Effective Pandas

Patterns for Data Manipulation

Effective Pandas

Patterns for Data Manipulation

Matt Harrison

Technical Editors: Lawrence Gray, Alexandre Batisse, Edward Krueger,

hairysun.com

Contents

Forward

Python is easy to learn. You can learn the basics in a day and be productive. With only an understanding of Python, moving to pandas can be difficult or confusing. It borrows some ideas from NumPy that are not common in the wider Python ecosystem. This book is meant to aid you in mastering pandas.

I have taught Python and pandas to many people over the years, in large corporate environments, small startups, and in Python and Data Science conferences. I have seen what trips people up, and confuses them. With the correct background, an attitude of acceptance, and a deep breath, much of this confusion evaporates.

Having said this, pandas is an excellent tool. Many use it around the world to great success. I hope to empower you to do this as well.

Cheers!

Matt

Chapter 1

Introduction

I have been using Python in some professional capacity or another since the turn of the century. One of the trends that I have seen in that time is the uptake of Python for various aspects of data science—gathering data, cleaning data, analysis, machine learning, and visualization. The pandas library has seen much uptake in this area.

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

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

My description of pandas is: pandas is an in-memory analysis tool, which has SQL-like constructs, essential statistical and analytic support, as well as graphing capability. Because pandas is built on top of Cython and NumPy, it has less memory overhead and runs quicker than pure 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 with pandas, you don't need to. This work is done for you. If you use pandas and the vectorized operations, you are getting close to C-level speeds for numeric work but writing Python.

1.1 Who this book is 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 *Illustrated Guide to Python 3* that provides all the background necessary.

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

1.2 Data in this Book

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

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

1.3 Hints, Tables, and Images

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

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

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

Chapter 2

Installation

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

2.1 Anaconda

With that out of the way, let's address the installation of pandas. The easiest and least painful way to install pandas on most platforms is to use the Anaconda distribution[2]. 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 itself is freely available, though commercial support is available as well.

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

```
$ conda install pandas
```

Note

This book shows commands run from the UNIX command prompt. They are prefixed by the prompt `$`. Unless otherwise noted, these commands will run on the Windows command prompt as well. 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__
'1.3.2'
```

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, `'1.3.2'` does not have the prompt in front of it. The book also includes the secondary Python prompt, `...` for code that is longer than a single line.

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

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

2.2 Pip

If you aren't using Anaconda, I recommend you use pip[3] to install pandas. The pandas library will install on Windows, Mac, and Linux via pip.

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

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

Using `virtualenv`[4] will alleviate the need for superuser access during installation. Because `virtualenv` uses pip, it can download and install newer releases of pandas 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
```

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

```
$ source pandas-env/bin/activate
(pandas-env)$ python
>>> import pandas
>>> pandas.__version__
'1.3.2'
```

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
```

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__
'1.3.2'
```

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

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

[4]http://www.virtualenv.org

Figure 2.1: Jupyter home page.

2.3 Jupyter Overview

I recommend you use Jupyter (or a program that connects to it) as a data exploration tool. I use Jupyter classic, though there are other options: JupyterLab, connecting to Jupyter via PyCharm, VSCode, Emacs, as well as Google Colab. Jupyter classic will give you basic functionality and is included in many cloud environments.

Jupyter notebook is an environment for combining interactive coding and text in a web browser. This allows us to easily share code and narrative around that code. An example that was popular in the scientific community was the discovery of gravitational waves.[5]

The name Jupyter is a rebranding of an open-source project previously known as iPython Notebook. The rebranding was to emphasize that although the backend is written in Python, Jupyter supports various *kernels* to run other languages, including Julia (the "Ju" portion), Python ("pyt"), and R ("er"). All popular data science programming languages.

The architecture of Jupyter includes a server running various kernels. Using a *notebook* we can interact with a kernel. Typically we use a web browser to do this, but other interfaces exist, such as an emacs mode (ein), PyCharm, or VSCode.

To install Jupyter, type:

```
$ pip install notebook
```

Once Jupyter is installed, launch it with this command:

```
$ jupyter-notebook
```

Then navigate to https://localhost:8888 and you should be presented with the Jupyter home page.

Click on the dropdown button on the right that says "New" and select Python 3.

At this point, you are presented with a notebook with an empty cell. Jupyter is a *modal* environment. There are two modes, command mode and edit mode. Command mode is for creating and manipulating cells. Edit mode is for changing what is inside of a single cell.

There are many commands for both modes. If you are in command mode (and you will know that because the box around the cell is blue), you can type "h", and it will bring up a pop-up with

[5] https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html

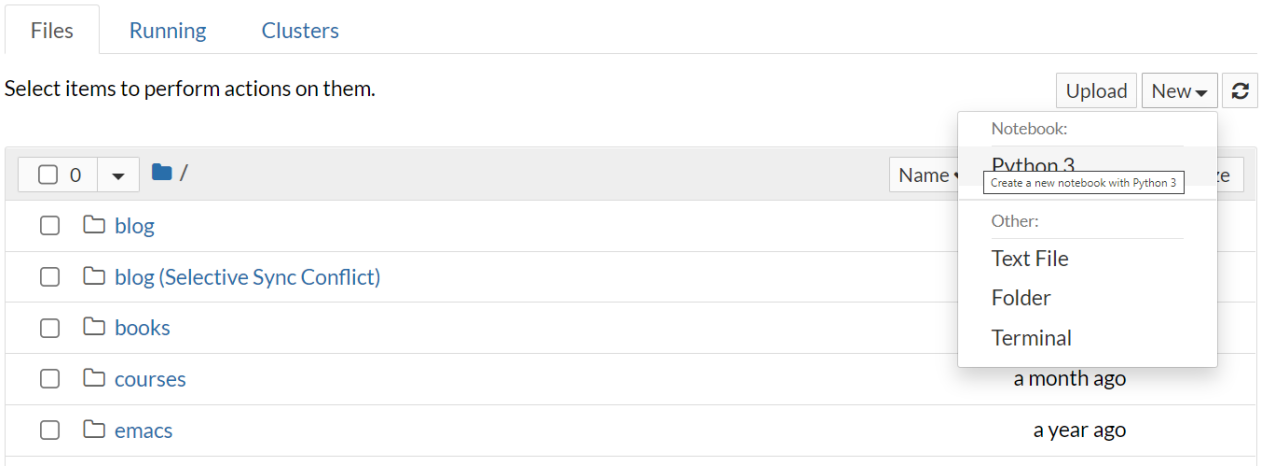


Figure 2.2: Creating a Python 3 Jupyter notebook.

the keyboard shortcuts for both command and edit mode. Don't worry about memorizing all of them. Here are the commands you will be using most of the time in command mode:

- h - Bring up help (ESC to dismiss)
- a - Create cell above
- b - Create cell below
- x - Cut cell
- c - Copy cell
- v - Paste cell below
- Enter - Go into Edit Mode
- m - Change cell type to Markdown
- y - Change cell type to code
- ii - Interrupt kernel
- 00 - Restart kernel
- Ctr-Enter - Execute cell

When you click on a cell or type Enter, you go into *edit mode*. You will see that the outline turns green if you are in edit mode. In edit mode, you have basic editing functionality. A few keys to know:

- Ctr-Enter - Run cell (execute Python code, render Markdown)
- ESC - Go back to command mode
- TAB - Tab completion
- Shift-TAB - Bring up tooltip (ESC to dismiss)

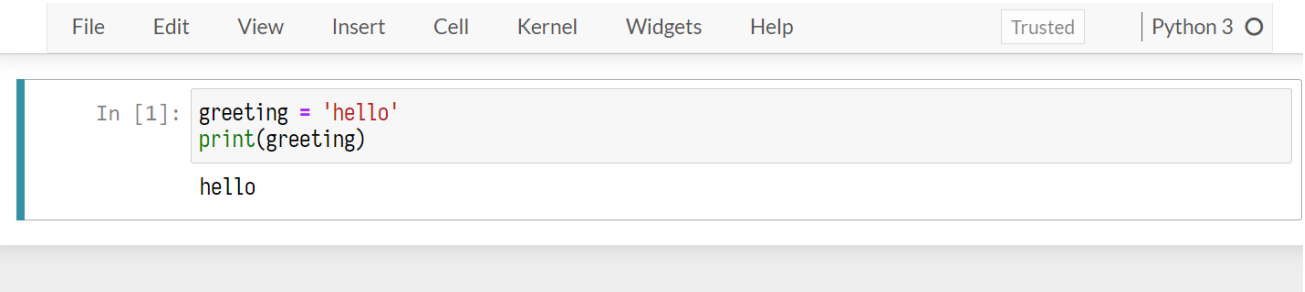


Figure 2.3: Running a cell in Jupyter with basic Python commands.

2.4 Summary

In this chapter, we saw how to set up a Python environment using Anaconda or Pip. We also introduced the Jupyter notebook. I recommend that you get comfortable with Jupyter. Not only is it free and open-source, but many large cloud providers also offer Jupyter in their environments.

2.5 Exercises

1. Install pandas on your machine (using Anaconda or pip).
2. Install Jupyter on your machine.
3. Launch Jupyter and run the following in a cell:

```
import pandas
pandas.show_versions()
```

Chapter 3

Data Structures

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 data and tabular data. This table shows their analogs in the spreadsheet and database world.

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

Figure 3.1: Different dimensions of pandas data structures

An analogy with the spreadsheet world illustrates the basic 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 are referring to a `Series`).

Diving into these core data structures a little more is helpful because a bit of understanding goes a long way towards better use of the library. 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 in an optimal way).

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 just leads to confusion.

3.1 Summary

The pandas library includes two main 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.

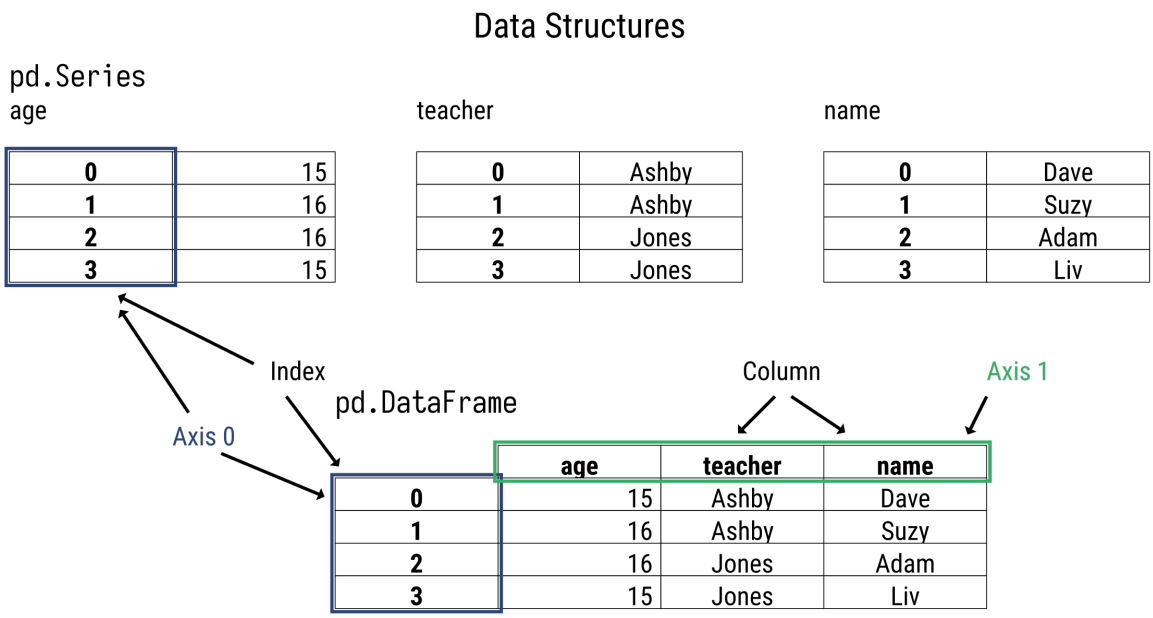


Figure 3.2: Figure showing the relation between the main data structures in pandas. Namely, that a dataframe can have on or many series.

3.2 Exercises

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

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

Chapter 4

Series Introduction

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 counts of songs artists composed. We will use this to explore the series:

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 one that follows. 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 that are 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 print the return value of the last invocation (even if the `print` statement is missing) automatically. If you desire to use the code samples found in this book, leave the interpreter prompts out.

4.1 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, dates, as well as 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 of the operations performed on a `Series` operate directly on the index or by index lookup.

4.2 The pandas `Series`

With that background in mind, let's look at how to 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
```

When the interpreter prints our series, pandas makes a 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 values of the index—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 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`, `dtype` meaning data type, and `int64` meaning 64-bit integer), but in general, the values of a `Series` can hold strings, floats,

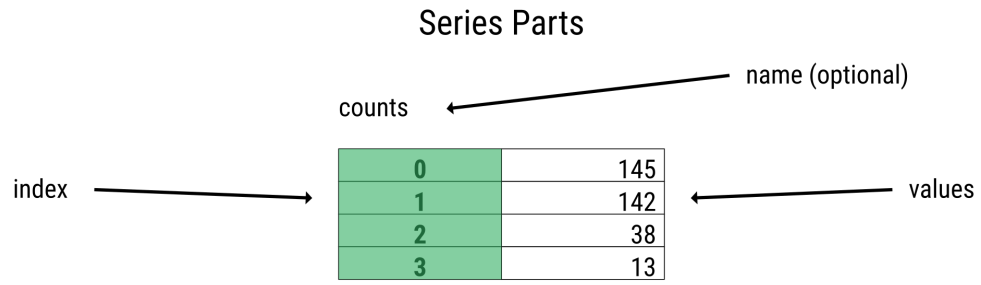


Figure 4.1: The parts of a Series.

booleans, or arbitrary Python objects. To get the best speed (and to leverage vectorized operations), the values should be of the same type, 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'])
```

Note that the dtype that we see when we print 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
```

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

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

The actual data (or values) for a series 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 instance at 0x...>
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.

The `object` data type is also used for a series with string values. In addition, 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 rather as an `int64` or `float64`, which allow 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 that you would get if the type were `datetime64[ns]`. A series with string data, on the other hand, has the type of `object`. Don't worry; we will see how to convert types later in the book.

4.3 The `NaN` value

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

Here is a series that has `NaN` in it:

```
>>> 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`, which `int64` does not. When pandas sees numeric data (`2`) as well as the `np.nan`, it coerced the `2` to a float value.

Below is an example of how pandas ignores `NaN`. The `.count` method, which counts the number of values in a series, disregards `NaN`. 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 `NaN` value at index location `Clapton`:

```
>>> nan_series.count()
1
```

You can inspect the number of entries (including missing values) with the `.size` property:

```
>>> nan_series.size
2
```

Note

If you load data from a CSV file, an empty value for an otherwise numeric column will become NaN. Later, methods such as .fillna and .dropna will explain how to deal with NaN.

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.

4.4 Optional Integer Support for NaN

The int64 type does not support missing data. Many considered that a wart of pandas. As of pandas 0.24, there is optional support for another integer type that can hold missing values denoted as <NA> below. The documentation calls this type the *nullable integer type*. When you create a series, you can pass in dtype='Int64' (note the capitalization):

```
>>> nan_series2 = pd.Series([2, None],
...     index=['Ono', 'Clapton'],
...     dtype='Int64')
>>> nan_series2
Ono           2
Clapton    <NA>
dtype: Int64
```

Operations on these series still ignore NaN or <NA>:

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

Note

You can use the .astype method to convert columns to the nullable integer type. Just use the string 'Int64' as the type:

```
>>> nan_series.astype('Int64')
Ono           2
Clapton    <NA>
dtype: Int64
```

I generally ignore 'Int64' as I tend to clean up missing data. Also, when you ingest data in pandas, most functions use 'int64' (in lowercase) by default.

4.5 Similar to NumPy

The Series object behaves similarly to a NumPy array. As shown below, both types respond to index operations:

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

They both have methods in common:

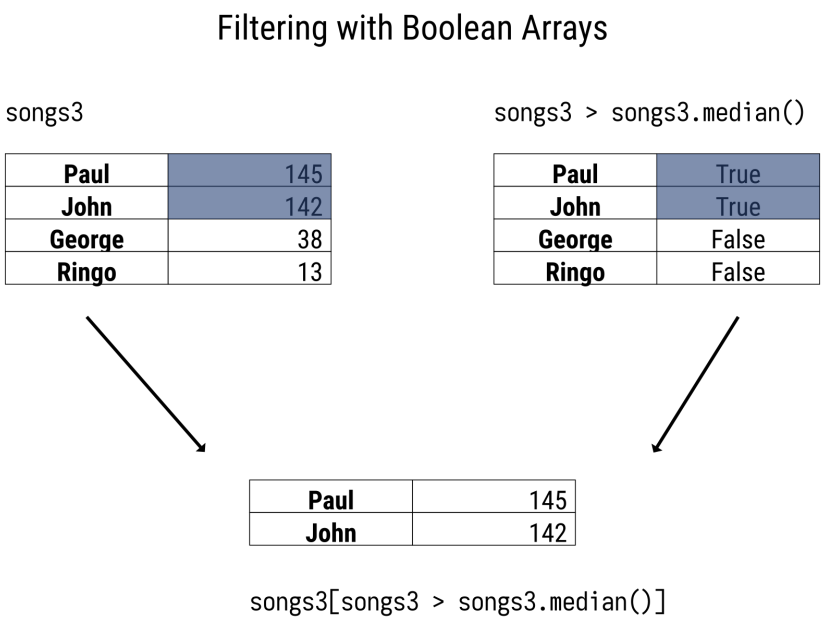


Figure 4.2: Filtering a series with a boolean array.

