindergarten and business schools still teach it today. As such, Excel drives most of the business world.

Note

You must ensure the `openpyxl` library is installed to use Excel support. Simply installing the pandas library usually will not install full Excel support. Using pip is usually sufficient:

```
$ pip install openpyxl
```

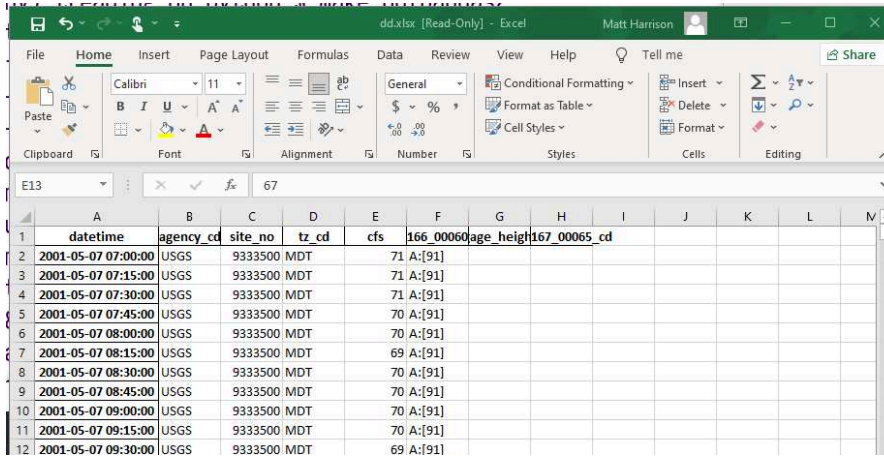
Let's export the data to Excel:

```
>>> dd.to_excel('/tmp/dd.xlsx')
Traceback (most recent call last):
  ...
ValueError: Excel does not support datetimes with timezones.
Please ensure that datetimes are timezone unaware before writing to Excel.
```

Whoops! That didn't quite work. We will need to strip the timezone information before exporting to Excel.

Note that exporting to Excel is a bit slower than writing CSV files. (Also note that Excel reads CSV files, so if you can deal without the limited formatting and type information that pandas inserts in an XLSX file, you might be ok with sending out CSV files to your Excel-junkie friends.):

```
>>> (dd
...  .reset_index()
...  .assign(datetime=lambda df_: df_.datetime.dt.tz_convert(tz=None))
...  .set_index('datetime')
...  .to_excel('/tmp/dd.xlsx')
... )
```



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	datetime	agency_cd	site_no	tz_cd	cfs	166_00060	age_heigh	167_00065_cd					
2	2001-05-07 07:00:00	USGS	9333500	MDT	71	A:[91]							
3	2001-05-07 07:15:00	USGS	9333500	MDT	71	A:[91]							
4	2001-05-07 07:30:00	USGS	9333500	MDT	71	A:[91]							
5	2001-05-07 07:45:00	USGS	9333500	MDT	70	A:[91]							
6	2001-05-07 08:00:00	USGS	9333500	MDT	70	A:[91]							
7	2001-05-07 08:15:00	USGS	9333500	MDT	69	A:[91]							
8	2001-05-07 08:30:00	USGS	9333500	MDT	70	A:[91]							
9	2001-05-07 08:45:00	USGS	9333500	MDT	70	A:[91]							
10	2001-05-07 09:00:00	USGS	9333500	MDT	70	A:[91]							
11	2001-05-07 09:15:00	USGS	9333500	MDT	70	A:[91]							
12	2001-05-07 09:30:00	USGS	9333500	MDT	69	A:[91]							

Figure 35.1: Excel export of pandas data frame.

Another benefit of Excel is writing a spreadsheet with multiple sheets. In this example, we write the 2010 data on one sheet, and 2011 data to another:

```
>>> writer = pd.ExcelWriter('/tmp/dd2.xlsx')
>>> dd2 = (dd
...  .reset_index()
...  .assign(datetime=lambda df_: df_.datetime.dt.tz_convert(tz=None))
...  .sort_values('datetime')
...  .set_index('datetime')
... )
>>> (dd2
...  .loc['2010':'2010-12-31']
...  .to_excel(writer, sheet_name='2010')
... )
>>> (dd2
...  .loc['2011':'2011-12-31']
...  .to_excel(writer, sheet_name='2011')
... )
```

35.6 Parquet

Parquet is a standard format for folks dealing with large amounts of data. It is a columnar format designed to be fast to read and write. It is also intended to be portable across different languages. It is a binary format, so it is not human-readable.

Let's export the data to Parquet:

```
>>> dd.to_parquet('/tmp/dd.parquet')
```

Let's roundtrip the data to make sure it is the same. As of pandas 2.2, I can't use the `pyarrow` backend when reading [1] because it has timezone information.:

```
>>> dd2 = pd.read_parquet('/tmp/dd.parquet')
>>> dd2.equals(dd)
Traceback (most recent call last)
  ...
ArrowInvalid: Timestamps already have a timezone: 'America/Denver'.
     Cannot localize to 'utc'.
```

Let's see if we can discern the differences between the two data frames.

```
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  string[pyarrow]
[right]: string[pyarrow]
```

This is odd because the types of the *agency_cd* column look the same.

```
>>> dd.agency_cd.dtype
string[pyarrow]

>>> dd2.agency_cd.dtype
string[pyarrow]
```

However, as of pandas 2.2, there are two different `string[pyarrow]` types.

```
>>> type(dd.agency_cd.dtype), type(dd2.agency_cd.dtype)
(pandas.core.dtypes.dtypes.ArrowDtype,
     pandas.core.arrays.string_.StringDtype)
```

[1] https://github.com/pandas-dev/pandas/issues/56282

If we ignore the types, the data frames are the same:

```
>>> pd.testing.assert_series_equal(dd2.agency_cd, dd.agency_cd,
...                                check_dtype=False)
```

35.7 Feather

Here is an option that is a relative newcomer. Feather is a binary file format for persisting columnar data in data frames. This is not a surprise because the creator of pandas works on it. Feather tends to be fast and keeps type information (for the most part). It is also supposed to be supported by other languages if you have to process data in R, Julia, or other languages.

> **Note**
>
> You will need to install the `feather-format` library to leverage this functionality.

Let's try exporting our data:

```
>>> (dd
...  .to_feather('/tmp/dd.fea')
... )
```

Let's see how this did in preserving our information:

```
>>> dd2 = pd.read_feather('/tmp/dd.fea')
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  string[pyarrow]
[right]: string[pyarrow]
```

Looks like this has the same issues as Parquet. Let's ignore the types:

```
>>> pd.testing.assert_frame_equal(dd2, dd, check_dtype=False)
```

Awesome! It looks like this works. Feather is relatively quick and supports most datatypes.

35.8 SQL

You can stick a data frame into a SQL table with the `.to_sql` method. In this example, we will create a SQLite database and insert our data into a table named *dd*.

Note

You will need to install `sqlalchemy` for SQL functionality.

```
% pip install sqlachemy
```

```
>>> import sqlite3
>>> con = sqlite3.connect('dd.db')
>>> dd.to_sql('dd', con, if_exists='replace')
539305
```

Let's read from the database:

```
>>> import sqlalchemy as sa
>>> eng = sa.create_engine('sqlite:///dd.db')
>>> sa_con = eng.connect()
>>> dd2 = pd.read_sql('dd', sa_con, index_col='datetime',
...     dtype_backend='pyarrow')
>>> pd.testing.assert_frame_equal(dd2, dd, check_dtype=False)
Traceback (most recent call last)
  ...
AssertionError: DataFrame.index are different

DataFrame.index classes are different
[left]:  Index([2001-05-07 01:00:00-06:00, 2001-05-07 01:15:00-06:00,
       2001-05-07 01:30:00-06:00, 2001-05-07 01:45:00-06:00,
       2001-05-07 02:00:00-06:00, 2001-05-07 02:15:00-06:00,
       2001-05-07 02:30:00-06:00, 2001-05-07 02:45:00-06:00,
       2001-05-07 03:00:00-06:00, 2001-05-07 03:15:00-06:00,
       ...
       2020-09-28 07:15:00-06:00, 2020-09-28 07:30:00-06:00,
       2020-09-28 07:45:00-06:00, 2020-09-28 08:00:00-06:00,
       2020-09-28 08:15:00-06:00, 2020-09-28 08:30:00-06:00,
       2020-09-28 08:45:00-06:00, 2020-09-28 09:00:00-06:00,
       2020-09-28 09:15:00-06:00, 2020-09-28 09:30:00-06:00],
      dtype='object', name='datetime', length=539305)
[right]: DatetimeIndex(['2001-05-07 01:00:00-06:00',
                '2001-05-07 01:15:00-06:00',
                '2001-05-07 01:30:00-06:00',
                '2001-05-07 01:45:00-06:00',
                '2001-05-07 02:00:00-06:00',
```

```
               '2001-05-07 02:15:00-06:00',
               '2001-05-07 02:30:00-06:00',
               '2001-05-07 02:45:00-06:00',
               '2001-05-07 03:00:00-06:00',
               '2001-05-07 03:15:00-06:00',
               ...
               '2020-09-28 07:15:00-06:00',
               '2020-09-28 07:30:00-06:00',
               '2020-09-28 07:45:00-06:00',
               '2020-09-28 08:00:00-06:00',
               '2020-09-28 08:15:00-06:00',
               '2020-09-28 08:30:00-06:00',
               '2020-09-28 08:45:00-06:00',
               '2020-09-28 09:00:00-06:00',
               '2020-09-28 09:15:00-06:00',
               '2020-09-28 09:30:00-06:00'],
              dtype='datetime64[ns, America/Denver]',
    name='datetime', length=539305, freq=None)

>>> print(dd2)
                    agency_cd  site_no tz_cd    cfs  gage_height
datetime
2001-05-07 01:00:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:15:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:30:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:45:00      USGS  9333500   MDT  70.00         <NA>
2001-05-07 02:00:00      USGS  9333500   MDT  70.00         <NA>
...                       ...      ...   ...    ...          ...
2020-09-28 08:30:00      USGS  9333500   MDT   9.53         6.16
2020-09-28 08:45:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:00:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:15:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:30:00      USGS  9333500   MDT   9.20         6.15

[539305 rows x 5 columns]
```

We could read the table from the database, but it was not equal to the original data. Closer inspection reveals that our index with timezone-aware dates was stored with timezone data, but this information was dropped when the data came out from the database.

Here is an example of using the `sqlite3` command-line tool to inspect the database:

```
$ sqlite3 dd.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
```

```
sqlite> .schema
CREATE TABLE IF NOT EXISTS "dd" (
"datetime" TIMESTAMP,
  "agency_cd" TEXT,
  "site_no" INTEGER,
  "tz_cd" TEXT,
  "cfs" REAL,
  "144166_00060_cd" TEXT,
  "gage_height" REAL,
  "144167_00065_cd" TEXT
);
CREATE INDEX "ix_dd_datetime"ON "dd" ("datetime");
sqlite> SELECT * FROM dd LIMIT 1;
2001-05-07 01:00:00-06:00|USGS|9333500|MDT|71.0|A:[91]||
sqlite>
```

If we update the index with timezone information, our dataframe is equal to the original data:

```
>>> pd.testing.assert_frame_equal(dd2
...     .reset_index()
...     .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, utc=True)
...           .dt.tz_convert('America/Denver'))
...     .set_index('datetime'),
...  dd, check_dtype=False
... )
```

35.9 JSON

Those who implement backend services often need to serialize data with JavaScript Object Notation (JSON). The pandas library has a `.to_dict` method to format data as a dictionary. It also has a `.to_json` method which supports exporting data formatted as JSON in multiple layouts.

Let's try out `.to_dict` first. While not strictly JSON, they are both dictionary representations (with JSON being serialized as a string):

```
>>> obj = dd.to_dict()
```

The result of `.to_dict` is a dictionary of dictionaries. The outer dictionary is keyed by the column names. The inner dictionary is keyed by the index. The values are the data in the data frame. Here is an example of the export of the first two rows of data:

```
>>> dd.head(2).to_dict()
{'agency_cd': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 'USGS',
```

```
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 'USGS'},
 'site_no': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 9333500,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'):
     9333500},
 'tz_cd': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 'MDT',
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 'MDT'},
 'cfs': {Timestamp('2001-05-07 01:00:00-0600', tz='America/Denver'):
     71.0,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 71.0},
 'gage_height': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): None,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): None}}
```

There is no corresponding `pd.read_dict` function. Rather, there is a class method on the data frame called `.from_dict`. Let's see how round tripping works with this method:

```
>>> dd2 = pd.DataFrame.from_dict(obj)
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: DataFrame.index are different

Attribute "names" are different
[left]:  [None]
[right]: ['datetime']
```

It looks like the index name was dropped. Not a big deal, let's fix that and try again:

```
>>> pd.testing.assert_frame_equal((dd2
...   .rename_axis(index='datetime')), dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  object
[right]: string[pyarrow]
```

Now it is complaining about type mismatches. Remember, I've been focusing on the optimized PyArrow types in this book, but the dataframe from `.from_dict` uses legacy types.

```
>>> dd2.dtypes
agency_cd       object
site_no          int64
tz_cd           object
cfs            float64
gage_height    float64
dtype: object
```

Let's address the types and test for equality again:

```
>>> pd.testing.assert_frame_equal((dd2
...   .rename_axis(index='datetime')
...   .astype(dict(dd.dtypes))), dd)
```

Note

Dictionary exports do not support duplicated index names. Unlike `.to_json` (when called with `orient='columns'` which raises a `ValueError`), it will silently drop data.

Ok, now on to `.to_json`. For pyarrow types (as of pandas 2.2), we need to specify the `orient` and `lines` parameter as below. We also need to push the index into a column since the *records* orientation that is required for pyarrow loading doesn't support index entries:

```
>>> dd.reset_index().to_json('/tmp/dd.json.gz', orient='records',
...      index=False, lines=True)
>>> dd2 = (pd.read_json('/tmp/dd.json.gz', orient='records', lines=True,
...                     dtype_backend='pyarrow', engine='pyarrow')
...   .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, utc=True)
...               .dt.tz_convert('America/Denver')))
>>> pd.testing.assert_frame_equal(dd2, dd.reset_index())
Traceback (most recent call last)
  ...
AssertionError: DataFrame.iloc[:, 0] (column name="datetime") are
     different

DataFrame.iloc[:, 0] (column name="datetime") values are different
     (100.0 %)
[index]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
     17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
     34, ...
[left]:  [989218800000, 989219700000, 989220600000, 989221500000,
     989222400000, 989223300000, 989224200000, 989225100000,
     989226000000, 9892...
[right]: [9892188000000000000, 9892197000000000000, 9892206000000000000,
```

```
    989221500000000000, 989222400000000000, 989223300000000000,
    9892242000...
```

These are not equal because the dates were stored as integers (milliseconds past the UNIX epoch of 1970). Let's put them back into *America/Denver* dates:

```
>>> dd3 = (dd2
...  .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, unit='ms')
...          .dt.tz_localize(tz='UTC')
...          .dt.tz_convert('America/Denver'))
...  .set_index('datetime')
... )

>>> print(dd3)
                          agency_cd  site_no tz_cd    cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.00
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.00
...                             ...      ...   ...    ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT   9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.20

                          gage_height
datetime
2001-05-07 01:00:00-06:00        <NA>
2001-05-07 01:15:00-06:00        <NA>
2001-05-07 01:30:00-06:00        <NA>
2001-05-07 01:45:00-06:00        <NA>
2001-05-07 02:00:00-06:00        <NA>
...                               ...
2020-09-28 08:30:00-06:00        6.16
2020-09-28 08:45:00-06:00        6.15
2020-09-28 09:00:00-06:00        6.15
2020-09-28 09:15:00-06:00        6.15
2020-09-28 09:30:00-06:00        6.15

[539305 rows x 5 columns]
```

Let's check if they are equal now:

```
>>> pd.testing.assert_frame_equal(dd3, dd)
```

Note

The `.to_json` method exports dates as epoch integers in strings:

```
>>> dd.head()
                          agency_cd   ... gage_height  144167_00065_cd
datetime                              ...
2001-05-07 01:00:00-06:00      USGS   ...         NaN              NaN
2001-05-07 01:15:00-06:00      USGS   ...         NaN              NaN
2001-05-07 01:30:00-06:00      USGS   ...         NaN              NaN
2001-05-07 01:45:00-06:00      USGS   ...         NaN              NaN
2001-05-07 02:00:00-06:00      USGS   ...         NaN              NaN

[5 rows x 7 columns]

>>> dd.head().to_json()[:60]
'{"agency_cd":{"989218800000":"USGS","989219700000":"USGS",'
```

When we read the JSON, pandas converts the epoch integers into naive dates (they have no timezone information).

Table 35.1: Chapter Methods

Method	Description
`df.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding='utf8', compression='infer', quoting=csv.QUOTE_MINIMAL, quotechar='"', line_terminator=os.linesep, chunksize=None, date_format=None, doublequote=True, escapechar=None, decimal='.', errors='strict', storage_options=None)`	Write to a CSV file (or stdout if not specified). Can specify `float_format` with `'%.3f'` (.1234 to .123).

