

Treading on Python Series



EFFECTIVE PANDAS 2

Opinionated Patterns for Data Manipulation

Matt Harrison

Effective Pandas2

Matt Harrison

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.

pandas¹ 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.

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.

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.

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.

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.

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.

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 distribution¹. 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:

```
$ 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.

Pip

If you aren't using Anaconda, I recommend you use `pip`² 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-a11-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'
```

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'
```

Summary

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

Exercises

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

1. <https://anaconda.com/downloads>

2. <http://pip-installer.org/>

Using Jupyter

A standard tool for engineers, scientists, and data people is Jupyter¹. 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.

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.

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

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.

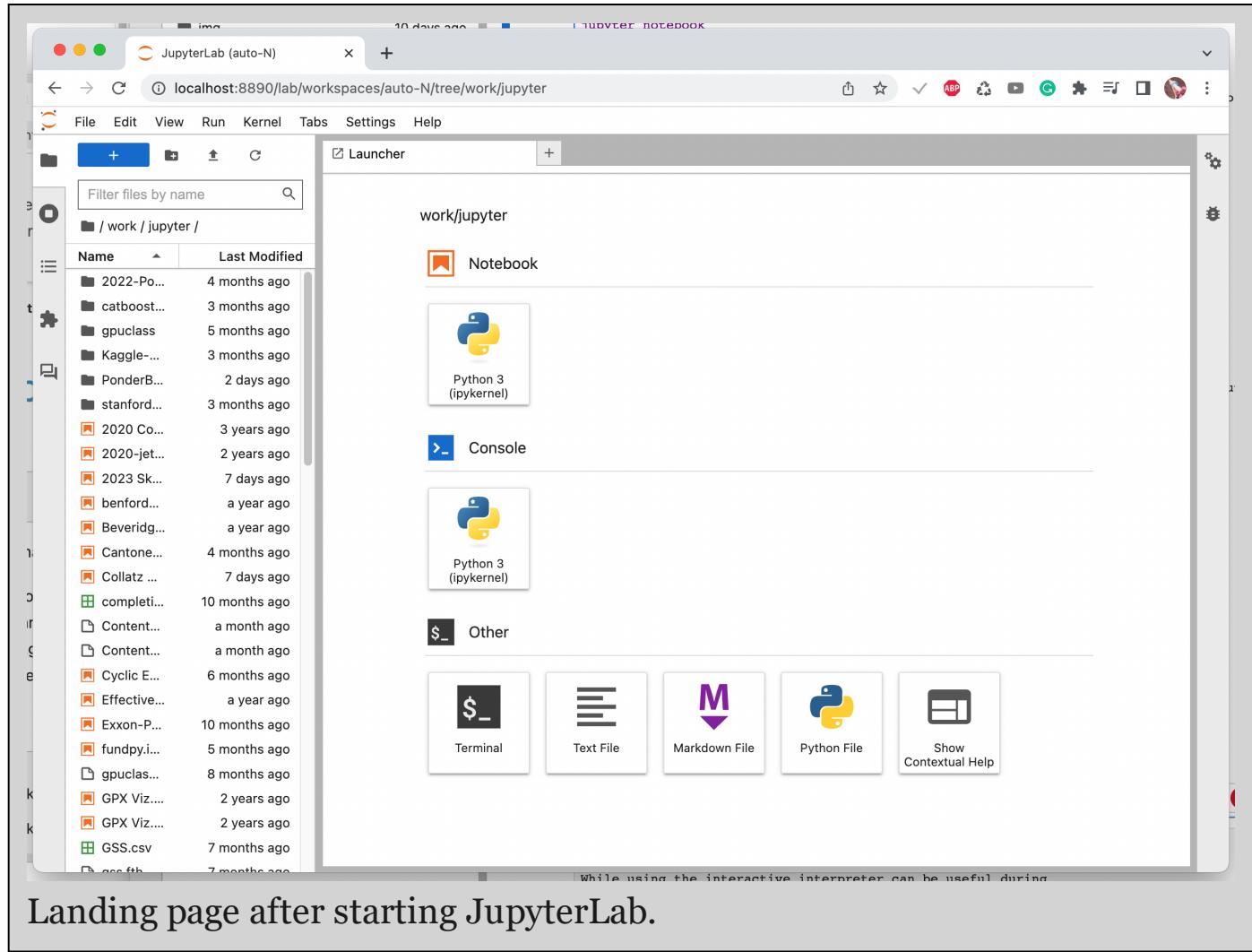
Using Lab

To install JupyterLab, run the command:

```
$ pip install jupyterlab
```

You can launch the tool with the command:

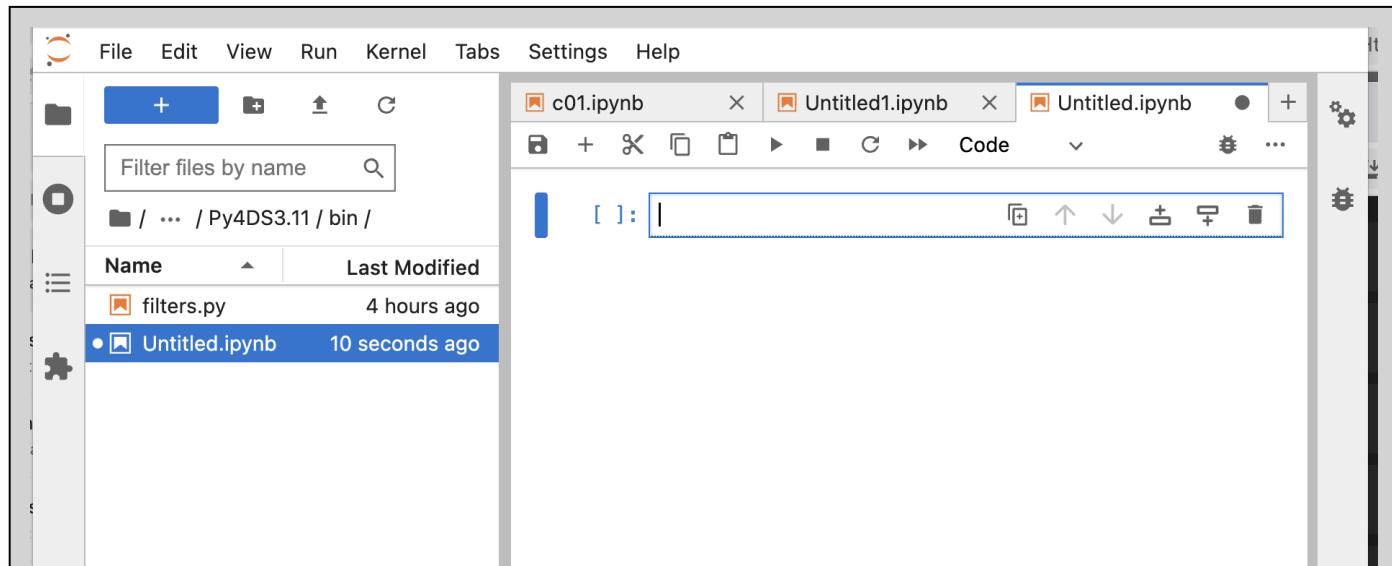
```
$ jupyter lab
```



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.

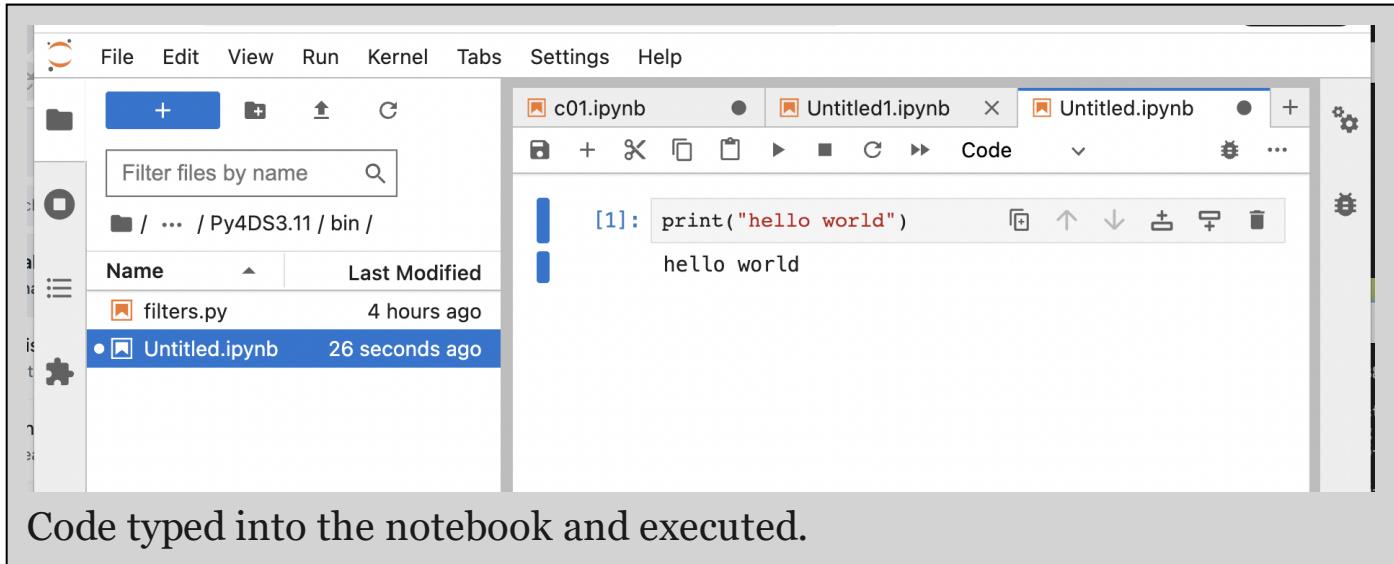
Type your code into the first cell of the notebook. Since you are doing the hello world example, type:

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



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

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.



Code typed into the notebook and executed.

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.

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.

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 **above** the current cell.

`b`

Inserts a new cell **below** 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.

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.

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.

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?

1. <https://jupyter.org/>

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`.

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.

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`).

Data Structures

pd.Series
age

0	15
1	16
2	16
3	15

teacher

0	Ashby
1	Ashby
2	Jones
3	Jones

name

0	Dave
1	Suzy
2	Adam
3	Liv

Index
Axis 0

pd.DataFrame

Column
Axis 1

	age	teacher	name
0	15	Ashby	Dave
1	16	Ashby	Suzy
2	16	Jones	Adam
3	15	Jones	Liv

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

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.

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.

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?

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.

Table of Types

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

Table of Pandas 1 and Pandas 2 types

Data Type	SQL	Python	Pandas 2	Pandas 1
Integer	INT	int	'int64[pyarrow]' pd.ArrowDType(pa.Int64) ¹	'int64'
Float	FLOAT	float	'float64[pyarrow]' pd.ArrowDType(pa.Float64) ¹	'float64'
String	VARCHAR	str	pd.ArrowDType(pa.String) ¹	object, str
Date	DATE	datetime	'timestamp[ns] [pyarrow]' pd.ArrowDType(pa.Timestamp) ¹	'datetime64[ns]'
Categorical	N/A	N/A	'dictionary' ²	'category'
Boolean	BOOLEAN	bool	'bool[pyarrow]' pd.ArrowDType(pa.Bool) ¹	bool

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.

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:

```
// 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.

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.

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.

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  1267650600228229401496703205376
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
```

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.

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 programmer, 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.

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]
```

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]
```

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.

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?
-

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.

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.

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:

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.

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.

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
```

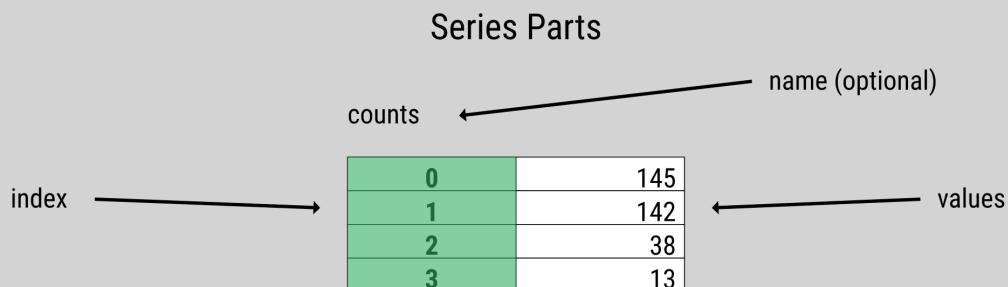
Pandas 2 introduced a new backend, the pyarrow backend. By default, pandas still uses the NumPy backend, which is the `int64` type. To use pyarrow we need to specify the type, '`int64[pyarrow]`'.

```
>>> 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.



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.

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.

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]
```

Filtering with Boolean Arrays

`songs3`

Paul	145
John	142
George	38
Ringo	13

`songs3 > songs3.median()`

Paul	True
John	True
George	False
Ringo	False

Paul	145
John	142

`songs3[songs3 > songs3.median()]`

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”¹.

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', 'x1'], dtype=dict_type)
>>> s
0    m
1    l
2    xs
3    s
4    x1
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', 'x1'], 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', 'x1'], 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', 'x1', ])
...     .cat.reorder_categories(['xs', 's', 'm', 'l', 'x1'],
...                           ordered=True)
... )
0    s
1    m
2    l
dtype: category
Categories (5, object): ['xs' < 's' < 'm' < 'l' < 'x1']
```

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 of functionality explored in this chapter.

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

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.

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.
-

1. Type `import this` into an interpreter to see the Zen of Python. Or search for “PEP 20”.

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 website¹. This site has data on the efficiency of makes and models of cars sold in the US since 1984.

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
```

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.

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).

In []: city.|

The screenshot shows a Jupyter Notebook cell with the prefix "city." followed by a period. A dropdown menu lists several method names starting with "city.", with "abs" highlighted. Other visible methods include add, add_prefix, add_suffix, agg, aggregate, align, all, any, and append.

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

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`.
- 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.

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.

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.
-

1. <https://www.fueleconomy.gov/feg/download.shtml>

Operators (& Dunder Methods)

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.

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]`.

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`.

Duplicate Index Alignment

`s1`

1	10
2	20
2	30

`s2`

2	35
2	44
4	53

`s1 + s2`

1	nan
2	55.00
2	64.00
2	65.00
2	74.00
4	nan

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`.

Duplicate Index Alignment

s1

1	10
2	20
2	30

s2

2	35
2	44
4	53

`s1.add(s2, fill_value=0)`

1	10.00
2	55.00
2	64.00
2	65.00
2	74.00
4	53.00

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.

Remember that the index entries align before performing math operations.

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
```

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).

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.

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.

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.

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

Method	Operator	Description
s.rsub(s2)	s2 - s	Subtracts series
s.mul(s2)	s * s2	Multiplies series
s.multiply(s2)		
s.rmul(s2)	s2 * s	Multiplies series
s.div(s2)	s / s2	Divides series
s.truediv(s2)		
s.rdiv(s2)	s2 / s	Divides series
s.rtruediv(s2)		
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).

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.

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.

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.

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:

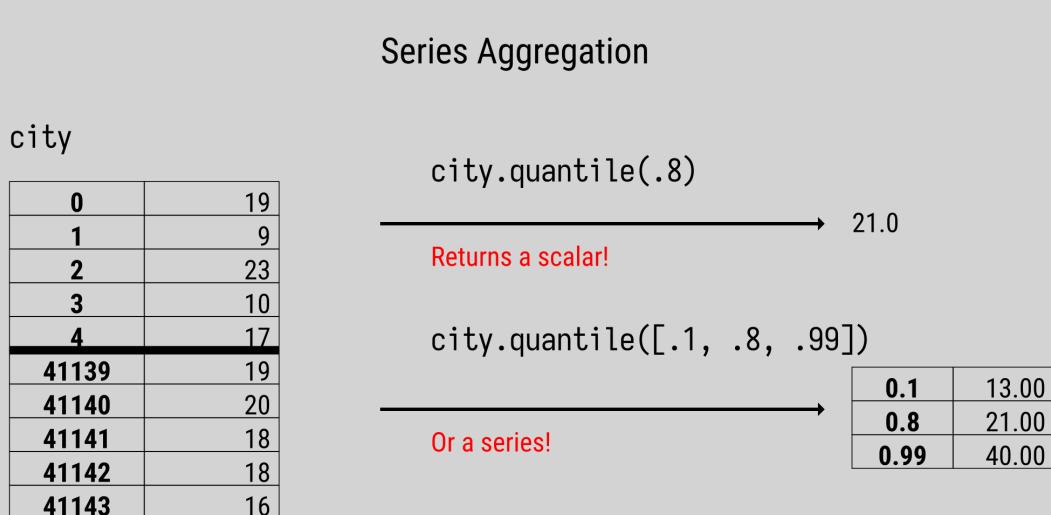
```
>>> city_mpg.mean()  
18.369045304297103
```

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]
```



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.