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

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, like 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
```

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

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

NumPy also has filtering by 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

Both NumPy and pandas have adopted the convention of using import statements in combination 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 code that I write for clients. They have the potential to 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"[6].

4.6 Categorical Data

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

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

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

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

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

```
>>> s = pd.Series(['m', 'l', 'xs', 's', 'xl'], dtype='category')
>>> s
0     m
1     l
2    xs
3     s
4    xl
dtype: category
Categories (5, object): ['l', 'm', 's', 'xl', 'xs']
```

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
```

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

```
>>> s2 = pd.Series(['m', 'l', 'xs', 's', 'xl'])
>>> size_type = pd.api.types.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 just `'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 do 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 just need to make sure that we specify all of the members of the category or pandas will throw a `ValueError`:

```
>>> s.cat.reorder_categories(['xs','s','m','l', 'xl'],
...                          ordered=True)
0     m
1     l
2    xs
3     s
4    xl
dtype: category
```

```
Categories (5, object): ['xs' < 's' < 'm' < 'l' < 'xl']
```

Note

String and datetime series have a `str` and `dt` attribute that 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
```

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

Table 4.1: Series Overview Attributes and Methods

4.7 Summary

The `Series` object is a one-dimensional data structure. It can hold numerical data, time data, strings, or arbitrary Python objects. If you are dealing with numeric data, using pandas rather than a Python list will benefit you. Pandas is faster, consumes less memory, and comes with built-in methods that are very useful to manipulate the data. 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.

4.8 Exercises

1. Using Jupyter, create a series with the temperature values for the last seven days. Filter out the values below the mean.

2. Using Jupyter, create a series with your favorite colors. Use a categorical type.

Chapter 5

Series Deep Dive

There are many operations you can do with a `Series`. In this chapter, we will introduce many of them.

We will pull data from the US Fuel Economy website[7]. This site has data on the efficiency of makes and models of cars sold in the US since 1984.

5.1 Loading the Data

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

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)
>>> city_mpg = df.city08
>>> highway_mpg = df.highway08
```

Let's look at the data:

```
>>> city_mpg
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: int64
```

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

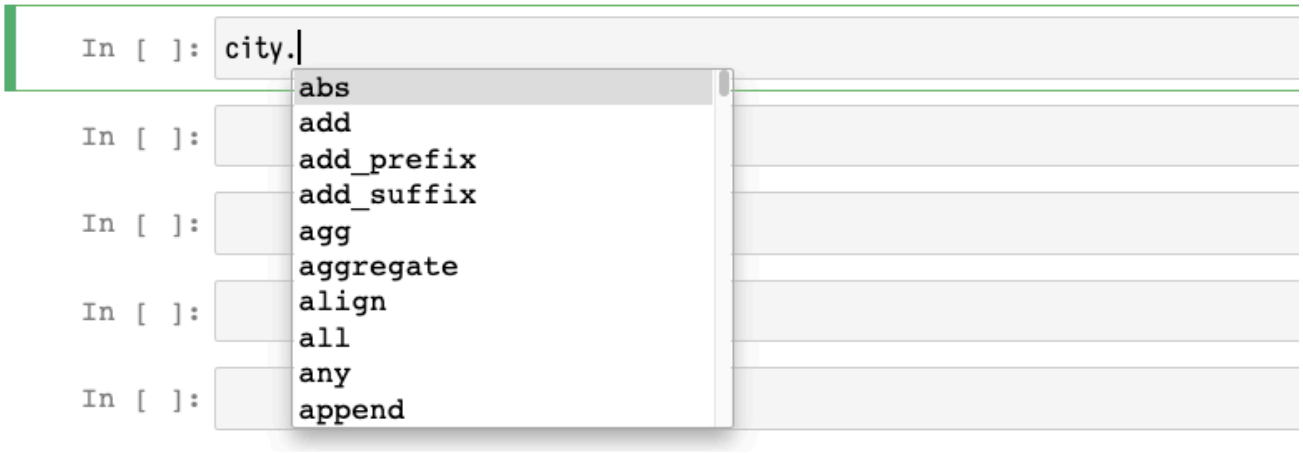


Figure 5.1: Jupyter will pop up a list of options for completions when you hit TAB following a period.

```
>>> highway_mpg
0         25
1         14
2         33
3         12
4         23
          ..
41139     26
41140     28
41141     24
41142     24
41143     21
Name: highway08, Length: 41144, dtype: int64
```

It looks like each series has around 40,000 integer entries. Because the type of this series is `int64`, we know that none of the values are missing.

5.2 Series Attributes

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

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

Wow! There are over 400 attributes on 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 it will pop up a list of completions. (Other tools are also able to do this for Python objects).

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

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

5.3 Summary

In this chapter, we introduced the notion that pandas objects have a large number of attributes and methods. Do not let this overwhelm you. You don't need to memorize all of the methods.

5.4 Exercises

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

Chapter 6

Operators (& Dunder Methods)

6.1 Introduction

This chapter, will review some of the operators and magic or *dunder methods* found in series. 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 look at how this works with a pandas series.

6.2 Dunder Methods

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

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

Under the covers, Python runs this:

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

A Python integer object that has a `.__add__` method responds to the `+` operation. Because a `Series` object has this 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
3         11.0
4         20.0
          ...
41139     22.5
41140     24.0
41141     21.0
41142     21.0
41143     18.5
Length: 41144, dtype: float64
```

Note that the type of the result is `float64`.

6.3 Index Alignment

Of note, you can apply most math operations on a series with another series, and you can also use 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`.

6.4 Broadcasting

When you perform math operations with a scalar, pandas *broadcasts* the operation to all values. In the above case, the values are added together. 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.)

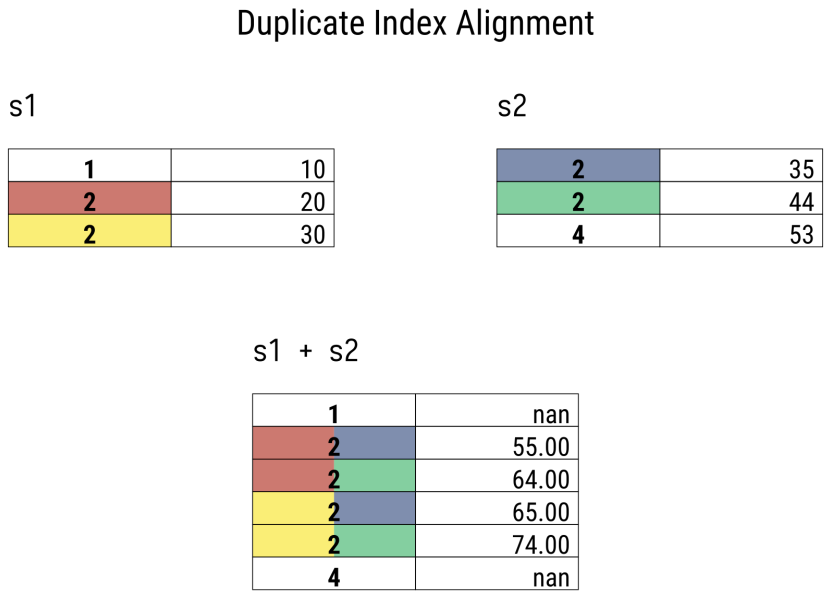


Figure 6.1: The index entries align before operating. If they are not unique, you will get a combinatoric explosion of index entries. Notice that each 2 name from s1 matches each 2 name from the index in s2.

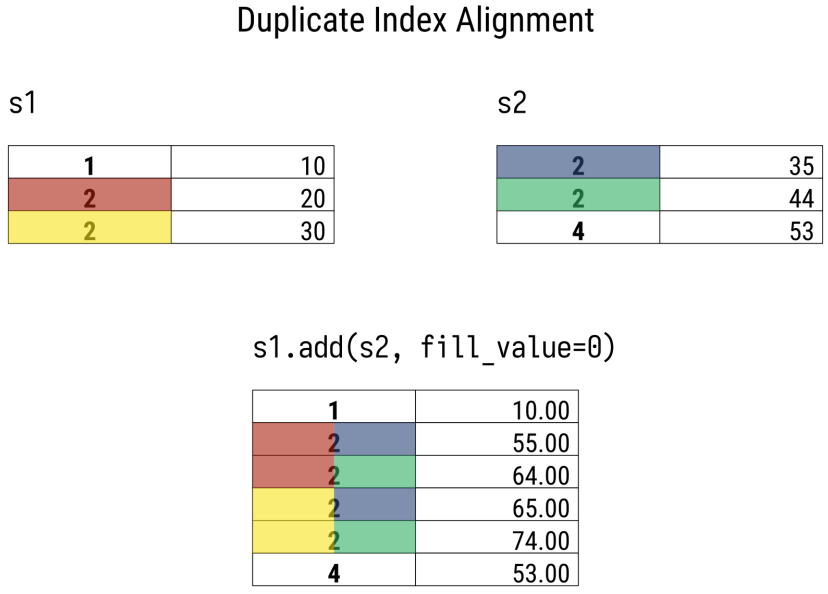


Figure 6.2: One upside to the operation methods like `.add` is that you can specify a fill value. The index entries will still align before performing the operation.

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

6.5 Iteration

Note that there is also a `.__iter__` method on a series, and you can loop over the items in a series. However, I recommend avoiding using a `for` loop with a series. That is a *code smell*, indicating that you are probably doing things the wrong way. 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 they make the code easier to understand.

6.6 Operator Methods

You might wonder why pandas also provides methods for the standard operators. 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
```

6.7 Chaining

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

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

```
>>> ((city_mpg +
...   highway_mpg)
...   / 2
... )
0         22.0
1         11.5
2         28.0
3         11.0
4         20.0
          ...
41139     22.5
41140     24.0
41141     21.0
41142     21.0
41143     18.5
Length: 41144, dtype: float64
```

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
3         11.0
4         20.0
          ...
41139     22.5
41140     24.0
41141     21.0
41142     21.0
41143     18.5
Length: 41144, dtype: float64
```

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

Method	*Operator*	*Description*
`s.add(s2)`	`s + s2`	Adds series
`s.radd(s2)`	`s2 + s`	Adds series
`s.sub(s2)`	`s - s2`	Subtracts series
`s.rsub(s2)`	`s2 - s`	Subtracts series
`s.mul(s2)` `s.multiply(s2)`	`s * s2`	Multiplies series
`s.rmul(s2)`	`s2 * s`	Multiplies series
`s.div(s2)` `s.truediv(s2)`	`s / s2`	Divides series

`s.rdiv(s2)` `s.rtruediv(s2)`	`s2 / s`	Divides series
`s.mod(s2)`	`s % s2`	Modulo of series division
`s.rmod(s2)`	`s2 % s`	Modulo of series division
`s.floordiv(s2)`	`s // s2`	Floor divides series
`s.rfloordiv(s2)`	`s2 // s`	Floor divides series
`s.pow(s2)`	`s ** s2`	Exponential power of series
`s.rpow(s2)`	`s2 ** s`	Exponential power of series
`s.eq(s2)`	`s2 == s`	Elementwise equals of series
`s.ne(s2)`	`s2 != s`	Elementwise not equals of series
`s.gt(s2)`	`s > 2`	Elementwise greater than of series
`s.ge(s2)`	`s >= 2`	Elementwise greater than or equals of series
`s.lt(s2)`	`s < 2`	Elementwise less than of series
`s.le(s2)`	`s <= 2`	Elementwise less than or equals of series
`np.invert(s)`	`~s`	Elementwise inversion of boolean series (no pandas method).
`np.logical_and(s, s2)`	`s & s2`	Elementwise logical and of boolean series (no pandas method).
`np.logical_or(s, s2)`	`s | s2`	Elementwise logical or of boolean series (no pandas method).

Table 6.1: Math Methods and Operators

6.8 Summary

Pandas series respond to most common math operations. You can use the operator directly, and 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.

6.9 Exercises

With a dataset of your choice:

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

Chapter 7

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:

- 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)

Aggregations allow you to take detailed data and collapse it to a single value. This chapter will explore how to do that on a series.

7.1 Aggregations

If we want to calculate the mean value of a series, we can use an aggregation method, `.mean`:

```
>>> 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 you can pass in a list of levels. In the latter case, the result of calling `.quantile` no longer returns a scalar but a `Series` object:

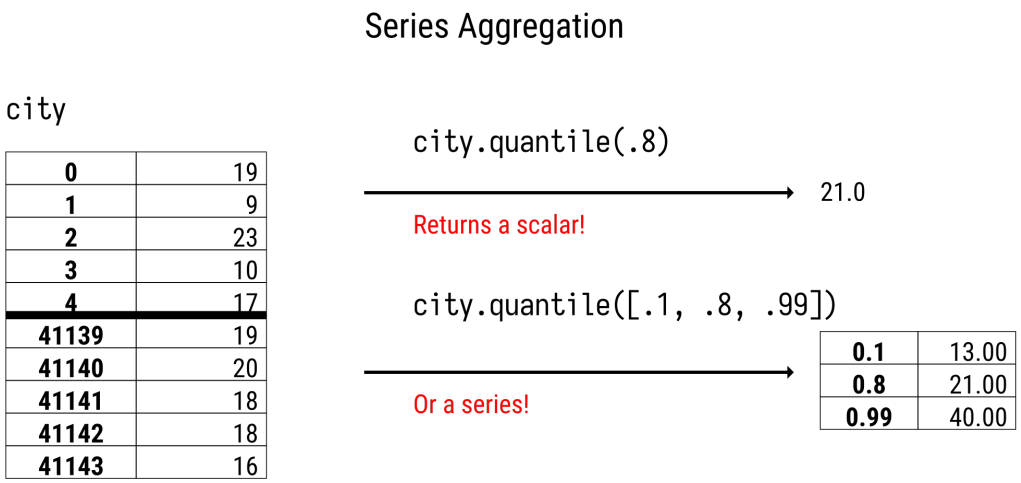


Figure 7.1: Aggregation collapses a series to a scalar value. However, the `.quantile` method also accepts a list of quantile levels and will return a `Series` object in that case.

```
>>> 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: float64
```

7.2 Count and Mean of an Attribute

Here is a neat trick in pandas to calculate aggregates. 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 percent of cars with mileage greater than 20, we can use the following code:

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

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
```

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 stands 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 count of `True` values. 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.

There are a bunch of aggregate methods found on a series, and they are listed in the table below.

7.3 .agg and Aggregation Strings

Finally, 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')
```

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 aggregations 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`:

Method	*Description*
`'all'`	Returns `True` if every value is truthy.
`'any'`	Returns `True` if any value is truthy.
`'autocorr'`	Returns Pearson correlation of series with shifted self. Can override `lag` as keyword argument(default is 1).
`'corr'`	Returns Pearson correlation of series with other series. Need to specify `other`.
`'count'`	Returns count of non-missing values.
`'cov'`	Return covariance of series with other series. Need to specify `other`.
`'dtype'`	Type of the series.
`'dtypes'`	Type of the series.
`'empty'`	`True` if no values in series.
`'hasnans'`	`True` if missing values in series.

`'idxmax'`	Returns index value of maximum value.
`'idxmin'`	Returns index value of minimum value.
`'is_monotonic'`	`True` if values always increase.
`'is_monotonic_decreasing'`	`True` if values always decrease.
`'is_monotonic_increasing'`	`True` if values always increase.
`'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.
`'sum'`	Return the sum of the series.

Table 7.1: Aggregation strings and descriptions

Below is a table of various aggregation methods and properties.

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

`s.quantile(q=.5, interpolation='linear')`	Returns 50% quantile by default. *Note* returns `Series` if `q` is a list.
`s.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased standard error of mean.
`s.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns sample standard deviation.
`s.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased variance.
`s.skew(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased skew.
`s.kurtosis(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased kurtosis.
`s.nunique(dropna=True)`	Returns count of unique items.
`s.count(level=None)`	Returns count of non-missing items.
`s.size`	Number of items in series. (Property)
`s.is_unique`	`True` if all values are unique
`s.is_monotonic`	`True` if all values are increasing
`s.is_monotonic_increasing`	`True` if all values are increasing
`s.is_monotonic_decreasing`	`True` if all values are decreasing

Table 7.2: Aggregation methods and properties

7.4 Summary

In this chapter, we discussed ways to summarize data in a series. As you begin to analyze data, you will find many of these keep popping up. One thing to keep in mind is that they also apply to a `DataFrame`.

7.5 Exercises

With a dataset of your choice:

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

Chapter 8

Conversion Methods

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 have 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`.

8.1 Automatic Conversion

In pandas 1.0, a new conversion method was introduced, `.convert_dtypes`. This tries to convert a `Series` to a type that supports `pd.NA`. In the case of our `city_mpg` series, it will change the type from `int64` to `Int64`:

```
>>> city_mpg.convert_dtypes()
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: Int64
```

I find that `.convert_dtypes` is a little too magical for me. I prefer a little more explicit control over what happens to my data.

To specify a type for a series, you can try to use the `.astype` method. Our city mileage can be held in a 16-bit integer, however an 8-bit 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')
0         19
1          9
2         23
3         10
4         17
          ..
41139     19
41140     20
41141     18
```

```
41142    18
41143    16
Name: city08, Length: 41144, dtype: Int16

>>> city_mpg.astype('Int8')
Traceback (most recent call last):
  ...
TypeError: cannot safely cast non-equivalent int64 to int8
```

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, giving you memory to process more data.