Method	Description

`pd.read_csv(filepath_or_buffer,`
`  sep=',',`
`  header='infer',`
`  names=None,`
`  index_col=None,`
`  usecols=None,`
`  squeeze=False,`
`  prefix='',`
`  mangle_dupe_cols=True,`
`  dtype=None,`
`  engine=None,`
`  converters=None,`
`  true_values=None,`
`  false_values=None,`
`  skipinitialspace=False,`
`  skiprows=None,`
`  skipfooter=0,`
`  nrows=None,`
`  na_values=None,`
`  keep_default_na=True,`
`  na_filter=True,`
`  verbose=False,`
`  skip_blank_lines=True,`
`  parse_dates=False,`
`  infer_datetime_format=False,`
`  keep_date_col=False,`
`  date_parser=None,`
`  dayfirst=False,`
`  cache_dates=True,`
`  iterator=False,`
`  chunksize=None,`
`  compression='infer',`
`  thousands=None,`
`  decimal='.',`
`  lineterminator=None,`
`  quotechar='"',`
`  quoting=0,`
`  doublequote=True,`
`  escapechar=None,`
`  comment=None,`
`  encoding=None,`
`  encoding_errors='strict',`
`  dialect=None,`
`  error_bad_lines=None,`
`  ...)`
	Create a dataframe from a CSV file. Use `sep` to parse files with other delimiters. Use `names` to specify column names. Specify missing numeric values with `na_values`. Set `dayfirst=True` to use dates in a non US-centric environment.

Method	Description
`df.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options=None)`	Write an Excel formatted file or instance `ExcelWriter`.
`pd.ExcelWriter(path, engine=None, date_format=None, datetime_format=None, mode='w', storage_options=None, if_sheet_exists=None, engine_kwargs=None, **kwargs)`	Create a class for writing dataframes into sheets.

Method	Description
`pd.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=False, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, mangle_dupe_cols=True, storage_options=None)`	Create a dataframe from Excel file or dictionary (mapping sheet name to dataframe) if sheet_name is a list.
`df.to_feather(path)`	Write a Feather formatted file.
`pd.read_feather(path, columns=None, use_threads=True, storage_options=None)`	Create a dataframe from a Feather file.
`sqlite3.connect(database, timeout=None, detect_types=None, isolation_level=None, check_same_thread=None, factory=None, cached_statements=None, uri=None)`	Open a connection to a SQLite database. Use `database=':memory:'` to create RAM database.
`sa.create_engine(url, **kwargs)`	Create a SQLAlchemy engine from a database connection string.
`eng.connect()`	Get the database connection from a SQLAlchemy engine.

Method	Description
`df.to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None)`	Create a SQL table with `name` from the dataframe. Store the results in database specified by connection `con`. Can specify `'replace'` or `'append'` for `if_exists`.
`pd.read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None,)`	Create a dataframe from a SQL query.
`df.to_dict(orient='dict', into=dict)`	Serialize a dataframe into a dictionary. Orientation can be `'dict'` (column to dict of index to value), `'list'` (column to list of values), `'series'` (column to series), `'split'` (dictionary with index, columns, and data keys), `'records'` (list of dictionary (column to value)), `'index'` (dictionary of index to dictionary of column to value).
`pd.Dataframe.from_dict(data, orient='columns', dtype=None, columns=None)`	Create a dataframe from a dictionary. Orientation can be `'columns'` (like `'dict'` in `.to_dict`) or `'index'`.
`df.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=None, lines=False, compression='infer', index=True, indent=None, storage_options=None)`	Serialize a dataframe to JSON. Orientation can be `'columns'` (column to dict of index to value), `'list'` (column to list of values), `'series'` (column to series), `'split'` (dictionary with index, columns, and data keys), `'records'` (list of dictionary (column to value)), `'index'` (dictionary of index to dictionary of column to value), `'data'` (list of values), `'values'` (values array), `'table'` (dictionary of schema and data). Can change date format with `'iso'` (ISO8601).

Method	Description

`pd.read_json(path_or_buf=None,`
`  orient=None,`
`  typ='frame',`
`  dtype=None,`
`  convert_axes=None,`
`  convert_dates=True,`
`  keep_default_dates=True,`
`  numpy=False,`
`  precise_float=False,`
`  date_unit=None,`
`  encoding='utf-8',`
`  encoding_errors='strict',`
`  lines=False,`
`  chunksize=None,`
`  compression='infer',`
`  nrows=None,`
`  storage_options=None)`
	Create a dataframe from JSON.

`df.round(decimals=0)`	Create a dataframe with decimals rounded to given places.
`df.equals(other)`	Compares two dataframes if they have the same shape and values. Columns should have the same type.

`.to_parquet(path,`
`  engine='auto',`
`  compression='snappy',`
`  index=None,`
`  partition_cols=None,`
`  storage_options=None,`
`  **kwargs)`
	Export a dataframe to a Parquet file. Use the pyarrow engine by default if avaliable. Partiction columns can be specified as a list of column names. Storage options are passed to the backend file-system. For example if accessing a file on S3, you can pass `storage_options={'key': 'value'}` for host, port, and other options.

`pd.read_parquet(path,`
`  engine='auto',`
`  columns=None,`
`  storage_options=None,`
`  use_nullable_dtypes=None,`
`  filesystem=None,`
`  filters=None,`
`  **kwargs)`
	Create a dataframe from a Parquet file. Use the pyarrow engine by default if avaliable. You can specify specific columns to load. Storage options are passed to the backend file-system. For example, if accessing a file on S3, you can pass `storage_options={'key': 'value'}` for host, port, and other options.

35.10 Summary

There are many formats for exporting data with pandas. As I keep mentioning in this book, you will want to double-check your data after

exporting it to know what is in there. Some formats, like CSV, lose most type information. Others try to preserve it but may get hung up on timezones or rounding issues.

35.11 Exercises

With a dataset of your choice:

1. Export the data from a dataframe into a CSV file.
2. Export the data from a dataframe into a SQLite database.
3. Export the data from a dataframe into a Feather file.
4. Export the data from a dataframe into JSON.

Chapter 36

Styling Dataframes

In this chapter, I will demonstrate how to style a dataframe inside of Jupyter.

36.1 Loading the Data

We are going to use the Dirty Devil dataset for this section.

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...      .set_index('datetime')
...     )

>>> dd = tweak_river(df)
```

```
>>> print(dd)
```

```
                          agency_cd  site_no tz_cd   cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                             ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

Now that we have the basic data, I will do some aggregations and column creation. See if you can review the following code and determine what it is doing. I'll explain right after showing it, but after going through this book, you should start practicing reading code and making sure that you can understand what it is doing. You will need the sparklines library if you want to follow along.

```
>>> import sparklines
>>> agg_flow = (dd
...  #.resample('M')  # resample .agg doesn't support named aggregations
...  .groupby(pd.Grouper(freq='M'))
...  .agg(cfs=('cfs', 'median'),
...       total_flow=('cfs', lambda ser:(ser*15*60).sum()),
...       gage_height=('gage_height', 'median'),
```

```
...       flow_trend=('cfs', lambda ser: sparklines.sparklines(
...           ser
...           .fillna(0)
...           .resample('2D')
...           .median()
...           .fillna(0))
...           [0])
...       )
...   .assign(quarterly_flow=lambda df_: df_
...           .total_flow
...           .resample('Q')
...           .transform('sum'),
...         percent_quarterly_flow=lambda df2_: df2_
...           .total_flow / df2_.quarterly_flow,
...         off_goal=lambda df3_: df3_.percent_quarterly_flow-.33,
...         cost=lambda df4_: df4_.total_flow * .0002)
... )
>>> print(agg_flow.iloc[:,:4])
                            cfs   total_flow  gage_height  \
datetime
2001-05-31 00:00:00-06:00   47.0  105383700.0         <NA>
2001-06-30 00:00:00-06:00   23.0   17843400.0         <NA>
2001-07-31 00:00:00-06:00   17.0    7781400.0         <NA>
2001-08-31 00:00:00-06:00   52.5  192848220.0         <NA>
2001-09-30 00:00:00-06:00   26.0   42819300.0         <NA>
...                          ...          ...          ...
2020-05-31 00:00:00-06:00  21.25   60721029.0         6.51
2020-06-30 00:00:00-06:00   10.2   24475410.0         6.28
2020-07-31 00:00:00-06:00   10.8   67073337.0         6.05
2020-08-31 00:00:00-06:00   0.32   11042316.0         5.48
2020-09-30 00:00:00-06:00   5.79   10369692.0         6.01

                                 flow_trend
datetime
2001-05-31 00:00:00-06:00
2001-06-30 00:00:00-06:00
2001-07-31 00:00:00-06:00
2001-08-31 00:00:00-06:00
2001-09-30 00:00:00-06:00
...                                     ...
2020-05-31 00:00:00-06:00
2020-06-30 00:00:00-06:00
2020-07-31 00:00:00-06:00
2020-08-31 00:00:00-06:00
2020-09-30 00:00:00-06:00
```

```
[233 rows x 4 columns]
```

There might have been a curveball in here… the sparklines library. Let's skip that for now and describe the rest of the chain.

Group by the months in the index (note that I'm using named aggregations and that, as the comment states, the result of the `.resample` method does not support named aggregations). For each group, calculate the median of the *cfs* column, calculate *total_flow* from the the *cfs* column (it is the 15-minute value, so we multiply it by 15 to get the minutes and 60 to get the seconds), and create a *flow_trend* column that uses sparklines.

After grouping, we are going to make some more columns. *quarterly_flow* resamples our monthly data to the quarterly level and sums it up. *percent_quarterly_flow* divides *total_flow* by *quarterly_flow*. The *off_goal* column assumes that each month should contribute 33% of the quarterly water flow and measures how far off we are from that goal. The *cost* column calculates expense, assuming it costs two-hundredths of a cent per cubic foot of water.

36.2 Sparklines

A sparkline[1] is a small plot drawn without axes or coordinates created by Edward Tufte. The intent is to show a general trend. The sparklines[2] library in Python is a Unicode bar chart implementation of this idea.

If you have a series of numbers, you can create a Unicode string that represents them:

```
>>> import sparklines
>>> sparklines.sparklines(range(10))
['▁▂▃▄▅▆▇█']
```

So let's revisit this chunk of code:

```
...         flow_trend=('cfs', lambda ser: sparklines.sparklines(
...             ser
...             .resample('2D')
...             .median()
...             .fillna(0))
...             [0])
```

We use the *cfs* column, resample to every two days (remember this series, `ser`, is data for every 15 minutes for a single month), calculate the median value of river flow, and fill in missing values with zero. This gives us a series with the median two-day value. We pass this data into the sparklines library

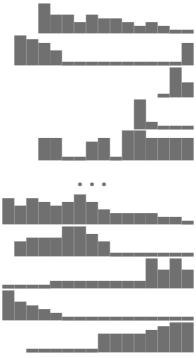
[1] This creative use of embedding sparklines was inspired by this tweet https://twitter.com/pmbaumgartner/status/1084645440224559104

[2] https://github.com/deeplook/sparklines

to generate a Unicode bar plot. The sparklines library returns a list with the string inside of it, so we pull the chart out of the list.

The resulting column looks like this:

```
>>> agg_flow.flow_trend
datetime
2001-05-31 00:00:00-06:00
2001-06-30 00:00:00-06:00
2001-07-31 00:00:00-06:00
2001-08-31 00:00:00-06:00
2001-09-30 00:00:00-06:00
                          ...
2020-05-31 00:00:00-06:00
2020-06-30 00:00:00-06:00
2020-07-31 00:00:00-06:00
2020-08-31 00:00:00-06:00
2020-09-30 00:00:00-06:00
Freq: M, Name: flow_trend, Length: 233, dtype: object
```



36.3 The .style Attribute

Up to this point, most of the results of our chains have been a series or dataframe. The `.style` attribute of a dataframe allows you to chain, but you can only chain more styling methods, you cannot update the dataframe. If you want to style the output, you should do that as your chain's last step(s).

36.4 Formatting

One thing you can do with styling is control the formatting. Let's make the *cost* column show dollar signs, change the format of the *datetime* column, convert *percent_quarterly_flow* to a percentage and add a plus or minus to the *off_goal* column. This is done with the `.format` method.

For each column, we pass a dictionary with the column name as the key and the format string as the value. The format string syntax is the same as the format string syntax for the `.str.format` method. Inside the curly braces, you can specify the format string following a colon. For example, the format string for the *cost* column is `'${:,.0f}'`. The dollar sign will be added to the front of the number, the comma will be added to separate thousands, and the `.0f` will round the number to zero decimal places.

The *datetime* column is formatted with the format string `'{:%Y-%m}/01'`. The curly braces are replaced with the value of the column and the `:%Y-%m` is a special format string that formats the date as `YYYY-MM` and the `/01` adds the day and formats it as `YYYY-MM/01`.

I use a dictionary comprehension because the format strings are the same for the *cfs*, *total_flow*, and *quarterly_flow* columns. The dictionary comprehension creates a dictionary with the column name as the key and

the format string as the value. The ** operator unpacks the dictionary comprehension into the other dictionary.

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
         na_rep='Missing')
)
```

✓ 0.1s

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Figure 36.1: Changing the style of the columns.

36.5 Embedding Bar Plots

Next, we will embed a bar plot in the cell background. We use the `.bar` method for that.

The *cfs* bars are clipped (via `vmax`) to the 95% quantile, otherwise, they don't show up due to outliers. The *off_goal* bars specify two colors to distinguish positive from negative.

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
         na_rep='Missing')
 .bar(subset='cfs', color='#c87fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef78c')
 .highlight_max(color='lightgreen', axis='index')
)
```

✓ 0.2s Python

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480
8	2002/01/01	136.0	197745300.0	Missing		638525700.0	31.0%	-2.0%	$39,549

Figure 36.2: Adding bar plots to *cfs* and *off_goal* columns. Highlighting missing and maximum values.

36.6 Highlighting

There are a few styling methods to highlight values. You can highlight missing values, the minimum, the maximum, a range, or a quantile range. Our example highlights missing and maximum values.

36.7 Heatmaps and Gradients

You can shade the background based on the value of the cell. We demoed this in the cross-tabulation section. Here, we will use a red colormap to color the *cost* column. We will set `vmax` to indicate that anything over $25,000 is over budget.

Depending on the data, you may want to choose a different colormap. For correlations, you want to use a diverging colormap. For positive numeric data, you may consider an increasing or continuous colormap. You can use a Matplotlib colormap. They are shown at the end of the chapter.

36.8 Captions

The `.set_caption` allows you to specify text for a caption. This will appear before the dataframe.

36.9 CSS Properties

The `.set_properties` method lets you set CSS properties to each cell.

The `.map` method will also let you place CSS properties. You pass in a function that takes the value of the cell and returns a string with the CSS properties for that cell.

Another way to set CSS styling is with the `.set_table_styles` method. This method allows you to specify the selector and its properties.

36.10 Stickiness

If you find it annoying to lose the column headers when scrolling down a dataframe or losing the index when scrolling to the side, you are in luck. The `.set_sticky` method will keep the headers in place when scrolling. Note, however, that you should call this method at the end of your chain because you might lose the stickiness if you set some CSS styles after it.

36.11 Hiding the Index

Finally, you can hide the index. In the image, you can see that we have made the index disappear. Through this book, we have seen that unless you are doing grouping, I recommend keeping essential data out of the index (and then moving it out when you are done with grouping).

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
         na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds', vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color: {"#4589ae" if val > 50 else "#a05cbc"}',
      subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props': 'background-color: pink; font-size:14pt;'}])
)
```

✓ 0.1s

Dirty Devil River Flow

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.2%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Figure 36.3: Using `.set_properties` to set CSS properties on the *datetime* column. Notice that the *datetime* column is gray. Using `.map` to set CSS properties on the *cfs* column. Notice that the font and background of *cfs* has changed. Using `.set_table_styles` to set CSS properties on hovering. Notice that when you hover over a cell, the style gets set.

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
         na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds', vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color: {"#4589ae" if val > 50 else "#a05cbc"}',
      subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props': 'background-color: pink; font-size:14pt;'}])
 .set_sticky(axis='columns')
 .hide(axis='index')
)
```

✓ 0.1s

Dirty Devil River Flow

datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.2%	$21,077
2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Figure 36.4: Using `.hide` to hide the index.

If you display a dataframe, removing the index can remove a potential distraction. Use the `.hide` method accessed from the `.style` attribute to hide it when displaying data in Jupyter.

36.12 Final Styling Code

The final styling code is shown below.

```
(agg_flow
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow',
                                       'quarterly_flow']}},
    na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds',
                      vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color:
        {"#4589ae" if val > 50 else "#a05cbc"}',
         subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props':

                    'background-color: pink; font-size:14pt;'}])
 .set_sticky(axis='columns')
 .hide(axis='index')
)
```

36.13 Display Options

Pandas has a few options for displaying dataframes.

You can inspect the default values by printing them off of the `pd.options.display` attribute.

I generally leave these alone, but often, my students ask how to view more rows or columns. The `display.max_rows` option finds the default value for the number of displayed rows.

```
>>> pd.options.display.max_rows
10
```

You can override the default by setting the value to a different number. But I recommend using the context manager `pd.option_context` to change the value temporarily. You do that like this:

```
>>> with pd.option_context('display.max_rows', 4,
...                        'display.max_columns', 2):
...     print(agg_flow)
                            cfs  ...        cost
datetime                         ...
2001-05-31 00:00:00-06:00  47.00  ...  21076.7400
2001-06-30 00:00:00-06:00  23.00  ...   3568.6800
...                          ...  ...         ...
2020-08-31 00:00:00-06:00   0.32  ...   2208.4632
2020-09-30 00:00:00-06:00   5.79  ...   2073.9384

[233 rows x 8 columns]
```

You can use the `pd.describe_option` function to get more information about the option.

```
>>> pd.describe_option('display.max_rows')
display.max_rows : int
    If max_rows is exceeded, switch to truncate view. Depending on
    `large_repr`, objects are either centrally truncated or printed as
    a summary view. 'None' value means unlimited.

    In case python/IPython is running in a terminal and `large_repr`
    equals 'truncate' this can be set to 0 and pandas will auto-detect
    the height of the terminal and print a truncated object which fits
    the screen height. The IPython notebook, IPython qtconsole, or
    IDLE do not run in a terminal and hence it is not possible to do
    correct auto-detection.
    [default: 60] [currently: 10]
```

Table 36.1: Display Options in Pandas

Option Name	Description
`chop_threshold`	Controls at what level floating point numbers are truncated (chopped).
`colheader_justify`	Controls the justification of the headers. Values can be 'right', 'left', etc.
`date_dayfirst`	Displays the date with the day first, such as '13/01/2020' for January 13, 2020.
`date_yearfirst`	Displays the date with the year first, such as '2020/01/13' for January 13, 2020.