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.

.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`:

Aggregation strings for `.agg` method

Method	Description
'all'	Returns <code>True</code> if every value is truthy.
'any'	Returns <code>True</code> if any value is truthy.
'autocorr'	Returns Pearson correlation of series with shifted self. Can override <code>lag</code> as keyword argument(default is 1).

Method	Description
'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.
'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.

Method	Description
'sum'	Return the sum of the series.

Below is a table of various aggregation methods and properties.

Aggregation methods and properties

Method	Description
s.agg(func=None, axis=0, *args, **kwargs)	Returns a scalar if <code>func</code> is a single aggregation function. Returns a series if a list of aggregations are passed to <code>func</code> .
s.all(axis=0, bool_only=None, skipna=True, level=None)	Returns <code>True</code> if every value is truthy. Otherwise <code>False</code>
s.any(axis=0, bool_only=None, skipna=True, level=None)	Returns <code>True</code> if at least one value is truthy. Otherwise <code>False</code>
s.autocorr(lag=1)	Returns Pearson correlation between <code>s</code> and shifted <code>s</code>
s.corr(other, method='pearson')	Returns correlation coefficient for ' <code>pearson</code> ', ' <code>spearman</code> ', ' <code>kendall</code> ', 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 <code>s</code> values.
s.quantile(q=.5, interpolation='linear')	Returns 50% quantile by default. <i>Note</i> returns series if <code>q</code> is a list.

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

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.

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.

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.

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.

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:

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()
)
```

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`.

Type and strings for column conversion

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

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.

Aggregation methods and properties

Method	Description
<code>s.convert_dtypes(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True)</code>	Convert types to appropriate pandas 1 types (that support NA). Doesn't try to reduce size of integer or float types.

Method	Description
<code>s.astype(dtype, copy=True, errors='raise')</code>	Cast series into particular type. If <code>errors='ignore'</code> then return original series on error.
<code>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)</code>	Convert <code>arg</code> (a series) into datetime. Use <code>format</code> to specify strftime string.
<code>s.to_numpy(dtype=None, copy=False, na_value=object, **kwargs)</code>	Convert the series to a NumPy array.
<code>s.values</code>	Convert the series to a NumPy array.
<code>s.to_frame(name=None)</code>	Return a dataframe representation of the series.
<code>pd.CategoricalDtype(categories=None, ordered=False)</code>	Create a type for categorical data.
<code>pd.ArrowDtype(pa.string())</code>	Create a PyArrow string type for casting

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.

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.

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.

.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

make

0	Oldsmobile
1	Chrysler
2	Ford
3	Jeep
4	BMW
15	Chevrolet
16	Mitsubishi
17	GMC
18	Chevrolet
19	Suzuki

make.isin(top5)

0	False
1	False
2	True
3	False
4	False
15	True
16	False
17	True
18	True
19	False

make.where(
make.isin(top5),
other='Other')

0	Other
1	Other
2	Ford
3	Other
4	Other
15	Chevrolet
16	Other
17	GMC
18	Chevrolet
19	Other

top5

['Chevrolet', 'Ford', 'Dodge', 'GMC', 'Toyota']

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

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)

>>> %%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.

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.

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 with 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 faster.

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 traditional *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]
```

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.

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]
```

Missing Data for Series

data

1	0.00
2	10.00
3	18.00
4	12.00
5	nan
6	7.00
7	8.00

→
(data
.dropna())

1	0.00
2	10.00
3	18.00
4	12.00
6	7.00
7	8.00

→
(data
.ffill())

Also bfill

↓
(data
.interpolate())

1	0.00
2	10.00
3	18.00
4	12.00
5	9.50
6	7.00
7	8.00

(data
.fillna(data
.mean()))

1	0.00
2	10.00
3	18.00
4	12.00
5	12.00
6	7.00
7	8.00

1	0.00
2	10.00
3	18.00
4	12.00
5	9.17
6	7.00
7	8.00

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.

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.

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
```

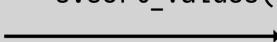
Sorting Values

The `.sort_values` Method

s

0	40
1	20
2	30
3	20
4	10

`s.sort_values()`



4	10
1	20
3	20
2	30
0	40

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]
```

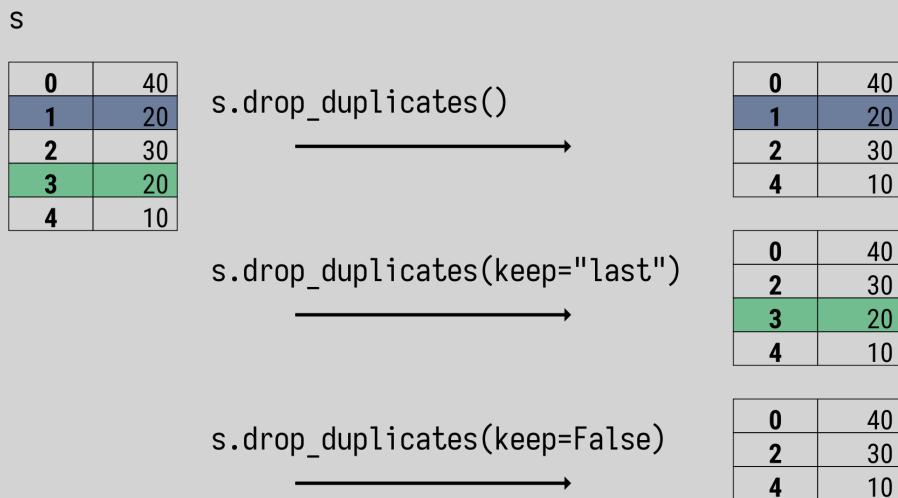
Sorting the Index

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]
```

Dropping Duplicates

The `.drop_duplicates` Method



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]
```

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.

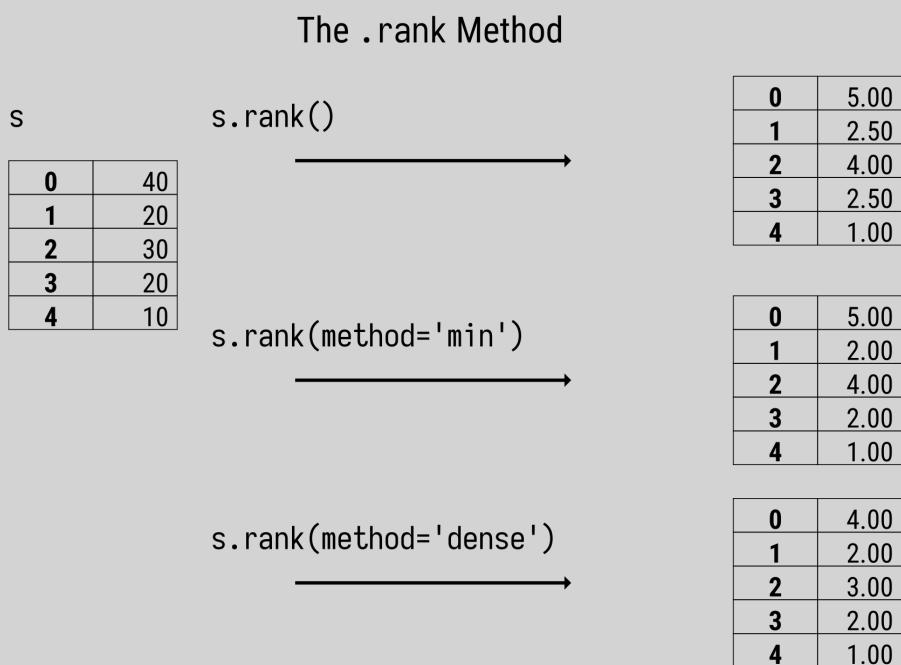
```
>>> city_mpg.rank()
0      27060.5
1      235.5
2     35830.0
...
41141   23528.0
41142   23528.0
41143   15479.0
Name: city08, Length: 41144, dtype: double[pyarrow]
```

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]
```



The `.rank` method has various options for dealing with ties.

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]
```

The .replace Method

s	s.replace(to_replace=[40, 10], value=[42, 9.8])
0 40	0 42.00
1 20	1 20.00
2 30	2 30.00
3 20	3 20.00
4 10	4 9.80

s	s.replace(to_replace={40: 42, 10: 9.8})
0 40	0 42.00
1 20	1 20.00
2 30	2 30.00
3 20	3 20.00
4 10	4 9.80

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]
```

The .replace Method for Series

s	s.replace(to_replace='Suzy', value='Suzanne')		<table border="1"><tr><td>0</td><td>Dave</td></tr><tr><td>1</td><td>Suzanne</td></tr><tr><td>2</td><td>Adam</td></tr><tr><td>3</td><td>Liv</td></tr></table>	0	Dave	1	Suzanne	2	Adam	3	Liv
0	Dave										
1	Suzanne										
2	Adam										
3	Liv										
	s.replace(to_replace={'Suzy': 'Suzanne'})		<table border="1"><tr><td>0</td><td>Dave</td></tr><tr><td>1</td><td>Suzanne</td></tr><tr><td>2</td><td>Adam</td></tr><tr><td>3</td><td>Liv</td></tr></table>	0	Dave	1	Suzanne	2	Adam	3	Liv
0	Dave										
1	Suzanne										
2	Adam										
3	Liv										
	s.replace(to_replace='z.*', value='zanne', regex=True)		<table border="1"><tr><td>0</td><td>Dave</td></tr><tr><td>1</td><td>Suzanne</td></tr><tr><td>2</td><td>Adam</td></tr><tr><td>3</td><td>Liv</td></tr></table>	0	Dave	1	Suzanne	2	Adam	3	Liv
0	Dave										
1	Suzanne										
2	Adam										
3	Liv										

The .replace method illustrating different replacement mechanisms.

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]
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]

```

Manipulation methods and properties

Method	Description
<code>s.apply(func, convert_dtype=True, args=(), **kwds)</code>	Pass in a NumPy function that works on the series, or a Python function that works on a single value. <code>args</code> and <code>kwds</code> are arguments for <code>func</code> . Returns a series, or dataframe if <code>func</code> returns a series.
<code>s.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False)</code>	Pass in a boolean series/dataframe, list, or callable as <code>cond</code> . If the value is <code>True</code> , keep it, otherwise use <code>other</code> value. If it is a function, it takes a series and should return a boolean sequence.

Method	Description
<code>s.case_when(caselst)</code>	Use <code>caselst</code> (list of tuples of (boolean array, result), to simulate if then statements
<code>s.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)</code>	Pass in a scalar, dict, series, or dataframe for <code>value</code> . If it is a scalar, use that value, otherwise use the index from the old value to the new value.
<code>s.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)</code>	Perform interpolation with missing values. <code>method</code> may be <code>linear</code> , <code>time</code> among others.
<code>s.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)</code>	Return a new series with values clipped to <code>lower</code> and <code>upper</code> .
<code>s.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)</code>	Return a series with values sorted. The <code>kind</code> option may be ' <code>quicksort</code> ', ' <code>mergesort</code> ' (stable), or ' <code>heapsort</code> '. <code>na_position</code> indicates location of NaNs and may be ' <code>first</code> ' or ' <code>last</code> '.
<code>s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)</code>	Return a series with index sorted. The <code>kind</code> option may be ' <code>quicksort</code> ', ' <code>mergesort</code> ' (stable), or ' <code>heapsort</code> '. <code>na_position</code> indicates location of NaNs and may be ' <code>first</code> ' or ' <code>last</code> '.
<code>s.drop_duplicates(keep='first', inplace=False)</code>	Drop duplicates. <code>keep</code> may be ' <code>first</code> ', ' <code>last</code> ', or <code>False</code> . (If <code>False</code> , it removes all values that were duplicated).

Method	Description
<code>s.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)</code>	Return a series with numerical ranks. <code>method</code> 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). <code>na_option</code> allows you to specify NaN handling. 'keep' (stay at NaN), 'top' (move to smallest), 'bottom' (move to largest).
<code>s.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad')</code>	Return a series with new values. <code>to_replace</code> can be many things. If it is a string, number, or regular expression, you can replace it with a scalar <code>value</code> . It can also be a list of those things which requires <code>values</code> to be a list of the same size. Finally, it can be a dictionary mapping old values to new values.
<code>pd.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates='raise', ordered=True)</code>	Bin values from <code>x</code> (a series). If <code>bins</code> is an integer, use equal-width bins. If <code>bins</code> is a list of numbers (defining minimum and maximum positions) use those for the edges. <code>right</code> defines whether the right edge is open or closed. <code>labels</code> allows you to specify the bin names. Out of bounds values will be missing.
<code>pd.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')</code>	Bin values from <code>x</code> (a series) into <code>q</code> equal-sized bins (10 for quantiles, 4). Alternatively, can pass in a list of quantile edges. Out of bounds values will be missing.

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.

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.

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.

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.

The .rename Method for Series

s

0	Dave
1	Suzy
2	Adam
3	Liv

s.rename(index={0: 'first'})



first	Dave
1	Suzy
2	Adam
3	Liv

```
def to_str(val):
    return f"idx-{val}"
s.rename(to_str)
```



idx-0	Dave
idx-1	Suzy
idx-2	Adam
idx-3	Liv

s2 = pd.Series(['a', 'b', 'c', 'd'])
s.rename(index=s2)



a	Dave
b	Suzy
c	Adam
d	Liv

s.rename(index="first")



0	Dave
1	Suzy
2	Adam
3	Liv

Only changes the name attribute!

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.

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',
```

```
'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]
```

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]

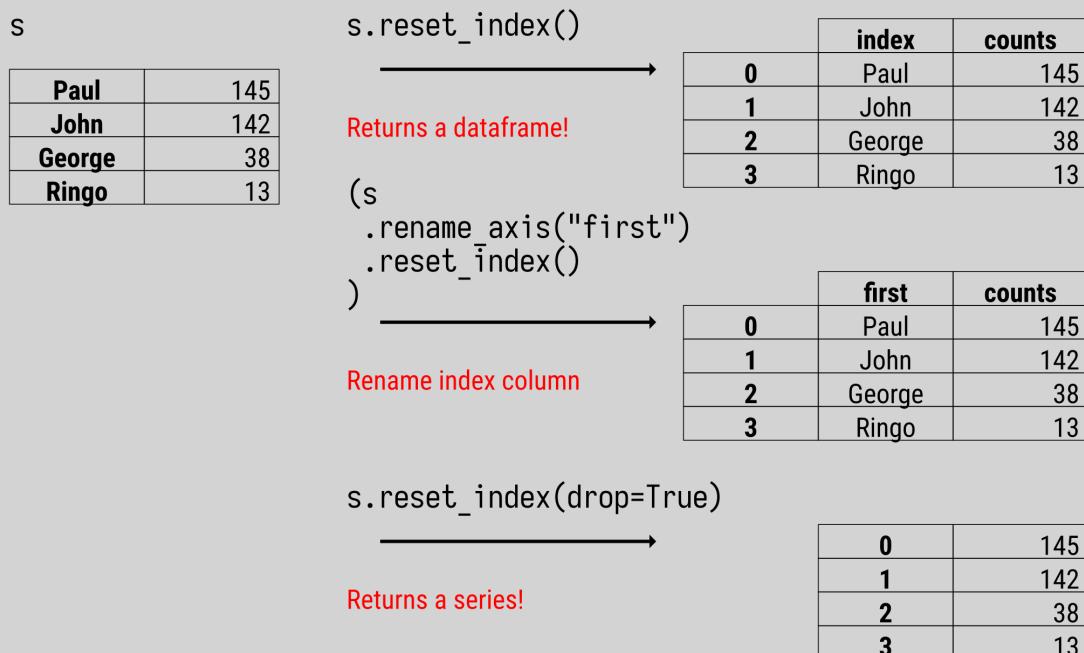
To drop the current index and return a series, use the `drop=True` parameter:

```

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

```

The `.reset_index` Method for Series



The `.reset_index` method will return a DataFrame or a Series with the index changed to a monotonically increasing index.