You can use NumPy to inspect limits on integer and float types:

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

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

>>> np.finfo('float16')
finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)

>>> np.finfo('float64')
finfo(resolution=1e-15, min=-1.7976931348623157e+308,
      max=1.7976931348623157e+308, dtype=float64)
```

8.2 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 strings and is stored as an `object`. To get the amount of memory that includes the strings, we need to use the `.memory_usage` method:

```
>>> make = df.make
>>> make.nbytes
329152

>>> make.memory_usage()
329280

>>> make.memory_usage(deep=True)
2606395
```

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:

```
>>> (make
...  .astype('category')
...  .memory_usage(deep=True)
... )
95888
```

8.3 String and Category Types

The `.astype` method can also convert numeric series to strings if you pass `str` into it. Note the `dtype` in the example below:

```
>>> city_mpg.astype(str)
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: object
```

To convert to a categorical type, you can pass in `'category'` as a type:

```
>>> city_mpg.astype('category')
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]
```

A categorical series is useful for string data and can result in large memory savings. This is because pandas stores Python strings when you have string data. When you convert it to categorical data, pandas no longer uses Python strings for each value but optimizes it, so repeating values are not duplicated. You still have all of the functionality found off of the `.str` attribute, but it comes with potentially large memory savings (if you have many duplicate values) and performance boosts as you do not need to perform as many string operations.

8.4 Ordered Categories

To create ordered categories, you need to define your own `CategoricalDtype`:

```
>>> 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]
```

The section on categories below will discuss more of their features.
The following table lists the types that you can pass into `.astype`.

String or Type	*Description*
`str` `'str'`	Convert type to Python string
`'string'`	Convert type to pandas string (supports `pd.NA`)
`int` `'int'` `'int64'`	Convert type to NumPy int64
`'int32'` `'uint32'`	Convert type to 32 signed or unsigned NumPy integer (can also use 16 and 8).
`'Int64'`	Convert type to pandas Int64 (supports `pd.NA`). Might complain when you convert floats or strings.
`float` `'float'` `'float64'`	Convert type to NumPy float64 (can also support 32 or 16).
`'category'`	Convert type to categorical (supports `pd.NA`). Can also use instance of `CategoricalDtype`.
dates	Don't use this for date conversion, use `pd.to_datetime`.

Table 8.1: Type and strings for column conversion

8.5 Converting to 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 as well. I find pandas objects to be a lot 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 just 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, there are many conversion methods to export data into other formats, including CSV, Excel, HDF5, SQL, JSON, 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. If you want to add timezone information, it is a little more involved. The section on dates will discuss this.

Method	*Description*
`s.convert_dtypes(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True)`	Convert types to appropriate pandas 1 types (that support `NA`). Doesn't try to reduce size of integer or float types.
`s.astype(dtype, copy=True, errors='raise')`	Cast series into particular type. If `errors='ignore'` then return original series on error.
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, format=None, exact=True, unit=None, infer_datetime_format=False, origin='unix', cache=True)`	Convert `arg` (a series) into datetime. Use `format` to specify strftime string.
`s.to_numpy(dtype=None, copy=False,na_value=object, **kwargs)`	Convert the series to a NumPy array.
`s.values`	Convert the series to a NumPy array.
`s.to_frame(name=None)`	Return a dataframe representation of the series.
`pd.CategoricalDtype(categories=None, ordered=False)`	Create a type for categorical data.

Table 8.2: Aggregation methods and properties

8.6 Summary

Having the correct types is very convenient. Not only does it save memory, but it also enables operations that are otherwise tedious. Whenever I teach students the fundamentals of data analysis, I make sure that they go through each column and determine what the correct type for that column is.

8.7 Exercises

With a dataset of your choice:

1. Convert a numeric column to a smaller type.
2. Calculate the memory savings by converting to smaller numeric types.
3. Convert a string column into a categorical type.
4. Calculate the memory savings by converting to a categorical type.

Chapter 9

Manipulation Methods

I consider manipulation methods to be the workhorses of pandas. When I have a dataset that 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 of the methods we discuss here manipulate the series values but preserve the index. In this chapter, we will explore these methods.

9.1 `.apply` and `.where`

The `.apply` is a curious method, and I often tell my students to avoid it, but sometimes it comes in handy. This method allows you to apply a function element-wise to every value. If you pass in a NumPy function that works on an array, it will broadcast the operation to the series.

However, usually, when I see this method is used, it is a code smell. How so? Because the `.apply` method typically operates on each individual value in the series, the function is called once for every value. If you have one million values in a series, it will be called one million times. 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'm going to look at the *make* column from my dataset. This has the company that made 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
3              Dodge
4             Subaru
              ...
41139         Subaru
41140         Subaru
41141         Subaru
41142         Subaru
41143         Subaru
Name: make, Length: 41144, dtype: object

>>> make.value_counts()
Chevrolet                            4003
Ford                                 3371
Dodge                                2583
GMC                                  2494
Toyota                               2071
                                     ...
Superior Coaches Div E.p. Dutton        1
Vixen Motor Company                     1
London Coach Co Inc                     1
Panoz Auto-Development                  1
Qvale                                   1
Name: make, Length: 136, dtype: int64
```

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

```
>>> top5 = make.value_counts().index[:5]
>>> def generalize_top5(val):
...     if val in top5:
...         return val
...     return 'Other'

>>> make.apply(generalize_top5)
0         Other
1         Other
2         Dodge
3         Dodge
4         Other
          ...
41139     Other
41140     Other
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: object
```

Note that when we have already defined a function to pass into `.apply` that we do not call that function. In the above example, we are not calling `generalize_top5`, 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 idiomatic 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

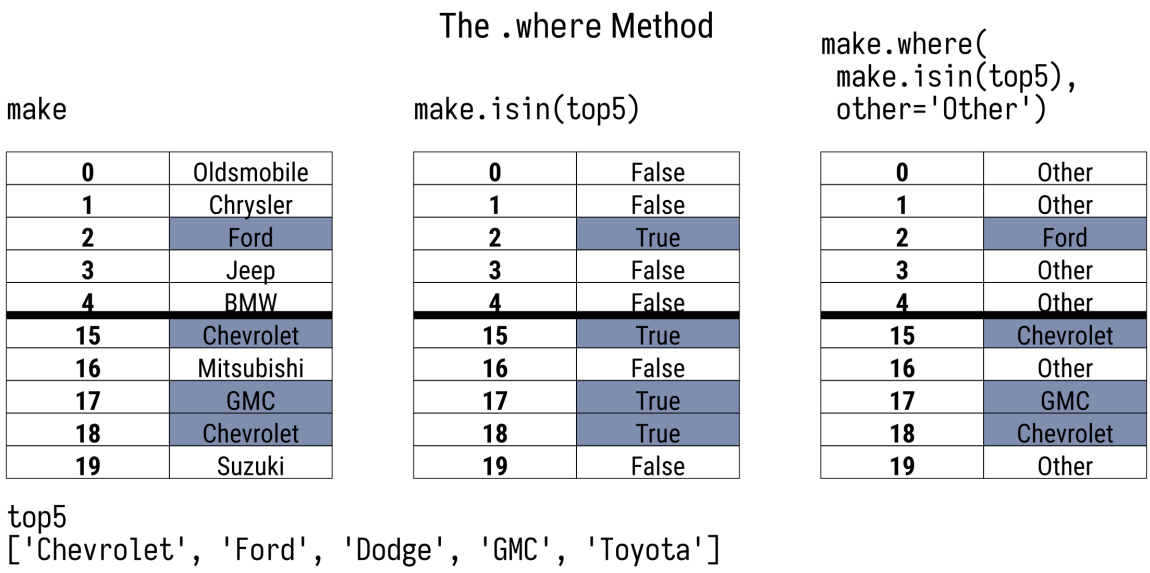


Figure 9.1: The .where method keeps the values where the index is True and uses the other parameter to specify values for False.

below) where the boolean array is true, 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
3         Dodge
4         Other
          ...
41139     Other
41140     Other
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: object
```

The .where method is optimized and if you look at the timings it is about six times faster:

```
>>> %%timeit
>>> make.apply(generalize_top5)
23.3 ms ± 3.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

>>> %%timeit
>>> make.where(make.isin(top5), 'Other')
4.49 ms ± 1.94 ms per loop (mean ± std. dev. of 7 runs, 100 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
3         Dodge
4         Other
          ...
41139     Other
41140     Other
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: object
```

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 ignore `.mask` since it is the complement.

9.2 If Else with Pandas

I'm going to 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. 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
3         Dodge
4         Other
          ...
41139     Other
41140     Other
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: object
```

To replicate this in pandas, I would need to chain calls to `.where`:

```
>>> (make
...   .where(make.isin(top5), 'Top10')
...   .where(make.isin(top10), 'Other')
... )
```

```
0        Other
1        Other
2        Dodge
3        Dodge
4        Other
         ...
41139    Other
41140    Other
41141    Other
41142    Other
41143    Other
Name: make, Length: 41144, dtype: object
```

Another option is to use the `select` function found in the NumPy library. This function works with a pandas series. The interface takes a list of boolean arrays and a list with corresponding replacement values. Finally, you can give it a default value:

```
>>> import numpy as np
>>> np.select([make.isin(top5), make.isin(top10)],
...    [make, 'Top10'], 'Other')
array(['Other', 'Other', 'Dodge', ..., 'Other', 'Other', 'Other'],
      dtype=object)
```

Note that this returns a NumPy array. You can wrap it in a `Series` if you desire. I like this syntax for longer if statements than chaining `.where` calls because I think it is easier to understand:

```
>>> pd.Series(np.select([make.isin(top5), make.isin(top10)],
...    [make, 'Top10'], 'Other'), index=make.index)
0        Other
1        Other
2        Dodge
3        Dodge
4        Other
         ...
41139    Other
41140    Other
41141    Other
41142    Other
41143    Other
Length: 41144, dtype: object
```

9.3 Missing Data

Filling in missing data is another common operation, and this is important because many machine learning algorithms do not work if there is missing data. Also, it is prudent to be aware of how much data is missing to make sure you are getting 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 entries that are missing. 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 have more context, and a dataframe gives 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
8144       Ford
8146       Ford
          ...
34563     Tesla
34564     Tesla
34565     Tesla
34566     Tesla
34567     Tesla
Name: make, Length: 206, dtype: object
```

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 that names 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 is referring 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.

9.4 Filling In Missing Data

It looks like the cylinder information is missing from cars that are electric. 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    NaN
7139    NaN
8143    NaN
8144    NaN
8146    NaN
         ..
34563   NaN
34564   NaN
34565   NaN
34566   NaN
34567   NaN
Name: cylinders, Length: 206, dtype: float64
```

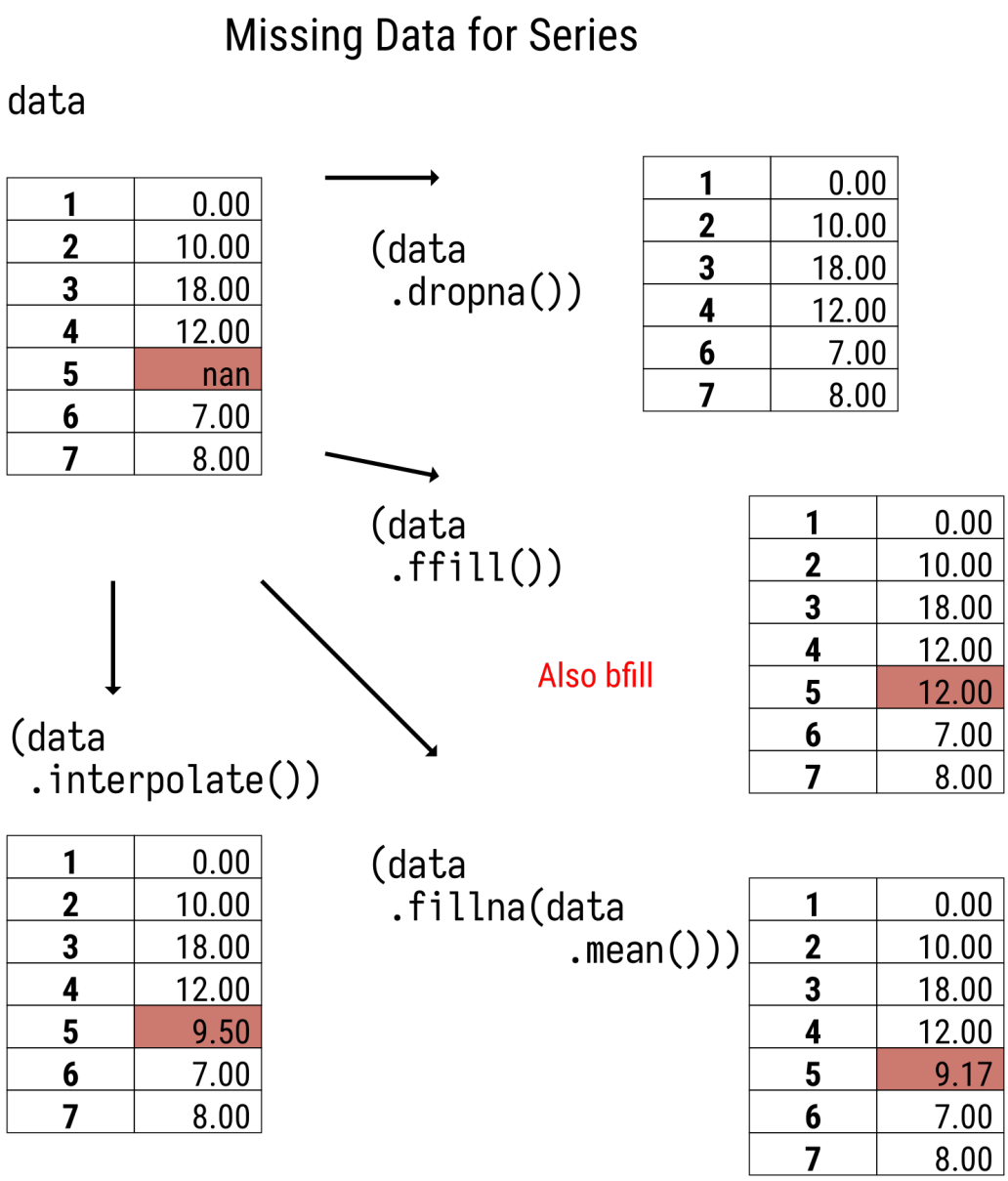


Figure 9.2: We can drop missing data or fill it in with other values.

```
>>> cyl.fillna(0).loc[7136:7141]
7136    6.0
7137    6.0
7138    0.0
7139    0.0
7140    6.0
7141    6.0
Name: cylinders, dtype: float64
```

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 would need to assign it to this new result. Usually, I end up chaining each command and build up a sequence of operations.

9.5 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 temperature measurements, `temp`, you could fill in the values using this:

```
>>> temp = pd.Series([32, 40, None, 42, 39, 32])
>>> temp
0    32.0
1    40.0
2     NaN
3    42.0
4    39.0
5    32.0
dtype: float64

>>> temp.interpolate()
0    32.0
1    40.0
2    41.0
3    42.0
4    39.0
5    32.0
dtype: float64
```

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*, the interpolation of the values around the missing value.

9.6 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
3      10
4      17
       ..
442    15
443    15
444    15
445    15
446    31
Name: city08, Length: 447, dtype: int64
```

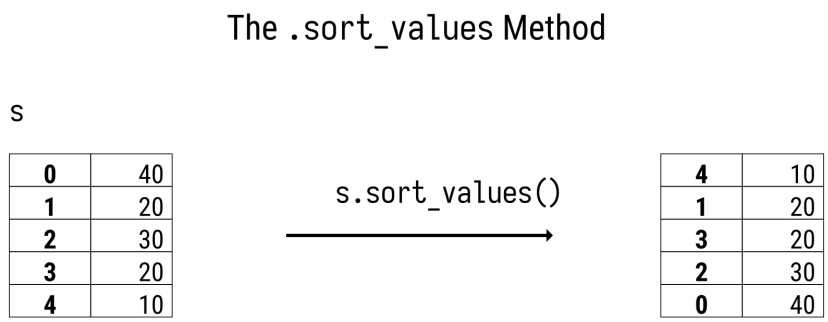


Figure 9.3: The `.sort_values` method will return a new series with the values sorted (and the original labels in the corresponding order).

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
3      11
4      17
       ..
442    15
443    15
444    15
445    15
446    27
Name: city08, Length: 447, dtype: int64
```

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:

```
if upper is not None:
    subset = self.to_numpy() <= upper
    result = result.where(subset, upper)
if lower is not None:
    subset = self.to_numpy() >= lower
    result = result.where(subset, lower)
```

9.7 Sorting Values

There are other manipulation methods that 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
```

```
34557      6
37161      6
21060      6
35887      6
         ...
34563    138
34564    140
32599    150
31256    150
33423    150
Name: city08, Length: 41144, dtype: int64
```

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
3         11.0
4         20.0
          ...
41139     22.5
41140     24.0
41141     21.0
41142     21.0
41143     18.5
Length: 41144, dtype: float64
```

9.8 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
3         10
4         17
          ..
41139     19
41140     20
41141     18
41142     18
41143     16
Name: city08, Length: 41144, dtype: int64
```

9.9 Dropping Duplicates

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 found 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

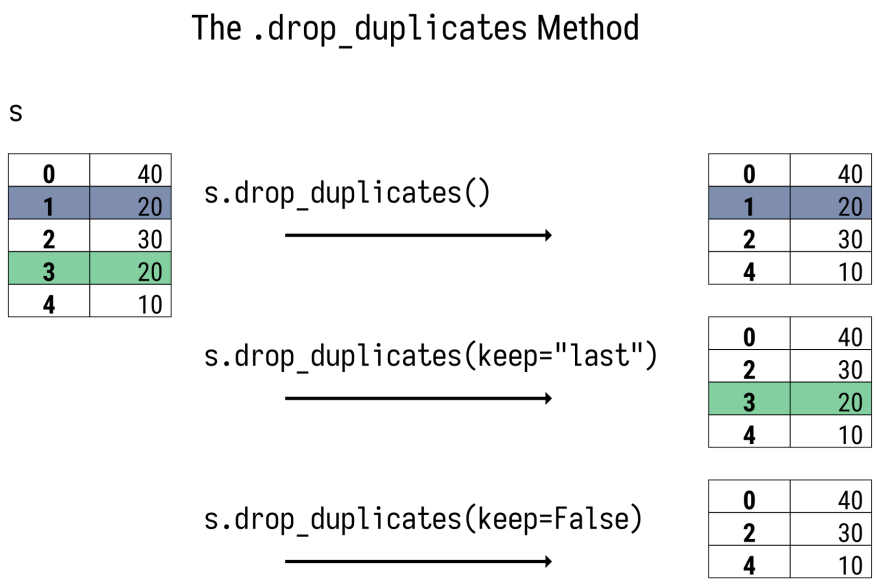


Figure 9.4: The `.drop_duplicates` method will return a new series that drops the values after they appear more than once by default. The behavior can be changed with the `keep` parameter.

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
3          10
4          17
         ...
34364     127
34409     114
34564     140
34565     115
34566     104
Name: city08, Length: 105, dtype: int64
```