Option Name	Description
`encoding`	Sets the default encoding for outputting objects.
`expand_frame_repr`	Whether to stretch the DataFrame representation across the console or not.
`float_format`	Formatter for floating point numbers.
`html`	Settings related to rendering DataFrames as HTML.
`large_repr`	Sets the method of displaying large DataFrames.
`max_categories`	Maximum number of categories displayed when printing a categorical column.
`max_columns`	Maximum number of columns displayed when printing a DataFrame.
`max_colwidth`	Maximum width of each column.
`max_dir_items`	Maximum number of items displayed in the `dir()` of a Pandas object.
`max_info_columns`	Maximum number of columns for which a frame will be considered narrow (for printing info).
`max_info_rows`	Maximum number of rows for which to display a summary (for printing info).
`max_rows`	Maximum number of rows to display.
`max_seq_items`	Maximum number of elements in each sequence (rows, columns) to print.
`memory_usage`	Specifies whether total memory usage of the DataFrame elements (including index) should be shown when printing.
`min_rows`	The minimum number of rows to show in the DataFrame output.
`multi_sparse`	Controls sparsifying of hierarchical indices.
`notebook_repr_html`	Whether to use HTML representation in an IPython notebook.
`pprint_nest_depth`	Controls the depth to which nested structures are printed.
`precision`	Sets the precision of floating point numbers.
`show_dimensions`	Whether to print the dimensions of the DataFrame.
`unicode`	Controls whether to use Unicode characters to pretty-print DataFrame objects.
`width`	Width of the display in characters.

Table 36.2: Styling Methods

Method	Description
`.format( formatter=None, subset=None, na_rep=None, precision=None, decimal='.', thousands=None, escape=None)`	Return a `Styler`. `formatter` can be a string, a callable that takes a value and returns the string representation, or a dictionary mapping column names to Python format specifiers or callables. `subset` is a column or list of columns to apply (if not using a dictionary `formatter`). Use `na_rep` to specify alternate representation for missing numbers. Use `precision` to specify floating point decimal places. Use `decimal` to change decimal separator. Use `thousands` to specify character to insert for thousands separator. The `escape` parameter can specify `html` or `latex` to provide properly escaped cells.
`.bar( subset=None, axis=0, color='#d65f5f', width=100, align='left', vmin=None, vmax=None)`	Return a `Styler`. Draw a bar chart in cell background. `subset` is a column or list of columns to apply to. If you specify a two-tuple for `color`, the first is for negative values. `width` is the percentage of the cell to use. `align` defaults to `left` side, you can specify `zero` for the center of the cell, or `mid` for center to right aligned if all values are negative or (max-min)/2. Use `vmin` and `vmax` to clip values.
`.highlight_max( null_color='red', subset=None, axis=0, props=None)`	Return a `Styler` that highlights maximum values. You can specify CSS properties with `props`.
`.highlight_null( null_color='red', subset=None, props=None)`	Return a `Styler` that highlights missing values. You can specify CSS properties with `props`.
`.background_gradient( cmap='PuBu', low=0, high=0, axis=0, subset=None, text_color_threshold=0.408, vmin=None, vmax=None, gmap=None)`	Return a `Styler` that highlights background colors based on values. Use `cmap` to specify a Matplotlib colormap.
`.set_caption(caption)`	Return a `Styler`. Create HTML caption. If using LaTex, can specify a tuple with full and short captions.

Method	Description
`.set_properties( subset=None, **kwargs)`	Return a `Styler`. Set CSS properties on each cell. You can specify them as keyword arguments, but will probably need to use an unpacked dictionary since many CSS properties have dashes in them (ie: `**{'background-color': 'red'}`).
`.map( func, subset=None, **kwargs)`	Return a `Styler`. Set CSS properties on each cell. The `func` takes the current value of the cell and returns a string with the CSS properties. You can pass additional arguments to `func` with `kwargs`.
`.set_table_styles( table_styles, axis=0, overwrite=True)`	Return a `Styler`. Set CSS properties on table, columns, rows, or HTML selectors. `table_styles` can be a list of dictionaries (mapping `'selector'` to CSS selector, and `'props'` to CSS properties) or a dictionary (mapping column names (or index names if `axis=1`) to row CSS selectors (a list of the selector and the property).
`.set_sticky( axis=0, pixel_size=None, levels=None)`	Return a `Styler`. Sets columns to sticky if `axis=1`. Set index to stick if `axis=0`. Make sure you call this as one of the last styling operations, otherwise it might not work.
`.hide(subset=None, axis=0, level=None, names=False)`	Return a `Styler`. Hide the index or columns or specified values.

36.14 Summary

In this chapter, we demonstrated many of the styling features of pandas. There are other features that we didn't explain. Feel free to explore those and see if they will be useful. We also demonstrated how to create a sparkplot as Unicode.

36.15 Exercises

With a dataset of your choice:

1. Color the background of the first two columns blue.
2. Format the numeric values by specifying precision and thousands separator.
3. Include a bar plot in a column.
4. Set a background gradient for a column.
5. Make the column headers sticky.

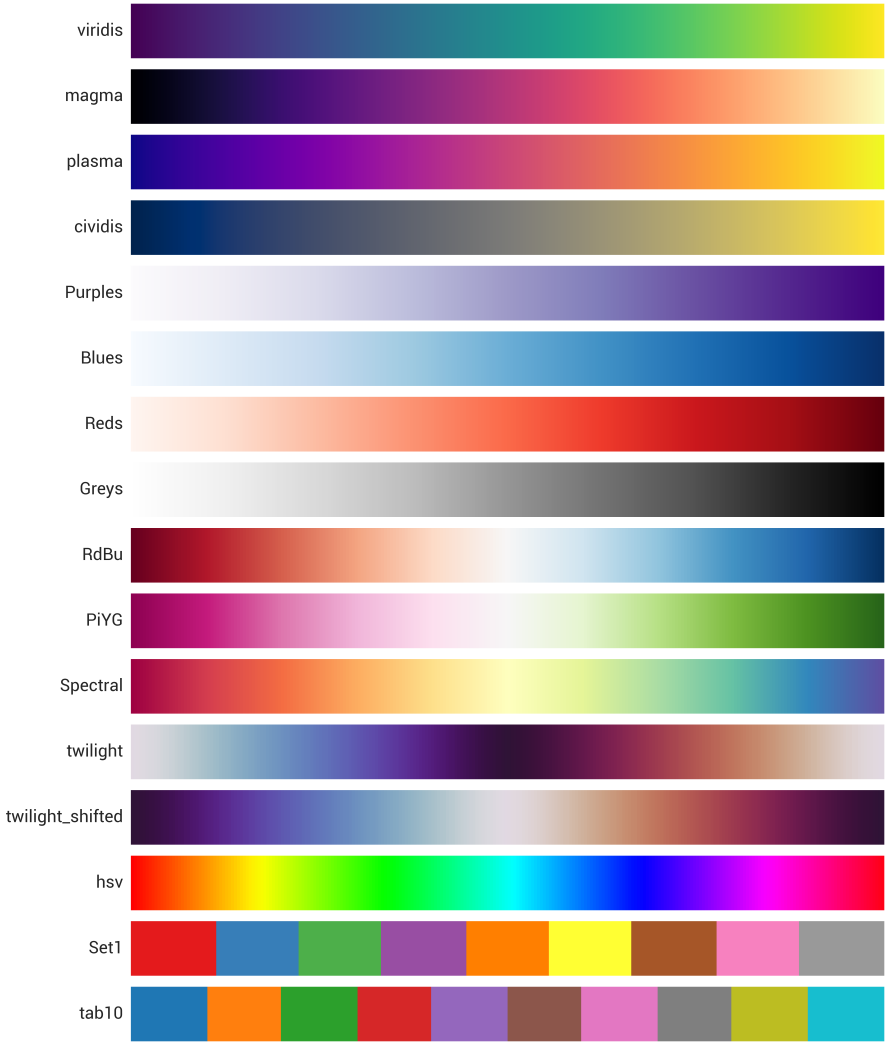


Figure 36.5: Select Matplotlib colormaps. Continuous (viridis through cividis). Increasing (Purples through Greys). Diverging (RdBu through Spectral). Cyclic (twilight through hsv). Categorical (Set1 and tab10).

Chapter 37

Debugging Pandas

In this chapter, we will explore various techniques for debugging Pandas.

37.1 Checking if Dataframes are Equal

The first technique we will explore is checking whether two dataframes are equal. This is especially useful after serializing and deserializing data and, unfortunately, is a little more complicated than it should be. We can use the `.equals` method to check if two dataframes are equal, but if they are not, diagnosing the problem is hard.

Let's step through an example with our Dirty Devil data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
```

```
...                 'gage_height'] ]
...     )

>>> dd = tweak_river(df)
>>> print(dd)

                       datetime agency_cd  ...   cfs gage_height
0      2001-05-07 01:00:00-06:00      USGS  ...  71.0        <NA>
1      2001-05-07 01:15:00-06:00      USGS  ...  71.0        <NA>
2      2001-05-07 01:30:00-06:00      USGS  ...  71.0        <NA>
3      2001-05-07 01:45:00-06:00      USGS  ...  70.0        <NA>
4      2001-05-07 02:00:00-06:00      USGS  ...  70.0        <NA>
...                          ...       ...  ...   ...         ...
539300 2020-09-28 08:30:00-06:00      USGS  ...  9.53        6.16
539301 2020-09-28 08:45:00-06:00      USGS  ...   9.2        6.15
539302 2020-09-28 09:00:00-06:00      USGS  ...   9.2        6.15
539303 2020-09-28 09:15:00-06:00      USGS  ...   9.2        6.15
539304 2020-09-28 09:30:00-06:00      USGS  ...   9.2        6.15

[539305 rows x 6 columns]
```

Now, let's roundtrip this through JSON and evaluate whether we get the same data back:

```
>>> dd2 = pd.read_json(dd.to_json(), dtype_backend='pyarrow')
>>> dd.equals(dd2)
False
```

Nope, the data is different! Our task is to find out why `dd` and `dd2` are different. The `.equals` method is not particularly helpful in helping us figure out why they aren't equal. Let's dive in a little more.

Let's see if the `.eq` method will help us out. It returns a dataframe of booleans indicating where values are equal:

```
>>> print(dd.eq(dd2))
       datetime agency_cd  ...   cfs gage_height
0         False      True  ...  True        <NA>
1         False      True  ...  True        <NA>
2         False      True  ...  True        <NA>
3         False      True  ...  True        <NA>
4         False      True  ...  True        <NA>
...         ...       ...  ...   ...         ...
539300    False      True  ...  True        True
539301    False      True  ...  True        True
539302    False      True  ...  True        True
539303    False      True  ...  True        True
```

```
539304     False      True  ...  True        True

[539305 rows x 6 columns]
```

We can use the `.sum` or `.mean` trick to quantify the counts or percentages of values that are the same.

```
>>> (dd
...    .eq(dd2)
...    .sum()
... )
datetime            0.0
agency_cd      539305.0
site_no        539305.0
tz_cd          539305.0
cfs            491257.0
gage_height    413649.0
dtype: double[pyarrow]
```

The pandas library has a function hidden away in the `testing` namespace that helps a little, `pd.testing.assert_frame_equal`. This function is meant to be used for the core developers of pandas when developing and testing the library, but let's try it here:

```
>>> pd.testing.assert_frame_equal(dd, dd2)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="datetime") are different

Attribute "dtype" are different
[left]:  datetime64[ns, America/Denver]
[right]: timestamp[ns][pyarrow]
```

Ok, it hints that the *datetime* column has different types. As we saw in the JSON serialization section, we lose timezone information when we serialize. Let's address that and try again:

```
>>> from_json = (dd2
...     .assign(datetime=dd2.datetime
...         .dt.tz_localize('UTC')
...         .dt.tz_convert('America/Denver'))
... )
>>> pd.testing.assert_frame_equal(dd, from_json)

Traceback (most recent call last)
  ...
```

```
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
    name="datetime") are different

Attribute "dtype" are different
[left]:  datetime64[ns, America/Denver]
[right]: timestamp[ns, tz=America/Denver][pyarrow]
```

Now it complains that the types are different. We can ask the method to ignore type information:

```
>>> pd.testing.assert_frame_equal(dd, from_json, check_dtype=False)
```

In this case, no assertion is raised, it is quiet! However `.equals` still fails:

```
>>> dd.equals(from_json)
False
```

Let's change the types using `.astypes` and try again.

```
>>> pd.testing.assert_frame_equal(dd, from_json.astype(dict(dd.dtypes)))
```

It doesn't complain, so we should feel confident they are now equal.
For good measure, let's check `.equals`:

```
>>> (dd.equals(from_json.astype(dict(dd.dtypes))))
False
```

That seems weird. The `.equals` method is returning `False`.
Let's try the `check_exact` parameter for `assert_frame_equals` and see if we can see what is different:

```
>>> pd.testing.assert_frame_equal(dd, from_json.astype(dict(dd.dtypes)),
...  check_exact=True
... )
```

This works as well. We will do a little more exploring. I'm going to store my changes to `dd2` in `dd3`.

```
dd3 = from_json.astype(dict(dd.dtypes))
```

Let's use the `.eq` method combined with `.all` to see the differences. It looks like some of the values in the *cfs* column differ.

```
>>> dd.eq(dd3).all()
datetime        True
agency_cd       True
site_no         True
tz_cd           True
cfs            False
gage_height    False
dtype: bool
```

Let's examine those with the `.ne` method. This method will return a boolean array where the values are not equal in a series:

```
>>> print(dd[dd.cfs.ne(dd3.cfs)])
                      datetime agency_cd  ...   cfs gage_height
96246  2007-07-03 19:45:00-06:00      USGS  ...   1.7        <NA>
96247  2007-07-03 20:00:00-06:00      USGS  ...   1.7        <NA>
96248  2007-07-03 20:15:00-06:00      USGS  ...   1.7        <NA>
96249  2007-07-03 20:30:00-06:00      USGS  ...   1.7        <NA>
96250  2007-07-03 20:45:00-06:00      USGS  ...   1.7        <NA>
...                          ...       ...  ...   ...         ...
538678 2020-09-21 21:00:00-06:00      USGS  ...  6.56        6.06
538728 2020-09-22 09:30:00-06:00      USGS  ...  6.56        6.06
538735 2020-09-22 11:15:00-06:00      USGS  ...  6.56        6.06
538739 2020-09-22 12:15:00-06:00      USGS  ...  6.56        6.06
538753 2020-09-22 15:45:00-06:00      USGS  ...  6.56        6.06

[1867 rows x 6 columns]
```

The index here are rows where values are different. Ok, let's look at the values for *cfs* from row label *96246* from both of the datasets:

```
>>> dd.loc[96246].cfs, dd3.loc[96246].cfs
(1.7, 1.7000000000000002)
```

We found a culprit! It looks like we have rounding issues.

Here is a little function I wrote to help diagnose where dataframes are not the same:

```
>>> def cmp_dfs(df1, df2, round_amt=3):
...     diff_cols = set(df1.columns) ^ set(df2.columns)
...     if diff_cols:
...         print(f'Different columns {diff_cols}')
...     if df1.shape != df2.shape:
...         print(f'Different shapes {df1.shape} {df2.shape}')
...     bad = False
...     for col in df1.columns:
...         s1 = df1[col]
...         s2 = df2[col]
...         if s1.equals(s2):
...             continue
...         bad = True
...         if s1.dtype != s2.dtype:
...             print(f'{col} types differ {s1.dtype} vs {s2.dtype}')
...         if s1.dtype in [float, 'double[pyarrow]']:
...             if s1.round(round_amt).equals(s2.round(round_amt)):
...                 print(f'{col} has rounding differences'
```

```
...                     f'{df1[s1.ne(s2)][col].dropna().iloc[0]} '
...                     f'vs {df2[s1.ne(s2)][col].dropna().iloc[0]}')
...         else:
...             diff = (df1
...                     .loc[s1.ne(s2)]
...                     .assign(other=s2)
...                     .loc[[col, "other"]]
...                     .dropna())
...             print(f'{col} differs {diff}')
...     if not bad:
...         print('Same')

>>> cmp_dfs(dd, dd3)
cfs has rounding differences1.7 vs 1.7000000000000002
gage_height has rounding differences3.28 vs 3.2800000000000002
```

Feel free to leverage this function and the others described in this section to discover why your dataframes are not equal.

Hopefully this section gave you some insight into determining what equality really means for your dataframes.

37.2 Debugging Chains

In this section, we will explore debugging chains of operations on dataframes or series. I have taught thousands of people pandas during my career. I've also seen a lot of pandas code from clients and students. Almost universally, it is messy code. I get it. I used to write pandas code that way too. Making liberal use of chaining and creating functions to tweak my data has gone a long way toward remedying my ails.

I have been a vocal proponent of chaining on social media. Occasionally, I will hear someone protest that they don't like chaining. When asked why, they usually flounder. Excuses like excess code, copying data (yes, there are copies, but no more than non-chained pandas), and "hard to debug" are common complaints. I don't buy excess code. In fact, I think chaining produces less code. The pandas library is an in-memory library that works by copying data. This argument is a moot point. Let's address the debugging complaint.

I'm going to show a "tweak" function that I created to analyze fuel economy data.

Let's load the raw data:

```
>>> import pandas as pd
>>> autos = pd.read_csv('https://github.com/mattharrison/datasets/raw/'
...     'master/data/vehicles.csv.zip', dtype_backend='pyarrow',
...     engine='pyarrow')
```

```
>>> print(autos)
       barrels08  barrelsA08  ...  phevHwy  phevComb
0      15.695714         0.0  ...        0         0
1      29.964545         0.0  ...        0         0
2      12.207778         0.0  ...        0         0
3      29.964545         0.0  ...        0         0
4      17.347895         0.0  ...        0         0
...          ...         ...  ...      ...       ...
41139  14.982273         0.0  ...        0         0
41140   14.33087         0.0  ...        0         0
41141  15.695714         0.0  ...        0         0
41142  15.695714         0.0  ...        0         0
41143  18.311667         0.0  ...        0         0

[41144 rows x 83 columns]
```

Here is my tweak function:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...             .groupby(tz_offset)
...             [time_col]
...             .transform(lambda s: pd.to_datetime(s)
...                 .dt.tz_localize(s.name, ambiguous=True)
...                 .dt.tz_convert(tz_name))
...            )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                   'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
```

```
...                     .astype('int8[pyarrow]'),
...                offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...                str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                          autos.createdOn.str.slice(-4)),
...                createdOn=lambda df_: to_tz(df_, 'str_date',
...                          'offset', 'America/New_York'),
...                ffs=autos.eng_dscr.str.contains('FFS')
...               )
...        .astype(types)
...        .loc[:, final_cols]
...      )
```

Say you came across this `tweak_autos` function and wanted to understand what it does. First of all, realize that it is written like a recipe, step by step:

- Limit the columns to `col_cols` to make it easier to deal with.
- Create various columns (`.assign`).
- Convert column types (`.astype`).
- Keep only the columns in `final_cols`.