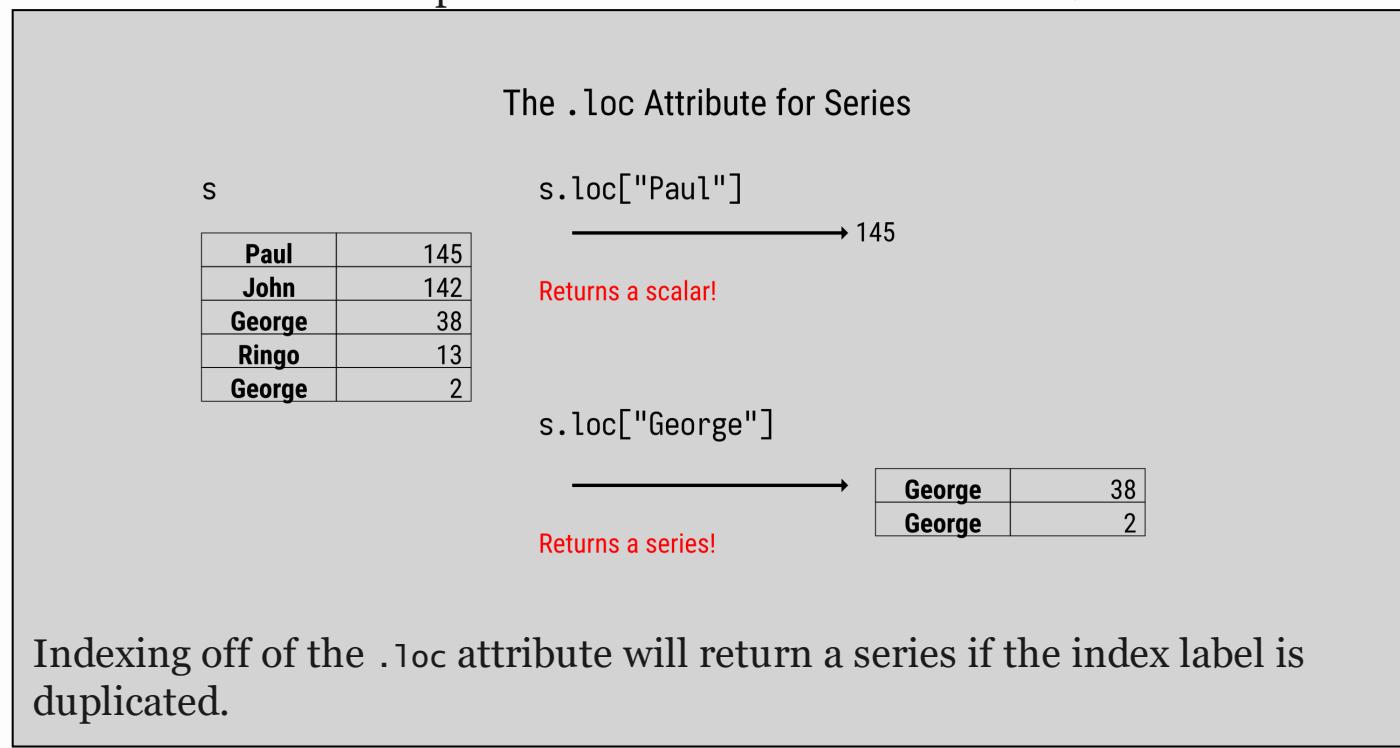
Note that you can sort the values and the index with `.sort_values` and `.sort_index` respectively. Because those keep the same index but rearrange the order, they do not impact operations that align on the index.

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.



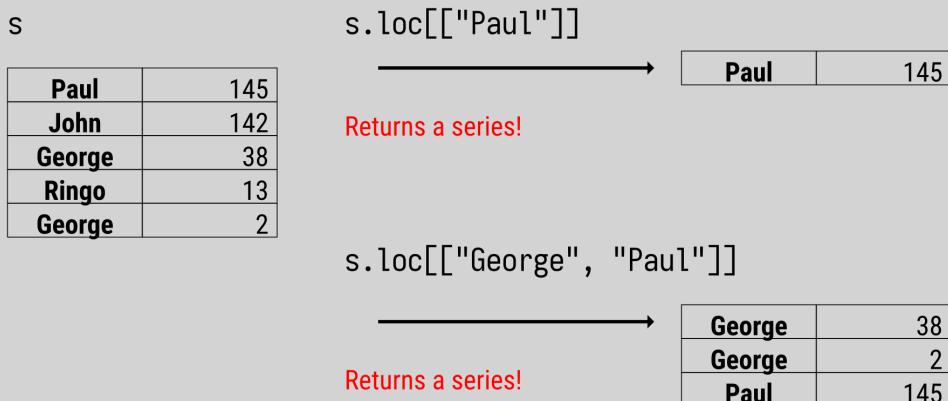
If you pass in a scalar with the label of an index, you need to be careful. If there are duplicate labels in the index, it will return a series, but if there is

only one value for that label, it will return a scalar. In the example below '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
```

The .loc Attribute for Series



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

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]

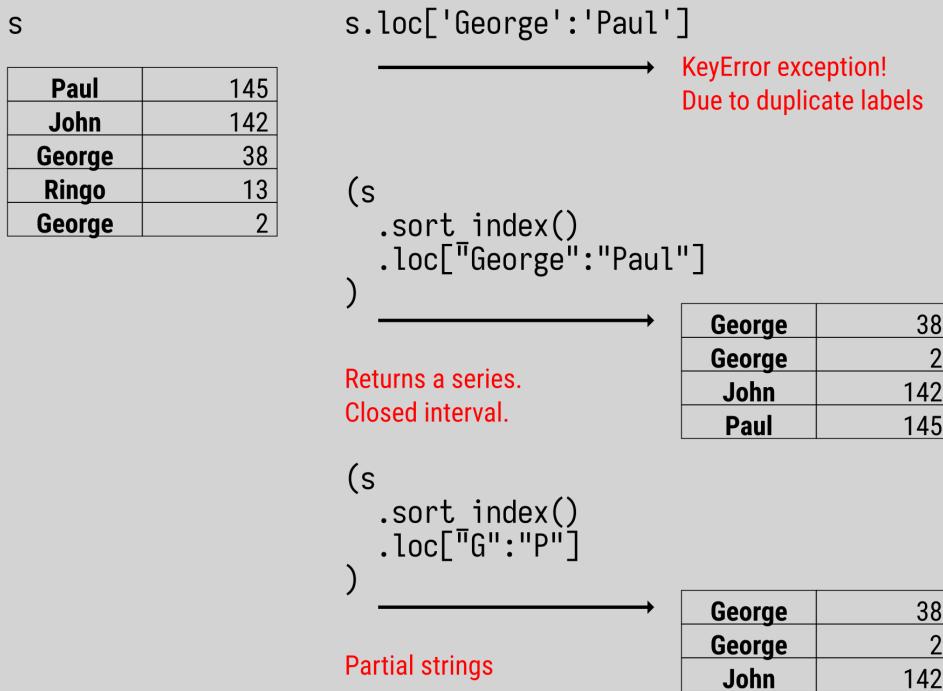
>>> city2.loc[['Ferrari', 'Lamborghini']]
Ferrari      9
Ferrari     12
```

```

Ferrari      11
...
Lamborghini   8
Lamborghini   8
Lamborghini   8
Name: city08, Length: 357, dtype: int64[pyarrow]

```

The .loc Attribute for Series



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.

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]
```

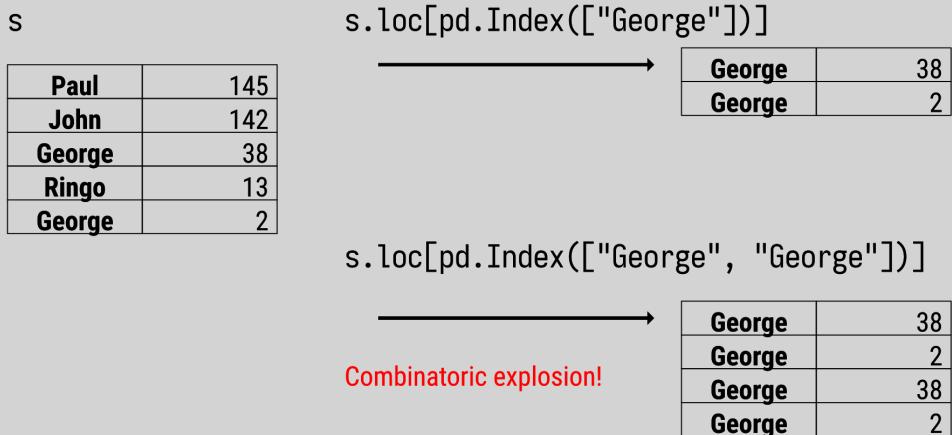
Note that when slicing with `.loc`, it follows the *closed interval*. The closed interval includes both the start index and the final index. This behavior differs from the *half-open interval* found in Python's slicing behavior for strings and lists (which includes the start index, going up to but not 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.

The .loc Attribute for Series



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
```

The .loc Attribute for Series

cost	
Gum	1.00
Cookie	2.25
Melon	3.99
Roll	0.99
Carrots	2.79

```
(cost  
    .mul(inflation)  
    .loc[gt3]  
)
```

Filters on intermediate value

Melon	4.39
Carrots	3.07

```
def gt3(ser):  
    return ser > 3  
  
inflation = 1.10
```

```
(cost  
    .mul(inflation)  
    .loc[cost > 3]  
)
```

Filters on original value

Melon	4.39
-------	------

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.

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:

```
>>> 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.

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
```

The <code>.iloc</code> Attribute for Series		
s	s.iloc[0]	
Paul	145	
John	142	
George	38	
Ringo	13	
George	2	

Returns a scalar!

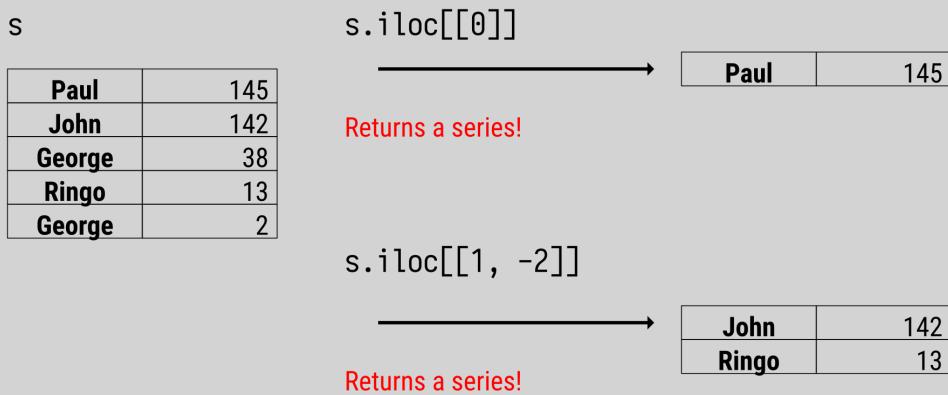
Indexing off of the `.iloc` attribute will return a scalar by location in the series.

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
```

The .iloc Attribute for Series



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

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:

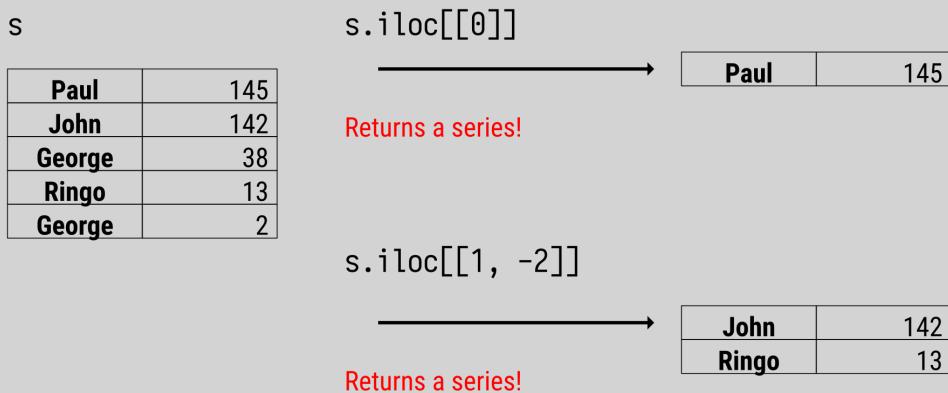
```
>>> city2.iloc[0:5]  
Alfa Romeo    19  
Ferrari        9  
Dodge         23  
Dodge         10  
Subaru       17  
Name: city08, dtype: int64[pyarrow]
```

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]
```

The `.iloc` Attribute for Series



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

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
```

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.

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]
```

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]
```

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]
```

Reindexing

The `.reindex` Method for Series

s

Paul	145
John	142
George	38
Ringo	13
George	2

`s.reindex(['Paul', 'John', 'Eric'])`

ValueError exception!
Due to duplicate labels

`s.iloc[:-1].reindex(['Paul', 'John', 'Eric'])`

Returns a series!

Paul	145.00
John	142.00
Eric	nan

The `.reindex` method will *conform* an index to a new index.

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):
...
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
```

Method	Description
<code>s.rename(index=None, *, level=None, errors='ignore')</code>	Return a series with updated <code>.name</code> attribute if <code>index</code> is a scalar. If <code>index</code> 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).
<code>s.index</code>	Returns the index of the series.
<code>s.reset_index(level=None, drop=False, name=None, inplace=False)</code>	Return a dataframe (or series when <code>drop=True</code>) with a new integer index.
<code>s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)</code>	Return a series with the index sorted. The <code>kind</code> option may be ' <code>quicksort</code> ', ' <code>mergesort</code> ' (stable), or ' <code>heapsort</code> '. <code>na_position</code> indicates the location of NaNs and may be ' <code>first</code> ' or ' <code>last</code> '.
<code>s.loc[idx]</code>	Slice series by names. <code>idx</code> 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).
<code>s.iloc[idx]</code>	Slice series by index position. <code>idx</code> 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).
<code>s.head(n=5)</code>	Return a series with the first <code>n</code> values.
<code>s.tail(n=5)</code>	Return a series with the last <code>n</code> values.

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

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`.

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 as 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'.

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.

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, don't 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.

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.

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'
```

String Methods for Series

data

1	suz
2	john
3	fred
4	george



(data
.str.capitalize())

1	Suz
2	John
3	Fred
4	George

(data
.str.find("e"))

1	-1
2	-1
3	2
4	1

(data
.str
.startswith('f'))

1	False
2	False
3	True
4	False

(data
.str
.extract(r'([a-e])',
expand=False))

1	nan
2	nan
3	e
4	e

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

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:

```
>>> '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.

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 2.0.2¹, 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]
```

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  
.  
,      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.

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
```

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

```

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)
```

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)

```

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.

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', 'À')
0      Alfa 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', 'À')
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': 'Àudi', 'Acura': 'Àcura',
...                 'Ashton Martin': 'Àshton Martin',
...                 'Alfa Romeo': 'Àlfa Romeo'})
0      Alfa 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', 'À', regex=True)
0      Alfa 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.

String methods

Method	Description
<code>.str.capitalize()</code>	Capitalize strings
<code>.str.casefold()</code>	Lowercase Unicode/caseless strings.
<code>.str.cat(others=None, sep='', na_rep=None, join='inner')</code>	If <code>others</code> is <code>None</code> , return a string with values separated by <code>sep</code> . Otherwise, align the index (if <code>others</code> series) and concatenate values.
<code>.str.center(width, fillchar='')</code>	Center align strings
<code>.str.contains(pat, case=True, flags=0, na=np.nan, regex=True)</code>	Return a boolean array if <code>pat</code> matches values.
<code>.str.count(pat, flags=0)</code>	Return series with the count of how many times <code>pat</code> occurs in each value.
<code>.str.decode(encoding)</code>	Works with bytestrings to decode them to Unicode strings.
<code>.str.encode(encoding)</code>	Encode Unicode string to bytestring.
<code>.str.endswith(pat, na=np.nan)</code>	Return boolean array if value ends with <code>pat</code> .