9.10 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. You can specify `'min'` to put equal values in the same rank, and `'dense'` to not skip any positions:

```
>>> city_mpg.rank()
0         27060.5
1           235.5
2         35830.0
```

```
3          607.5
4        19484.0
          ...
41139    27060.5
41140    29719.5
41141    23528.0
41142    23528.0
41143    15479.0
Name: city08, Length: 41144, dtype: float64

>>> city_mpg.rank(method='min')
0        25555.0
1          136.0
2        35119.0
3          336.0
4        17467.0
          ...
41139    25555.0
41140    28567.0
41141    21502.0
41142    21502.0
41143    13492.0
Name: city08, Length: 41144, dtype: float64

>>> city_mpg.rank(method='dense')
0        14.0
1         4.0
2        18.0
3         5.0
4        12.0
         ...
41139    14.0
41140    15.0
41141    13.0
41142    13.0
41143    11.0
Name: city08, Length: 41144, dtype: float64
```

9.11 Replacing Data

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

```
>>> make.replace('Subaru', 'スバル')
0        Alfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4              スバル
           ...
41139          スバル
41140          スバル
41141          スバル
41142          スバル
41143          スバル
```

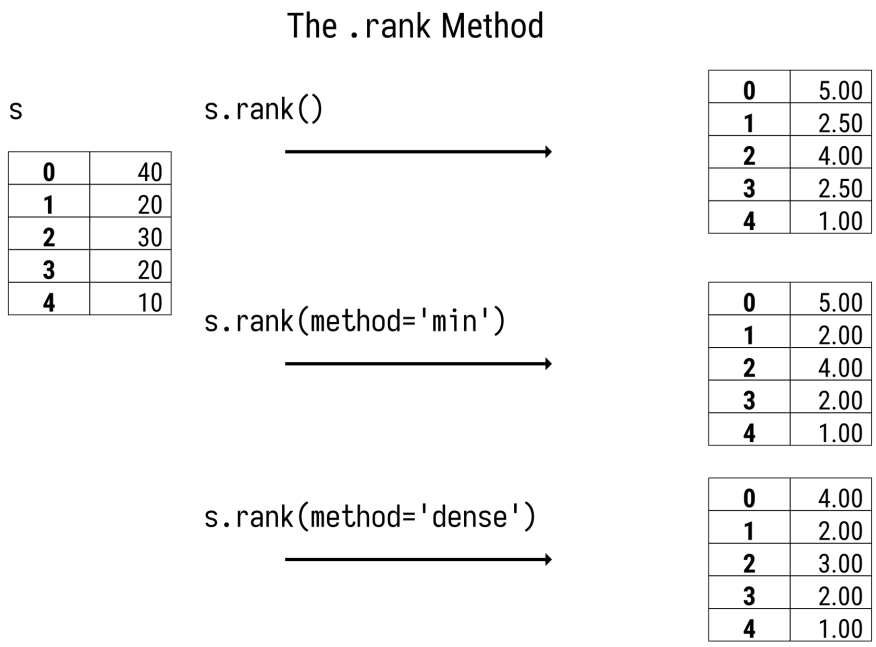


Figure 9.5: The .rank method has various options for dealing with ties.

```
Name: make, Length: 41144, dtype: object
```

The to_replace parameter's value can contain a regular expression if you provide the regex=True parameter. In this example we use regular expression *capture groups* (they are specified in the expression by the parentheses). In 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
3               Dodge
4              Subaru
              ...
41139          Subaru
41140          Subaru
41141          Subaru
41142          Subaru
41143          Subaru
Name: make, Length: 41144, dtype: object
```

9.12 Binning Data

You can bin data as well. Using the cut function, you can create bins of equal width:

```
>>> pd.cut(city_mpg, 10)
0        (5.856, 20.4]
1        (5.856, 20.4]
```

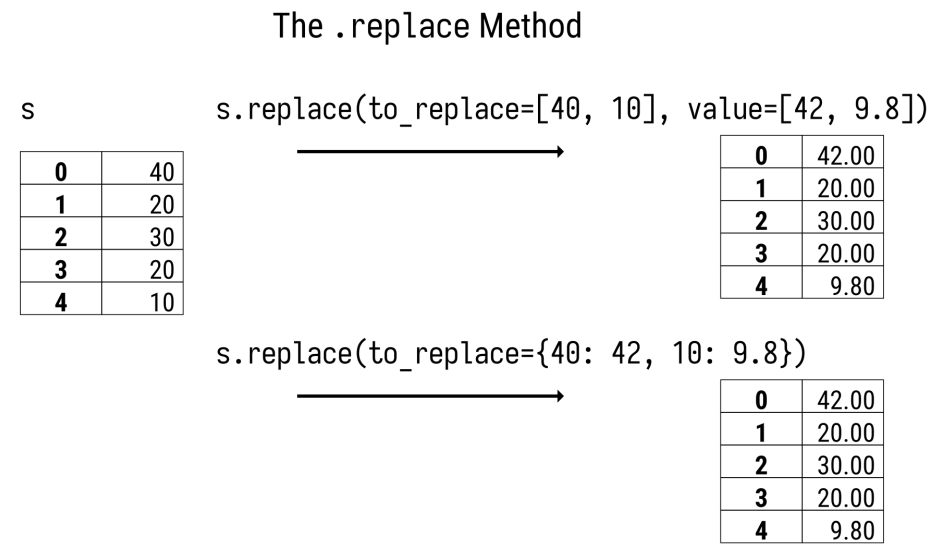


Figure 9.6: The `.replace` method illustrating lists and dictionaries.

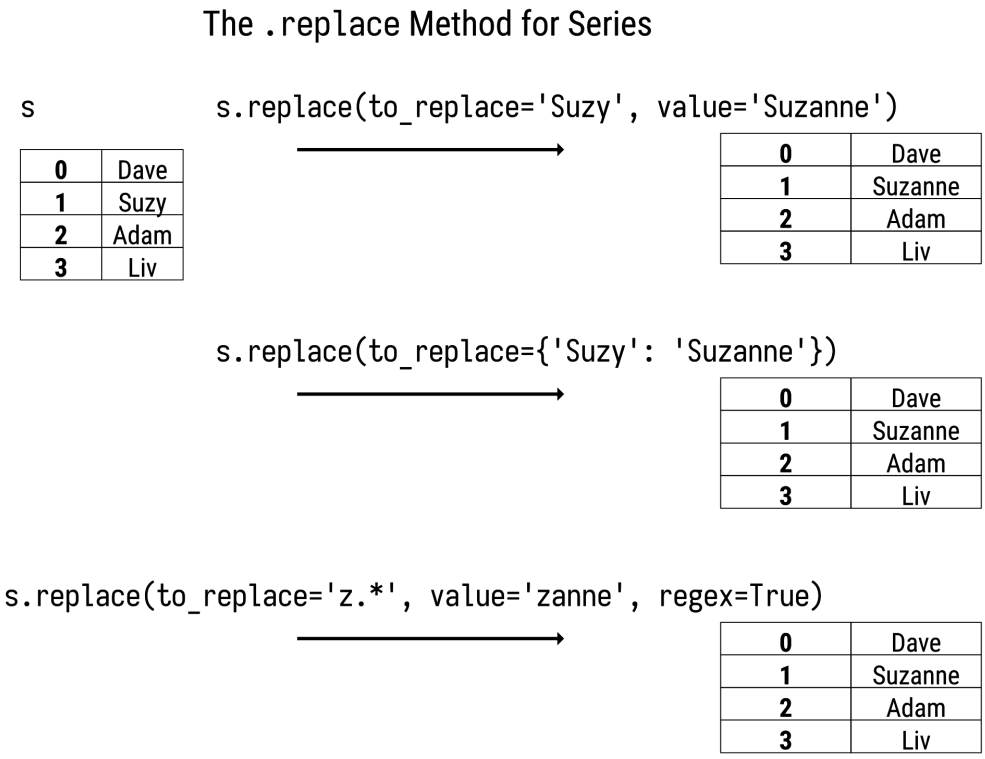


Figure 9.7: The `.replace` method illustrating different replacement mechanisms.

```
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 results of this call is a series with 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):

```
>>> 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 10 bins that had approximately the same number of entries in each bin (rather that each bin width being the same), use the `qcut` function:

```
>>> 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]): [(5.999, 13.0] < (13.0, 14.0] < ...
   (18.0, 20.0] < (20.0, 21.0] < (21.0, 24.0] < (24.0, 150.0]]
```

Both of these functions allow you to set the labels to use instead of 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]
```

Method	*Description*
`s.apply(func, convert_dtype=True, args=(), **kwds)`	Pass in a NumPy function that works on the series, or a Python function that works on a single value. `args` and `kwds` are arguments for `func`. Returns a series, or dataframe if `func` returns a series.
`s.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False)`	Pass in a boolean series/dataframe, list, or callable as `cond`. If the value is `True`, keep it, otherwise use `other` value. If it is a function, it takes a series and should return a boolean sequence.
`np.select(condlist, choicelist, default=0)`	Pass in a list of boolean arrays for `condlist`. If the value is true use the corresponding value from `choicelist`. If multiple conditions are True, only use the first. Returns a NumPy array.
`s.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)`	Pass in a scalar, dict, series, or dataframe for `value`. If it is a scalar, use that value, otherwise use the index from the old value to the new value.
`s.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)`	Perform interpolation with missing values. `method` may be `linear`, `time` among others.
`s.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)`	Return a new series with values clipped to `lower` and `upper`.
`s.sort_values(axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)`	Return a series with values sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates location of NaNs and may be `'first'` or `'last'`.
`s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)`	Return a series with index sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates location of NaNs and may be `'first'` or `'last'`.
`s.drop_duplicates(keep='first', inplace=False)`	Drop duplicates. `keep` may be `'first'`, `'last'`, or `False`. (If `False`, it removes all values that were duplicated).

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

Table 9.1: Manipulation methods and properties

9.13 Summary

In this chapter, we explored many methods and functions that are useful for changing the data. We saw how to use function application with the `.apply` method, but try to avoid that and use `np.select` instead to get better performance. We discussed various ways to deal with missing data. We saw that we can 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.

9.14 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 `np.select`.
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 to 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 that is not in the top `n` most frequent values with `'Other'`.
9. Using a numeric column, bin it into 10 groups that have the same width.
10. Using a numeric column, bin it into 10 groups that have equal sized bins.

Chapter 10

Indexing Operations

Indexing is an overloaded term in the pandas world. Both a series and a dataframe have an index (the labels down the left side for each row). In addition, 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 both changing the index and accessing parts of a series with the indexing operators.

10.1 Prepping the Data and Renaming the Index

To ease explaining the various operations, I'm going to 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. If these are both integer values, it can be a little confusing but becomes more clear if the index has string labels.

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

```
>>> city2 = city_mpg.rename(make.to_dict())
>>> city2
Alfa Romeo    19
Ferrari        9
Dodge         23
Dodge         10
Subaru        17
              ..
Subaru        19
Subaru        20
Subaru        18
Subaru        18
Subaru        16
Name: city08, Length: 41144, dtype: int64
```

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)
```

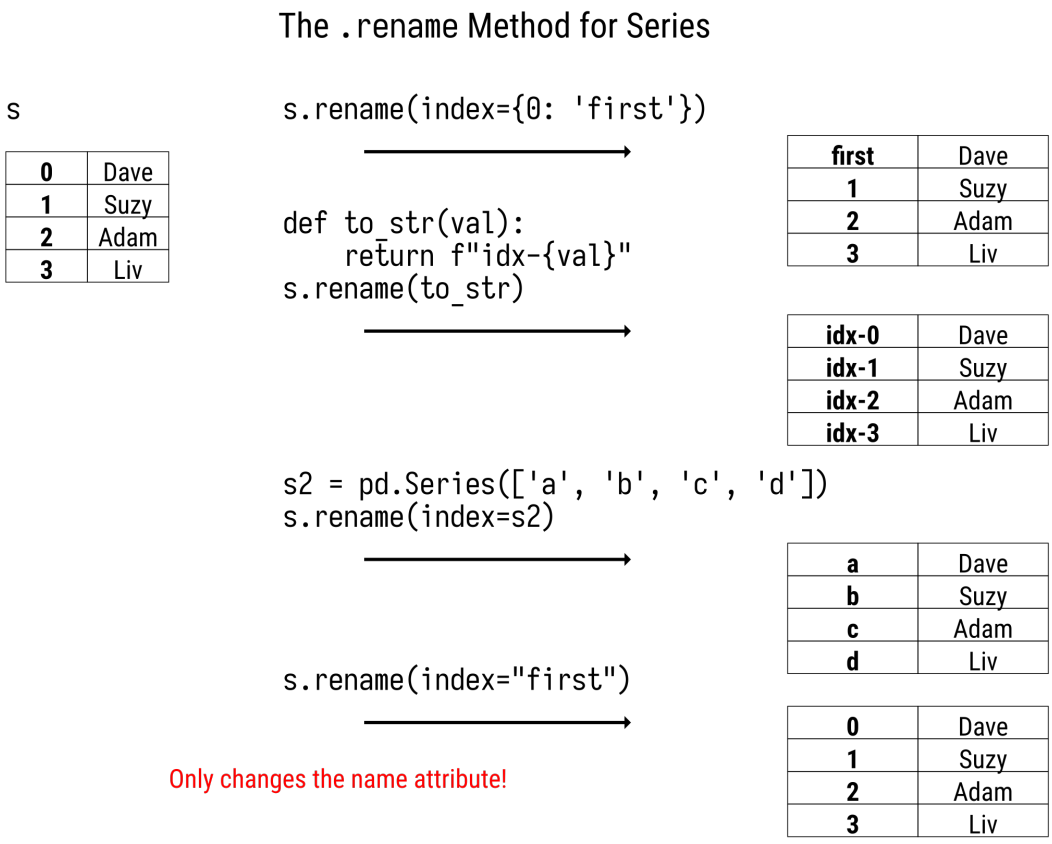


Figure 10.1: The `.rename` method will return a new series with the original values but new index labels. If you pass in a scalar value it will change the `.name` attribute of the series on the new series it returns, leaving the index intact.

The `.rename` method also accepts a series, a scalar, 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
Dodge         10
Subaru        17
              ..
Subaru        19
Subaru        20
Subaru        18
Subaru        18
Subaru        16
Name: city08, Length: 41144, dtype: int64
```

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
Dodge         10
Subaru        17
              ..
Subaru        19
Subaru        20
Subaru        18
Subaru        18
Subaru        16
Name: citympg, Length: 41144, dtype: int64
```

10.2 Resetting the Index

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

```
>>> city2.reset_index()
            index  city08
0      Alfa Romeo      19
1         Ferrari       9
2           Dodge      23
3           Dodge      10
4          Subaru      17
...           ...     ...
41139      Subaru      19
41140      Subaru      20
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
3        10
4        17
         ..
41139    19
41140    20
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int64
```

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

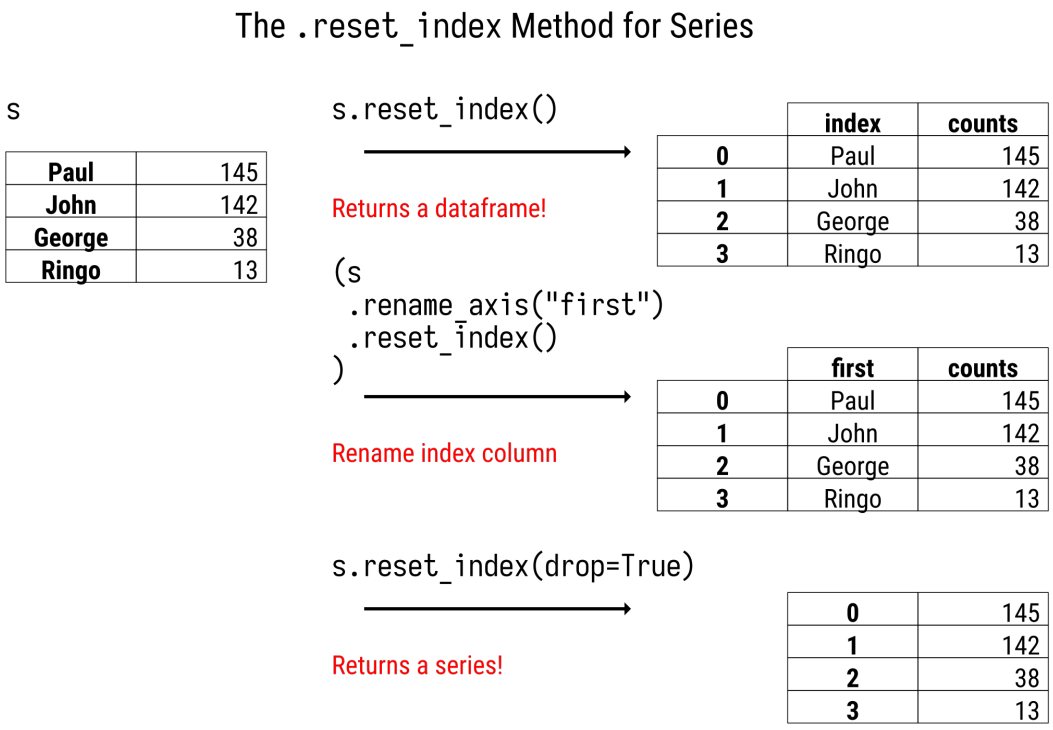


Figure 10.2: The `.reset_index` method will return a dataframe or a series with the index changed to a monotonically increasing index.

10.3 The `.loc` Attribute

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

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

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

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

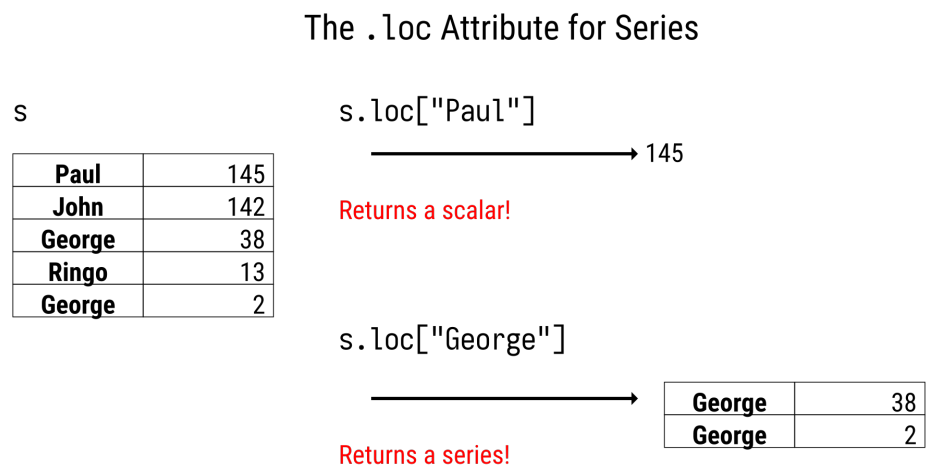


Figure 10.3: Indexing off of the .loc attribute will return a series if the index label is duplicated.