Haters of chaining say there is no way to debug this. I have a few ways to debug the chain. The first is using comments. I comment out all of the operations and then go through them one at a time. This comes in really handy to visually see what is happening as the chain progresses. Let's look at all four steps with debugging.

First, pulling out the columns. The raw data has 83 columns. I don't want all of those, so I limit the columns to have less distraction during my data cleanup. I would make a simple function that did just that. Then, I would validate that the function works as intended:

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...       .loc[:, orig_cols]
...     )

>>> print(tweak_autos(autos))
      city08  comb08  ...                createdOn  year
0         19      21  ...  Tue Jan 01 00:00:00 E...  1985
1          9      11  ...  Tue Jan 01 00:00:00 E...  1985
2         23      27  ...  Tue Jan 01 00:00:00 E...  1985
```

```
3          10      11  ...  Tue Jan 01 00:00:00 E...  1985
4          17      19  ...  Tue Jan 01 00:00:00 E...  1993
...       ...     ...  ...                       ...   ...
41139      19      22  ...  Tue Jan 01 00:00:00 E...  1993
41140      20      23  ...  Tue Jan 01 00:00:00 E...  1993
41141      18      21  ...  Tue Jan 01 00:00:00 E...  1993
41142      18      21  ...  Tue Jan 01 00:00:00 E...  1993
41143      16      18  ...  Tue Jan 01 00:00:00 E...  1993

[41144 rows x 14 columns]
```

The next step is creating the new columns that I care about. I would add them one at a time to an `.assign` method. Checking with each of them that they work. Here's the first column. Updating the *drive* column, by filling in missing values and then converting it to a category:

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              )
...     )
```

Let's check that it works. I would do this by calling `.value_counts` on the new column:

```
>>> (tweak_autos(autos)
...   .drive
...   .value_counts()
... )
drive
Front-Wheel Drive             14236
Rear-Wheel Drive              13831
4-Wheel or All-Wheel Drive     6648
All-Wheel Drive                3015
4-Wheel Drive                  1460
Other                          1189
2-Wheel Drive                   507
Part-time 4-Wheel Drive         258
Name: count, dtype: int64
```

That's better. Here's the rest of the columns. I'm not going to go over checking each column here. But I hope you get the idea.

The *trany* (transmission) column appears to encode two pieces of data, the number of speeds and whether the automobile is automatic or manual. I'm going to pull out those features into their own columns and then we delete *trany*. The *eng_desc* (engine description) column appears almost freeform. I will create an *indicator column* indicating whether the string *FFS* (fuel feedback system) was in the description. Then, I will discard the *eng_desc* column later.

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...             .groupby(tz_offset)
...             [time_col]
...             .transform(lambda s: pd.to_datetime(s)
...                 .dt.tz_localize(s.name, ambiguous=True)
...                 .dt.tz_convert(tz_name))
...            )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...     )
```

The next step is to check the types. I do that with the `.dtypes` attribute.

```
>>> tweak_autos(autos).dtypes
city08          int64[pyarrow]
comb08          int64[pyarrow]
```

```
highway08     int64[pyarrow]
cylinders     int64[pyarrow]
displ        double[pyarrow]
                 ...
automatic      bool[pyarrow]
speeds         int8[pyarrow]
offset       string[pyarrow]
str_date     string[pyarrow]
ffs            bool[pyarrow]
Length: 19, dtype: object
```

These types are looking pretty good. In this case, pandas limits the rows shown. So, I don't know if other columns have problematic types. Rather than try and bump that limit, I would look at the value counts to see the unique types.

```
>>> (tweak_autos(autos)
...    .dtypes
...    .value_counts())
int64[pyarrow]                     7
string[pyarrow]                    6
bool[pyarrow]                      2
double[pyarrow]                    1
category                           1
datetime64[ns, America/New_York]   1
int8[pyarrow]                      1
Name: count, dtype: int64
```

From this output, it looks like we have many `int64[pyarrow]` types that I can probably shrink. Let's check by using the `.describe` method and look at the *max* row. (In this case, I will transpose it to see a little more data):

```
>>> print(tweak_autos(autos)
...    .describe()
...    .T)
              count         mean  ...     75%     max
city08      41144.0    18.369045  ...    20.0   150.0
comb08      41144.0    20.616396  ...    23.0   136.0
highway08   41144.0    24.504667  ...    28.0   124.0
cylinders   40938.0     5.717084  ...     6.0    16.0
displ       40940.0     3.294238  ...     4.3     8.4
fuelCost08  41144.0  2362.335942  ...  2700.0  7400.0
range       41144.0     0.793506  ...     0.0   370.0
year        41144.0  2001.535266  ...  2011.0  2020.0
speeds      41144.0     5.325029  ...     6.0    20.0

[9 rows x 8 columns]
```

Let's inspect the string columns and see if it makes sense to convert those to categoricals:

```
>>> print(tweak_autos(autos)
...   .select_dtypes('string')
...   .nunique())
eng_dscr     557
make         136
model       4058
trany         37
offset         2
str_date     269
dtype: int64
```

We are going to drop *eng_dscr*, *trany*, and *str_date*. The other columns could be categories.

I'm going to make a dictionary with the types for the columns and pass that into .astype:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...             .groupby(tz_offset)
...             [time_col]
...             .transform(lambda s: pd.to_datetime(s)
...                 .dt.tz_localize(s.name, ambiguous=True)
...                 .dt.tz_convert(tz_name))
...            )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...            'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...            'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...            'make': 'category', 'model': 'category',
...            'cylinders': 'int8[pyarrow]'}
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
```

```
...             offset=autos.createdOn
...                    .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                    .replace('EDT', 'EST5EDT'),
...             str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                       autos.createdOn.str.slice(-4)),
...             createdOn=lambda df_: to_tz(df_, 'str_date',
...                       'offset', 'America/New_York'),
...             ffs=autos.eng_dscr.str.contains('FFS')
...            )
...      .astype(types)
...     )
```

Let's check the types now:

```
>>> tweak_autos(autos).dtypes
city08         int16[pyarrow]
comb08         int16[pyarrow]
highway08       int8[pyarrow]
cylinders       int8[pyarrow]
displ         double[pyarrow]
                  ...
automatic       bool[pyarrow]
speeds          int8[pyarrow]
offset        string[pyarrow]
str_date      string[pyarrow]
ffs             bool[pyarrow]
Length: 19, dtype: object
```

That is looking good. Our next step is to drop columns that we don't need. I generally do this in two steps. First, I use the `.drop` method to remove the columns I don't want. Then, I inspect the columns of the result and change the `.drop` into a `.loc`.

Why go through the hassle? This is a lesson the creator of Polars, Ritchie Vink, taught me. I had never considered it before, but once he said it, it made perfect sense. Ritche said rather than dropping columns, you should select the columns that you want to keep. I guess this is a variation of the *Robustness Principle* or *Postel's Law*: be strict in what you do and tolerant of what others do.

The variation is that you should focus on the columns you want, not those you don't want. This can also protect you in the future. If a dataset adds features and you only worry about dropping specific columns, the dimensions of your data (the number of columns) could change, which is problematic for applications like machine learning.

Because I'm a lazy programmer and don't want to type out all of the columns manually, I use `.drop` and inspect the `.columns` attribute to "type" the columns for me. Then, I copy those columns and replace the `.drop` with `.loc`:

Here's my .drop:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...            'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...            'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...            'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      .drop(columns=['trany', 'eng_dscr', 'offset', 'str_date'])
...     )
```

Now let's get the columns that I want from this:

```
>>> tweak_autos(autos).columns
Index(['city08', 'comb08', 'highway08', 'cylinders', 'displ',
       'drive', 'fuelCost08', 'make', 'model', 'range', 'createdOn',
```

```
       'year', 'automatic', 'speeds', 'ffs'],
      dtype='object')
```

And change the `.drop` to `.loc`

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...           'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...           'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...           'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...           'displ', 'drive', 'fuelCost08', 'make', 'model', 'range',
...           'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      #.drop(columns=['trany', 'eng_dscr', 'offset', 'str_date'])
```

```
...         .loc[:, final_cols]
...     )
```

Let's make sure that it works:

```
>>> print(tweak_autos(autos))
       city08  comb08  ...  speeds    ffs
0          19      21  ...       5   True
1           9      11  ...       5  False
2          23      27  ...       5   True
3          10      11  ...       3   <NA>
4          17      19  ...       5   True
...       ...     ...  ...     ...    ...
41139      19      22  ...       4   True
41140      20      23  ...       5   True
41141      18      21  ...       4   True
41142      18      21  ...       5   True
41143      16      18  ...       4   True

[41144 rows x 15 columns]
```

If you were to come across this function and wanted to debug it, I would comment out the operations in the chain. Then, I would run the function, uncommenting and inspecting the output as I went through the code.

This is effectively what I did to create the chain in the first place.

Don't let a long chain scare you. Look at it as steps of a recipe. If you know what each step is doing, you will be in a good place.

Commenting out chain operations is an effective debugging technique.

37.3 Debugging Chains Part II

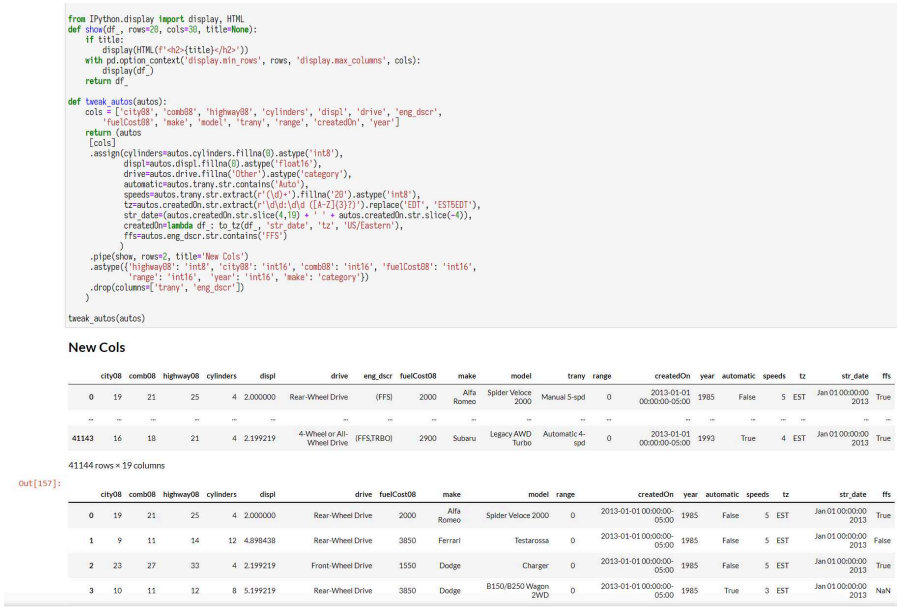
I won't stop with the debugging techniques. Here's another one that allows you to look at the intermediate state after any method call in a chain. Remember that the `.pipe` method will pass the current state of a dataframe or series into a function. This function can return anything but normally returns a dataframe or a series.

Imagine a function that returns the dataframe (or series) that was passed into it, but it also prints out the representation on the screen. That is what the `show` function below does. This function leverages the `display` function in Jupyter to create an optional HTML header and display the dataframe as HTML rather than a string version:

```
>>> from IPython.display import display, HTML
>>> def show(df_, rows=20, cols=30, title=None):
...     if title:
...         display(HTML(f'<h2>{\hangindent=1em\hangafter=1title}</h2>'))
```

```
...     with pd.option_context('display.min_rows', rows,
...                            'display.max_columns', cols):
...         display(df_)
...     return df_
```

Let's stick `show` into the `tweak_autos` function right after the new columns are created but before we convert the types. The image shows the new output.



```
from IPython.display import display, HTML
def show(df_, rows=20, cols=30, title=None):
    if title:
        display(HTML(f'<h2>{title}</h2>'))
    with pd.option_context('display.min_rows', rows, 'display.max_columns', cols):
        display(df_)
    return df_

def tweak_autos(autos):
    cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ', 'drive', 'eng_dscr',
        'fuelCost08', 'make', 'model', 'trany', 'range', 'createdOn', 'year']
    return (autos
     [cols]
     .assign(cylinders=autos.cylinders.fillna(0).astype('int8'),
             displ=autos.displ.fillna(0).astype('float16'),
             drive=autos.drive.fillna('Other').astype('category'),
             automatic=autos.trany.str.contains('Auto'),
             speeds=autos.trany.str.extract(r'(\d)+').fillna('20').astype('int8'),
             tz=autos.createdOn.str.extract(r'\d\d:\d\d ([A-Z]{3}?)').replace('EDT', 'EST5EDT'),
             str_date=(autos.createdOn.str.slice(4,19) + ' ' + autos.createdOn.str.slice(-4)),
             createdOn=lambda df_: to_tz(df_, 'str_date', 'tz', 'US/Eastern'),
             ffs=autos.eng_dscr.str.contains('FFS')
            )
     .pipe(show, rows=2, title='New Cols')
     .astype({'highway08': 'int8', 'city08': 'int16', 'comb08': 'int16', 'fuelCost08': 'int16',
              'range': 'int16',  'year': 'int16', 'make': 'category'})
     .drop(columns=['trany', 'eng_dscr'])
    )

tweak_autos(autos)
```

New Cols

	city08	comb08	highway08	cylinders	displ	drive	eng_dscr	fuelCost08	make	model	trany	range	createdOn	year	automatic	speeds	tz	str_date	ffs
0	19	21	25	4	2.000000	Rear-Wheel Drive	(FFS)	2000	Alfa Romeo	Spider Veloce 2000	Manual 5-spd	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
41143	16	18	21	4	2.199219	4-Wheel or All-Wheel Drive	(FFS,TRBO)	2900	Subaru	Legacy AWD Turbo	Automatic 4-spd	0	2013-01-01 00:00:00-05:00	1993	True	4	EST	Jan 01 00:00:00 2013	True

41144 rows × 19 columns

Out[157]:

	city08	comb08	highway08	cylinders	displ	drive	fuelCost08	make	model	range	createdOn	year	automatic	speeds	tz	str_date	ffs
0	19	21	25	4	2.000000	Rear-Wheel Drive	2000	Alfa Romeo	Spider Veloce 2000	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
1	9	11	14	12	4.898438	Rear-Wheel Drive	3850	Ferrari	Testarossa	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	False
2	23	27	33	4	2.199219	Front-Wheel Drive	1550	Dodge	Charger	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
3	10	11	12	8	5.199219	Rear-Wheel Drive	3850	Dodge	B150/B250 Wagon 2WD	0	2013-01-01 00:00:00-05:00	1985	True	3	EST	Jan 01 00:00:00 2013	NaN

Figure 37.1: Inserting `show` function inside of chain to debug intermediate state.

Another helpful tool during chaining is to inspect the shape of the intermediate dataframes to ensure that you are not accidentally removing all the rows or that you don't have a combinatoric explosion of data following a merge. You could leverage `.pipe` with a function that prints out the shape of the data:

```
>>> def shape(df_):
...     print(df_.shape)
...     return df_
```

37.4 Debugging Chains Part III

We are on a roll with debugging. Let's keep going!

Another complaint that people who justify not using chains is that they really want to have the intermediate states of each operation. For example, they might write the `tweak_autos` chain like this:

```
cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
        'drive', 'eng_dscr', 'fuelCost08', 'make', 'model',
        'trany', 'range', 'createdOn', 'year']
autos2 = autos[cols]
cyl_nona = autos.cylinders.fillna(0)
cyl_int8 = cyl_nona.astype('int8')
autos2['cylinders'] = cyl_int8
displ_nona = autos.displ.fillna(0)
displ_float16 = displ_nona.astype('float16')
autos2['displ'] = displ_float16
...
autos2.drop(columns=['trany', 'eng_dscr'], inplace=True)
```

I left out much of the column updating and type changing, but I think you get the point: most users pull out a column, mess with it, and finally stick it back in. Anti-chainers claim that this ability to inspect the state using any of these variables is useful. (Nevermind that the variables sit around in global memory, wasting space.)