Method	Description
<code>.str.extract(pat, flags=0, expand=True)</code>	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 <code>expand=False</code> .
<code>.str.findall(pat, flags=0)</code>	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 <i>match</i> and has match number.
<code>.str.find(sub, start=None, end=None)</code>	Return the lowest index of <code>sub</code> . -1 if not found.
<code>.str.findall(pat, flags=0)</code>	Return a series with a list of matches for each value.
<code>.str.get(i)</code>	Return a series with the result of <code>val[i]</code> for each value (<code>val</code>) in the series.
<code>.str.get_dummies(sep=' ')</code>	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 <code>sep</code> .
<code>.str.index(sub, start=None, end=None)</code>	Return the lowest index of <code>sub</code> . <code>ValueError</code> if not found.
<code>.str.isalnum()</code>	Return boolean array if characters are alphanumeric.
<code>.str.isalpha()</code>	Return boolean array if characters are alphabetic.
<code>.str.isdecimal()</code>	Return boolean array if characters are decimal.
<code>.str.isdigit()</code>	Return boolean array if characters are digits.
<code>.str.islower()</code>	Return boolean array if characters are lowercase.

Method	Description
<code>.str.isnumeric()</code>	Return boolean array if characters are numeric.
<code>.strisspace()</code>	Return boolean array if characters are whitespace.
<code>.str.istitle()</code>	Return boolean array if characters are titlecase.
<code>.str.isupper()</code>	Return boolean array if characters are uppercase.
<code>.str.join(sep)</code>	Given a series with a list of strings in it, join each element with <code>sep</code> .
<code>.str.len()</code>	Return a series with length of each value (works with lists or collections).
<code>.str.ljust(width, fill=' ')</code>	Return a left justified series.
<code>.str.lower()</code>	Return a lowercase series.
<code>.str.lstrip(to_strip=None)</code>	Return a series with left stripped <code>to_strip</code> (whitespace default).
<code>.str.match(pat, case=True, flags=0, na=np.nan)</code>	Return a boolean array if <code>pat</code> matches values (anchored at the beginning). Use <code>.str.contains</code> to match anywhere in the string. (Use <code>.str.extract</code> to pull out the string.)
<code>.str.normalize(form)</code>	Return Unicode normal form for series. <code>form</code> can be 'NFC', 'NFKC', 'NFD', or 'NFKD'.
<code>.str.pad(width, side='left', fill=' ')</code>	Return a padded series of length <code>width</code> . <code>side</code> can be 'left', 'right', or 'both'.
<code>.str.partition(sep, expand=True)</code>	Return a dataframe with three columns: element before first <code>sep</code> , the <code>sep</code> , and the part after.
<code>.str.repeat(repeats)</code>	Return a series with values repeated <code>repeats</code> times. <code>repeats</code> can be a scalar or list.

Method	Description
<code>.str.replace(pat, repl, n=-1, case=True, flags=0, regex=True)</code>	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.
<code>.str.rfind(sub, start=None, end=None)</code>	Return highest index of sub. -1 if not found.
<code>.str.rindex(sub, start=None, end=None)</code>	Return highest index of sub. ValueError if not found.
<code>.str.rjust(width, fill=' ')</code>	Return a right justified series.
<code>.str.rpartition(sep, expand=True)</code>	Return a dataframe with three columns: element before last sep, the sep, and the part after.
<code>.str.rsplit(pat, n=-1, expand=False)</code>	Return a Series (if expand=False) with a list of values split from the right side limited to n splits.
<code>.str.rstrip(to_strip=None)</code>	Return a series with rightstripped to_strip (whitespace default).
<code>.str.slice(start=None, stop=None, step=None)</code>	Return a series. Equivalent to s[start:stop:step].
<code>.str.slice_replace(start=None, stop=None, repl=None)</code>	Return a series with slice replaced by the value of repl.
<code>.str.split(pat, n=-1, expand=False)</code>	Return a Series (if expand=False) with a list of values split by sep limited to n splits.
<code>.str.startswith(pat, na=np.nan)</code>	Return boolean array if value starts with pat.
<code>.str.strip(to_strip=None)</code>	Return a series with left and right stripped to_strip (whitespace default).
<code>.str.swapcase()</code>	Return swapcase series.
<code>.str.title()</code>	Return titlecase series.

Method	Description
<code>.str.translate(table)</code>	Return series using a dictionary <code>table</code> to replace characters. <code>table</code> maps code points to new code points (numbers not strings). Keys mapped to <code>None</code> are deleted.
<code>.str.upper()</code>	Return uppercase series.
<code>.str.wrap(width)</code>	Return a line wrapped series limited to <code>width</code> .
<code>.str.zfill(width)</code>	Return a series limited to <code>width</code> left padded with '0'.

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.

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.

1. <https://github.com/pandas-dev/pandas/issues/56268>

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.

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>

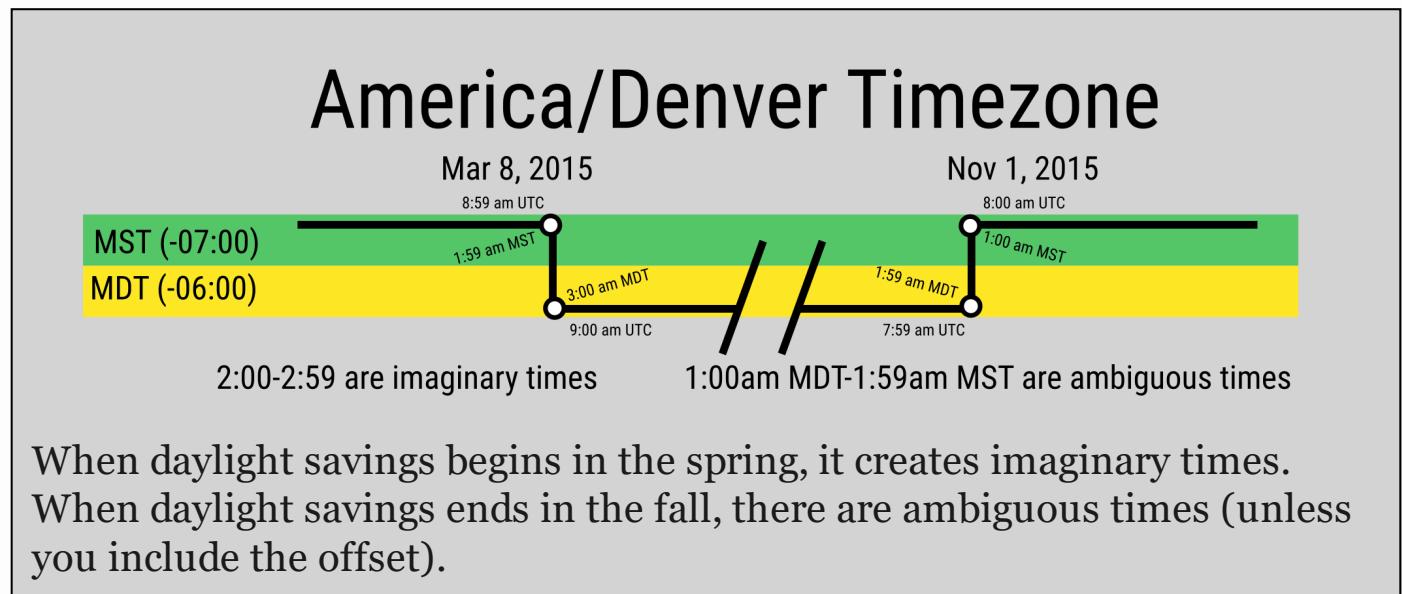
Getting the correct time zone name is essential and might be confusing or complicated. As I said, I live in Salt Lake City. If I search for “Time zone for Salt Lake City”, I get “Mountain Daylight Time” or “GMT-6”. Neither of which is a time zone. You might also see “US/Mountain”, “MST”, or “MDT”. These 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.



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 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.

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]
```

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, -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]
```

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]
```

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    14258016000000000000  
1    14258034000000000000  
2    14258052000000000000  
     ...  
16   14463684000000000000  
17   14463702000000000000  
18   14463720000000000000  
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]
```

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 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

Code	Meaning	Sample
<code>%%</code>	Percent sign	<code>%</code>

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]` [2](#). `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 of timedelta attributes

Attribute	Meaning
<code>.as_unit(unit)</code>	Change the units of the duration (supports 's', 'ms', 'us', 'ns')
<code>.ceil(freq)</code>	Round up to specified units (supports 'D', 'H', 'T' ('min'), 's', 'ms', 'us', 'ns')
<code>.components</code>	Property with dataframe of duration components
<code>.days</code>	Property with number of days
<code>.floor(freq)</code>	Round down to specified units (supports 'D', 'H', 'T' ('min'), 's', 'ms', 'us', 'ns')
<code>.freq</code>	Frequency of the duration
<code>.microseconds</code>	Property with number of microseconds
<code>.nanoseconds</code>	Property with number of nanoseconds

Attribute	Meaning
.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 <code>datetime.timedelta</code> objects
.total_seconds()	Return total seconds
.unit	Property to get the unit of the duration (use <code>.as_unit</code> 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:

```
>>> (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.

.dt methods and Properties

Method	Description
<code>.ceil(freq=None, ambiguous=None, nonexistent=None)</code>	Return ceiling according to offset alias in <code>freq</code> . The <code>nonexistent</code> parameter controls DST time issues.
<code>.date</code>	Property with a series of Python <code>datetime.date</code> objects.
<code>.day</code>	Property with a series of day of month.
<code>.day_name(locale='en_us')</code>	Return the string day of week.
<code>.dayofweek</code>	Property with a series of date of week as number (0 is Monday).
<code>.dayofyear</code>	Property with a series of day of the year.
<code>.days_in_month</code>	Property with a series of number of days in month.
<code>.daysinmonth</code>	Property with a series of number of days in month.
<code>.floor(freq=None, ambiguous=None, nonexistent=None)</code>	Return floor according to offset alias in <code>freq</code> . The <code>nonexistent</code> parameter controls DST time issues.
<code>.hour</code>	Property with a series of hour of date.
<code>.is_leap_year</code>	Property with a series of booleans if date is leap year.
<code>.is_month_end</code>	Property with a series of booleans if date is end of month.

Method	Description
.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).
.strftime(date_format)	Return a series with string dates. Formatted using strftime format codes.
.time	Property with a series of Python <code>datetime.time</code> objects.
.timetz	Property with a series of Python <code>datetime.time</code> objects with time zone information.
.to_period(freq)	Return a series with pandas <code>Period</code> objects.

Method	Description
<code>.to_pydatetime()</code>	Return a numpy array with <code>datetime.datetime</code> objects.
<code>.tz</code>	Property with time zone.
<code>.tz_convert(tz)</code>	Convert from one time zone aware series to another.
<code>.tz_localize(tz, ambiguous=None, nonexistent=None)</code>	Convert from naive to time zone aware.
<code>.week</code>	Property with a series of week of date (numeric 1-53).
<code>.weekday</code>	Property with a series of date of week as number 0 is Monday.
<code>.weekofyear</code>	Property with a series of week of date (numeric 1-53).
<code>.year</code>	Property with a series of year of date.

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.

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.
-

1. https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

2. <https://github.com/pandas-dev/pandas/issues/56269>

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]
```

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]
```

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.

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

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]
```

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.

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]
```

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]

The .rolling Method

data		(data .rolling(3) .mean())	
0	2.00	0	nan
1	1.00	1	nan
2	3.00	2	2.00
3	5.00	3	3.00
4	nan	4	nan
5	2.00	5	nan
6	1.00	6	nan
7	8.00	7	3.67
8	6.00	8	5.00
9	10.00	9	8.00

The .rolling method slides a window along the data, allowing you to call an aggregate function.

Alternative .rolling using the .shift Method

data

0	2.00
1	1.00
2	3.00
3	5.00
4	nan
5	2.00
6	1.00
7	8.00
8	6.00
9	10.00



```
(data  
    .add(data.shift(1))  
    .add(data.shift(2))  
    .div(3)  
)
```

0	nan
1	nan
2	2.00
3	3.00
4	nan
5	nan
6	nan
7	3.67
8	5.00
9	8.00

0	nan
1	2.00
2	1.00
3	3.00
4	5.00
5	nan
6	2.00
7	1.00
8	8.00
9	6.00

0	nan
1	nan
2	2.00
3	1.00
4	3.00
5	5.00
6	nan
7	2.00
8	1.00
9	8.00

The .rolling method slide is similar to shifting the data for N-1 window size and then applying an aggregation.

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
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64

```

Below are methods that work on a `Rolling` object:

Rolling methods and properties

Method	Description
<code>r.agg(func=None, axis=0, *args, **kwargs)</code>	Returns a scalar if <code>func</code> is a single aggregation function. Returns a series if a list of aggregations are passed to <code>func</code> . (<code>aggregate</code> is a synonym.)
<code>r.apply(func, args=None, kwargs=None)</code>	Apply custom aggregation function to rolling group.
<code>r.corr(other, method='pearson')</code>	Returns correlation coefficient for 'pearson', 'spearman', 'kendall', or a callable.
<code>r.count(other, method='pearson')</code>	Returns count of non <code>NaN</code> values.
<code>r.cov(other, min_periods=None)</code>	Returns covariance.
<code>r.max(axis=None, skipna=None, level=None, numeric_only=None)</code>	Returns maximum value.
<code>r.min(axis=None, skipna=None, level=None, numeric_only=None)</code>	Returns minimum value.
<code>r.mean(axis=None, skipna=None, level=None, numeric_only=None)</code>	Returns mean value.
<code>r.median(axis=None, skipna=None, level=None, numeric_only=None)</code>	Returns median value.
<code>r.quantile(q=.5, interpolation='linear')</code>	Returns 50% quantile by default. <i>Note</i> returns series if <code>q</code> is a list.
<code>r.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)</code>	Returns unbiased standard error of mean.

Method	Description
<code>r.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)</code>	Returns sample standard deviation.
<code>r.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)</code>	Returns unbiased variance.
<code>r.skew(axis=None, skipna=None, level=None, numeric_only=None)</code>	Returns unbiased skew.

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 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.

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 <code>-MAY</code> 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)

Offset Alias	Date Offset	Description
'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
'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

Offset	Date Offset	Description
Alias		
'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.

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]
```

The .resample Method

data

1990/01/01	5.00
1990/01/10	2.70
1990/01/24	3.20
1990/02/01	0.00
1990/02/10	1.10
1990/02/24	8.00

The offset alias 'ME' aggregates at the month end level. The .transform method puts the results into the original index.