```
>>> city2.loc['Subaru']
Subaru    17
Subaru    21
Subaru    22
Subaru    19
Subaru    20
          ..
Subaru    19
Subaru    20
Subaru    18
Subaru    18
Subaru    16
Name: city08, Length: 885, dtype: int64

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

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

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

>>> city2.loc[['Ferrari', 'Lamborghini']]
Ferrari          9
Ferrari         12
Ferrari         11
Ferrari         10
Ferrari         11
                ..
Lamborghini      6
Lamborghini      8
Lamborghini      8
Lamborghini      8
Lamborghini      8
```

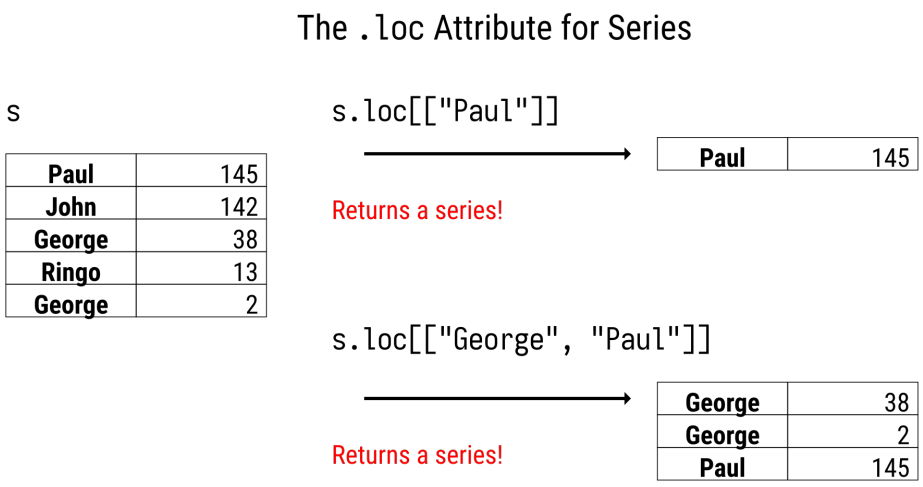


Figure 10.4: Indexing off of the `.loc` attribute with a list of index names will return a series.

```
Name: city08, Length: 357, dtype: int64
```

This next option might seem a little weird if you are used to normal list slicing with Python. When we slice sequences, we use integer index position, however with `.loc` we can use a slice with string values. You need to be aware that if join will first need to 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
Ferrari         9
Ferrari        10
               ..
Lamborghini    12
Lamborghini     9
Lamborghini     8
Lamborghini    13
Lamborghini     8
Name: city08, Length: 11210, dtype: int64
```

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

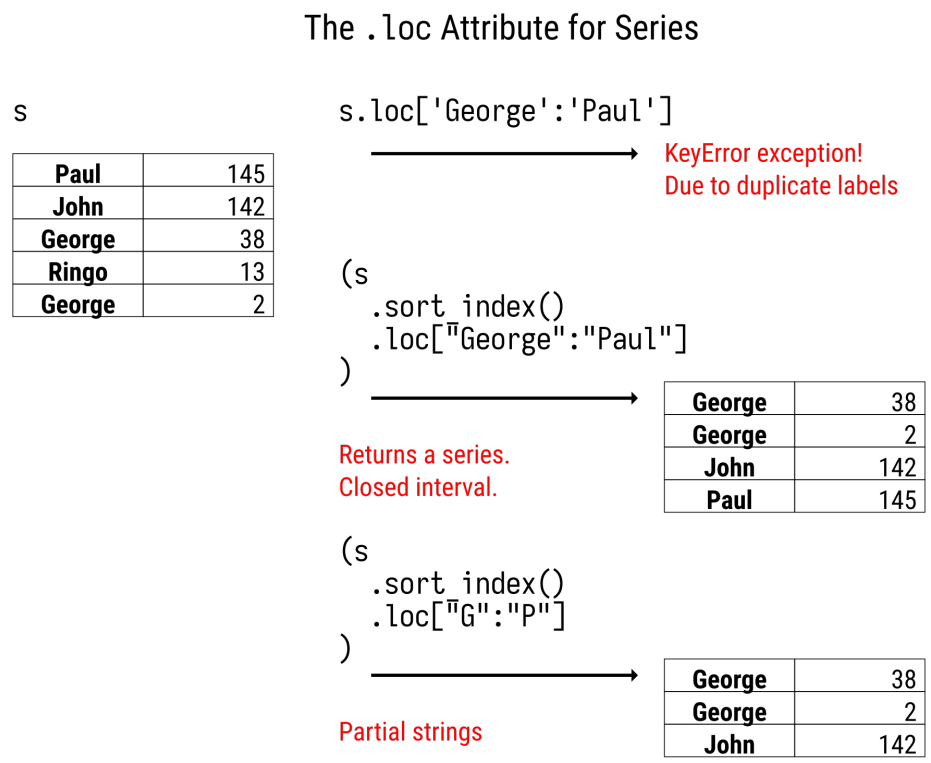


Figure 10.5: Indexing off of the .loc attribute with a slice will return a series. Note that slicing with labels is *closed* and includes the end value.

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
Federal Coach    14
Federal Coach    13
                 ..
Isuzu            15
Isuzu            15
Isuzu            15
Isuzu            27
Isuzu            18
Name: city08, Length: 9040, dtype: int64
```

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.

An example will make this more clear. 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:

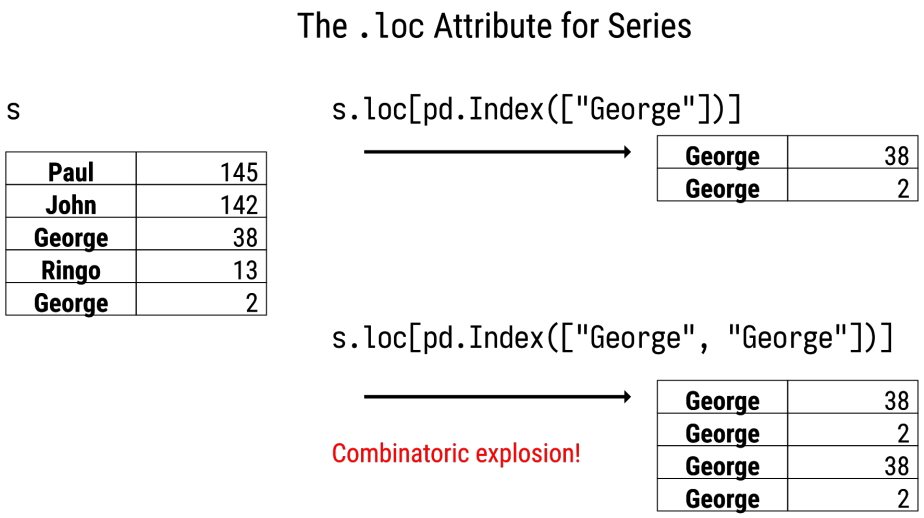


Figure 10.6: The `.loc` attribute will accept an `Index` in an indexing operation (no pun intended). Be careful with duplicate index labels, as that may lead to a combinatoric explosion.

```
>>> idx = pd.Index(['Dodge'])
>>> city2.loc[idx]
Dodge    23
Dodge    10
Dodge    12
Dodge    11
Dodge    11
         ..
Dodge    18
Dodge    17
Dodge    14
Dodge    14
Dodge    11
Name: city08, Length: 2583, dtype: int64
```

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    11
Dodge    11
         ..
Dodge    18
Dodge    17
Dodge    14
Dodge    14
Dodge    11
Name: city08, Length: 5166, dtype: int64
```

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
Dodge         False
Subaru        False
              ...
Subaru        False
Subaru        False
Subaru        False
Subaru        False
Subaru        False
Name: city08, Length: 41144, dtype: bool
```

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

```
>>> city2.loc[mask]
Nissan     81
Toyota     81
Toyota     81
Ford       74
Nissan     84
          ...
Tesla     140
Tesla     115
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64
```

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

Note

The .loc attribute cap pulling out values by specifying index name as well by using a boolean array. By using a boolean array, you can extract almost any data from a series. This becomes even more powerful when you use it with dataframes and can 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. For series objects, this might seem like overkill, but it comes in very handy with dataframes.

Here is an example. I have a series with old pricing information from last year. I know that 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:

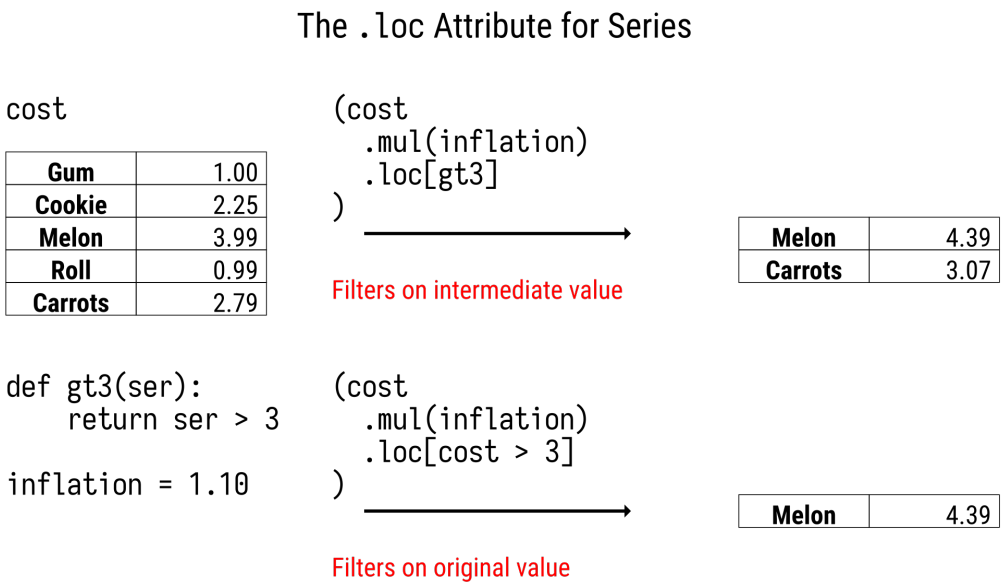


Figure 10.7: The `.loc` attribute will accept a function. This function accepts the current series it was called on and should return a scalar, list, slice, or index.

```
>>> 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, (ie using the old series instead of the chained intermediate values) I get the wrong answer:

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

Note

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

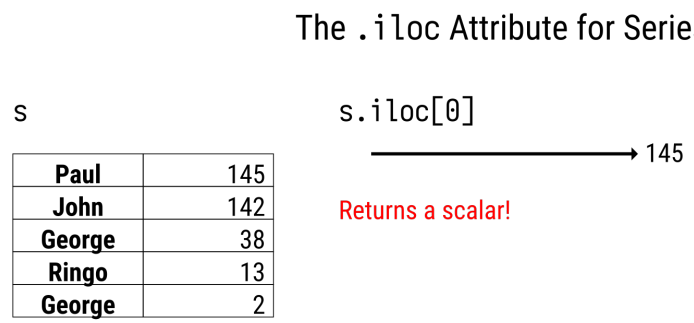


Figure 10.8: Indexing off of the .iloc attribute will return a scalar by location in the series.

```
>>> 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 there is an implicit `return` statement in the lambda function. Also, you can only put an *expression* in it, you can have a *statement*. So it is limited to a single line of code.

10.4 The .iloc Attribute

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

- A scalar index position (an integer)
- A list of index positions
- A slice of positions (half-open interval so it 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.

In the examples below we will pull out the first value and last value by slicing off of `.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:

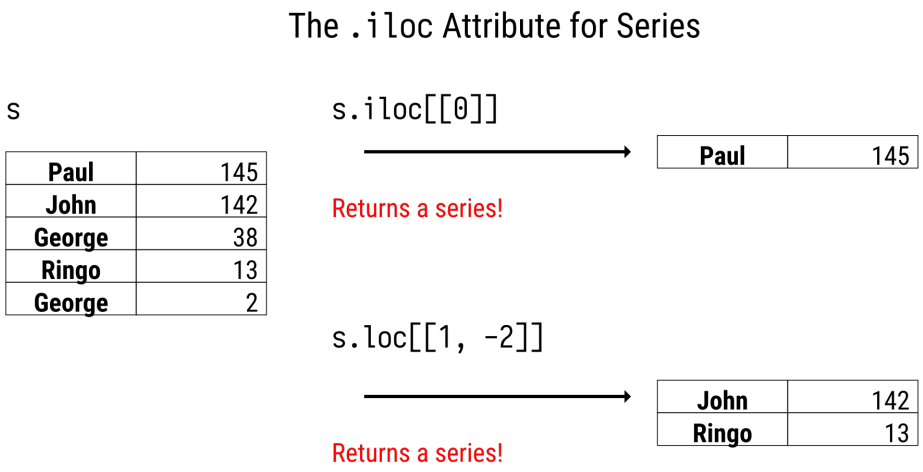


Figure 10.9: Indexing off of the `.iloc` attribute with a list will return a series of values at the locations in the list.

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

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 in it or multiple index values. The following code will return a series with the first, second, and last values:

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

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

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

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    19
Subaru    20
```

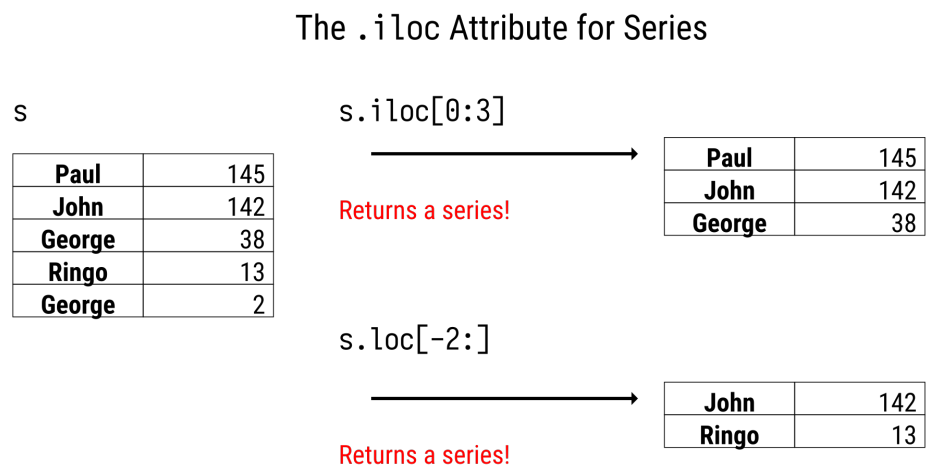


Figure 10.10: Indexing off of the .iloc attribute with a slice uses the half-open interval of positions.

```
Subaru    18
Subaru    18
Subaru    16
Name: city08, dtype: int64
```

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:

```
>>> city2.iloc[mask.to_numpy()]
Nissan     81
Toyota     81
Toyota     81
Ford       74
Nissan     84
          ...
Tesla     140
Tesla     115
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64

>>> city2.iloc[list(mask)]
Nissan     81
Toyota     81
Toyota     81
Ford       74
Nissan     84
          ...
```

```
Tesla     140
Tesla     115
Tesla     104
Tesla      98
Toyota     55
Name: city08, Length: 236, dtype: int64
```

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.

10.5 Heads and Tails

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

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

10.6 Sampling

While the previous two methods allow us to inspect the data, sampling the data can be a better choice. Often the first few entries of the data may be incomplete, test data, or not representative of all of the 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
```

10.7 Filtering Index Values

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

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

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

Using `like` we can do substring matches:

```
>>> city2.filter(like='rd')
Ford    18
Ford    16
Ford    17
Ford    17
Ford    15
        ..
Ford    26
Ford    19
Ford    21
Ford    18
Ford    19
Name: city08, Length: 3371, dtype: int64
```

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

```
>>> city2.filter(regex='(Ford)|(Subaru)')
Subaru    17
Subaru    21
Subaru    22
Ford      18
Ford      16
          ..
Subaru    19
Subaru    20
Subaru    18
Subaru    18
Subaru    16
Name: city08, Length: 4256, dtype: int64
```

10.8 Reindexing

The `.reindex` method allows you to 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
0          19.0
10         23.0
```

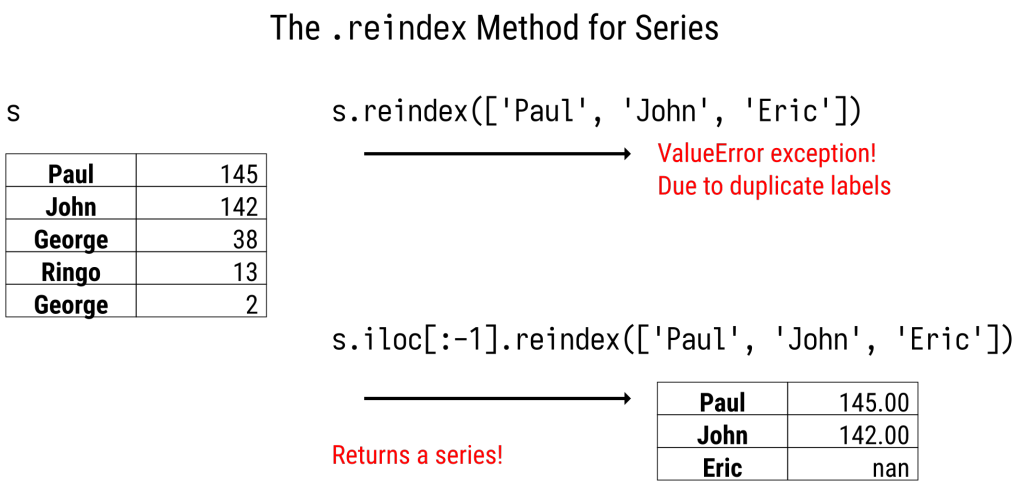


Figure 10.11: The `.reindex` method will *conform* an index to a new index.

```
20         14.0
2000000     NaN
Name: city08, dtype: float64
```

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*
`s.rename(index=None, *, level=None, errors='ignore')`	Return a series with updated `.name` attribute if `index` is a scalar. If `index` is a function series, or dictionary, return a series with updated index mapped from input (functions work on index name, series and dictionaries map the index name to a new value).
`s.index`	Returns the index of the series.
`s.reset_index(level=None, drop=False, name=None, inplace=False)`	Return a dataframe (or series when `drop=True`) with a new integer index.