Admittedly, the intermediate state might be useful during the development (in fact, you saw that I was inspecting the internal state as I built up the chain), but that utility quickly fades away during analysis and also creates a mess. I don't care about the intermediate state when my chain is done. I care about the output. The intermediate state is just noise.

If you really want the intermediate state of the dataframe, guess what? You can get that by leveraging `.pipe`. Below is a function, `get_var`, that will create a global variable with the contents of the intermediate value of a dataframe. Just shim this function into the chain with `.pipe`:

```
>>> def get_var(df, var_name):
...     globals()[var_name] = df
...     return df
```

Let's use `get_var` to create a variable, `new_cols`, with the state of `tweak_autos` immediately after creating the new columns:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
```

```
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                    'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.replace('', 'Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .pipe(get_var, 'new_cols')
...      .astype(types)
...      .loc[:, final_cols]
...     )
```

Let's inspect the intermediate state stored in `new_cols`:

```
>>> tweak_autos(autos)
>>> print(new_cols)
       city08  comb08  ...              str_date    ffs
0          19      21  ...  Jan 01 00:00:00 2013   True
1           9      11  ...  Jan 01 00:00:00 2013  False
2          23      27  ...  Jan 01 00:00:00 2013   True
3          10      11  ...  Jan 01 00:00:00 2013   <NA>
4          17      19  ...  Jan 01 00:00:00 2013   True
...       ...     ...  ...                   ...    ...
41139      19      22  ...  Jan 01 00:00:00 2013   True
41140      20      23  ...  Jan 01 00:00:00 2013   True
41141      18      21  ...  Jan 01 00:00:00 2013   True
```

```
41142      18      21  ...  Jan 01 00:00:00 2013   True
41143      16      18  ...  Jan 01 00:00:00 2013   True

[41144 rows x 19 columns]
```

You can use the `.pipe` method to debug intermediate states of chained operations.

37.5 Debugging Chains Part IV

Another option for debugging code in Jupyter is to leverage the pdb debugger. In Jupyter notebook, there are two main options to do this. One is to run the command `%debug` command immediately after encountering an exception. The other way to invoke the debugger is to explicitly invoke the `set_trace` function.

Let's look at the first option. I will insert a link into the chain to call an `err` function that raises an exception. When we run this, it will raise an exception:

```
>>> def err(*args):
...     1/0

>>> def tweak_autos(autos):
...     cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      [cols]
...      .assign(cylinders=autos.cylinders.fillna(0).astype('int8'),
...              displ=autos.displ.fillna(0).astype('float16'),
...              drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .pipe(err)
```

```
...         .astype({'highway08': 'int8', 'city08': 'int16',
...                  'comb08': 'int16', 'fuelCost08': 'int16',
...                  'range': 'int16',  'year': 'int16',
...                  'make': 'category'})
...         .drop(columns=['trany', 'eng_dscr'])
...     )

>>> res = tweak_autos(autos)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

This raises an exception. But if you run this in Jupyter, you can drop into a debugger after raising the exception. In a new cell, run the command `%debug`.

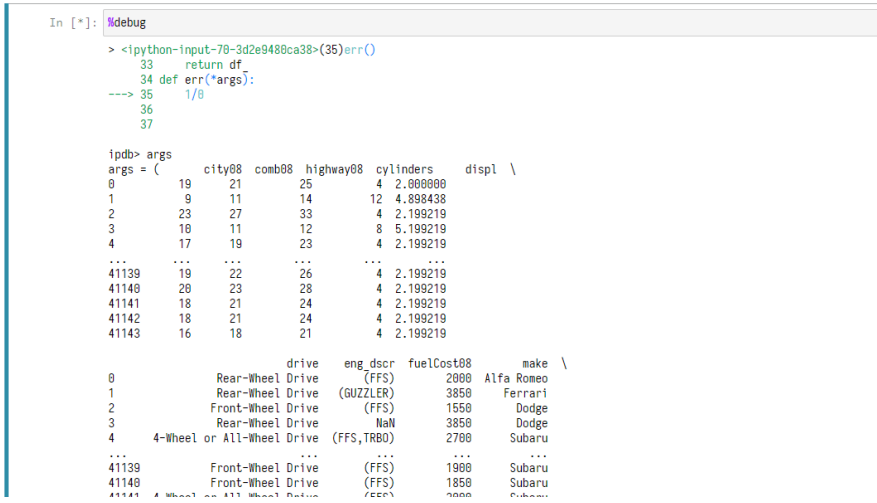


Figure 37.2: Run the `%debug` cell magic after executing a cell that raises and exception.

You are now in the debugger. Here is a brief overview of the pdb commands that I find useful:

- `h` - (help) Show the commands.
- `l` - (list) List code around break.
- `s` - (step) Step into function/method.
- `w` - (where) Show where you are in stack.
- `u` - (up) Move up in the stack.
- `d` - (down) Move down in the stack.
- `c` - (continue) Continue running code.
- `q` - (quit) Quit running code.

Another mechanism to drop into the debugger is to call the `set_trace` function. Replace `err` with this function:

```
>>> from IPython.core.debugger import set_trace
>>> def err(*args):
...     set_trace()
```

Note

While the debugger is running in Jupyter, no other cells can run. Make sure you type `c` or `q` to finish your debugging session before executing other cells.

37.6 Debugging Apply (and Friends)

It can be confusing to keep track of what pandas passes around when you call `.apply`, `.assign`, `.groupby(...) .apply`, `.groupby(...) .agg`, `.groupby(...) .transform`, `.pipe`, and others. What is getting passed in? A series, dataframe, group? One answer is to look at the documentation, which is generally good (although there are some holes). Also, it can be useful to have access to the object being passed around so you can play with it in Jupyter and figure out what you want your `.apply` (or `.groupby(...) .apply` or `.groupby(...) .agg` ...) to do.

We can take a similar approach to debugging with `.pipe` and create a function to help us. The `debug_var` function accepts an item (this is what we want to check). This function will store the item in the `debug_item` variable (we can overwrite this if we desire) for future inspection. Then, the function raises a `DebugException` to prevent further processing. We will pass this function into `.apply`.

Here is the function:

```
>>> class DebugException(Exception):
...     pass

>>> def debug_var(thing, *, name='debug_item', raise_ex=True):
...     globals()[name] = thing
...     if raise_ex:
...         raise DebugException
...     return thing
```

Make sure you have our normal tweak function.

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
```

```
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                   'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      .loc[:, final_cols]
...     )
```

Let's use the function to explore how `.apply` works. What gets passed into the `.apply` method? Plug in the `debug_var` function and find out. Let's use it on the Fuel Economy data:

```
>>> autos2 = tweak_autos(autos)
>>> autos2.apply(debug_var, name='this')
Traceback (most recent call last)
  ...
DebugException:
```

What is `this`?

```
>>> this
0       19
1        9
2       23
3       10
```

```
4        17
         ..
41139    19
41140    20
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int16[pyarrow]
```

It looks like `this` is a single column (or series). The `.apply` method will call our function on every single column.

I've removed the stack trace from the exception above, but I try to convince my students that they should try to understand it. In a previous section, we discussed the debugger and how to step through the stack to explore what is happening.

Let's re-run this, but with the `axis=1` parameter to see what gets passed into our function:

```
>>> autos2.apply(debug_var, axis=1)
Traceback (most recent call last):
  ...
DebugException

>>> debug_item
city08                                19
comb08                                21
highway08                             25
cylinders                              4
displ                                2.0
drive                  Rear-Wheel Drive
fuelCost08                          2000
make                          Alfa Romeo
model                Spider Veloce 2000
range                                  0
createdOn     2013-01-01 00:00:00-05:00
year                                1985
automatic                          False
speeds                                 5
tz                                   EST
str_date           Jan 01 00:00:00 2013
ffs                                 True
Name: 0, dtype: object
```

It looks like it is passing in a row represented as a series.

Let's try it with `.assign`:

```
>>> (autos2
...  .assign(new_col=debug_var)
... )
Traceback (most recent call last):
  ...
DebugException
```

```
>>> debug_item
       city08  comb08  highway08  ...   tz              str_date    ffs
0          19      21         25  ...  EST  Jan 01 00:00:00 2013   True
1           9      11         14  ...  EST  Jan 01 00:00:00 2013  False
2          23      27         33  ...  EST  Jan 01 00:00:00 2013   True
3          10      11         12  ...  EST  Jan 01 00:00:00 2013    NaN
4          17      19         23  ...  EST  Jan 01 00:00:00 2013   True
...       ...     ...        ...  ...  ...                   ...    ...
41139      19      22         26  ...  EST  Jan 01 00:00:00 2013   True
41140      20      23         28  ...  EST  Jan 01 00:00:00 2013   True
41141      18      21         24  ...  EST  Jan 01 00:00:00 2013   True
41142      18      21         24  ...  EST  Jan 01 00:00:00 2013   True
41143      16      18         21  ...  EST  Jan 01 00:00:00 2013   True

[41144 rows x 17 columns]
```

```
>>> (autos2
...  .assign(new_col=debug_var)
... )
Traceback (most recent call last)
  ...
DebugException:
```

```
>>> print(debug_item)
       city08  comb08  ...  speeds    ffs
0          19      21  ...       5   True
1           9      11  ...       5  False
2          23      27  ...       5   True
3          10      11  ...       3  False
4          17      19  ...       5   True
...       ...     ...  ...     ...    ...
41139      19      22  ...       4   True
41140      20      23  ...       5   True
41141      18      21  ...       4   True
41142      18      21  ...       5   True
41143      16      18  ...       4   True

[41144 rows x 15 columns]
```

It looks like `debug_item` is the whole dataframe.

Let's try it when we call `.groupby(...).agg` with a dictionary:

```
>>> (autos2.groupby('make').agg({'city08': debug_var}))
Traceback (most recent call last):
  ...
DebugException

>>> debug_item
Series([], Name: city08, dtype: int16)
```

Looks like `debug_item` is the *city08* column.

You get the idea. With the intermediate variable in hand, you should be able to make progress on your analysis.

Note

In addition to creating a variable, you can also combine this technique with the `%debug` cell magic. This will drop you into a debugger at the point that the exception was raised.

37.7 Memory Usage

Because pandas requires that you load your data into RAM, you need to be aware of the size of your data. Because pandas doesn't mutate data (in general), you will need some overhead to work with data. I typically recommend that my clients have 3-10x more memory than the size of the data they are analyzing.

One way to explore the data is to look at the `.info` method. Just remember to use the `memory_usage='deep'` option so you take into account any Python objects the dataframe might use (strings for example):

```
>>> dd.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 539305 entries, 0 to 539304
Data columns (total 6 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   datetime     539305 non-null  datetime64[ns, America/Denver]
 1   agency_cd    539305 non-null  string[pyarrow]
 2   site_no      539305 non-null  int64[pyarrow]
 3   tz_cd        539305 non-null  string[pyarrow]
 4   cfs          493124 non-null  double[pyarrow]
 5   gage_height  433377 non-null  double[pyarrow]
dtypes: datetime64[ns, America/Denver](1), double[pyarrow](2),
     int64[pyarrow](1), string[pyarrow](2)
memory usage: 24.2 MB
```

Another option is to use the 3rd party library *memory-profiler*. You can install this with pip:

```
pip install memory-profiler
```

If you are using Jupyter, you will want to load the extension to access the `%%memit` cell magic. Run this command in a cell in Jupyter:

```
%load_ext memory_profiler
```

Now, you can leverage the `%%memit` cell magic. This will run a cell and track from the operating system's point of view how much memory the process has allocated. It also reports how much the memory usage has grown:

```
>>> %%memit
>>> dd = tweak_river(df)
peak memory: 304.42 MiB, increment: 254.99 MiB
```

If you find that you are using too much memory, consider:

- Sampling rows to limit the data
- Only loading columns you need
- Changing types to more efficient types (i.e., using `'int8[pyarrow]'` instead of `'int64[pyarrow]'` when representing human ages, or using `'category'` for categorical data)
- Acquiring more memory (or using a machine with more memory)

37.8 Copy On Write

One of the best features of pandas 2 is called copy on write. In legacy pandas, when you operated on a dataframe, pandas would often make a copy of the data. This was a problem when you had a large dataframe and limited memory. In pandas 2, pandas will only make a copy of the data if you modify the data. This is a vast improvement.

But you need to enable this feature. You can do this by setting `pd.options.mode.copy_on_write` to `True`. Before I do that, I'm going to use the `psutil` library to see how much memory the `tweak_jb` function is using.

Make sure that you install `psutil` (using `pip` or your favorite tool):

```
pip install psutil
```

Let's write a helper function to see how much memory is being used by a process. Because I need to track the previous memory usage, I'm going to create a class:

```
import psutil

import os

class MemoryTracker:
    def __init__(self):
        self.previous_memory = self._get_process_memory()

    def _get_process_memory(self):
        process = psutil.Process(os.getpid())
        memory_info = process.memory_info()
        return memory_info.rss / (1024 ** 2)  # Convert bytes to megabytes

    def __call__(self, df, txt=''):
        current_memory = self._get_process_memory()
        memory_growth = current_memory - self.previous_memory
        print(f"{txt} Process memory usage: {current_memory:.2f} MB"\
              f" (growth: {memory_growth:.2f} MB)")
        self.previous_memory = current_memory
        return df
```

Now, I'm going to instrument the `tweak_jb` function to see how much memory it uses as the chain of operations is applied.

```
>>> import catboost as cb
>>> import numpy as np
>>> import pandas as pd

>>> import collections

>>> def get_uniq_cols(jb):
...     counter = collections.defaultdict(list)
...     for col in sorted(jb.columns):
...         period_count = col.count('.')
...         if period_count >= 2:
...             part_end = 2
...         else:
...             part_end = 1
...         parts = col.split('.')[:part_end]
...         counter['.'.join(parts)].append(col)
...     uniq_cols = []
...     for cols in counter.values():
...         if len(cols) == 1:
...             uniq_cols.extend(cols)
...     return uniq_cols
```

```
>>> def prep_for_ml(df):
...     # remove pandas/pyarrow types
...     return (df
...      .assign(**df.select_dtypes(['number', 'bool']).astype(float),
...              **{col:df[col].astype(str).fillna('')
...                for col in df.select_dtypes(['object',
...                                             'category', 'string'])})
...     )

>>> def predict_col(df, col):
...     df = prep_for_ml(df)
...     missing = df.query(f'~{col}.isna()')
...     cat_idx = [i for i,typ in enumerate(df.drop(columns=[col]).dtypes)
...                if str(typ) == 'object']
...     X = (missing
...          .drop(columns=[col])
...          .values
...         )
...     y = missing[col]
...     model = cb.CatBoostRegressor(iterations=20, cat_features=cat_idx)
...     model.fit(X,y, cat_features=cat_idx)
...     pred = model.predict(df.drop(columns=[col]))
...     return df[col].where(~df[col].isna(), pred)

>>> def tweak_jb_mt(jb, mem_tracker):
...     uniq_cols = get_uniq_cols(jb)
...     return (jb
...      .pipe(mem_tracker, txt='Start')
...      [uniq_cols]
...      .pipe(mem_tracker, txt='After uniq_cols')
...      .rename(columns=lambda c: c.replace('.', '_'))
...      .pipe(mem_tracker, txt='After rename')
...      .assign(age=lambda df_:df_.age.str.slice(0,2)
...                                .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
...          company_size=lambda df_:df_.company_size.replace(
...            {'Just me': '1', '': pd.NA,
...            'Not sure': pd.NA, 'More than 5,000': '5000', '2–10': '2',
...            '11–50':'11','51–500': '51', '501–1,000':'501',
...            '1,001–5,000':'1001'}).astype('int64[pyarrow]'),
...          country_live=lambda df_:df_.country_live.astype('category'),
...          employment_status=lambda df_:df_.employment_status
...             .fillna('Other').astype('category'),
...          is_python_main=lambda df_:df_.is_python_main
```

```
...             .astype('category'),
...          team_size=lambda df_:df_.team_size
...             .str.split(r'-', n=1, expand=True)
...             .iloc[:,0].replace('More than 40 people', '41')
...             .where(df_.company_size!=1, '1')
...             .replace('', pd.NA)
...             .astype('int8[pyarrow]'),
...          years_of_coding=lambda df_:df_.years_of_coding.astype(str)
...             .replace('Less than 1 year', '.5')
...             .str.extract(r'(\.?\d+)').astype('float64[pyarrow]'),
...          python_years=lambda df_:df_.python_years
...              .replace('Less than 1 year', '.5')
...              .str.extract(r'(?P<python_years>\.?\d+)')
...              .astype('float64[pyarrow]'),
...          python3_ver=lambda df_:df_.python3_version_most
...              .str.replace('_', '.')
...              .str.extract(r'(?P<python3_ver>\d\.\d)'),
...          use_python_most=lambda df_:df_.use_python_most
...              .fillna('Unknown'))
...      .pipe(mem_tracker, txt='After assign')
...       .assign(
...             team_size=lambda df_:predict_col(df_, 'team_size')
...              .astype(int))
...      .pipe(mem_tracker, txt='After predict_col')
...   .loc[:, ['age', 'are_you_datascientist', 'company_size',
...       'country_live', 'employment_status',
...       'first_learn_about_main_ide', 'how_often_use_main_ide',
...       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
...       'missing_features_main_ide', 'nps_main_ide', 'python_years',
...       'python3_version_most', 'several_projects', 'team_size',
...       'use_python_most', 'years_of_coding', 'python3_ver']]
...      .pipe(mem_tracker, txt='After loc')
... )