```
(data  
    .resample('ME')  
    .sum()  
)
```

```
(data  
    .resample('ME')  
    .transform('sum')  
)
```

1990/01/31	10.90
1990/02/28	9.10

1990/01/01	10.90
1990/01/10	10.90
1990/01/24	10.90
1990/02/01	9.10
1990/02/10	9.10
1990/02/24	9.10

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.

Resampler Methods for a Series

Method	Description
<code>.agg(func, *args, **kwargs)</code>	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 dataframe for multiple functions.
<code>.aggregate(func, *args, **kwargs)</code>	Same as <code>.agg</code>
<code>.apply(func, *args, **kwargs)</code>	Same as <code>.agg</code>
<code>.asfreq(fill_value=None)</code>	Return values at frequency (like <code>.reindex</code>)
<code>.backfill(limit=None)</code>	Backfill the missing values.
<code>.bfill(limit=None)</code>	Same as <code>.backfill</code>
<code>.count()</code>	Count of non-missing items in group.
<code>.ffill(limit=None)</code>	Forward fill the missing values.
<code>.fillna(method, limit=None)</code>	Method (' <code>ffill</code> ', ' <code>bfill</code> ', or ' <code>nearest</code> ') to use for filling in missing data for upsampling.
<code>.first()</code>	Return a series with the first value of each group.
<code>.get_group(name, obj=None)</code>	Return the series for grouping frequency of name.

Method	Description
.interpolate(method='linear', axis=0, limit=None, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)	Return a series with interpolated values.
.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.

Method	Description
.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.

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.

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]

Date Manipulation Methods

Method	Description
<pre>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)</pre>	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.
.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.

Method	Description
<code>.where(cond, other=nan, level=None, errors='raise', try_cast=False)</code>	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).
<code>.dropna()</code>	Return a series with missing values removed.
<code>.shift(periods=1, freq=None, fill_value=None)</code>	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.
<code>.rolling(window, min_periods=None, center=False, win_type=None, closed='right')</code>	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.
<code>.resample(rule, closed='left', label='left', convention='start', kind=None, level=None, origin='start_day', offset=None)</code>	Return Resampler object to aggregate on. Use rule to specify DateOffset, Timedelta, or offset alias string.
<code>.transform(func)</code>	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.
<code>.groupby(by=None, level=None, sort=True, group_keys=True, observed=False, dropna=True)</code>	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.

Method	Description
.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

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.

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.

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.

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.

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)
... )

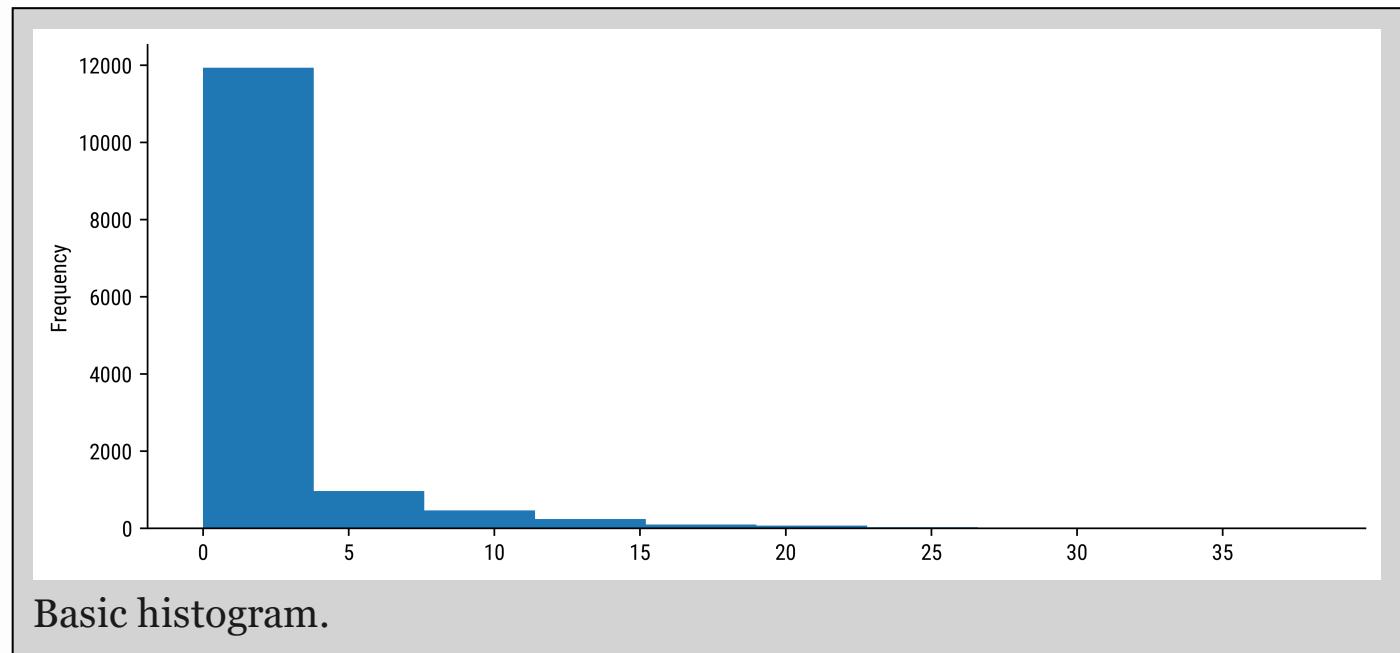
>>> snow
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
...
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: double[pyarrow]
```

The following plot attributes are available for plotting a series: `bar`, `barh`, `box`, `hist`, `kde`, and `line`. The next sections will explore them.

Histograms

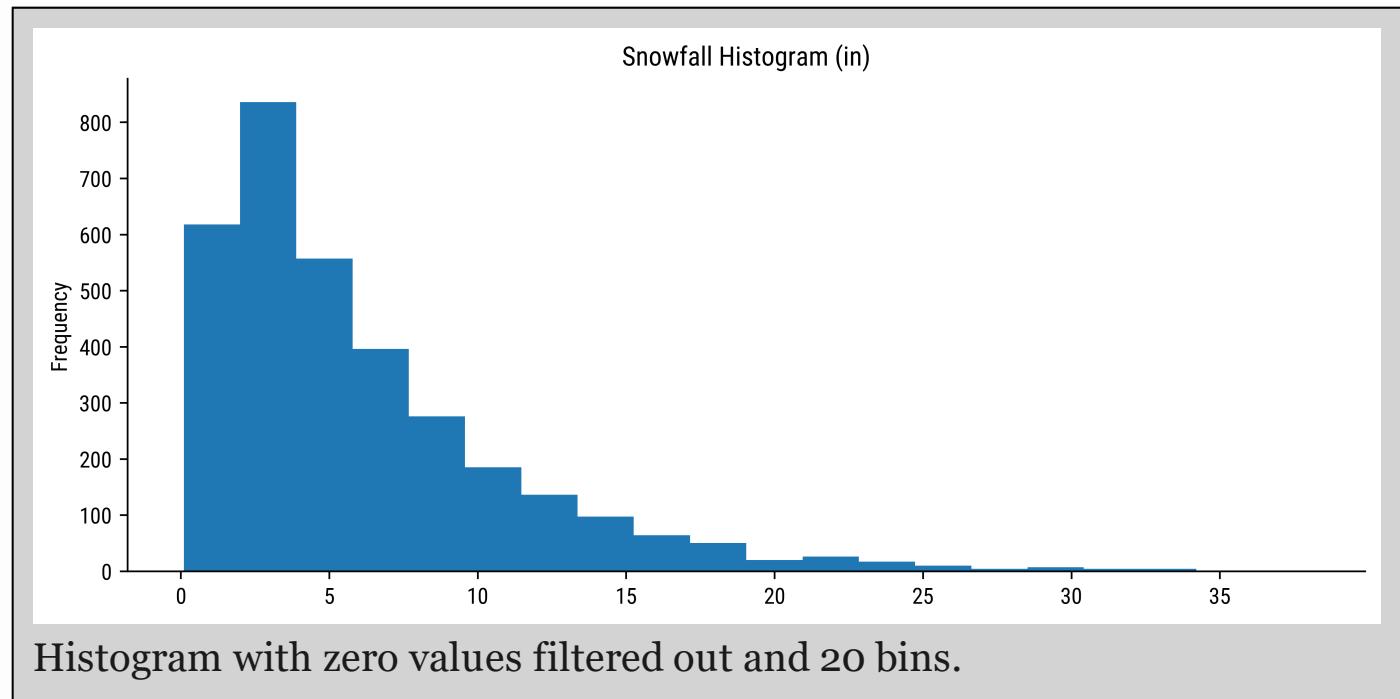
If you have continuous numeric data, plotting a histogram can give you insight into how the data is distributed:

```
snow.plot.hist()
```



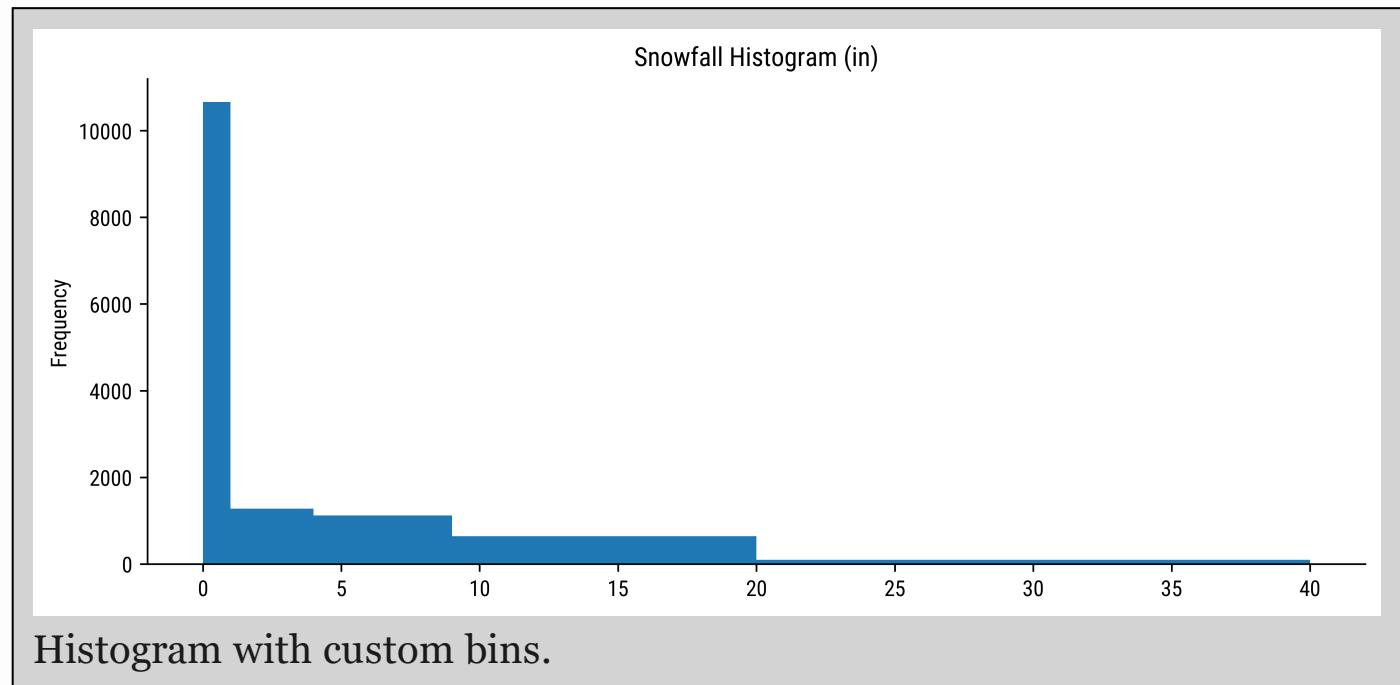
The snow data is heavily skewed. We might want to drop the zero entries and try again. We will also change the number of bins:

```
snow[snow>0].plot.hist(bins=20, title='Snowfall Histogram (in)')
```



Histogram with zero values filtered out and 20 bins.

You can also specify the bins directly with a list of numbers. The following code creates bins from 0, 1-4, 5-9, 10-20, and 20+:

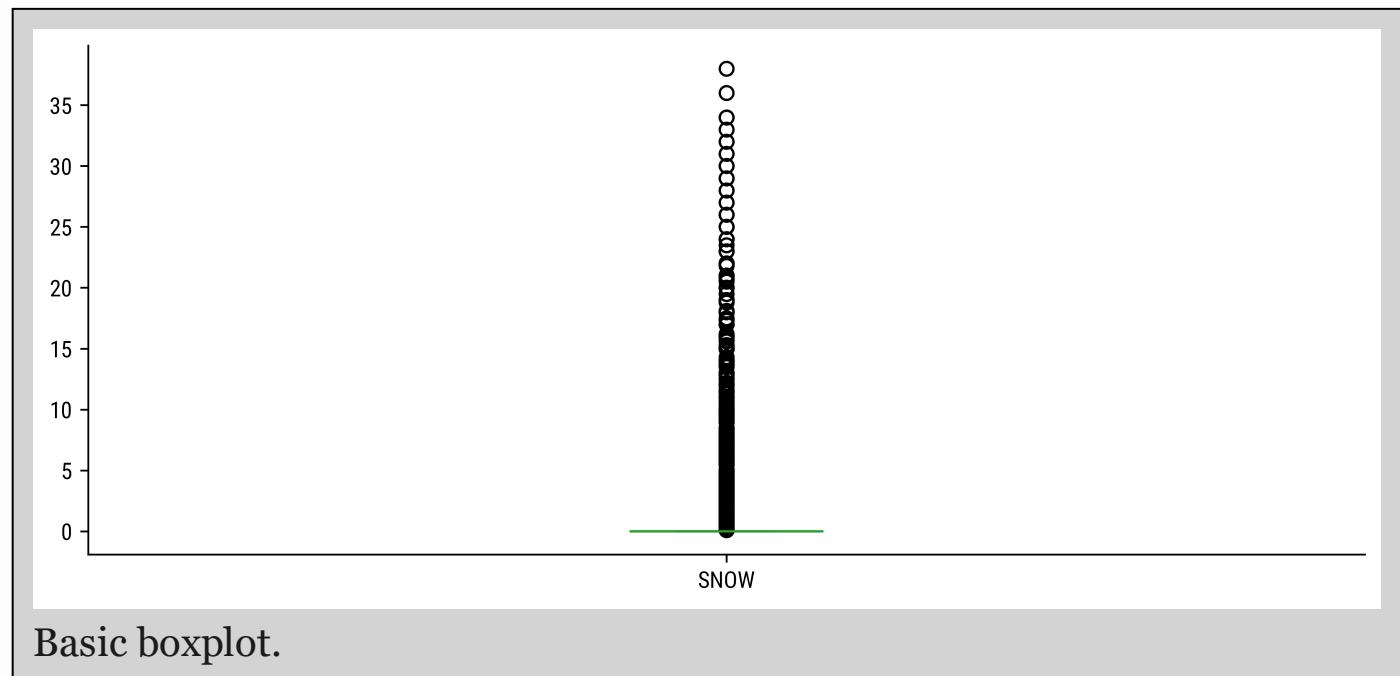


Histogram with custom bins.

Box Plot

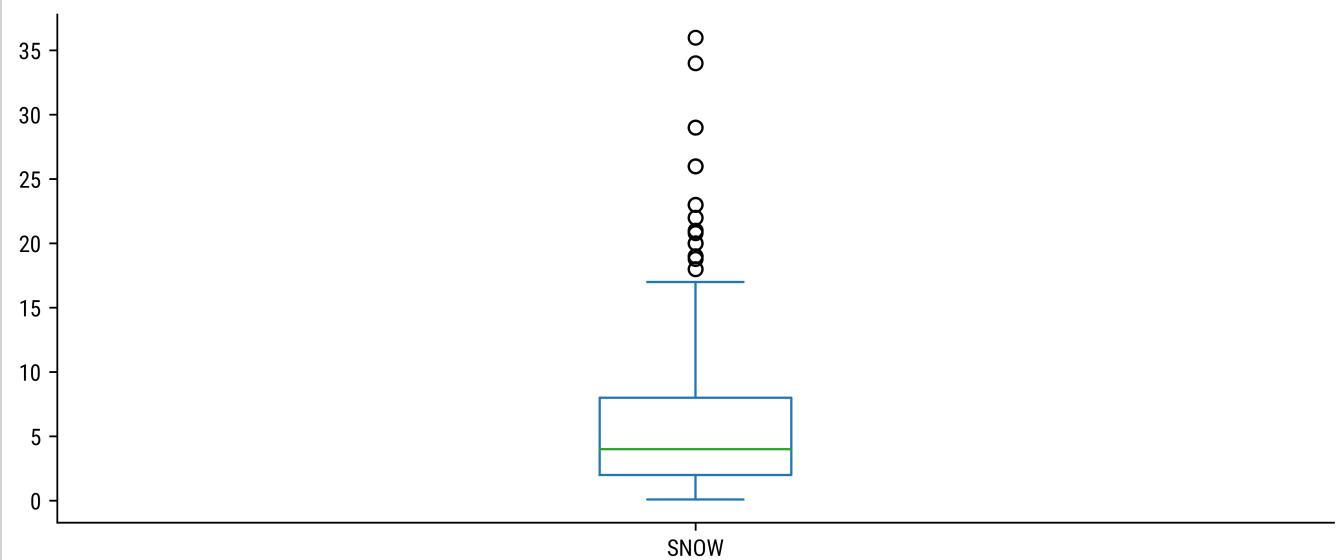
You can also create a boxplot to view the distribution of the data. In this example, it does not look much like a box. This is because most of the time, it doesn't snow, so the plot shows that any time it snows is considered an outlier:

```
>>> 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)]
...     .plot.box()
... )
```