`s.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True, ignore_index=False, key=None)`	Return a series with the index sorted. The `kind` option may be `'quicksort'`, `'mergesort'` (stable), or `'heapsort'`. `na_position` indicates the location of NaNs and may be `'first'` or `'last'`.
`s.loc[idx]`	Slice series by names. `idx` can be a scalar (pull out value at that name), list of names, slice with names (including end position), a boolean array, an index, or a function (that accepts the series and returns one of the previous items).
`s.iloc[idx]`	Slice series by index position. `idx` can be a scalar (pull out value at that index), list of indices, slice with index positions (half-open including start but not end index), a list of booleans, or a function (that accepts the series and returns one of the previous items).
`s.head(n=5)`	Return a series with the first `n` values.
`s.tail(n=5)`	Return a series with the last `n` values.
`s.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)`	Return a series with `n` random entries. Can also specify a fraction with `frac` (if `frac` > 1 specify `replace=True`).
`s.filter(items=None, like=None, regex=None, axis=None)`	Return a series with index values from `items` list, matching `like` substring, or when `regex` (regular expression) search matches.
`s.reindex(index=None, method=None, copy=True, level=None, limit=None, tolerance=None)`	Return a series with a conformed index.

Table 10.1: Indexing operation, methods, and properties

10.9 Summary

The index is a fundamental structure of pandas. Both a series and dataframe have an index. To get the most out of pandas, it is important that you 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, when we add a series to another series, pandas will align the index values and add the corresponding values for each index entry.

We also saw that we could index off of `.loc` and `.iloc` to pull out values by name and position, respectively. You will use both of these attributes often when dealing with pandas dataframes and series.

10.10 Exercises

With a dataset of your choice:

1. Inspect the index.
2. Sort the index.
3. Set the index to monotonically increasing integers starting from 0.

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

Chapter 11

String Manipulation

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

11.1 Strings and Objects

Before pandas 1.0, 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 that have Python types in them (such as a list, a dictionary, or a custom class). Also, the `object` type is used for mixed types. If you have a series that has numbers and strings in it, the type is also `object`.

Pandas 1.0 introduced the new `'string'` type. In addition to being more explicit than `object`, it supports missing values that are not `NaN`.

The *make* column has an `object` type by default:

```
>>> 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: object
```

You can convert it to a string type by using the `.astype` method:

```
>>> make.astype('string')
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: string
```

The main difference between the `'string'` type and strings stored in `object` (and `category`) type series is that the string methods return the nullable type when you use a `'string'` series. If the result of the string method is missing, pandas will use the newer types that have native pandas nullable types. Otherwise, the behavior is similar.

11.2 Categorical Strings

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

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

We will dive into categories later.

11.3 The `.str` Accessor

The `object`, `'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 adept at the Python string methods, many of the pandas variants should be second nature. Here is the Python string method `.lower`:

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

And here is the pandas method `.lower` that works on a series:

```
>>> make.str.lower()
0         alfa romeo
1            ferrari
2              dodge
3              dodge
4             subaru
             ...
41139         subaru
41140         subaru
41141         subaru
41142         subaru
```

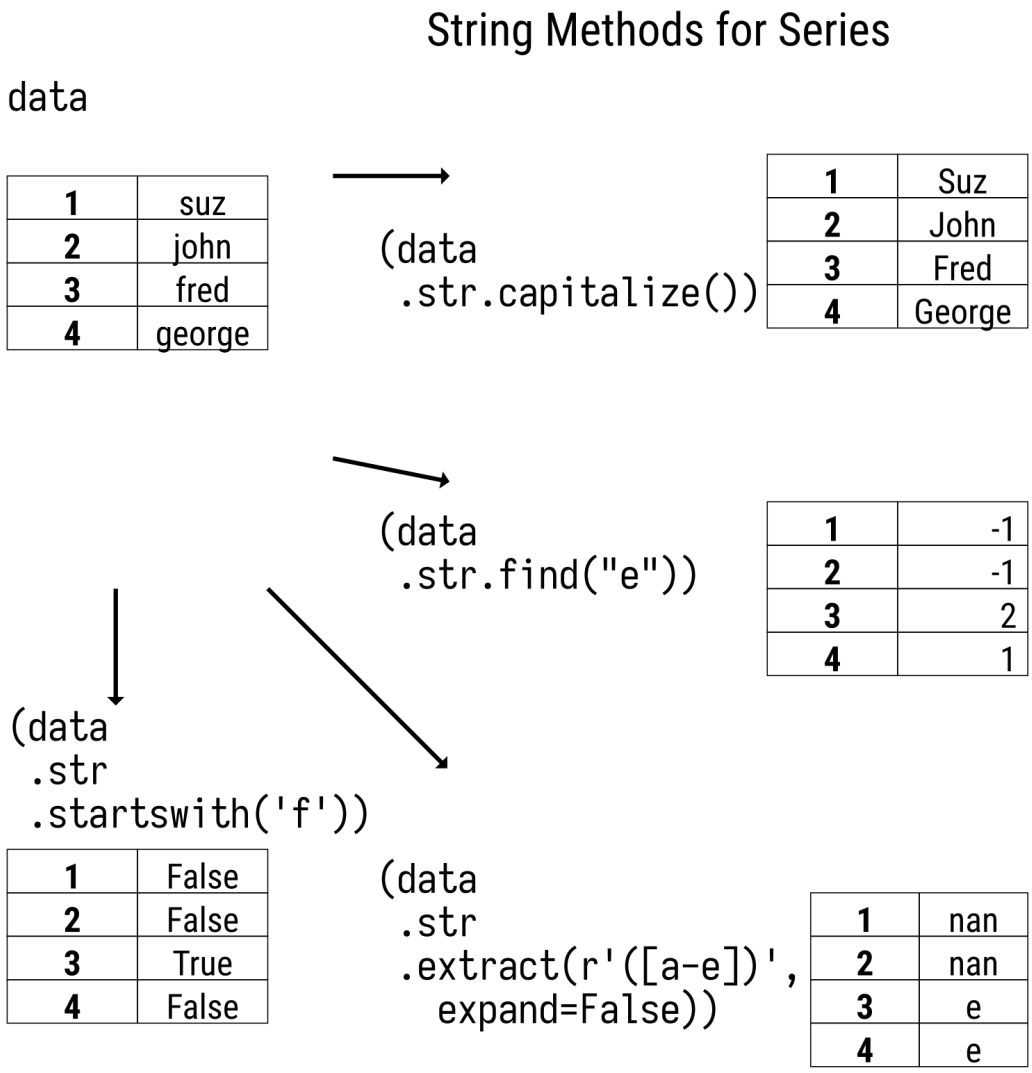


Figure 11.1: The .str accessor will allow you to manipulate strings in a series much like you can manipulate Python strings.

```
41143        subaru
Name: make, Length: 41144, dtype: object
```

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
3        -1
4        -1
         ..
41139    -1
41140    -1
41141    -1
41142    -1
41143    -1
Name: make, Length: 41144, dtype: int64
```

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

11.4 Searching

There are a few methods that leverage regular expressions to perform searching, replacing, and splitting. This book will not go deep into regular expressions as there are books solely devoted to that subject.

To find all of the non alphabetic characters (disregarding space), you could use this code:

```
>>> make.str.extract(r'([^a-z A-Z])')
          0
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
...     ...
41139   NaN
41140   NaN
41141   NaN
41142   NaN
41143   NaN

[41144 rows x 1 columns]
```

This returns a dataframe that has 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'([^a-z A-Z])', expand=False)
...    .value_counts()
... )
-    1727
```

```
.      46
,       9
Name: make, dtype: int64
```

Hint

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'([^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.

11.5 Splitting

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

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

```
>>> age = pd.Series(['0-10', '11-15', '11-15', '61-65', '46-50'])
>>> age
0     0-10
1    11-15
2    11-15
3    61-65
4    46-50
dtype: object
```

If we just 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: object
```

Having a series with a Python list makes it hard to manipulate the data. To remedy that, we can provide the `expand=True` parameter to retrieve a dataframe. 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(int)
... )
```

```
0     0
1    11
2    11
3    61
4    46
Name: 0, dtype: int64
```

This will bias our ages towards the low side. If you wanted to just 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(int)
... )
0    10
1    15
2    15
3    65
4    50
dtype: int64

>>> (age
...   .str[-2:]
...   .astype(int)
... )
0    10
1    15
2    15
3    65
4    50
dtype: int64
```

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

```
>>> (age
...   .str.split('-', expand=True)
...   .astype(int)
...   .mean(axis='columns')
... )
0     5.0
1    13.0
2    13.0
3    63.0
4    48.0
dtype: float64
```

We have not really 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    15
2    15
3    63
4    49
dtype: int64
```

11.6 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 look at making it quicker using Cython.

Cython is a superset of Python that can compile to native code. To enable it in Jupyter, you will need to 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. If you add types to Cython code, you can get a speed increase. I'll try that here:

```
%%cython
import random
cpdef int between_cy3(int x, int y):
    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)
)
```

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.str.split('-', expand=True).astype(int)
   .apply(lambda row: between_cy3(row[0], row[1]), axis=1))

31786620 function calls (31786601 primitive calls) in 12.334 seconds

 Ordered by: internal time
 List reduced from 308 to 10 due to restriction <10>

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1000000    1.533    0.000    5.190    0.000 series.py:928(__getitem__)
1000000    0.708    0.000    2.908    0.000 series.py:1034(_get_value)
```

```
1000006    0.674    0.000    2.075    0.000 generic.py:5489(__setattr__)
 500001    0.607    0.000    3.311    0.000 apply.py:937(series_generator)
1000000    0.591    0.000    1.390    0.000 base.py:5175(_get_values_for_loc)
1000000    0.534    0.000    0.809    0.000 range.py:379(get_loc)
 500006    0.501    0.000    0.501    0.000 {method 'split' of 'str' objects}
 500000    0.494    0.000    0.557    0.000 managers.py:1712(set_values)
 500003    0.461    0.000    1.327    0.000 series.py:627(name)
 500000    0.439    0.000    6.832    0.000 <string>:1(<lambda>)
```

I'm going to change the plan of attack and just send in two NumPy arrays and return a NumPy array:

```
%%cython
cimport numpy as np
import numpy as np
import random
cpdef np.ndarray[int] apply_between_cy4(np.ndarray[int] x, np.ndarray[int] y):
    cdef np.ndarray[int] res = np.empty(len(x), dtype='int32')
    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 8x faster on a dataset with 500,000 values:

```
(age
  .str.split('-', expand=True)
  .astype(int)
 .pipe(lambda df_: apply_between_cy4(df_.iloc[:, 0].to_numpy(dtype='int32'),
                                     df_.iloc[:, 1].to_numpy(dtype='int32')))
)
```

11.7 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 of the values I use `.replace`.

If I wanted to replace a capital "A" with the Unicode letter a with a ring above it, I could use this code:

```
>>> make.str.replace('A', 'Å')
0        Ålfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: object
```

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.replace('A', 'Å')
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: object
```

You can use a dictionary to specify complete replacements. (This is very explicit, but it might be problematic if you had 20,000 numeric values that had dashes in them, and you wanted to strip out the dashes for all 20,000 numbers. You would have to create a dictionary with all the entries, tedious work.):

```
>>> make.replace({'Audi': 'Åudi', 'Acura': 'Åcura',
...     'Ashton Martin': 'Åshton Martin',
...     'Alfa Romeo': 'Ålfa Romeo'})
0        Ålfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: object
```

Alternatively, you can specify that you mean to use a regular expression to replace just a portion of the strings with the `regex=True` parameter:

```
>>> make.replace('A', 'Å', regex=True)
0        Ålfa Romeo
1           Ferrari
2             Dodge
3             Dodge
4            Subaru
            ...
41139        Subaru
41140        Subaru
41141        Subaru
41142        Subaru
41143        Subaru
Name: make, Length: 41144, dtype: object
```

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 string operations are not vectorized. 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.

Method	*Description*
`.str.capitalize()`	Capitalize strings
`.str.casefold()`	Lowercase Unicode/caseless strings.
`.str.cat(others=None, sep='', na_rep=None, join='inner')`	If `others` is `None`, return a string with values separated by `sep`. Otherwise, align the index (if `others` series) and concatenate values.
`.str.center(width, fillchar=' ')`	Center align strings
`.str.contains(pat, case=True, flags=0, na=np.nan, regex=True)`	Return a boolean array if `pat` matches values.
`.str.count(pat, flags=0)`	Return series with the count of how many times `pat` occurs in each value.
`.str.decode(encoding)`	Works with bytestrings to decode them to Unicode strings.
`.str.encode(encoding)`	Encode Unicode string to bytestring.
`.str.endswith(pat, na=np.nan)`	Return boolean array if value ends with `pat`.
`.str.extract(pat, flags=0, expand=True)`	Return a dataframe with the first match from each regular expression capture group in its own column (use named groups for column names). Returns a series if `expand=False`.
`.str.extractall(pat, flags=0)`	Return a dataframe with all matches from each regular expression capture group in its own column (use named groups for column names). The dataframe has a multiindex, where the inner index is named *match* and has match number.
`.str.find(sub, start=None, end=None)`	Return the lowest index of `sub`. -1 if not found.
`.str.findall(pat, flags=0)`	Return a series with a list of matches for each value.
`.str.get(i)`	Return a series with the result of `val[i]` for each value (`val`) in the series.
`.str.get_dummies(sep='|')`	Return a dataframe with each value in its own column and a 0/1 indicating if the value is absent/appeared for that index label. If a string has multiple values they can be separated with `sep`.
`.str.index(sub, start=None, end=None)`	Return the lowest index of `sub`. `ValueError` if not found.
`.str.isalnum()`	Return boolean array if characters are alphanumeric.
`.str.isalpha()`	Return boolean array if characters are alphabetic.
`.str.isdecimal()`	Return boolean array if characters are decimal.
`.str.isdigit()`	Return boolean array if characters are digits.
`.str.islower()`	Return boolean array if characters are lowercase.
`.str.isnumeric()`	Return boolean array if characters are numeric.
`.str.isspace()`	Return boolean array if characters are whitespace.

`.str.istitle()`	Return boolean array if characters are titlecase.
`.str.isupper()`	Return boolean array if characters are uppercase.
`.str.join(sep)`	Given a series with a list of strings in it, join each element with `sep`.
`.str.len()`	Return a series with length of each value (works with lists or collections).
`.str.ljust(width, fill=' ')`	Return a left justified series.
`.str.lower()`	Return a lowercase series.
`.str.lstrip(to_strip=None)`	Return a series with left stripped `to_strip` (whitespace default).
`.str.match(pat, case=True, flags=0, na=np.nan)`	Return a boolean array if `pat` matches values (anchored at the beginning). Use `.str.contains` to match anywhere in the string. (Use `.str.extract` to pull out the string.)
`.str.normalize(form)`	Return Unicode normal form for series. `form` can be `'NFC'`, `'NFKC'`, `'NFD'`, or `'NFKD'`.
`.str.pad(width, side='left', fill=' ')`	Return a padded series of length `width`. `side` can be `'left'`, `'right'`, or `'both'`.
`.str.partition(sep, expand=True)`	Return a dataframe with three columns: element before first `sep`, the `sep`, and the part after.
`.str.repeat(repeats)`	Return a series with values repeated `repeats` times. `repeats` can be a scalar or list.
`.str.replace(pat, repl, n=-1, case=True, flags=0, regex=True)`	Return a series where `pat` is replaced by `repl`. `n` is the number of times to replace a value. `repl` can be a string or a callable that takes a match object and returns a string.
`.str.rfind(sub, start=None, end=None)`	Return highest index of `sub`. -1 if not found.
`.str.rindex(sub, start=None, end=None)`	Return highest index of `sub`. `ValueError` if not found.
`.str.rjust(width, fill=' ')`	Return a right justified series.
`.str.rpartition(sep, expand=True)`	Return a dataframe with three columns: element before last `sep`, the `sep`, and the part after.
`.str.rsplit(pat, n=-1, expand=False)`	Return a Series (if `expand=False`) with a list of values split from the right side limited to `n` splits.
`.str.rstrip(to_strip=None)`	Return a series with rightstripped `to_strip` (whitespace default).
`.str.slice(start=None, stop=None, step=None)`	Return a series. Equivalent to `s[start:stop:step]`.
`.str.slice_replace(start=None, stop=None, repl=None)`	Return a series with slice replaced by the value of `repl`.
`.str.split(pat, n=-1, expand=False)`	Return a Series (if `expand=False`) with a list of values split by `sep` limited to `n` splits.
`.str.startswith(pat, na=np.nan)`	Return boolean array if value starts with `pat`.
`.str.strip(to_strip=None)`	Return a series with left and right stripped `to_strip` (whitespace default).
`.str.swapcase()`	Return swapcase series.
`.str.title()`	Return titlecase series.
`.str.translate(table)`	Return series using a dictionary `table` to replace characters. `table` maps code points to new code points (numbers not strings). Keys mapped to `None` are deleted.
`.str.upper()`	Return uppercase series.

`.str.wrap(width)`	Return a line wrapped series limited to `width`.
`.str.zfill(width)`	Return a series limited to `width` left padded with `'0'`.

Table 11.1: String methods

11.8 Summary

The `object`, `'string'`, and `'category'` type series all can be used to store string data. They all have the `.str` accessor. If you are familiar with Python strings, you get much of the same functionality. In addition, there is the ability to manipulate with regular expressions.

11.9 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. Using a string column, create a series extracting the numeric values.
5. Using a string column, create a series extracting the non-ASCII values.
6. Using a string column, create a dataframe with the dummy columns for every character in the column.

Chapter 12

Date and Time Manipulation

Pandas allows you to create series with date and time information in them. In this chapter, we will explore common operations that you will need to perform with date data.

12.1 Date Theory

Let's talk about dates in brief. Coordinated Universal Time (UTC) 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* timezone, which is 6 or 7 hours offset of UTC depending on the time of year.

In short, a timezone 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 timezone but optionally an offset.

A note on timezone names. The public domain timezone database (also known as the Olsen database) from iana.org provides code and data regarding timezones and their history. From their documentation:

> Timezones 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 timezone name is important and might be confusing or difficult. As I said, I live in Salt Lake City. If I search for "Timezone for Salt Lake City", I get "Mountain Daylight Time" or "GMT-6". Neither of which is a timezone. You might also see "US/Mountain", "MST", or "MDT". These are not timezones either. These are deprecated names or offsets. The correct timezone name is "America/Denver". However, many applications support erroneous names.

I recommend prefacing your search with "IANA" (ie. "IANA Timezone for Salt Lake City") and then double checking your result in this Wikipedia article (which shows deprecated names)[8].

[8]https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

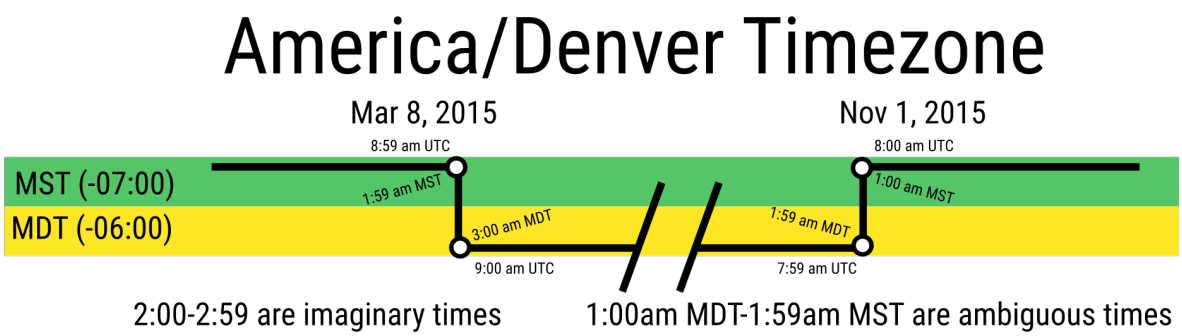


Figure 12.1: When daylight savings begins in the spring, it creates imaginary times. When daylight savings ends in the fall, there are ambiguous times (unless you include the offset).

It is important to have the offset information as well. Timezones that have 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 timezone, and an offset. If you are only concerned with duration, you can just use UTC time or seconds since UNIX epoch.

Let's introduce a few more terms before jumping to an example. A time without a timezone 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.

Naive time is ambiguous. 2:37 PM happens multiple times per day for each timezone.

1:29 AM US/Mountain might seem specific enough, but it is context-dependent. On the first Sunday in November, you also need offset information because it is ambiguous. There is 1:29 AM MDT, then after 1:59 AM MDT comes 1:00 AM MST, and there is another 1:29 AM for MST!

A general recommendation for programmers is to store dates in UTC times and then convert them to local time as needed. The ISO 8601 format is not sufficient to store precise local dates as it supports offset but not timezone. If you need local times I suggest you store one of these two options:

- UTC date and timezone
- Local date, offset, and timezone

Note

The pandas library can support dates stored in UTC values using the `datetime64[ns]` type. It also supports local times from a single timezone. It appears to (and by appear, I mean the operation goes without failure) support multiple timezones 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 timezones, I would probably break up the data by timezone, put each timezone in its own dataframe or series.

12.2 Loading UTC Time Data

Here is a series of strings with UTC dates. Let's convert it to a date series. You need to remember to pass the `utc=True` parameter to `pd.to_datetime`:

```
>>> col = pd.Series(['2015-03-08 08:00:00+00:00',
...      '2015-03-08 08:30:00+00:00',
...      '2015-03-08 09:00:00+00:00',
...      '2015-03-08 09:30:00+00:00',
...      '2015-11-01 06:30:00+00:00',
...      '2015-11-01 07:00:00+00:00',
...      '2015-11-01 07:30:00+00:00',
...      '2015-11-01 08:00:00+00:00',
...      '2015-11-01 08:30:00+00:00',
...      '2015-11-01 08:00:00+00:00',
...      '2015-11-01 08:30:00+00:00',
...      '2015-11-01 09:00:00+00:00',
...      '2015-11-01 09:30:00+00:00',
...      '2015-11-01 10:00:00+00:00'])

>>> utc_s = pd.to_datetime(col, utc=True)
>>> utc_s
0    2015-03-08 08:00:00+00:00
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
3    2015-03-08 09:30:00+00:00
4    2015-11-01 06:30:00+00:00
                ...
9    2015-11-01 08:00:00+00:00
10   2015-11-01 08:30:00+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 have the ability to leverage the `.dt` attribute.

Let's convert this series to the *America/Denver* timezone:

```
>>> 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
3    2015-03-08 03:30:00-06:00
4    2015-11-01 00:30:00-06:00
                ...
9    2015-11-01 01:00:00-07:00
10   2015-11-01 01:30:00-07:00
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
3    2015-03-08 03:30:00-06:00
4    2015-11-01 00:30:00-06:00
                ...
9    2015-11-01 01:00:00-07:00
10   2015-11-01 01:30:00-07:00
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]
```

12.3 Loading Local Time Data

If we want to load local date information, we need to have the date, the offset, and the timezone. Let's assume that we have localtime information in one series, and offset in another:

```
>>> time = pd.Series(['2015-03-08 01:00:00',
...   '2015-03-08 01:30:00',
...   '2015-03-08 02:00:00',
...   '2015-03-08 02:30:00',
...   '2015-03-08 03:00:00',
...   '2015-03-08 02:00:00',
...   '2015-03-08 02:30:00',
...   '2015-03-08 03:00:00',
...   '2015-03-08 03:30:00',
...   '2015-11-01 00:30:00',
...   '2015-11-01 01:00:00',
...   '2015-11-01 01:30:00',
...   '2015-11-01 02:00:00',
...   '2015-11-01 02:30:00',
...   '2015-11-01 01:00:00',
...   '2015-11-01 01:30:00',
...   '2015-11-01 02:00:00',
...   '2015-11-01 02:30:00',
...   '2015-11-01 03:00:00'])

>>> offset = pd.Series([-7, -7, -7, -7, -7, -6, -6,
...    -6, -6, -6, -6, -6, -6, -6, -7, -7, -7, -7, -7])
```

We want to apply the offset to the corresponding time. The mechanism in pandas is to use `.groupby` with `.transform` to do this. (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
3    2015-03-07 19:30:07-07:00
4    2015-03-07 20:00:07-07:00
                ...
14   2015-10-31 19:00:07-06:00
15   2015-10-31 19:30:07-06:00
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, it 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
3    2015-03-08 03:30:00-06:00
4    2015-03-08 04:00:00-06:00
                ...
14   2015-11-01 01:00:00-07:00
15   2015-11-01 01:30:00-07:00
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]
```

12.4 Converting Local time to UTC

If you have a series with local time information (stored as `datetime64[ns]` and not a string), you can use the `.dt.tz_convert` method to change it to UTC time:

```
>>> local.dt.tz_convert('UTC')
0    2015-03-08 08:00:00+00:00
1    2015-03-08 08:30:00+00:00
2    2015-03-08 09:00:00+00:00
3    2015-03-08 09:30:00+00:00
```

```
4    2015-03-08 10:00:00+00:00
                ...
14   2015-11-01 08:00:00+00:00
15   2015-11-01 08:30:00+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]
```

12.5 Converting to Epochs

If you have a series with UTC or local time information, you can get the seconds past the UNIX epoch using this code:

```
>>> secs = local.view(int).floordiv(1e9).astype(int)
>>> secs
0     1425801600
1     1425803400
2     1425805200
3     1425807000
4     1425808800
         ...
14    1446364800
15    1446366600
16    1446368400
17    1446370200
18    1446372000
Length: 19, dtype: int64
```

To load epoch information into UTC use the following:

```
>>> (pd.to_datetime(secs, unit='s')
...   .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
3    2015-03-08 09:30:00+00:00
4    2015-03-08 10:00:00+00:00
                ...
14   2015-11-01 08:00:00+00:00
15   2015-11-01 08:30:00+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]
```

12.6 Manipulating Dates

To further demo date manipulation, I am going to read in a dataset with snowfall levels from a local ski resort.

```
>>> url = 'https://github.com/mattharrison/datasets'+\
...     '/raw/master/data/alta-noaa-1980-2019.csv'
>>> alta_df = pd.read_csv(url)
```

I'm going to show working with a series with date information in them. Then we will look at a series that has 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)
>>> dates
0       1980-01-01
1       1980-01-02
2       1980-01-03
3       1980-01-04
4       1980-01-05
           ...
14155   2019-09-03
14156   2019-09-04
14157   2019-09-05
14158   2019-09-06
14159   2019-09-07
Name: DATE, Length: 14160, dtype: datetime64[ns]
```

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 useful.

Note that the type of `date` is `datetime64[ns]`. This gives us some super powers. It adds a `.dt` attribute to the series that allows 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
3          Viernes
4           Sábado
           ...
14155       Martes
14156    Miércoles
14157       Jueves
14158      Viernes
14159       Sábado
Name: DATE, Length: 14160, dtype: object
```

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 are they properties and not methods? A property is not parameterizable. You just get back the

results. Also note, that you do not put parentheses at the end of a property (ie, you do not *call* it). If you do, you will get an error stating that it is not callable.

The creators of the properties felt that there were no options to them. For example, `.is_month_end` just 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
3         False
4         False
          ...
14155     False
14156     False
14157     False
14158     False
14159     False
Name: DATE, Length: 14160, dtype: bool
```

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
3         04/01/80
4         05/01/80
            ...
14155     03/09/19
14156     04/09/19
14157     05/09/19
14158     06/09/19
14159     07/09/19
Name: DATE, Length: 14160, dtype: object
```

Code	*Meaning*	*Sample*
`%y`	Year (decimal)	14
`%Y`	Year (century)	2014
`%m`	Month (padded)	08
`%b`	Month (Abbrev locale)	Aug
`%B`	Month	August
`%d`	Day (padded)	04
`%a`	Weekday (Abbrev locale)	Mon
`%A`	Weekday (locale)	Monday
`%H`	Hour (24 padded)	22
`%I`	Hour (12 padded)	10
`%M`	Minutes (padded)	25
`%S`	Seconds (padded)	24
`%p`	AM/PM	PM
`%-d`	Day (unpadded unix*)	4
`%e`	Day (unpadded unix*)	4
`%c`	Locale representation	Mon Aug 4 22:25:24 2014
`%x`	Locale date	08/04/14
`%X`	Locale time	22:25:24
`%W`	Week num (Mon 1st)	31
`%U`	Week num (Sun 1st)	31
`%j`	Day of year (padded)	216
`%z`	UTC offset	+0000
`%Z`	Time Zone	MDT
`%%`	Percent sign	%

Figure 12.2: Table of strftime codes

Below is a table of `.dt` methods and properties.

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

`.is_year_start`	Property with a series of booleans if date is start of year.
`.microsecond`	Property with a series of microseconds of date.
`.minute`	Property with a series of minutes of date.
`.month`	Property with a series of month of date (numeric).
`.month_name(locale='en_us')`	Return a series of month of date (string).
`.nanosecond`	Property with a series of nanoseconds of date.
`.normalize()`	Return a series of dates converted to midnight.
`.quarter`	Property with series of quarter of date (numeric 1-4).
`.round(freq=None, ambiguous=None, nonexistent=None)`	Return round according to fixed frequency (cannot be end like `'ME'`) in `freq`. The `nonexistent` parameter controls DST time issues.
`.second`	Property with a series of seconds of date (numeric).
`.strftime(date_format)`	Return a series with string dates. Formatted using strftime format codes.
`.time`	Property with a series of Python `datetime.time` objects.
`.timetz`	Property with a series of Python `datetime.time` objects with timezone information.
`.to_period(freq)`	Return a series with pandas `Period` objects.
`.to_pydatetime()`	Return a numpy array with `datetime.datetime` objects.
`.tz`	Property with timezone.
`.tz_convert(tz)`	Convert from one timezone aware series to another.
`.tz_localize(tz, ambiguous=None, nonexistent=None)`	Convert from naive to timezone aware.
`.week`	Property with a series of week of date (numeric 1-53).
`.weekday`	Property with a series of date of week as number 0 is Monday.
`.weekofyear`	Property with a series of week of date (numeric 1-53).
`.year`	Property with a series of year of date.

Table 12.1: `.dt` methods and Properties

12.7 Summary

In the chapter, we explored converting series into date series. We discussed timezones, offsets, local time, and UTC time. If you have UTC time, you can convert it into a timezone. If you have local time, you will need an offset information to convert it into a timezone (as many local times have ambiguous times). If you have a series with multiple timezone dates in it, I recommend leaving it as UTC because pandas will not allow you to work on the dates unless you split them out into one timezone.

12.8 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 timezone.
4. Convert a date column into epoch values.

5. Convert an epoch number into UTC.

Chapter 13

Dates in the Index

If you have dates in the index, you can do some powerful manipulation and aggregation of your data.

We are going to shift gears and look at data that has a date as an index. We will look at the amount of snow that fell each day at the ski resort:

```
>>> snow = (alta_df
...     .SNOW
...     .rename(dates)
... )

>>> snow
1980-01-01    2.0
1980-01-02    3.0
1980-01-03    1.0
1980-01-04    0.0
1980-01-05    0.0
              ...
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64
```

13.1 Finding Missing Data

Let's look for missing data. There are a few methods that help with dealing with missing data in time data. We can check is 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   NaN
1985-09-12   NaN
1985-09-19   NaN
1986-02-07   NaN
1986-06-26   NaN
              ..
2017-04-26   NaN
```

```
2017-09-20   NaN
2017-10-02   NaN
2017-12-23   NaN
2018-12-03   NaN
Name: SNOW, Length: 365, dtype: float64
```

With a date index, we can provide partial date strings to the `.loc` indexing attribute. This will let us inspect around 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
1985-09-04    0.0
1985-09-05    0.0
              ...
1985-09-16    0.0
1985-09-17    0.0
1985-09-18    0.0
1985-09-19    NaN
1985-09-20    0.0
Name: SNOW, Length: 20, dtype: float64
```

13.2 Filling In Missing Data

Often we have time series data with missing values. For example, in the snow data, the value for the date 1985-09-19 is missing. (See previous code.)

This value looks like it could be filled in with zero (as this is the end of summer):

```
>>> (snow
...    .loc['1985-09':'1985-09-20']
...    .fillna(0)
... )
1985-09-01    0.0
1985-09-02    0.0
1985-09-03    0.0
1985-09-04    0.0
1985-09-05    0.0
              ...
1985-09-16    0.0
1985-09-17    0.0
1985-09-18    0.0
1985-09-19    0.0
1985-09-20    0.0
Name: SNOW, Length: 20, dtype: float64
```

However, these values in January, the middle of the winter, might not be zero. (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 do with missing data is the 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    NaN
1988-01-02    0.0
1988-01-03    0.0
```

```
             ...
1988-01-06    6.0
1988-01-07    4.0
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: float64
```

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 back fill 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
1988-01-02    0.0
1988-01-03    0.0
             ...
1988-01-06    6.0
1988-01-07    4.0
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: float64

>>> (snow
...      .loc['1987-12-30':'1988-01-10']
...      .bfill()
... )
1987-12-30    6.0
1987-12-31    5.0
1988-01-01    0.0
1988-01-02    0.0
1988-01-03    0.0
             ...
1988-01-06    6.0
1988-01-07    4.0
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: float64
```

13.3 Interpolation

We can also interpolate using `.interpolate`. By default this does a linear interpolation for the missing values:

```
>>> (snow
...      .loc['1987-12-30':'1988-01-10']
...      .interpolate()
... )
1987-12-30    6.0
```

```
1987-12-31    5.0
1988-01-01    2.5
1988-01-02    0.0
1988-01-03    0.0
              ...
1988-01-06    6.0
1988-01-07    4.0
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, Length: 12, dtype: float64
```

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 it.)

This is a good example of the `.where` method. Here is a truth table for *winter* and *snow* values.

Winter	*Snow*
True (I)	True (II)
False (III)	False (IV)

When it is winter and we are missing snow values, we will interpolate. This corresponds to sections I and IV. When it is not winter and snow values are missing we will fill in 0 (sections III and IV). Recall that the `.where` method keeps values where the first parameter is `True`, so we invert the conditions with `~`:

```
>>> winter = (snow.index.quarter == 1) | (snow.index.quarter== 4)
>>> (snow
...     .where(~(winter & snow.isna()), snow.interpolate())
...     .where(~(~winter & snow.isna()), 0)
... )
1980-01-01    2.0
1980-01-02    2.5
1980-01-03    1.0
1980-01-04    0.0
1980-01-05    0.0
              ...
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64
```

And we can validate the values to make sure that it worked:

```
>>> (snow
...     .where(~(winter & snow.isna()), snow.interpolate())
...     .where(~(~winter & snow.isna()), 0)
...     .loc[['1985-09-19','1988-01-01']]
... )
1985-09-19    0.0
1988-01-01    2.5
Name: SNOW, dtype: float64
```

Note

These `.where` statements can get confusing with double negatives. I like to work in Jupyter, where I can quickly try code and validate the results. Please do likewise!

13.4 Dropping Missing Values

We can also just 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
1988-01-03    0.0
1988-01-05    2.0
1988-01-06    6.0
1988-01-07    4.0
1988-01-08    9.0
1988-01-09    5.0
1988-01-10    2.0
Name: SNOW, dtype: float64
```

Be careful with the method and only use it after talking to a subject matter expert who confirms that it is ok 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.