>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...    '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> mt = MemoryTracker()
>>> jb2 = tweak_jb_mt(jb, mt)
Start Process memory usage: 585.14 MB (growth: 0.00 MB)
After uniq_cols Process memory usage: 585.17 MB (growth: 0.03 MB)
After rename Process memory usage: 585.17 MB (growth: 0.00 MB)
After assign Process memory usage: 607.91 MB (growth: 22.73 MB)
Learning rate set to 0.5
0:  learn: 2.9758568    total: 84.5ms   remaining: 1.61s
1:  learn: 2.8841040    total: 95.8ms   remaining: 862ms
```

```
2:  learn: 2.8443484    total: 113ms    remaining: 643ms
3:  learn: 2.8105584    total: 128ms    remaining: 511ms
4:  learn: 2.7922983    total: 139ms    remaining: 417ms
5:  learn: 2.7803329    total: 153ms    remaining: 358ms
6:  learn: 2.7756137    total: 178ms    remaining: 330ms
7:  learn: 2.7706510    total: 189ms    remaining: 284ms
8:  learn: 2.7571563    total: 212ms    remaining: 259ms
9:  learn: 2.7564631    total: 231ms    remaining: 231ms
10: learn: 2.7503591    total: 253ms    remaining: 207ms
11: learn: 2.7494745    total: 275ms    remaining: 183ms
12: learn: 2.7481258    total: 285ms    remaining: 154ms
13: learn: 2.7477180    total: 310ms    remaining: 133ms
14: learn: 2.7449738    total: 342ms    remaining: 114ms
15: learn: 2.7409940    total: 359ms    remaining: 89.7ms
16: learn: 2.7408640    total: 382ms    remaining: 67.4ms
17: learn: 2.7365108    total: 403ms    remaining: 44.8ms
18: learn: 2.7346780    total: 416ms    remaining: 21.9ms
19: learn: 2.7287662    total: 443ms    remaining: 0us
After predict_col Process memory usage: 643.61 MB (growth: 35.70 MB)
After loc Process memory usage: 643.62 MB (growth: 0.02 MB)
```

Now, I'm going to enable copy on write and see how much memory is used:

```
>>> pd.options.mode.copy_on_write = True

>>> mt2 = MemoryTracker()
>>> jb3 = tweak_jb_mt(jb, mt2)
Start Process memory usage: 603.28 MB (growth: 0.00 MB)
After uniq_cols Process memory usage: 603.28 MB (growth: 0.00 MB)
After rename Process memory usage: 603.28 MB (growth: 0.00 MB)
After assign Process memory usage: 604.98 MB (growth: 1.70 MB)
Learning rate set to 0.5
0:  learn: 2.9758568    total: 12.4ms   remaining: 236ms
1:  learn: 2.8841040    total: 35.4ms   remaining: 319ms
2:  learn: 2.8443484    total: 50.7ms   remaining: 287ms
3:  learn: 2.8105584    total: 60.8ms   remaining: 243ms
4:  learn: 2.7922983    total: 76.4ms   remaining: 229ms
5:  learn: 2.7803329    total: 88.6ms   remaining: 207ms
6:  learn: 2.7756137    total: 100ms    remaining: 186ms
7:  learn: 2.7706510    total: 122ms    remaining: 183ms
8:  learn: 2.7571563    total: 135ms    remaining: 164ms
9:  learn: 2.7564631    total: 155ms    remaining: 155ms
10: learn: 2.7503591    total: 170ms    remaining: 139ms
11: learn: 2.7494745    total: 186ms    remaining: 124ms
12: learn: 2.7481258    total: 211ms    remaining: 114ms
```

```
13: learn: 2.7477180    total: 227ms    remaining: 97.3ms
14: learn: 2.7449738    total: 254ms    remaining: 84.6ms
15: learn: 2.7409940    total: 271ms    remaining: 67.7ms
16: learn: 2.7408640    total: 295ms    remaining: 52ms
17: learn: 2.7365108    total: 314ms    remaining: 34.9ms
18: learn: 2.7346780    total: 328ms    remaining: 17.3ms
19: learn: 2.7287662    total: 347ms    remaining: 0us
After predict_col Process memory usage: 643.20 MB (growth: 38.22 MB)
After loc Process memory usage: 643.20 MB (growth: 0.00 MB)
```

It looks like copy on write is using less memory. I encourage you to turn on copy on write to save memory.

37.9 Timing Information

In addition to how much memory your data is using, you probably want your code to run as fast as possible. Throughout this book, we have emphasized best practices, but we have also seen that pandas often has two (or three or four) ways of doing something.

When clients ask what is faster, my general response is, "it depends". And that is true. If you compare two pieces of code and benchmark them on a small amount of data, there is no guarantee that the fast code will still be faster when bombarded with more data. (Pay special attention to `.apply`, `.query`, and date conversion.)

After saying, "It depends," I follow that up with, "Benchmark it and see." You can use the `%%time` cell magic to measure the clock time of a cell in Jupyter:

```
>>> %%time
>>> dd = tweak_river(df)
CPU times: user 228 ms, sys: 8.8 ms, total: 237 ms
Wall time: 235 ms
```

Another cell magic that provides timing information is `%%timeit`. This will run the cell a few times and give you the mean and standard deviation of the runtime:

```
>>> %%timeit
>>> dd = tweak_river(df)
233 ms ± 9.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Table 37.1: Chapter Methods

Method	Description
`df.equals(other)`	Compares two dataframes if they have the same shape and values. Columns should have the same type.
`df.eq(other, axis='columns', level=None)`	Return dataframe with same index and columns but boolean values indicating whether values are the same elementwise.
`df.ne(other, axis='columns', level=None)`	Return dataframe with same index and columns but boolean values indicating whether values are different elementwise.
`pd.testing.assert_frame_equal(left, right, check_dtype=True, check_index_type='equiv', check_column_type='equiv', check_frame_type=True, check_names=True, by_blocks=False, check_exact=False, check_datetimelike_compat=False, check_categorical=True, check_like=False, check_freq=True, check_flags=True, rtol=1e-05, atol=1e-08, obj='DataFrame')`	Utility function to determine if two dataframes are the same. Can change numeric tolerance with `rtol` (relative tolerance) and `atol` (absolute tolerance).
`df.round(decimals=0)`	Create a dataframe with decimals rounded to given places.
`.pipe(func, *args, **kwargs)`	Apply a function to a dataframe. Return the result of function.
`IPython.display.display( *objs, include=None, exclude=None, metadata=None, transient=None, display_id=None, **kwargs)`	Displays `objs` in Jupyter.
`df.info(verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None)`	Print summary of dataframe to stdout. Use `memory_usage='deep'` to show object column memory usage.

37.10 Summary

In this chapter, we have shown various techniques for understanding what happens when you use pandas. One of the keys to success with pandas is understanding what operations do to your data and validating that the operation worked as you expected. We also showed how to profile memory usage and timing.

37.11 Exercises

With a dataset of your choice, create a tweak function to perform a chain of operations.

1. Use the debugger to step into the chain of your tweak function.
2. Capture an intermediate state of your chain into a variable.
3. Time how long the tweak function takes to run.
4. Determine how much memory the tweak function needs to run.

Chapter 38

Refactoring Pandas Code

This chapter will explore a project from the book, *Algorithmic Short Selling with Python* [1]. The primary focus here is not to discuss the book's content but to take some Python code from the book's project and refactor it to improve readability and testability. I believe this coding style represents much of the pandas code in the wild.

I strongly recommend this book if you're interested in algorithmic short-selling.

38.1 Starting Code

```
# Chapter 13: Portfolio Management System

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf


K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
```

[1] https://github.com/PacktPublishing/Algorithmic-Short-Selling-with-Python-Published-by-Packt/blob/main/Chapter%2013/Chapter%2013.ipynb

```
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])

raw_data = yf.download(tickers= ticker_list, period='6mo',
                       interval = "1d",group_by = 'column',
                       auto_adjust = True,
                        # had a typo w/ treads=True
                        prepost = True, threads = True, proxy = None)

price_df = round( raw_data['Close'],2)

print(price_df.shape)

bm_cost = price_df[bm_ticker][0]
bm_price = price_df[bm_ticker][-1]

port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
port['Cost'] = price_df.iloc[0,:]
port['Price'] = price_df.iloc[-1,:]

print(port)
```

The existing code structure starts with library imports. This is followed by a number of global variables.

The code calls the finance `download` function to fetch six months of stock ticker information at the one-day interval. This is stored in the `price_df` variable. Another dataframe, `port`, is created and printed.

I have changed this code a little from the original. I removed the start and end dates. I also read the data from yfinance into its own data frame. To make this code easy to reproduce I want to limit the amount of times I'm hitting a web service. I will just read the data from a CSV instead.

```
raw_data.to_csv('data/raw-yfinance.csv')
```

With that data in place, few more columns are derived:

```
# Chapter 13: Portfolio Management System

price_df['bm returns'] = round(
    np.exp(np.log(price_df[bm_ticker]/
    price_df[bm_ticker].shift()).cumsum()) - 1, 3)
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
```

```
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round(
    (rLong_MV + rShort_MV).div(abs(rMV).sum(axis=1)),3)

price_df['rReturns_Long'] = round(
    np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
price_df['rReturns_Short'] = - round(
    np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
price_df['rReturns'] = price_df['rReturns_Long'] + price_df['rReturns_Short']

MV = price_df.mul(port['Shares'])
Long_MV = MV[MV >0].sum(axis=1)
Short_MV = MV[MV <0].sum(axis=1)
price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
price_df['Net'] = round(
    (Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)

price_df['Returns_Long'] = round(
    np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
price_df['Returns_Short'] = - round(
    np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
price_df['Returns'] = price_df['Returns_Long'] + price_df['Returns_Short']

MV_Beta = MV.mul(port['Beta'])
Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
```

Finally, some plots are created:

```
# Chapter 13: Portfolio Management System
price_df[['bm returns','Returns','Gross','rNet_Beta','rNet' ]].plot(
    figsize=(20,8),grid=True, secondary_y=['Gross'],
    style= ['r.-','k','g--','g:','g-o','b:','c','c:'],
    title = 'bm returns, Returns, Gross, rNet_Beta, rNet')

price_df[['bm returns','Returns','rReturns','rReturns_Long',
          'rReturns_Short']].plot(
    figsize=(20,8),grid=True,
    style= ['r.-','k','b--o','b--^','b--v','g-.','g:','b:'],
    title='bm returns, Returns, rReturns, rReturns_Long, rReturns_Short')
```

```
price_df[['bm returns','Returns','rReturns',
          'rReturns_Long','rReturns_Short','Returns_Long',
          'Returns_Short']].plot(
    figsize=(20,8),grid=True,secondary_y=['Gross'],
    style= ['r.-','k','b--o','b--^','b--v','k:^','k:v',],
    title= 'Returns: benchmark, Long / Short absolute & relative')

price_df[['bm returns',
          'rReturns_Long','rReturns_Short','Returns_Long',
          'Returns_Short']].plot(
    figsize=(20,8),grid=True,secondary_y=['Gross'],
    style= ['r.-','b--^','b--v','m:.','m:.',],
    title= 'Returns: benchmark, Long / Short absolute & relative')
```

Our plan is to refactor the existing code and wrap tests around it.

In the original code, various objects are derived from `rel_price`, such as `rLong_MV`, `rShort_MV`, `rMV_beta`, and `rLong_MV_Beta`. New columns in the `price_df` data frame are created from these derived objects. Some of these columns have calculations that are derived from other columns.

38.2 Code Review

This section focuses on the code we just listed. It's important to clarify that this code isn't inherently bad. In fact, it reflects how many people, including students and industry professionals, write pandas code. Nonetheless, I wish to discuss certain aspects I find less than ideal.

The first issue lies in the prevalent use of global variables. While in the context of Jupyter notebooks global variables are commonplace, they can cause serious problems such as shadowing variables, unexpected state, the time travel paradox (change a global variable's value halfway through your notebook, then rerun an earlier cell? Welcome to a world where Marty McFly might not be born), notebook amnesia (revisit a notebook only to discover that global values have all disappeared), variable name lazy reuse, reproducibility nightmare, order dependency, naming creativity, Murphy's Law (when things can go wrong with globals they will) and more. Ok, maybe I slightly exaggerated, this list should be a little longer.

This dichotomy between Jupyter's easy-going approach to global variables and the more disciplined software engineering perspective can often lead to confusion. While exploratory data analysis has a certain degree of leniency, relying on global variables becomes problematic quickly. Especially when you intend to move your code into production. We'll examine how to mitigate this by refactoring the code into functions.

The code's rigidity is another concern, stemming from heavy reliance on hard coding. A lack of precise inputs and outputs makes modifications a challenging task. We will create a more streamlined user interface, offering

functions with clearly defined inputs that return predictable outputs. This approach not only makes it easier to revisit the code in the future but also facilitates code sharing.

You should also note the absence of documentation within this code. Though this chapter won't delve extensively into documentation, awareness of its importance is crucial. Comments can serve as simple starting points for documentation, but once you've refactored the code into functions, they become excellent candidates for more detailed documentation. Jupyter's shift-tab inspection feature is invaluable, allowing you to review your function's documentation and usage instructions quickly.

Lastly, this code lacks tests, a common occurrence given many individuals either lack the knowledge or the motivation to write tests. However, I firmly believe that tests are integral, especially for Python code, to establish confidence in your code's functionality. Consequently, I will demonstrate how to add tests to ensure the code maintains behavior as expected post-refactoring.

When encountering code in the future, I urge you to ask the following questions:

- Are global variables being used?
- Is the code organized into functions?
- Is the code usage clear?
- Is there adequate documentation?
- Are there tests in place?

If the answer to any of these questions is 'no', then there's room for improvement. Applying the principles and practices discussed in this chapter can enhance our ability to write robust, efficient, and maintainable pandas code.

38.3 Enhancing Your Coding Process

As we delve deeper into the intricacies of coding, I want to emphasize a habit I've cultivated over the years. Before typing away, I like to step back and contemplate my action plan. Taking the time to chart your path often leads to better code.

Before diving in, think about the code you're about to work on before diving in. Identify the main objects and what you want the output to be. Certain data structures might be candidates for global variable storage but should result from function calls. It's not advisable to pepper your code with global variables haphazardly.

Next, ponder on how you'll create the main objects. What parameters are needed for their creation? For instance, we have two main objects: a portfolio object (`port`) and a pricing object `priceDF`. However, there are also several intermediate variables created along the way. These intermediate objects are sources from which we derive columns for our `priceDF`.

Ask yourself, what role do these objects play? Are they crucial or merely side effects?

Also, consider how this code will be reused. We often focus on achieving functionality for the present moment, neglecting future use cases. What different parameters might be required? Perhaps, in the future, we'll need to look at data hourly rather than daily, or maybe we'll have different tickers in our portfolio.

As we go along, I'll be using the term "refactoring" quite often. *Refactoring* is a software engineering term indicating a change in the code's internal structure without altering its external behavior. Throughout this chapter, I'm not adding new features or making changes that would affect the output. We're simply altering how we achieve that output.

When refactoring, we want to have a test for the original code, and it should yield the same result after refactoring.

Identify your main objects, understand the steps to create them, plan for future needs, and ensure you have tests to validate the code's behavior. This strategy can significantly enhance your coding process, allowing you to produce high-quality, reusable, and reliable code.

38.4 Analyzing Raw Data

This section will look at financial data fetched from Yahoo using the finance `download` function.

The code creates a `price_df` DataFrame object, which fetches and stores our financial data. We also instantiate a `port` object, a portfolio object that contains the number of shares for our portfolio.

```
K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])

raw_data = pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])

price_df = round(raw_data['Close'],2)
```

38.5 Creating a get_tickers Function

To improve the structure and readability of our code, we will refactor the process of pulling data into a new function called `get_tickers`.

```
def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))
```

This function takes as inputs a list of ticker symbols, a period, and an interval. Running this function confirms that it works as expected. I'll tack on a `_rf` on my new object that stands for "refactored".

```
>>> port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
...                 'UPS','F']
>>> bm_ticker = '^GSPC'
>>> ticker_list = [bm_ticker] + port_tickers
>>> period = '6mo'
>>> price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
```

I will also validate that it returns the same result as `price_df`.

It's important to note that the `.equals` method differs from the `==` operator or the `.eq` method. Using `.eq` provides a DataFrame output with `True` or `False` for each cell, indicating whether the corresponding cells are equal.

```
>>> price_df.equals(price_df_rf)
True
```

In our case, `.equals` returns `True` for all values, confirming that our refactoring worked as intended.

38.6 Creating Portfolio Data

Now let's explore the the process of creating our portfolio data.

```
K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
```

```
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])
```

First, let's take a look at the existing code structure we have. The existing code comprises several variables:

- `K` - This is used in calculating the *Gross* column
- `lot` - This is not used
- `port_tickers` - The portfolio tickers
- `benchmark_ticker` - The benchmark tickers
- `ticker_list` - The combination of the tickers
- `df_data` - The seed data for the port data frame

We start by creating a data frame using the `df_data` dictionary. Then, we make a new column called *side*, reflecting whether the *Shares* column is positive or negative. (`np.sign` returns `-1` for negative numbers, `1` for positive numbers, and `0` for zero.)

```
bm_cost = price_df[bm_ticker][0]
bm_price = price_df[bm_ticker][-1]

port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
port['Cost'] = price_df.iloc[0,:]
port['Price'] = price_df.iloc[-1,:]
```

Next, we proceed to set up our benchmark information. We create a benchmark cost, `bm_cost`, which represents the initial cost of the benchmark item in our portfolio. We also compute the benchmark price, `bm_price`, which corresponds to the last item from the benchmark.