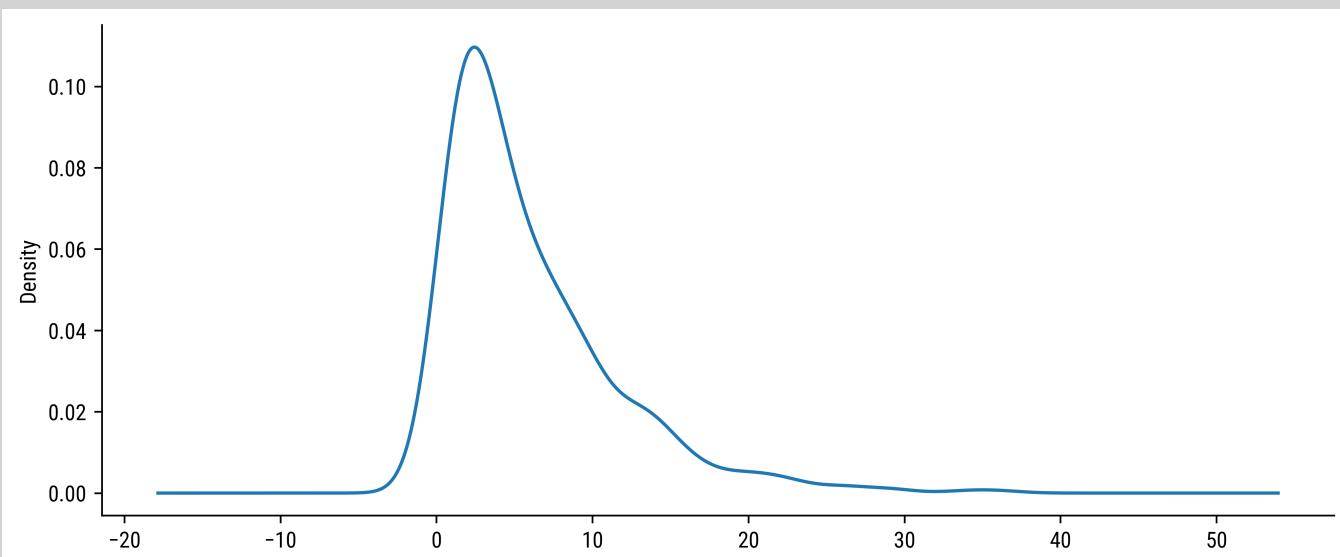


A better basic boxplot with snowfall levels for each January.

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()  
... )
```

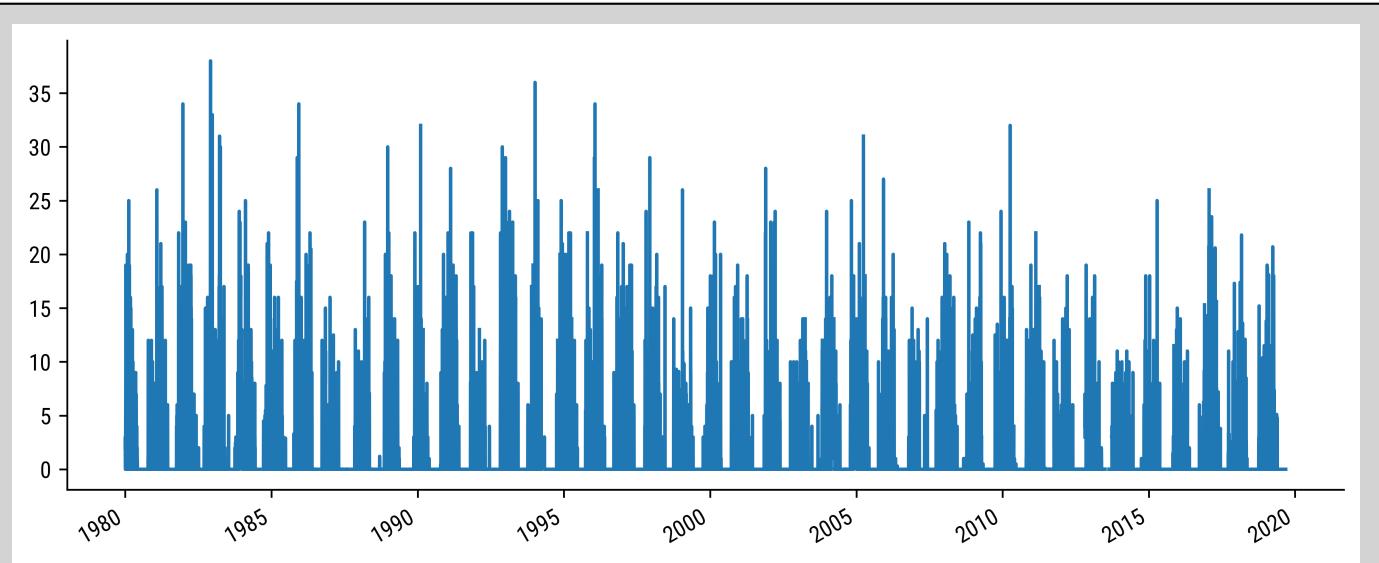


A basic kernel density estimate plot.

Line Plots

For numeric time series values, we can plot a line plot:

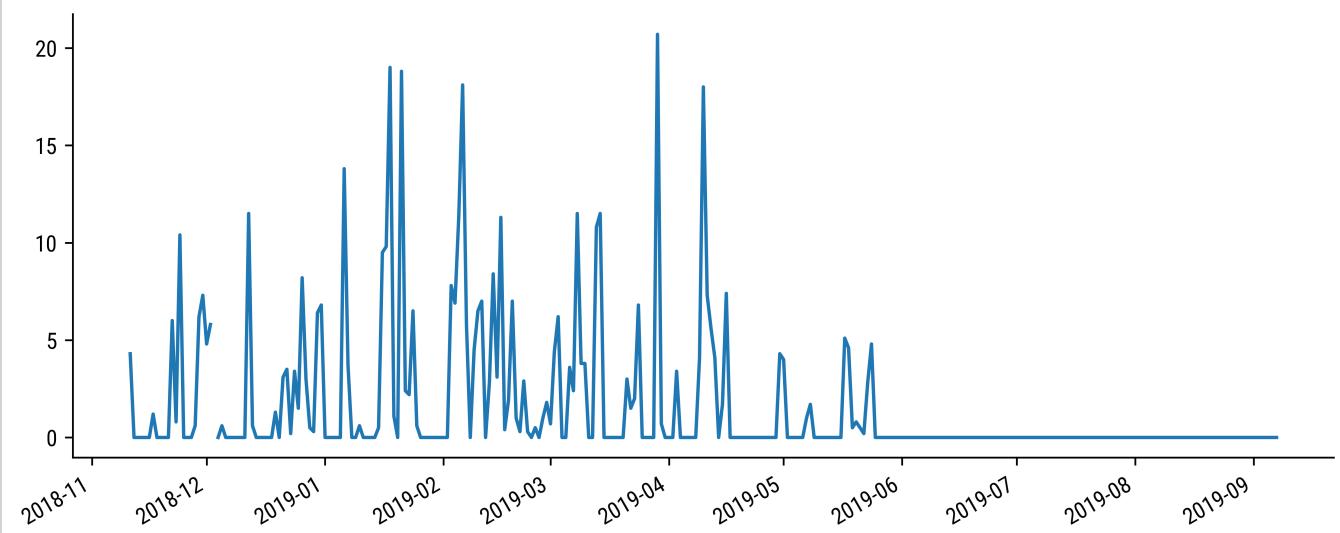
```
>>> snow.plot.line()
```



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()  
... )
```

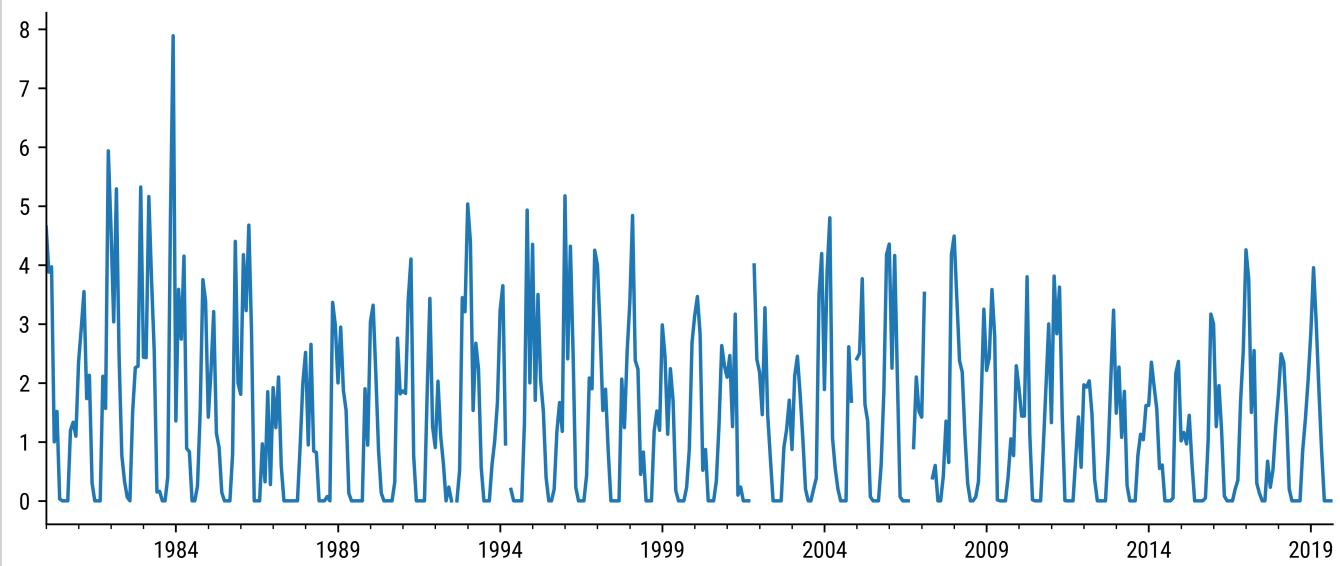
Note that by writing the code as above, I can easily comment out the line `.plot.line()` and inspect the series that will be plotted.



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('M')` with the '`ME`' offset alias and the `.mean` aggregation method:

```
>>> (snow
...     .resample('M')
...     .mean()
...     .plot.line()
... )
```

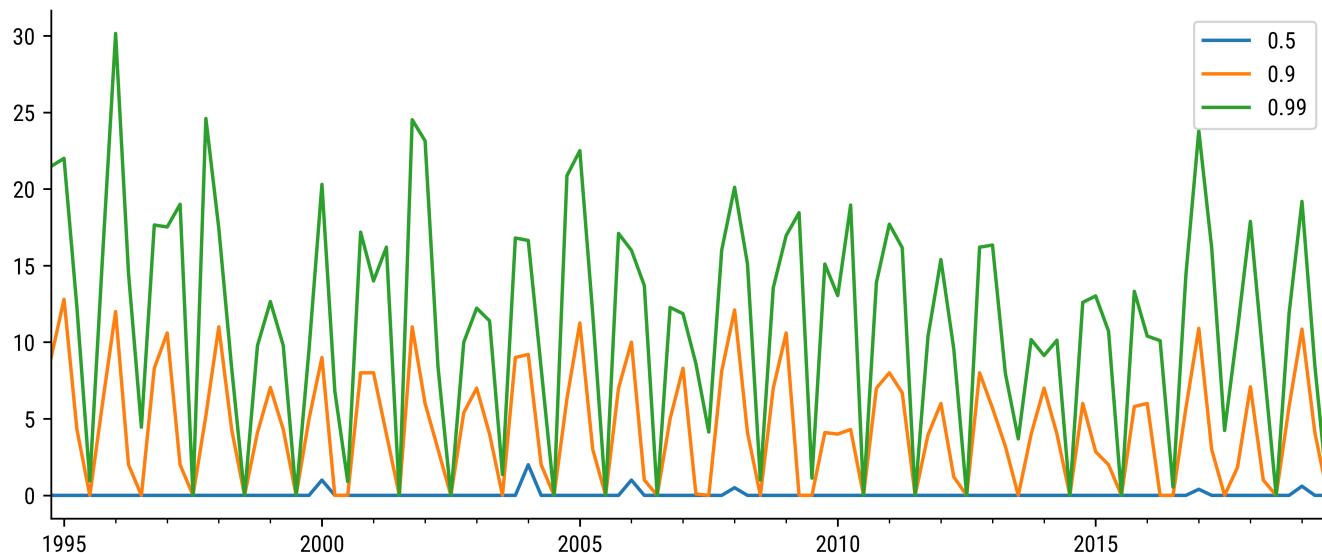


Resampled line plot.

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()  
...     .tail(100)  
...     .plot.line()  
... )
```



Resampled line plot from dataframe.

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]

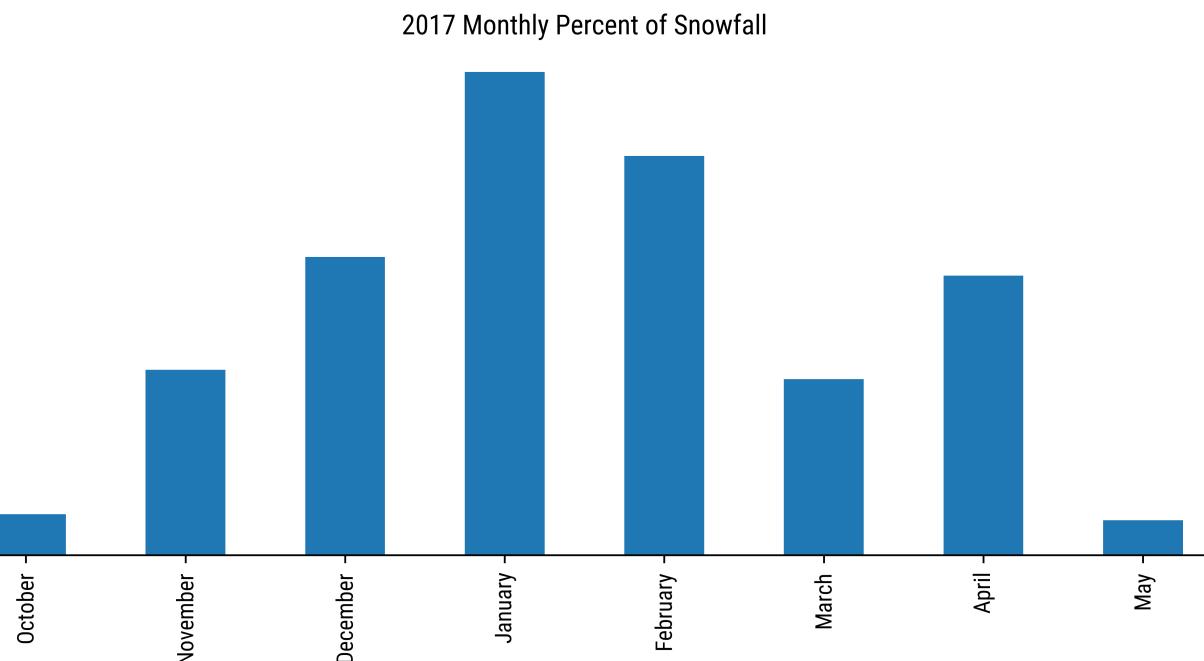
```

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')
... )

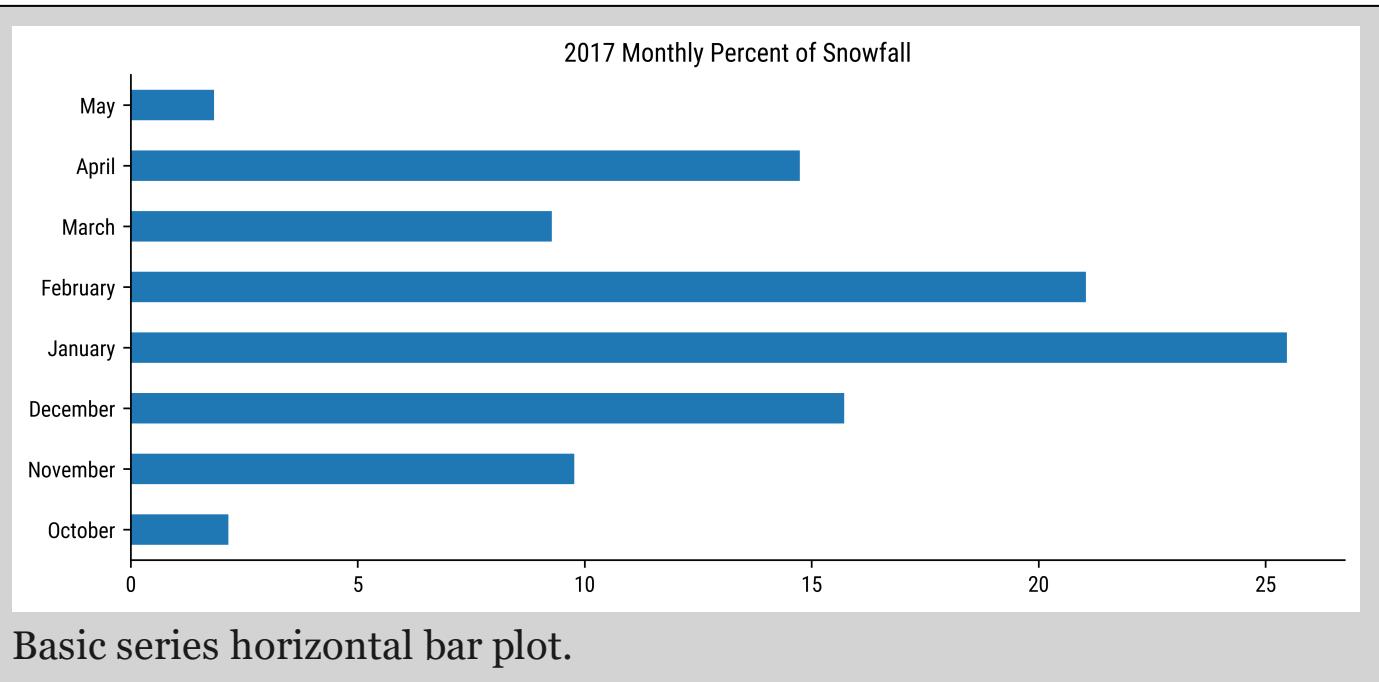
```

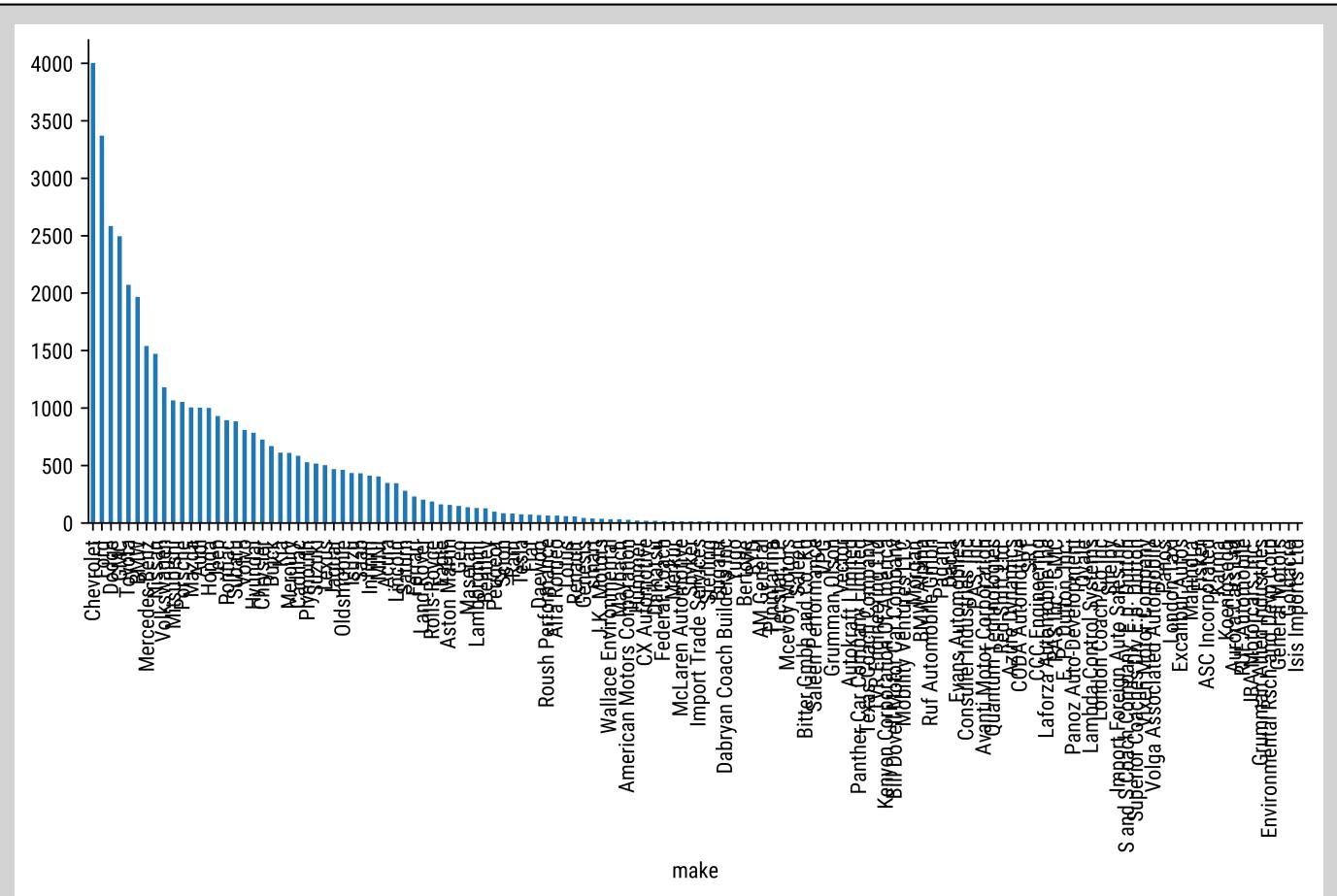


Basic series bar plot.

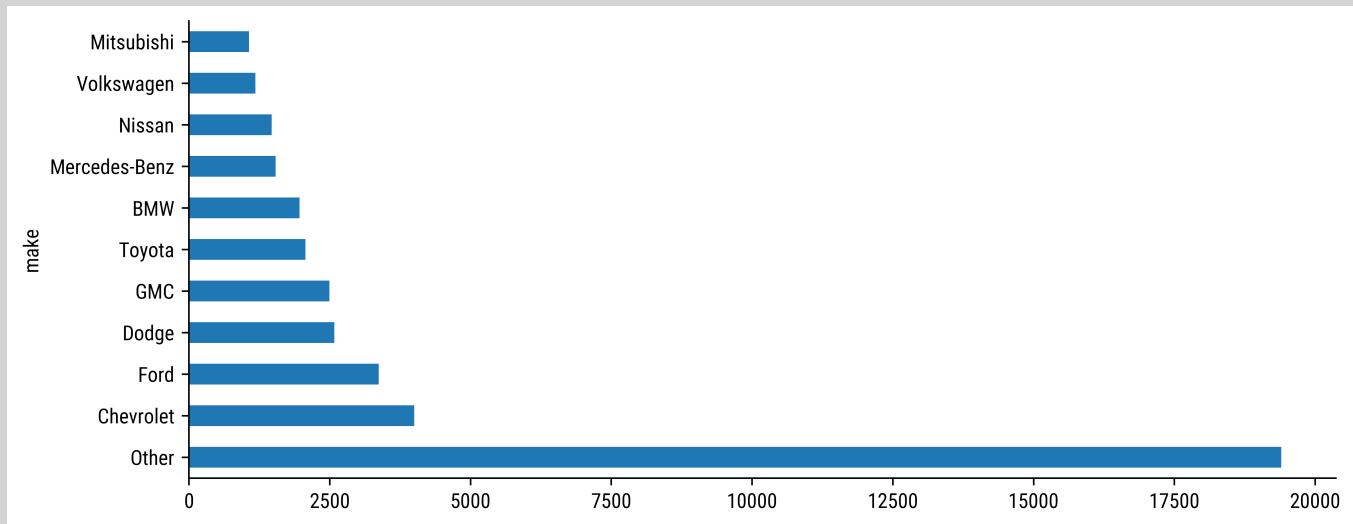
You can create a horizontal bar plot with the `.barh` method:

```
>>> (season2017
...     .resample('ME')
...     .sum()
...     .div(season2017.sum())
...     .mul(100)
...     .rename(lambda idx: idx.month_name())
...     .plot.barh(title='2017 Monthly Percent of Snowfall')
... )
```





Crowded bar plot.



Grouping long-tail members together for legible 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
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()
... )
```

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`.

Series Plotting Methods

Method	Description
<code>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)</code>	Common plot parameters. Use <code>ax</code> to use existing Matplotlib axes, <code>style</code> for color and marker style (see <code>matplotlib.marker</code>), <code>_ticks</code> to specify tick locations, <code>_lim</code> to specify tick limits, <code>_label</code> to specify x/y label (default to index/column name), <code>rot</code> to rotate labels, <code>fontsize</code> for tick label size, <code>colormap</code> for coloring, <code>position</code> , <code>table</code> to create table with data. Additional arguments are passed to <code>plt.plot</code>
<code>s.plot.bar(position=.5, color=None)</code>	Create a bar plot. Use <code>position</code> to specify label alignment (0-left, 1-right). Use <code>color</code> (string, list) to specify line color.
<code>s.plot.bahr(x=None, y=None, color=None)</code>	Create a horizontal bar plot. Use <code>position</code> to specify label alignment (0-left, 1-right). Use <code>color</code> (string, list) to specify line color.
<code>s.plot.hist(bins=10)</code>	Create a histogram. Use <code>bins</code> to change the number of bins. Can pass in a list of bin edges.
<code>s.plot.box()</code>	Create a boxplot.

Method	Description
<code>s.plot.kde(bw_method='scott', ind=None)</code>	Create a Kernel Density Estimate plot. Use <code>bw_method</code> to calculate estimator bandwidth (see <code>scipy.stats.gaussian_kde</code>). Use <code>ind</code> to specify evaluation points for PDF estimation (NumPy array of points, or integer with equally spaced points).
<code>s.plot.line(color=None)</code>	Create a line plot. Use <code>color</code> to specify line color.

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.

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.

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.

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/mattarrison/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]
```

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)
```

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.

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]
```

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']
```

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]

```

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']

```

Category Attributes and Methods

Method	Description
.astype(dtype)	Return a series converted to categories. Set <code>dtype</code> to 'category' for an unordered category, <code>CategoricalDType</code> for an ordered category.
<code>pd.CategoricalDType(categories, ordered=False)</code>	Create categorical type. Set <code>categories</code> to a list of categories.

Method	Description
<code>pd.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates='raise', ordered=True)</code>	Bin values from <code>x</code> (a series). If <code>bins</code> is an integer, use equal-width bins. If <code>bins</code> is a list of numbers (defining minimum and maximum positions), use those for the edges. <code>right</code> defines whether the right edge is open or closed. <code>labels</code> allows us to specify bin names. Out of bounds values will be missing.
<code>pd.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')</code>	Bin values from <code>x</code> (a series) into <code>q</code> 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.
<code>.cat.add_categories(new_categories)</code>	Return a series with the new categories added. The new values are added at the end (highest) if it is ordinal.
<code>.cat.as_ordered()</code>	Convert categorical series to an ordered series. Use <code>.reorder_categories</code> or <code>categoricalDtype</code> to specify the order.
<code>.cat.categories</code>	Property with the index of categories.
<code>.cat.codes</code>	Property with a series with category codes (index into a category).
<code>.cat.ordered</code>	Boolean property if series is ordered.
<code>.cat.remove_categories(removals)</code>	Return a series with the categories removed (replace with <code>NaN</code>).
<code>.cat.remove_unused_categories()</code>	Return a series with the categories removed that are being used.
<code>.cat.rename_categories(new_categories)</code>	Return a series with the categories replaced by a list (with new values) or a dictionary (mapping old to new values).
<code>.cat.reorder_categories(new_categories)</code>	Return a series with the categories replaced by a list.

Method	Description
<code>.cat.set_categories(new_categories, ordered=False, rename=False)</code>	Return a series with the categories replaced by a list.

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.

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?

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`.

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 OLTP¹ 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.

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. 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']
```

Dataframes

Using the pandas `DataFrame` object, the previous data structure could be created like this:

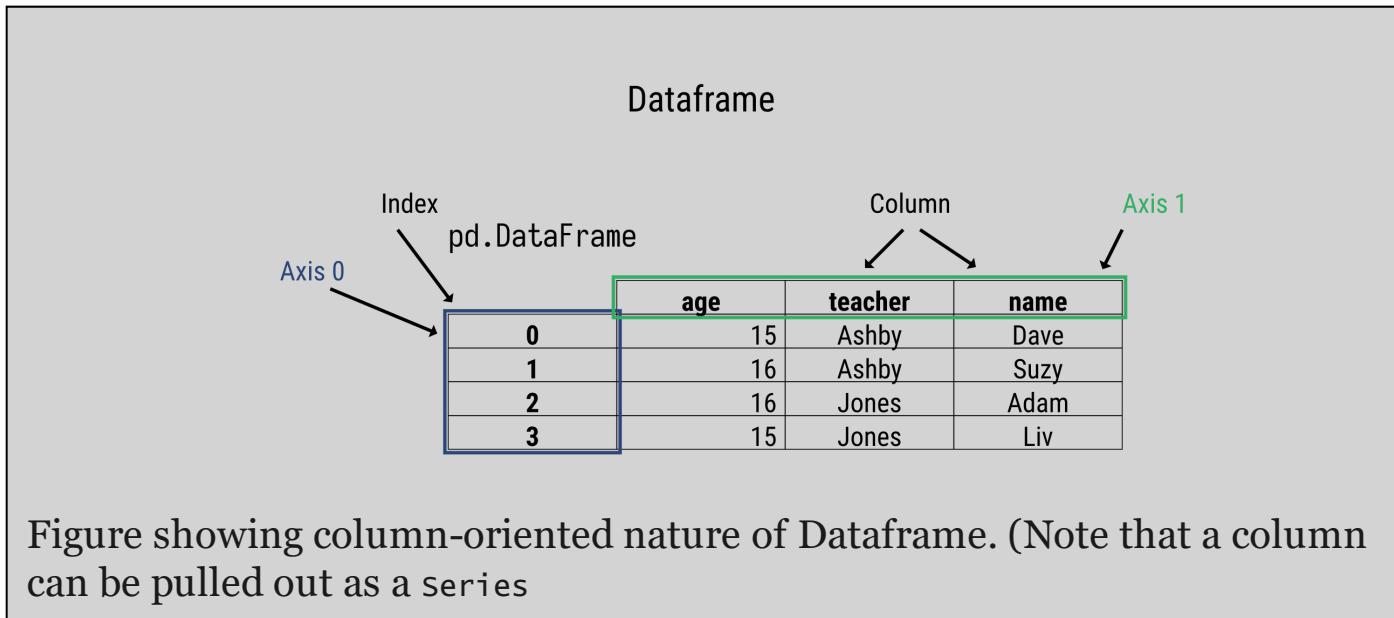
```
>>> import pandas as pd
>>> df = pd.DataFrame({
...     'growth':[.5, .7, 1.2],
...     'Name':['Paul', 'George', 'Ringo'] })

>>> print(df)
```

```

growth      Name
0    0.5    Paul
1    0.7  George
2    1.2  Ringo

```



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.

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:

types for `pd.read_csv` `dtypes` attribute

Type	Description
<code>float64[pyarrow]</code>	Floating point. Can specify different sizes, i.e., <code>float16</code> , <code>float32</code> , or <code>float64</code> .
<code>int64[pyarrow]</code>	Integer number. Can put <code>u</code> in front for <code>unsigned</code> . Can specify size, i.e., <code>int8</code> , <code>int16</code> , <code>int32</code> , or <code>int64</code> .
<code>datetime64[ns]</code>	Datetime number
<code>datetime64[ns, tz]</code>	Datetime number with timezone
<code>timedelta[ns]</code>	A difference between datetimes
<code>category</code>	Used to specify categorical columns
<code>object</code>	Used for other columns such as strings or Python objects
<code>pd.ArrowDtype(pa.string())</code>	Used for text data. Supports <code><NA></code> for missing values. Reported as <code>string[pyarrow]</code> in <code>dtype</code>

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
```