These values allow us to calculate the relative price, *rPrice*, and cost, *rCost*, to our benchmark. We place those in the portfolio data frame. We also add the non-relative versions of these to the portfolio. It's important to note a crucial point. The calculation of relative price depends on the `price_df` data frame. Hence, a dependency exists between these two objects. This is something to keep in mind during the refactoring process.

```
>>> print(port.loc[:, 'rCost':'Price'])
       rCost  rPrice    Cost   Price
QCOM   28.58   26.63  109.25  116.36
TSLA   32.79   59.85  125.35  261.47
NFLX   77.90   96.28  297.75  420.61
DIS    22.67   20.23   86.67   88.39
PG     39.32   34.29  150.30  149.79
MMM    30.92   22.95  118.20  100.27
IBM    35.92   29.97  137.30  130.91
BRK-B  79.19   77.12  302.69  336.91
```

```
UPS    45.11   39.46  172.42  172.37
F       2.75    3.27   10.51   14.28
```

38.7 Refactoring the Code

Let's refactor this code to create the `port` variable.

```
def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
            )
```

Upon refactoring, the code is written inside a function named `get_portfolio`. This function accepts the tickers, the dependent price data frame, and the benchmark data as arguments.

Let's try it out:

```
>>> print(get_portfolio(port_tickers, price_df_rf,
...                     price_df_rf.loc[:,bm_ticker])
...     .loc[:, 'rCost':'Price']
... )

       rCost  rPrice    Cost   Price
QCOM   28.58   26.63  109.25  116.36
TSLA   32.79   59.85  125.35  261.47
NFLX   77.90   96.28  297.75  420.61
DIS    22.67   20.23   86.67   88.39
PG     39.32   34.29  150.30  149.79
MMM    30.92   22.95  118.20  100.27
IBM    35.92   29.97  137.30  130.91
```

```
BRK-B  79.19   77.12  302.69  336.91
UPS    45.11   39.46  172.42  172.37
F       2.75    3.27   10.51   14.28
```

Chaining allows us to conduct operations one after another, where each operation returns a new object (a new data frame or series). The `.assign` method is especially useful here. Chaining forces us to think about the operations step by step, making it read like a recipe.

The use of a `lambda` function within the `.assign` method is particularly essential here. This is because, during chaining, we work with intermediate objects. The assign method accepts a function as an argument and passes in the current state of the data frame to that function. This allows us to work with the current state of the new data fram which now has a *Shares* column. There is no way to access the dataframe without the lambda inside of the `.assign`.

Next, we create the *rCost* column. This column is calculated based on the `price_df` data frame, and involves division of the first column by the first column of the benchmark data. Finally, it is multiplied by 1000 and rounded to two decimal places.

You might be wondering, how do we know if our *rCost* is correct? That's the crux of testing and verification. We should test the result of our calculation against known examples to ensure the accuracy of our code.

In fact, I'll just test the whole data frame:

```
>>> (get_portfolio(port_tickers, price_df_rf,
...   price_df_rf.loc[:,bm_ticker]).equals(port))
True
```

Yeah! It looks like we are doing well.

38.8 Notebook Reformatting

Let's delve into an often-overlooked aspect of data science work: **reformatting notebooks**. Unorganized notebooks can cause confusion and significantly slow down the workflow.

After you've executed some code refactoring, the best practice is to place this revised code at the top of your notebook. This way, it's straightforward to locate and execute.

You may want to restart your notebook to ensure your refactoring works as intended. This action can be performed simply by hitting 'zero' twice. A dialog box will pop up, prompting you to confirm if you want to restart the current kernel.

My top cell would not look like this:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
import pandas as pd
import yfinance as yf

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
           )

port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
period = '6mo'
price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
port_rf = get_portfolio(port_tickers, price_df_rf,
                        price_df_rf.loc[:,bm_ticker])
```

When performing this kind of refactoring, keep in mind the treatment of global variables. Global variables can affect your functions and, in some cases, may be inadvertently left in your refactored code. Restarting your notebook and running the code from scratch ensures that your refactored functions aren't reliant on any lingering global variables. If they were, you would encounter an error, indicating that your refactoring needs more attention.

Keeping your Jupyter notebooks tidy is highly recommended for both solo and collaborative projects. Whenever you complete a portion of your work, move it to the top, restart your notebook, and rerun the code. This practice not only streamlines your work but also eases collaboration. It also simplifies your workflow the following day, as you won't have to wade through a bunch of cells determining which ones need to run and in what order.

38.9 More Refactoring

In this section, we're going to see the rest of the codebase. Previously, we looked at the initial code and refactored it into some functions. There's a lot of code we haven't explored yet, so let's see that.

```
price_df['bm returns'] = round(np.exp(np.log(price_df[bm_ticker]/
                        price_df[bm_ticker].shift()).cumsum()) - 1, 3)
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round((rLong_MV + rShort_MV)
                         .div(abs(rMV).sum(axis=1)),3)

price_df['rReturns_Long'] = round(
    np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
price_df['rReturns_Short'] = - round(
    np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
price_df['rReturns'] = price_df['rReturns_Long'] + \
    price_df['rReturns_Short']

MV = price_df.mul(port['Shares'])
Long_MV = MV[MV >0].sum(axis=1)
Short_MV = MV[MV <0].sum(axis=1)
price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
price_df['Net'] = round((Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)

price_df['Returns_Long'] = round(
    np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
price_df['Returns_Short'] = -round(
    np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
```

```
price_df['Returns'] = price_df['Returns_Long'] + price_df['Returns_Short']

MV_Beta = MV.mul(port['Beta'])
Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
```

I've extracted the code directly from the GitHub repository. You'll notice we have a dataframe called `price_df` and several intermediate objects. These intermediate objects are then used to create various columns in `price_df`. We aim to refactor the code so our new version can replicate these columns with the same values.

Let's go through this code. First, we need to create the *bm returns* column. This column is derived from the *bm_ticker* column in the same dataframe. We use several operations such as `.shift`, `np.log`, `.cumsum`, and `np.exp` to calculate the returns. I'll use the `.assign` method to create that column:

```
>>> # round(np.exp(np.log(price_df[bm_ticker]/
>>> #      price_df[bm_ticker].shift()).cumsum()) - 1, 3)
>>> print(price_df_rf.assign(**{
...  'bm returns': price_df_rf[bm_ticker]/price_df_rf[bm_ticker]
...  .shift().apply(np.log).cumsum().apply(np.exp).sub(1).round(3)})
... )
                           BRK-B    DIS  ...    ^GSPC    bm returns
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39           NaN
2022-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82  1.006131e+00
2022-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25  2.605570e-04
2022-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22  6.722594e-08
2022-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28  1.807978e-11
...                           ...    ...  ...      ...           ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84  0.000000e+00
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59  0.000000e+00
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71  0.000000e+00
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69  0.000000e+00
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72  0.000000e+00

[124 rows x 12 columns]
```

Let's run the original calculation and compare the results:

```
>>> price_df['bm returns']
Date
2022-12-22 00:00:00-05:00      NaN
2022-12-23 00:00:00-05:00    0.006
2022-12-27 00:00:00-05:00    0.002
```

```
2022-12-28 00:00:00-05:00   -0.010
2022-12-29 00:00:00-05:00    0.007
                             ...
2023-06-15 00:00:00-04:00    0.158
2023-06-16 00:00:00-04:00    0.154
2023-06-20 00:00:00-04:00    0.148
2023-06-21 00:00:00-04:00    0.142
2023-06-22 00:00:00-04:00    0.143
Name: bm returns, Length: 124, dtype: float64
```

It turns out that my re-writing of the logic in a chain had a operator precedence error. Good thing I checked that the values were the same. Here is a fixed version:

```
>>> # round(np.exp(np.log(price_df[bm_ticker]/
>>> # price_df[bm_ticker].shift()).cumsum()) - 1, 3)
>>> print(price_df_rf.assign(**{
...     'bm returns': (price_df_rf[bm_ticker]/price_df_rf[bm_ticker]
...      .shift()).apply(np.log).cumsum().apply(np.exp).sub(1).round(3)})
... )
                           BRK-B    DIS  ...    ^GSPC  bm returns
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39         NaN
2022-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82       0.006
2022-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25       0.002
2022-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22      -0.010
2022-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28       0.007
...                           ...    ...  ...      ...         ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84       0.158
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59       0.154
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71       0.148
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69       0.142
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72       0.143

[124 rows x 12 columns]
```

The code calculates the returns many times. So I'm going to refactor this calculation into its own function:

```
def returns(df, col):
    return (df[col]
            .div(df[col].shift())
            .apply(np.log)
            .cumsum()
            .apply(np.exp)
            .sub(1)
            .round(3))
```

Now, let's run the function:

```
>>> print(price_df_rf
...   .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
... )
                           BRK-B    DIS  ...    ^GSPC  bm returns
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39         NaN
2022-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82       0.006
2022-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25       0.002
2022-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22      -0.010
2022-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28       0.007
...                           ...    ...  ...      ...         ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84       0.158
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59       0.154
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71       0.148
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69       0.142
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72       0.143

[124 rows x 12 columns]
```

In the `returns` function, we use `.apply` with `np.log` and `np.exp`. You might think that `.apply` is slow. And it is when you use it with Python functions. However, when you're applying a NumPy function to a pandas series, it is fast. This is because it doesn't perform the operation element by element, but instead applies the operation to the entire series at once due to the vectorized nature of NumPy functions.

If the `returns` function was slow during benchmarking, we might consider a NumPy, Cython, or Numba version. Here is a NumPy version:

```
from scipy.ndimage import shift

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(
        np.exp(np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)
```

Let's benchmarkk this new function:

```
>>> %%timeit
>>> np_returns(price_df_rf, bm_ticker)
46.4 µs ± 14.1 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
    each)
```

```
>>> %%timeit
>>> returns(price_df_rf, bm_ticker)
190 µs ± 8.69 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
    each)
```

On my machine the NumPy version is five times faster for this size of data. I'll leave it to the reader to try and implement a Cython and Numba version.

38.10 The rel_price Variable

This section looks at dependent objects—objects that our chain relies upon. The original code started with two original objects and created ancillary objects to develop columns that would feed back into the original objects.

The original code operated on a `priceDF` object. In my chain, however, I intend to make calls that will generate some global objects. Consequently, later operations dependent on these will be able to access those global objects and generate derived columns from them.

Consider this original code. The *RNet_Beta* column depends on *rLong_MV_Beta*, which depends on *rMV_Beta* and *rLong_MV* both of which depend on *rMV*, which itself depends on *rel_price*. With chaining, we might be able to just stick these variables into columns, however, *rep_price*, *rMV*, and *rMV_Beta* aren' series. They are dataframes. In my opinion, sticking these dataframes into an existing dataframe is not a win. It will make the code complicated.

```
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round((rLong_MV + rShort_MV)
    .div(abs(rMV).sum(axis=1)),3)
```

It seems counterintuitive to assign a dataframe into a column. So, how do we deal with this issue?

My solution is to create a new dataframe.

I'm going to create a new function called `make_var`. This function takes in a dataframe, a variable name, a function, and several arguments. It then inserts the variable name into the global namespace, calling the function I passed in with the dataframe and the arguments I passed into it.

This function allows me to pass any arguments into the underlying function, `fn`, that I choose.

I will create these variables inside a chain using this function:

```
def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df
```

Instead of using `.assign` to create a column called `relPrice2`, I'm going to use the `.pipe` method of pandas. The `.pipe` method allows you to pass in any function and return whatever you want.

I want to pass in a function that creates a global variable, and I'll return the current state of the dataframe.

We introduced a similar function during the debugging chapter. Now, we are using it in real-world code. I will use it to create these dataframes that other columns depend on:

```
>>> print(price_df_rf
...   .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
...   .pipe(make_var, var_name='rel_price2',
...         fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
...              .mul(1_000).round(2))
...   .pipe(make_var, var_name='rMV2',
...         fn=lambda df_: rel_price2.mul(port_rf['Shares']))
...   .pipe(make_var, var_name='rMV_Beta2',
...         fn=lambda df_: rMV2.mul(port_rf['Beta']))
...   .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
...           rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
...           rLong_MV_Beta=lambda df_: rMV_Beta2[rMV_Beta2 > 0]
...             .sum(axis='columns') / df_.rLong_MV,
...           rShort_MV_Beta=lambda df_: rMV_Beta2[rMV_Beta2 < 0]
...             .sum(axis='columns') / df_.rShort_MV,
...           rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta
...          )
... )
                           BRK-B    DIS  ...  rShort_MV_Beta  \
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...        0.659542
2022-12-23 00:00:00-05:00  306.49  88.01  ...        0.659418
2022-12-27 00:00:00-05:00  305.55  86.37  ...        0.654134
2022-12-28 00:00:00-05:00  303.43  84.17  ...        0.653003
2022-12-29 00:00:00-05:00  309.06  87.18  ...        0.657856
...                           ...    ...  ...             ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...        0.694658
2023-06-16 00:00:00-04:00  338.31  91.32  ...        0.692247
2023-06-20 00:00:00-04:00  338.67  89.75  ...        0.692130
```

```
2023-06-21 00:00:00-04:00  338.61  88.64  ...        0.684792
2023-06-22 00:00:00-04:00  336.91  88.39  ...        0.684952

                           rNet_Beta
Date
2022-12-22 00:00:00-05:00   0.343361
2022-12-23 00:00:00-05:00   0.343176
2022-12-27 00:00:00-05:00   0.348584
2022-12-28 00:00:00-05:00   0.349062
2022-12-29 00:00:00-05:00   0.344405
...                              ...
2023-06-15 00:00:00-04:00   0.310011
2023-06-16 00:00:00-04:00   0.312403
2023-06-20 00:00:00-04:00   0.311869
2023-06-21 00:00:00-04:00   0.318311
2023-06-22 00:00:00-04:00   0.318279

[124 rows x 17 columns]
```

While this technique can be powerful, it requires some caution. We don't want to create too many global variables, and we may need to consider whether to delete these global objects if they consume a significant amount of space.

For more complex operations, you could use something like a context manager to create a global variable during your data creation. When the context manager exits, it cleans up those global variables for you. This is not covered in this book, but it might be an interesting exercise if you're unfamiliar with how context managers work.

Here's my new chain with the new columns:

```
>>> print(price_df_rf
...   .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
...   .pipe(make_var, var_name='rel_price2',
...         fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
...                          .mul(1_000).round(2))
...   .pipe(make_var, var_name='rMV2',
...         fn=lambda df_: rel_price2.mul(port_rf['Shares']))
...   .pipe(make_var, var_name='rMV_Beta2',
...         fn=lambda df_: rMV2.mul(port_rf['Beta']))
...   .pipe(make_var, var_name='MV2',
...         fn=lambda df_: df_.mul(port_rf['Shares']))
...   .pipe(make_var, var_name='MV_Beta2',
...         fn=lambda df_: MV2.mul(port_rf['Beta']))
...   .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
...       rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
...       rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
```

```
...                             .sum(axis='columns') / df_.rLong_MV),
...       rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
...                             .sum(axis='columns') / df_.rShort_MV),
...       rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
...       rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
...                           .div(rMV.abs().sum(axis='columns')).round(3)),
...       rReturns_Long=lambda df_: returns(df_, 'rLong_MV'),
...       Long_MV=MV[MV > 0].sum(axis='columns'),
...       Short_MV=MV[MV < 0].sum(axis='columns'),
...       Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
...       Net=lambda df_:((df_.Long_MV + df_.Short_MV)
...                       .div(MV2.abs().sum(axis='columns')).round(3)),
...       Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
...       Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
...       Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,
...       Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
...                                .sum(axis='columns') / df_.Long_MV),
...       Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
...                                 .sum(axis='columns') / df_.Short_MV),
...       Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
...       )
... )
```

```
                           BRK-B    DIS  ...  Short_MV_Beta  \
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...       0.659551
2022-12-23 00:00:00-05:00  306.49  88.01  ...       0.659401
2022-12-27 00:00:00-05:00  305.55  86.37  ...       0.654096
2022-12-28 00:00:00-05:00  303.43  84.17  ...       0.653013
2022-12-29 00:00:00-05:00  309.06  87.18  ...       0.657848
...                           ...    ...  ...            ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...       0.694644
2023-06-16 00:00:00-04:00  338.31  91.32  ...       0.692241
2023-06-20 00:00:00-04:00  338.67  89.75  ...       0.692140
2023-06-21 00:00:00-04:00  338.61  88.64  ...       0.684778
2023-06-22 00:00:00-04:00  336.91  88.39  ...       0.684991

                           Net_Beta
Date
2022-12-22 00:00:00-05:00  0.343349
2022-12-23 00:00:00-05:00  0.343220
2022-12-27 00:00:00-05:00  0.348606
2022-12-28 00:00:00-05:00  0.349037
2022-12-29 00:00:00-05:00  0.344450
...                             ...
2023-06-15 00:00:00-04:00  0.310061
```

```
2023-06-16 00:00:00-04:00  0.312408
2023-06-20 00:00:00-04:00  0.311862
2023-06-21 00:00:00-04:00  0.318336
2023-06-22 00:00:00-04:00  0.318226

[124 rows x 29 columns]
```

Again, to ensure that this works, you could restart your notebook and run it with only the new code.

38.11 Code Porting and Validation

In this section, we will finalize refactoring our code, ensuring all elements from the original code are included in our refactored version.

A crucial feature in my columns is the reusability of the `np_returns` function I created earlier. This refactoring allows me to leverage all the returns logic into one location.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.ndimage import shift
import yfinance as yf

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(np.exp(
        np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
```

```
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
            )

def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df

def get_price(df, bm_ticker, port, K):
    return (df
     .assign(**{'bm returns': lambda df_: np_returns(df_, bm_ticker)})
     .pipe(make_var, var_name='rel_price2',
           fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
               .mul(1_000).round(2))
     .pipe(make_var, var_name='rMV2',
           fn=lambda df_: rel_price2.mul(port['Shares']))
     .pipe(make_var, var_name='rMV_Beta2',
           fn=lambda df_: rMV2.mul(port['Beta']))
     .pipe(make_var, var_name='MV2',
           fn=lambda df_: df_.mul(port['Shares']))
     .pipe(make_var, var_name='MV_Beta2',
           fn=lambda df_: MV2.mul(port['Beta']))
     .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
        rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
        rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
                                   .sum(axis='columns') / df_.rLong_MV),
        rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
                                    .sum(axis='columns') / df_.rShort_MV),
        rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
        rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
                          .div(rMV2.abs().sum(axis='columns')).round(3)),
        rReturns_Long=lambda df_: np_returns(df_, 'rLong_MV'),
        rReturns_Short=lambda df_: -np_returns(df_, 'rShort_MV'),
        rReturns=lambda df_:df_.rReturns_Long + df_.rReturns_Short,
        Long_MV=MV2[MV2 > 0].sum(axis='columns'),
        Short_MV=MV2[MV2 < 0].sum(axis='columns'),
        Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
        Net=lambda df_: ((df_.Long_MV + df_.Short_MV)
```

```
                        .div(MV2.abs().sum(axis='columns')).round(3)),
        Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
        Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
        Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,
        Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
                                 .sum(axis='columns') / df_.Long_MV),
        Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
                                 .sum(axis='columns') / df_.Short_MV),
        Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
        )
     #.rename(columns={'benchmark_returns': 'bm returns'})
     .loc[:, ['BRK-B', 'DIS', 'F', 'IBM', 'MMM', 'NFLX', 'PG', 'QCOM',
       'TSLA', 'UPS', '^GSPC', 'bm returns', 'rNet_Beta', 'rNet',
       'rReturns_Long', 'rReturns_Short', 'rReturns', 'Gross', 'Net',
       'Returns_Long', 'Returns_Short', 'Returns', 'Net_Beta']]
    )

def process_data():
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
    port_rf = get_portfolio(port_tickers, price_df_rf,
                            price_df_rf[bm_ticker])
    returns = get_price(price_df_rf, bm_ticker, port_rf, K)
    return returns

price_df_rf = process_data()
```

Let's check whether these are the same.

```
>>> pd.testing.assert_frame_equal(price_df, price_df_rf)
```

Success!

38.12 Summary

This chapter presented the various facets of refactoring using pandas. We began by discussing the importance of restructuring or refactoring code to make it cleaner, more efficient, and easier to understand. We started making functions and chains. We underscored the importance of comprehensive checks and validations after refactoring, introduced refined methods for validating the equivalence of DataFrames, and emphasized the need to stop and resolve issues as soon as they emerge.

38.13 Exercises

1. Reflect on your current practices in writing pandas code. Identify areas where you could apply the principles of refactoring to write more efficient and readable code.
2. Find a block of pandas code you have written and refactor it.
3. Rewrite `np_returns` in Cython.
4. Rewrite `np_returns` in Numba.
5. Convert the code in this chapter to use PyArrow types and ensure that the conversion didn't introduce errors.

Chapter 39

Refactoring Code and Unit Testing

This chapter will discuss how to refactor code and run sanity checks. We then introduced a more formal approach to testing, using a library called PyTest. Testing is something that software engineers do all the time, but many folks don't have experience with it. I want to show you how to get started testing your pandas code.

39.1 Using PyTest

Before proceeding, ensure PyTest is installed in your environment. You can install it by running `pip install pytest` in your terminal or command prompt.

When running tests, it's essential to have consistent tests. This means you get the same output every time you run the tests. In our code, we are using Yahoo Finance to download data. This dependency could lead to inconsistent test results since there's no guarantee that the data from Yahoo Finance will be the same if you run the tests in the future.

We already did this in the previous chapter, but I will repeat it here. (Normally I would only do this for testing purposes.) I will force data consistency by saving the downloaded data to a CSV file. In my test, I will read data from the file instead of calling the resource that might change. (I might want to have another test to check that the format of the data coming from Yahoo Finance doesn't change. But I would want this test to be distinct from one that ensures my logic works.)

```
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
period = '6mo'
price_df_rf = get_tickers(ticker_list=ticker_list, period=period)

price_df_rf.to_csv('data/tickers-raw.csv')
```

39.2 Writing Test Code

One of the significant challenges for Jupyter users is that they might not work in a terminal or a shell like software developers do. Software developers will write code and test in a text editor and then use a command line tool like pytest to run the tests and report the results. (They might have an IDE that does this for them, but you should understand what the underlying tool does before blindly trusting the IDE.)

Let's write some test code. If I were in Jupyter, I would use a Jupyter cell magic called `%writefile`. This cell magic allows us to write a file directly from Jupyter.

```
%%writefile test_pd_refactor.py

import pandas as pd
import pytest


@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                       dtype_backend='pyarrow', engine='pyarrow')

def test_basic(raw_price_df):
    assert len(raw_price_df) > 1
```

This test code imports the `pandas` and `pytest` libraries. I created a function, `test_basic`, that takes in a `DataFrame` and checks if its length is greater than one. This function uses a PyTest feature known as *fixtures*. Fixtures are dependencies necessary for tests to run. The `test_basic` test needs a dataframe to run. The function `raw_price_df` will provide that dataframe. Using a decorator, `pytest.fixture`, we register the fixture. Any test that wants to use that fixture just needs to put the name of the fixture function in the parameter list.

Here is where the challenge for Jupyter users is. Running the test. Software folks would do this from the command line. We can use the Jupyter feature to access the command line. We need to preface our code with an exclamation point (`!`).

We ran the test code using the command `pytest test_pd_refactored.py`.

Let's look at the output:

```
!pytest test_pd_refactored.py
test_pd_refactor.py .                                                [100%]

============================ 1 passed in 0.30s =============================
```

Following the file name is a period. That period is a convention that pytest uses to indicate that the test passed. On the right, it says 100 percent of the test from the file passed. At the bottom is a summary of the number of tests and the time it took to run them.

39.3 Writing More Compelling Test Cases

Our initial test case was relatively simple – checking if the length of the `DataFrame` was greater than one.

I'm going to test that the original code and the refactored code have the same behavior. I will create a file, `returns.py`, with a function to generate the original `price_df` called `orig_price_df`. It will also have the refactored logic, `process_data`.

```
%%writefile returns.py
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.ndimage import shift
import yfinance as yf

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(
        np.exp(np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
```

```
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
            )

def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df

def get_price(df, bm_ticker, port, K):
    return (df
     .assign(**{'bm returns': lambda df_: np_returns(df_, bm_ticker)})
     .pipe(make_var, var_name='rel_price2',
           fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
               .mul(1_000).round(2))
     .pipe(make_var, var_name='rMV2',
           fn=lambda df_: rel_price2.mul(port['Shares']))
     .pipe(make_var, var_name='rMV_Beta2',
           fn=lambda df_: rMV2.mul(port['Beta']))
     .pipe(make_var, var_name='MV2',
           fn=lambda df_: df_.mul(port['Shares']))
     .pipe(make_var, var_name='MV_Beta2',
           fn=lambda df_: MV2.mul(port['Beta']))
     .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
         rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
         rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
                                    .sum(axis='columns') / df_.rLong_MV),
         rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
                                     .sum(axis='columns') / df_.rShort_MV),
         rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
         rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
                           .div(rMV2.abs().sum(axis='columns')).round(3)),
         rReturns_Long=lambda df_: np_returns(df_, 'rLong_MV'),
         rReturns_Short=lambda df_: -np_returns(df_, 'rShort_MV'),
         rReturns=lambda df_:df_.rReturns_Long + df_.rReturns_Short,
         Long_MV=MV2[MV2 > 0].sum(axis='columns'),
         Short_MV=MV2[MV2 < 0].sum(axis='columns'),
         Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
         Net=lambda df_: ((df_.Long_MV + df_.Short_MV)
                          .div(MV2.abs().sum(axis='columns')).round(3)),
         Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
         Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
```

```
        Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,
        Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
                                 .sum(axis='columns') / df_.Long_MV),
        Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
                                  .sum(axis='columns') / df_.Short_MV),
        Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
            )
     #.rename(columns={'benchmark_returns': 'bm returns'})
     .loc[:, ['BRK-B', 'DIS', 'F', 'IBM', 'MMM', 'NFLX', 'PG', 'QCOM',
       'TSLA', 'UPS', '^GSPC', 'bm returns', 'rNet_Beta', 'rNet',
       'rReturns_Long', 'rReturns_Short', 'rReturns', 'Gross', 'Net',
       'Returns_Long', 'Returns_Short', 'Returns', 'Net_Beta']]
    )

def process_data():
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
    port_rf = get_portfolio(port_tickers, price_df_rf,
                            price_df_rf[bm_ticker])
    returns = get_price(price_df_rf, bm_ticker, port_rf, K)
    return returns

def orig_price_df(price_df):
    price_df = price_df.copy()
    K = 1000000
    lot = 100
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    port = pd.DataFrame(df_data,index=port_tickers)
    port['Side'] = np.sign(port['Shares'])

    bm_cost = price_df[bm_ticker][0]
    bm_price = price_df[bm_ticker][-1]
```

```
    port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
    port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
    port['Cost'] = price_df.iloc[0,:]
    port['Price'] = price_df.iloc[-1,:]
    price_df['bm returns'] = round(
        np.exp(np.log(price_df[bm_ticker]/
                price_df[bm_ticker].shift()).cumsum()) - 1, 3)
    rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

    rMV = rel_price.mul(port['Shares'])
    rLong_MV = rMV[rMV >0].sum(axis=1)
    rShort_MV = rMV[rMV <0].sum(axis=1)
    rMV_Beta = rMV.mul(port['Beta'])
    rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
    rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

    price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
    price_df['rNet'] = round(
        (rLong_MV + rShort_MV).div(abs(rMV).sum(axis=1)),3)

    price_df['rReturns_Long'] = round(
        np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
    price_df['rReturns_Short'] = -round(
        np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
    price_df['rReturns'] = price_df['rReturns_Long'] +\
             price_df['rReturns_Short']

    MV = price_df.mul(port['Shares'])
    Long_MV = MV[MV >0].sum(axis=1)
    Short_MV = MV[MV <0].sum(axis=1)
    price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
    price_df['Net'] = round(
        (Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)

    price_df['Returns_Long'] = round(
        np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
    price_df['Returns_Short'] = -round(
        np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
    price_df['Returns'] = price_df['Returns_Long'] + \
        price_df['Returns_Short']

    MV_Beta = MV.mul(port['Beta'])
    Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
    Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
    price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
    return price_df
```

Let's create a new fixture that depends on the first fixture to do something more interesting. This new fixture processes the raw data with the original logic. We then wrote another test to ensure that this processed `DataFrame` also has a length greater than one.

```
%%writefile test_pd_refactor.py
import numpy as np
import pandas as pd
import pytest

import returns

@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )
@pytest.fixture
def orig_price_df(raw_price_df):
    return returns.orig_price_df(raw_price_df)

def test_basic(orig_price_df):
    assert len(orig_price_df) > 1
```

Now let's run the new code:

```
!pytest test_pd_refactor.py
============================= test session starts
     ==============================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition
plugins: anyio-4.2.0
collected 1 item

test_pd_refactor.py .
        [100%]

============================== warnings summary
     ==============================
test_pd_refactor.py::test_basic

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:109: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
```

```
     value by position, use `ser.iloc[pos]`
    bm_cost = price_df[bm_ticker][0]

test_pd_refactor.py::test_basic

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:110: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_price = price_df[bm_ticker][-1]

-- Docs:
     https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 1 passed, 2 warnings in 0.73s
     ========================
```

Success!

39.4 Testing the Refactor

Now, let's add a test to ensure the refactoring works.

We will create the `test_refactor` function. This test ensures that our new `get_price` logic returns the same result as the previous code.

```
%%writefile test_pd_refactor.py
import numpy as np
import pandas as pd
#   from pandas.testing import assert_frame_equal
import pytest

import returns

@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )

@pytest.fixture
def raw_price_df_legacy(raw_price_df):
    return raw_price_df

@pytest.fixture
def orig_price_df(raw_price_df_legacy):
```

```
    return returns.orig_price_df(raw_price_df_legacy)

def test_basic(orig_price_df):
    assert len(orig_price_df) > 1

def test_refactor(orig_price_df, raw_price_df_legacy):
    price_df_rf = raw_price_df_legacy
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    port_rf = returns.get_portfolio(port_tickers, price_df_rf,
                                    price_df_rf[bm_ticker])
    new_price_df = returns.get_price(price_df_rf, bm_ticker, port_rf, K)
    pd.testing.assert_frame_equal(new_price_df, orig_price_df)
```

Let's run the tests with pytest.

```
!pytest test_pd_refactor.py
============================= test session starts
     ==============================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition
plugins: anyio-4.2.0
collected 2 items

test_pd_refactor.py ..
        [100%]

=============================== warnings summary
     ===============================
test_pd_refactor.py::test_basic
test_pd_refactor.py::test_refactor

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:109: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_cost = price_df[bm_ticker][0]

test_pd_refactor.py::test_basic
test_pd_refactor.py::test_refactor
```

```
    /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
   py:110: FutureWarning: Series.__getitem__ treating keys as positions
    is deprecated. In a future version, integer keys will always be
    treated as labels (consistent with DataFrame behavior). To access a
    value by position, use `ser.iloc[pos]`
   bm_price = price_df[bm_ticker][-1]

-- Docs:
    https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 2 passed, 4 warnings in 0.82s
    =========================
```

Awesome! We now have a test that confirms that our refactoring gives the same results as the original code.

39.5 ipytest

In this section, we will delve into another testing mechanism called ipytest. Unlike other command line tools, ipytest is run from Jupyter.

To use IPyTest, ensure it's installed using `pip install IPyTest`. Next, you will need to import the library and run the `autoconfig` function that sets up the library to work in Jupyter.

```
import ipytest
ipytest.autoconfig()
```

Now we should have the `%%ipytest` cell magic available.

This cell magic allows us to create the tests and run them in the same cell. Instead of having to write a file with the test code and then invoke `pytest` in a new cell, I can just stick `%%pytest` at the top of the cell with the test code and run that cell. The ipytest library will run the cell as if it were a pytest test.

```
%%ipytest
import pytest
import returns

@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )

@pytest.fixture
def raw_price_df_legacy(raw_price_df):
    return raw_price_df.astype(float)
```

```
@pytest.fixture
def orig_price_df(raw_price_df_legacy):
    return returns.orig_price_df(raw_price_df_legacy)


def test_basic(orig_price_df):
    assert len(orig_price_df) > 1

def test_refactor(orig_price_df, raw_price_df_legacy):
    price_df_rf = raw_price_df_legacy
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM',
                    'BRK-B','UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    port_rf = returns.get_portfolio(port_tickers, price_df_rf,
                                    price_df_rf[bm_ticker])
    new_price_df = returns.get_price(price_df_rf, bm_ticker, port_rf, K)
    pd.testing.assert_frame_equal(new_price_df, orig_price_df)
```

A beneficial aspect of this approach is its flexibility. We can use ipytest anywhere in our notebook. This way, we can run a quick test to ensure it works. This method is particularly beneficial for collaborations and sanity checks.

39.6 Summary

In this chapter, we introduced pytest a powerful testing framework for Python, along with ipytest. This was only a surface-level introduction, as diving deep into these topics merits a book (or two).

This combination of tools provides the means to write, test, and debug Python code more efficiently, making it easier to ensure code quality and correctness. The capability to run tests directly in Jupyter is a boon for data scientists and those who often use these notebooks for coding.

39.7 Exercises

1. Install pytest and IPyTest using pip. Once installed, write a small script to confirm that pytest is working correctly on your machine.
2. Set up IPyTest in a Jupyter notebook and run a simple test to verify it's correctly configured.
3. Write a pytest test for your pandas code.
4. Execute the test using ipytest.

Chapter 40

Conclusion

Congratulations! If you have read this far and start applying the techniques you have learned, you will have better pandas code than 90% of the pandas users. You will be able to write code that is more readable, more efficient, and more robust. You will be happy when you return to your code in six months and can still understand it.

If you are skeptical, I encourage you to try it out. Apply the patterns you have seen in this book:

- Chaining
- Leveraging *tweak* functions
- Using `.assign`
- Using `.pipe`
- Avoiding `.apply`
- Using PyArrow types
- Turn on copy on write
- Take advantage of Numba or Cython if necessary

I've had many readers comment to me over the years that they were skeptical about the patterns I was teaching, but after trying them out, they completely changed how they wrote pandas code. I hope you will have the same experience.

As an educator, I feel that I must also be open to learning from my students. If you have pandas patterns that you have discovered that are not in this book, please let me know. I would love to learn from you.

40.1 What's next?

Folks often ask what they should do after reading this book. I recommend practicing the patterns you have learned here. Try to apply them to your work or hobby projects. Practice will make you feel comfortable with these patterns, and you will have a deeper understanding of them once you apply them to your problems.

If you are looking for more pandas content, I have many courses at metasnake.com. These include courses on exploring analytics, refactoring and testing, sales reporting, and more.

40.2 Team Training

If you want to help your team learn these patterns, don't hesitate to contact me at matt@metasnake.com. I've helped some of the largest companies in the world improve their pandas code. I would be happy to help your team, too.

40.3 One More Thing

If you enjoyed this book, please leave a review on Amazon. This is the best way to say thank you to me and to help other people find this book. I don't have a large publisher behind me, so I rely on word of mouth to get the word out about this book. I would appreciate it if you took a moment to leave an honest review of this book.

I also appreciate any reviews or mentions on social media. Please tag me so I can thank you!

Index