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
```

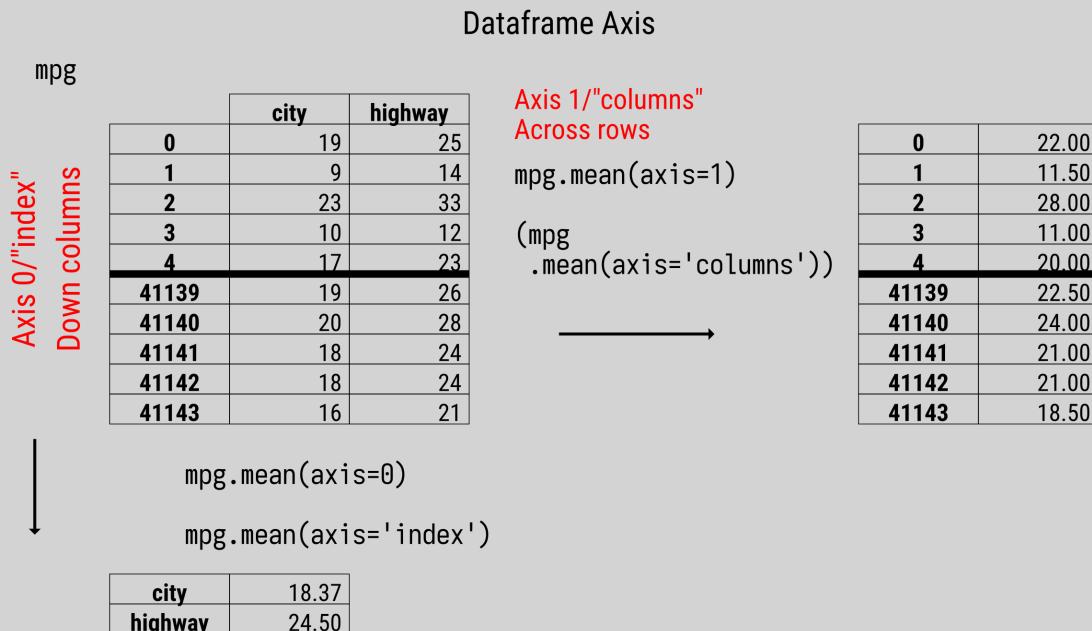


Figure showing the relation between axis 0 and axis 1. Note that when an operation is applied along axis 0, it is applied down the column. Likewise, operations along axis 1 operate across the values in the row.

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.

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.

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.
-

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.

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.

Getting the Data

Wikipedia has the data¹ 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:

```
(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/mattarrison/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-Rep  ...   7   5   5
4     4      James Madison  Democratic-Rep  ...  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.

Create a tweak_ Function

snow

	Obs Date	Precip.	Snowfall	T. Obs
0	1980/01/01	0.1	1	25
1	1980/01/02	T	0	18

String column

String column (has "T")

The lambda in the .assign method gets the intermediate dataframe!

```
def tweak_snow(df_):
    return (df_
        .rename(columns=lambda c: c.lower().replace(' ', '_').replace('.', '')) 
        .assign(obs_date=lambda df2: pd.to_datetime(df2.obs_date),
               precip=df_[['Precip.'].replace('T', 0).astype(float)]))
```

	obs_date	precip	snowfall	t_obs
0	1980-01-01	0.10	1	25
1	1980-01-02	0.00	0	18

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.

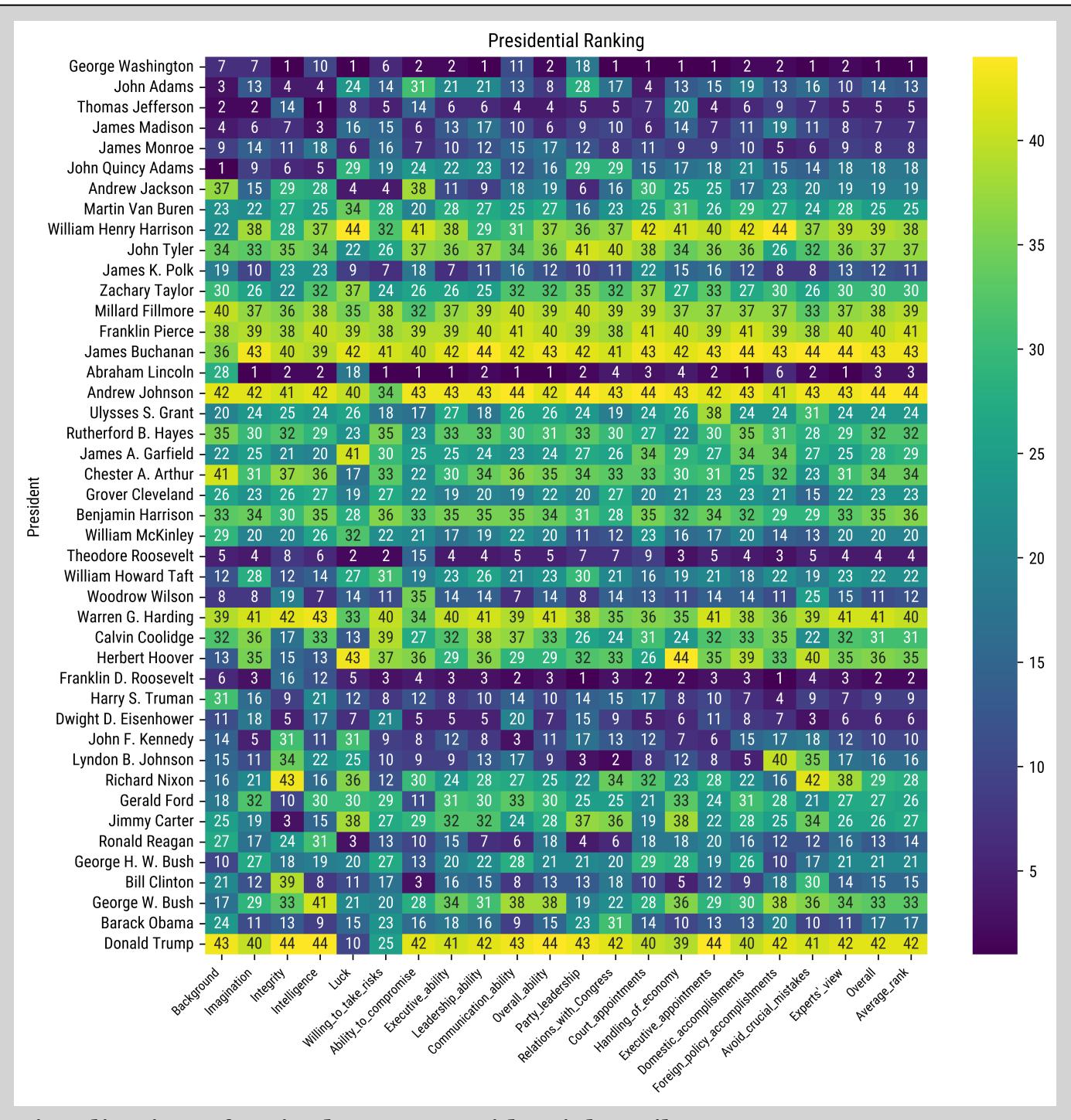
Another advantage of writing this as a function is that you can pull this out and leverage it in production code.

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 ² (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>
```



Visualization of United States presidential attributes.

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.

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-Rep...  ...      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]
```

Dataframe viewing Methods

Method	Description
<code>.head(n=5)</code>	Return a dataframe with the first <code>n</code> values.
<code>.tail(n=5)</code>	Return a dataframe with the last <code>n</code> values.
<code>s.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)</code>	Return a dataframe with <code>n</code> random entries. Can also specify a fraction with <code>frac</code> (if <code>frac > 1</code> , may specify <code>replace=True</code>).

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.

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.
-

1.
https://en.wikipedia.org/wiki/Historical_rankings_of_presidents_of_the_United_States
2. <https://github.com/pandas-dev/pandas/issues/56270>

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)
```

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!

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          ...         ...         ...         ...    ...
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
```

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:

$$x_{\text{normalized}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

The formula for standardization is:

$$x_{\text{standardized}} = \frac{x - \mu}{\sigma}$$

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           ...          ...          ...          ...
41           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  ...
2             2              12           3  ...
3             1              1           13  ...
4             3              5           6  ...
..            ..            ...
41            20             11           38  ...
42            16             28           32  ...
43            23             10           12  ...
44            42             39           43  ...
                                         ...
                                         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 - nums.min()) / (nums.max() - 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   ...          ...          ...          ...          ...
42   0.476190    0.261905    0.883721    ...    0.302326
43   0.380952    0.666667    0.744186    ...    0.767442
44   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]
```

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
Background	Intelligence	...	\
Imagination	127.162791	...	
Integrity	136.895349	...	
Intelligence	116.023256	...	
Luck	165.000000	...	
...	56.302326	...	
Foreign_policy_accomplishments	
Avoid_crucial_mistakes	121.232558	...	
Experts'_view	102.302326	...	
Overall	133.209302	...	
Average_rank	133.813953	...	
...	134.279070	...	
Background	Avoid_crucial_mistakes	...	\
Imagination	82.511628		
Integrity	121.616279		
Intelligence	123.023256		
Luck	102.302326		
...	115.162791		
Foreign_policy_accomplishments	
Avoid_crucial_mistakes	140.837209		
Experts'_view	165.000000		
Overall	148.302326		
Average_rank	144.186047		
...	144.069767		
Background	Experts'_view	Overall	\
Imagination	104.697674	111.651163	
Integrity	148.616279	151.523256	
Intelligence	119.418605	113.976744	
Luck	133.209302	133.813953	
...	113.441860	113.372093	
Foreign_policy_accomplishments	
Avoid_crucial_mistakes	143.279070	147.116279	
Experts'_view	148.302326	144.186047	
Overall	165.000000	162.372093	
Average_rank	162.372093	165.000000	
...	162.116279	164.837209	
Background	Average_rank		
Imagination	113.116279		
Integrity	151.267442		
Intelligence	114.116279		
Luck	134.279070		
...	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,
       1.11111111e-05, 1.11111111e-05, 1.11111111e-05, 1.11111111e-05,
       1.11111111e-05, 1.11111111e-05])

```

```
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
```

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.

Method	Description
<code>np.cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None)</code>	Calculate covariance matrix.

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.

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?

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.

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
```

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.

The .describe 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.describe()

- Summary statistics for numeric columns
- Use `include='all'` to show other types
- Count is non-NA values

↓

	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

The .describe method provides the count of non-missing values, the mean, standard deviation, minimum, maximum, and quartiles.

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.

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]
```

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

```

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]
```

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!

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 [2](#) 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()))
1.04 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)

```

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 solutions 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!

Dataframe Looping Methods

Method	Description
<code>.items()</code>	Iterate over a tuple of column name and series.
<code>.itertuples(index=True, name="Pandas")</code>	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).
<code>.sum(axis=0, skipna=True, level=None, numeric_only=None, min_count=0)</code>	Return sum over axis. Default of empty sequence is 0. Set min_count=1 to return nan.
<code>.min(axis=0, skipna=True, level=None, numeric_only=None)</code>	Return minimum over the axis.
<code>.max(axis=0, skipna=True, level=None, numeric_only=None)</code>	Return maximum over the axis.
<code>.idxmin(axis=0, skipna=True)</code>	Return the index of the first minimum value over the axis.
<code>.idxmax(axis=0, skipna=True)</code>	Return the index of the first maximum value over the axis.
<code>.agg(func=None, axis=0, *args, **kwargs)</code>	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.
<code>.describe(percentiles=[.25, .5, .75], include=None, exclude=None, datetime_is_numeric=False)</code>	Return summary statistics for dataframe.

Method	Description
<code>.apply(func=None, axis=0, raw=False, result_type=None, *args, **kwargs)</code>	Apply <code>func</code> over the axis. If <code>func</code> returns a sequence, then return a dataframe. If <code>func</code> returns a scalar, then return a series.
<code>s.case_when(caselist)</code>	Use <code>caselist</code> (list of tuples of (boolean array, result)), to simulate if then statements

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.

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.
-

1. <https://github.com/pandas-dev/pandas/issues/47950>

2. <https://github.com/cython/cython/issues/5811>

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.

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')),
            )
    )
)

```

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.

Dataframe Methods from this Chapter

Method	Description
<code>.astype(dtype, copy=True, errors='raise')</code>	Cast dataframe into <code>dtype</code> . (More common to use this on series.)
<code>.assign(**kwargs)</code>	Return a new dataframe with updated or new columns. <code>kwargs</code> 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 <code>kwargs</code> if you use a function.
<code>.memory_usage(index=True, deep=False)</code>	Return a series with the memory usage of each column in bytes. By default includes <code>index</code> . Use <code>deep=True</code> to show how much space object columns consume.
<code>.info(verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None)</code>	Print summary of dataframe to stdout. Use <code>memory_usage='deep'</code> to show object column memory usage.

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.

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?

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.

Loading the Data

We will look at a dataset of Python users from JetBrains¹.

Let’s load the data:

```
>>> import pandas as pd
>>> import numpy as np
>>> url = 'https://github.com/mattarrison/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.

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('.')[0: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]`.

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=λ 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.

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
...     [unq_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
...     [unq_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 CatBoost² 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 comprehension 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'])}
```

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

	learn	total	remaining		
0:	2.9758568	68.5ms	1.3s		
1:	2.8841040	80.5ms	724ms		
2:	2.8443484	92.7ms	525ms		
3:	2.8105584	103ms	414ms		
4:	2.7922983	112ms	335ms		
5:	2.7803329	120ms	280ms		
6:	2.7756137	129ms	239ms		
7:	2.7706510	136ms	205ms		
8:	2.7571563	145ms	177ms		
9:	2.7564631	153ms	153ms		
10:	2.7503591	161ms	132ms		
11:	2.7494745	170ms	113ms		
12:	2.7481258	178ms	95.6ms		
13:	2.7477180	185ms	79.3ms		
14:	2.7449738	193ms	64.4ms		
15:	2.7409940	202ms	50.6ms		
16:	2.7408640	212ms	37.4ms		
17:	2.7365108	221ms	24.6ms		
18:	2.7346780	229ms	12.1ms		
19:	2.7287662	237ms	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:

```

...
)

>>> 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.

Dataframe Chapter Methods

Method	Description
<code>.rename(mapper=None, index=None, columns=None, axis=0, copy=True, level=None, errors='ignore')</code>	Change axis labels. Pass <code>columns</code> or <code>index</code> as a dictionary (mapping old values to new values) or a function (accepting the old value and returning the new value).

Method	Description
<code>.replace(to_replace=None, value=None, limit=None, regex=False, method='pad')</code>	Replace values from <code>to_replace</code> (string, regular expression, number, series or list of the previous, dictionary (mapping replacement if <code>value</code> is <code>None</code>), series, or <code>None</code>) with <code>value</code> . If <code>to_replace</code> is a list and there is no <code>value</code> , you can <code>bfill</code> or <code>ffill</code> with <code>method</code> .
<code>.drop(labels=None, axis=0, index=None, columns=None, level=None, errors='raise')</code>	Drop rows or columns with specified labels. Use <code>columns='age'</code> rather than <code>labels='age'</code> , <code>axis='1'</code> .
<code>.dropna(axis=0, how='any', thresh=None, subset=None)</code>	Drop rows (<code>axis=0</code>) or columns (<code>axis=1</code>) with missing values. Require a certain amount missing with <code>thresh</code> . Limit columns with <code>subset</code> .
<code>.query(expr)</code>	Evaluate <code>expr</code> to filter the dataframe. Refer to variables by prefixing them with <code>@</code> . Use backticks around column names with spaces.
<code>.assign(**kwargs)</code>	Return a new dataframe with updated or new columns. <code>kwargs</code> 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 <code>kwargs</code> if you use a function.

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.

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.
-

1. <https://www.jetbrains.com/lp/python-developers-survey-2020/>

2. <https://catboost.ai>