13.5 Shifting Data

We can shift data up or down, which is useful for sequence data like time series. This method works on any pandas series but comes in really 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    NaN
1980-01-02    2.0
1980-01-03    3.0
1980-01-04    1.0
1980-01-05    0.0
              ...
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64

>>> snow.shift(-1)
1980-01-01    3.0
1980-01-02    1.0
1980-01-03    0.0
1980-01-04    0.0
1980-01-05    1.0
              ...
```

```
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    NaN
Name: SNOW, Length: 14160, dtype: float64
```

13.6 Rolling Average

To calculate the five day moving average, we can leverage `.shift` and do the following:

```
>>> (snow
...   .add(snow.shift(1))
...   .add(snow.shift(2))
...   .add(snow.shift(3))
...   .add(snow.shift(4))
...   .div(5)
... )
1980-01-01    NaN
1980-01-02    NaN
1980-01-03    NaN
1980-01-04    NaN
1980-01-05    1.2
              ...
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64
```

That was a little tedious to write. Thankfully, pandas has a trick up its sleeve. There is a `.rolling` method that allows us to specify a window size. This method returns a `Rolling` object that we can apply various aggregate methods to. If we apply `.mean` to it, we get a very similar result to above:

```
>>> (snow
...   .rolling(5)
...   .mean()
... )
1980-01-01    NaN
1980-01-02    NaN
1980-01-03    NaN
1980-01-04    NaN
1980-01-05    1.2
              ...
2019-09-03    0.0
2019-09-04    0.0
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:

Method	*Description*
`r.agg(func=None, axis=0, *args, **kwargs)`	Returns a scalar if `func` is a single aggregation function. Returns a series if a list of aggregations are passed to `func`. (`aggregate` is a synonym.)

`r.apply(func, args=None, kwargs=None)`	Apply custom aggregation function to rolling group.
`r.corr(other, method='pearson')`	Returns correlation coefficient for `'pearson'`, `'spearman'`, `'kendall'`, or a callable.
`r.count(other, method='pearson')`	Returns count of non `NaN` values.
`r.cov(other, min_periods=None)`	Returns covariance.
`r.max(axis=None, skipna=None, level=None, numeric_only=None`	Returns maximum value.
`r.min(axis=None, skipna=None, level=None, numeric_only=None)`	Returns minimum value.
`r.mean(axis=None, skipna=None, level=None, numeric_only=None)`	Returns mean value.
`r.median(axis=None, skipna=None, level=None, numeric_only=None)`	Returns median value.
`r.quantile(q=.5, interpolation='linear')`	Returns 50% quantile by default. *Note* returns `Series` if `q` is a list.
`r.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased standard error of mean.
`r.std(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns sample standard deviation.
`r.var(axis=None, skipna=None, level=None, ddof=1, numeric_only=None)`	Returns unbiased variance.
`r.skew(axis=None, skipna=None, level=None, numeric_only=None)`	Returns unbiased skew.

Table 13.1: Rolling methods and properties

13.7 Resampling

Because this series has dates as the index, it has more super powers. We can use the `.resample` method to 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('M')
...     .max()
... )
1980-01-31    20.0
1980-02-29    25.0
1980-03-31    16.0
1980-04-30    10.0
1980-05-31     9.0
              ...
2019-05-31     5.1
2019-06-30     0.0
2019-07-31     0.0
2019-08-31     0.0
2019-09-30     0.0
Freq: M, Name: SNOW, Length: 477, dtype: float64
```

The `'M'` string in the `.resample` call is what pandas calls an *offset alias*. This is a string that specifies a grouping frequency. Using *M* means group all values by the end of the month. If you look at the

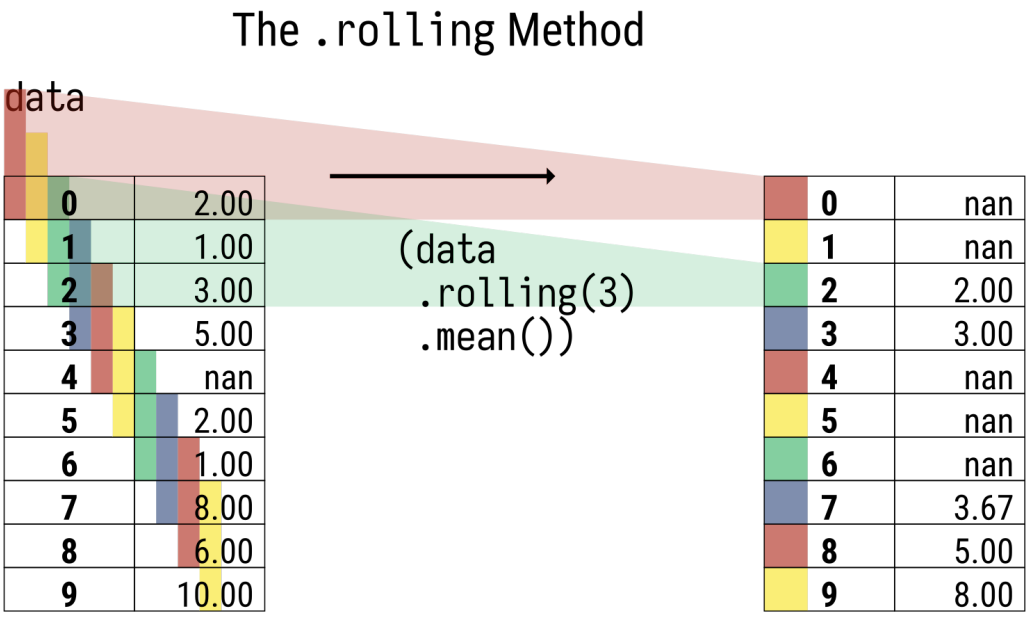


Figure 13.1: The `.rolling` method slides a window along the data, allowing you to call an aggregate function.

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 `'2M'` as the offset alias:

```
>>> (snow
...     .resample('2M')
...     .max()
... )
1980-01-31    20.0
1980-03-31    25.0
1980-05-31    10.0
1980-07-31     1.0
1980-09-30     0.0
              ...
2019-01-31    19.0
2019-03-31    20.7
2019-05-31    18.0
2019-07-31     0.0
2019-09-30     0.0
Freq: 2M, Name: SNOW, Length: 239, dtype: float64
```

If we want to aggregate the maximum value for each ski season, which normally ends in May, we could use the following code. This offset alias, `'A-MAY'`, indicates that we want an annual grouping (`'A'`), but ending in May of each year:

```
>>> (snow
...     .resample('A-MAY')
...     .max()
... )
1980-05-31    25.0
1981-05-31    26.0
```

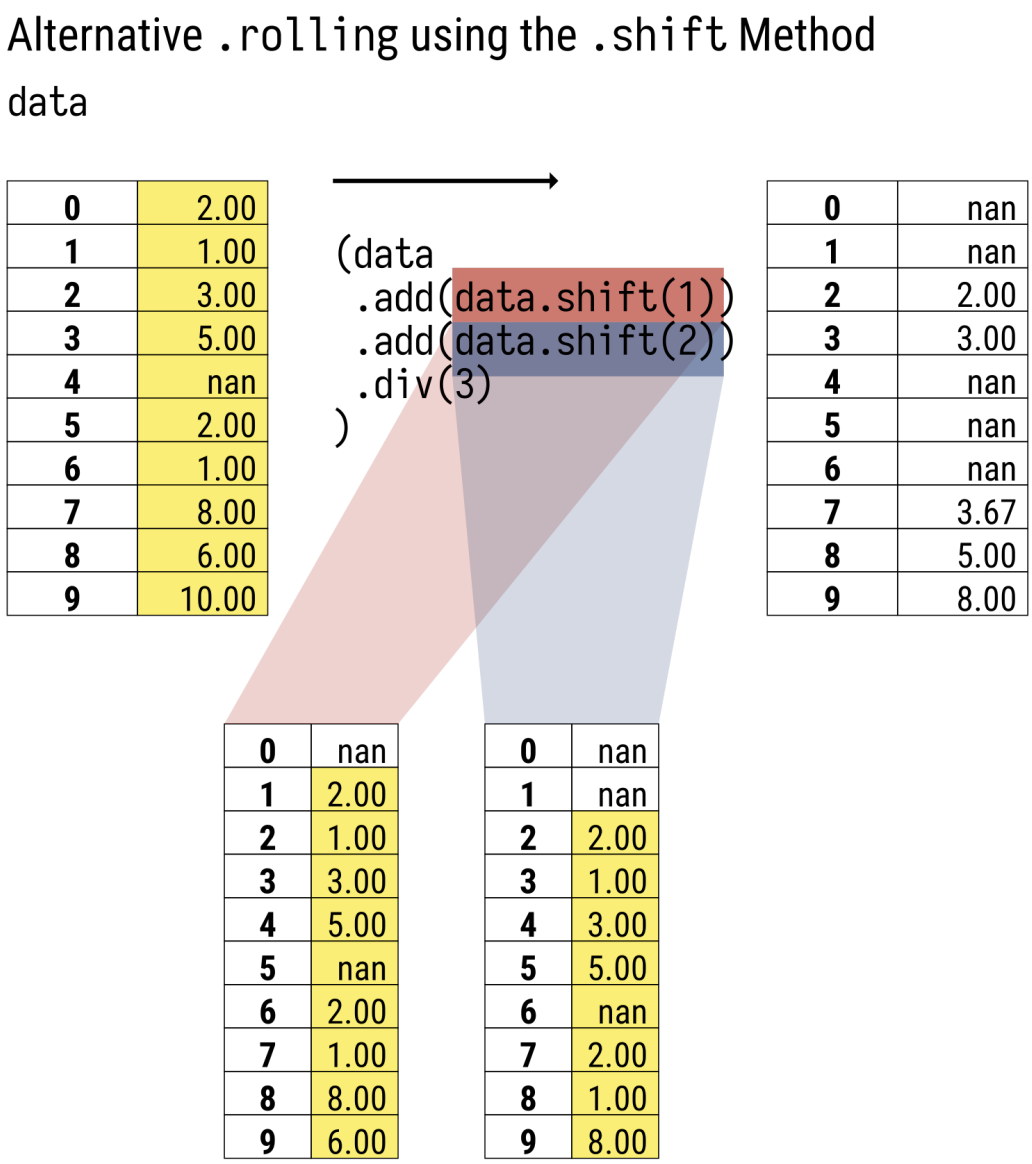


Figure 13.2: The `.rolling` method slide is similar to shifting the data for N-1 window size and then applying an aggregation.

```
1982-05-31    34.0
1983-05-31    38.0
1984-05-31    25.0
              ...
2016-05-31    15.0
2017-05-31    26.0
2018-05-31    21.8
2019-05-31    20.7
2020-05-31     0.0
Freq: A-MAY, Name: SNOW, Length: 41, dtype: float64
```

Below is a table of the offset aliases.

Offset Alias	*Date Offset*	*Description*
None	DateOffset	Default 1 day
'B'	BDay	Business day (weekday)
'C'	CDay	Custom business day
'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
'M'	MonthEnd	Month end
'MS'	MonthBegin	Month start
'BM'	BMonthEnd	Business month end
'BMS'	BMonthBegin	Business month start
'CBM'	CBMonthEnd	Custom business month end
'CBMS'	CBMonthBegin	Custom business month start
'SM'	SemiMonthEnd	Semi-month end (15th and month end)
'SMS'	SemiMonthBegin	Semi-month start (15th and month start)
'Q'	QuarterEnd	Quarter end (Can specify -JAN to end quarter in January)
'QS'	QuarterBegin	Quarter start
'BQ'	BQuarterEnd	Business quarter end
'BQS'	BQuarterBegin	Business quarter start
'REQ'	FY5253Quarter	Retail quarter end (52-53 week)
'A'	YearEnd	Calendar year end (Can specify -MAY to end year in May)
'AS' / 'BYS'	YearBegin	Calendar year start
'BA'	BYearEnd	Business year end
'BAS'	BYearBegin	Business year start
'RE'	FY5253	Retail year end (52-53 week)
'BH'	BusinessHour	Business hour
'CBH'	CustomBusinessHour	Custom business hour
'D'	Day	Day
'H'	Hour	Hour
'T' / 'min'	Minute	Minute
'S'	Second	Second
'L' / 'ms'	Milli	Millisecond
'U' / 'us'	Micro	Microsecond
'N'	Nano	Nanosecond

Figure 13.3: Offset aliases and date offset classes for `Grouper` and `.resample`

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.

13.8 Gathering Aggregate Values (But Keeping Index)

Below, instead of performing an aggregation with `.resample`, we leverage the `.transform` method, which works on aggregation groups but returns a series with the original index. This makes it easy to do things like calculate the percentage of quarterly snowfall the fell in a day:

```
>>> (snow
...     .div(snow
...            .resample('Q')
...            .transform('sum'))
...     .mul(100)
...     .fillna(0)
... )
1980-01-01    0.527009
1980-01-02    0.790514
1980-01-03    0.263505
1980-01-04    0.000000
1980-01-05    0.000000
                ...
2019-09-03    0.000000
2019-09-04    0.000000
2019-09-05    0.000000
2019-09-06    0.000000
2019-09-07    0.000000
Name: SNOW, Length: 14160, dtype: float64
```

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('M')
...      .sum()
...      .div(season2017
...             .sum())
...      .mul(100)
... )
2016-10-31     2.153969
2016-11-30     9.772637
2016-12-31    15.715995
2017-01-31    25.468688
2017-02-28    21.041085
2017-03-31     9.274033
2017-04-30    14.738732
2017-05-31     1.834862
Freq: M, Name: SNOW, dtype: float64
```

Here is a table of the operations you can use on a resample object.

Method	*Description*

`.agg(func, *args, **kwargs)`	Apply a function (to the group), string function name, list of functions, or dictionary (mapping column names to previous function/string/list). Returns a series if called with a single function, otherwise return a dateframe for multiple functions.
`.aggregate(func, *args, **kwargs)`	Same as `.agg`
`.apply(func, *args, **kwargs)`	Same as `.agg`
`.asfreq(fill_value=None)`	Return values at frequency (like `.reindex`)
`.backfill(limit=None)`	Backfill the missing values.
`.bfill(limit=None)`	Same as `.backfill`
`.count()`	Count of non-missing items in group.
`.ffill(limit=None)`	Forward fill the missing values.
`.fillna(method, limit=None)`	Method (`'ffill'`, `'bfill'`, or `'nearest'`) to use for filling in missing data for upsampling.
`.first()`	Return a series with the first value of each group.
`.get_group(name, obj=None)`	Return the series for grouping frequency of name.
`.interpolate(method='linear', axis=0, limit=None, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)`	Return a series with interpolated values.
`.last()`	Return a series with the final value from each group.
`.max()`	Return a series with maximum value from each group.
`.mean()`	Return a series with mean value from each group.
`.median()`	Return a series with median value from each group.
`.min()`	Return a series with minimum value from each group.
`.nearest(limit=None)`	Fill the missing values with nearest.
`.ngroups`	Property with number of groups in aggregation.
`.nunique()`	Return a series with the number of unique values from each group.
`.ohlc()`	Return a dataframe with columns for open, high, low, close.
`.pad(limit=None)`	Same as `.ffill`.
`.pipe(func, *args, **kwargs)`	Apply function to resampler object.
`.plot()`	Plot the groups.
`.prod()`	Return a series with the product of each group.
`.quantile(q=0.5)`	Return a series with the quantile. If `q` is a list, return a multi-index series.
`.sem()`	Return a series with the standard error of mean of each group.
`.size()`	Return a series with the size of each group (number of rows including missing values).
`.std()`	Return a series with the standard deviation of each group.
`.sum()`	Return a series with the sum of each group.
`.transform(function, *args, **kwargs)`	Return a series with the same index as the original (not grouped series). Function takes a group and returns a group with the same index.

`.var()`	Return a series with the variance of each group.

Table 13.2: Resampler Methods on a Series

13.9 Groupby Operations

There is also a `.groupby` method that acts as a generic sort of `.resample`, and I use this more on dataframes than series. But here is an example of creating a function that will determine ski season by looking at the index with date information. It considers a season to be from October to September:

```
>>> def season(idx):
...     year = idx.year
...     month = idx.month
...     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 total snowfall for each season:

```
>>> (snow
...   .groupby(season)
...   .sum()
... )
1980    457.5
1981    503.0
1982    842.5
1983    807.5
1984    816.0
        ...
2015    284.3
2016    354.6
2017    524.0
2018    308.8
2019    504.5
Name: SNOW, Length: 40, dtype: float64
```

Note

We could also do the above with `.resample` using an anchored offset alias. The index would be a date instead of an integer:

```
>>> (snow
...   .resample('A-SEP')
...   .sum()
... )
1980-09-30    457.5
1981-09-30    503.0
1982-09-30    842.5
1983-09-30    807.5
1984-09-30    816.0
              ...
2015-09-30    284.3
2016-09-30    354.6
2017-09-30    524.0
2018-09-30    308.8
2019-09-30    504.5
Freq: A-SEP, Name: SNOW, Length: 40, dtype: float64
```

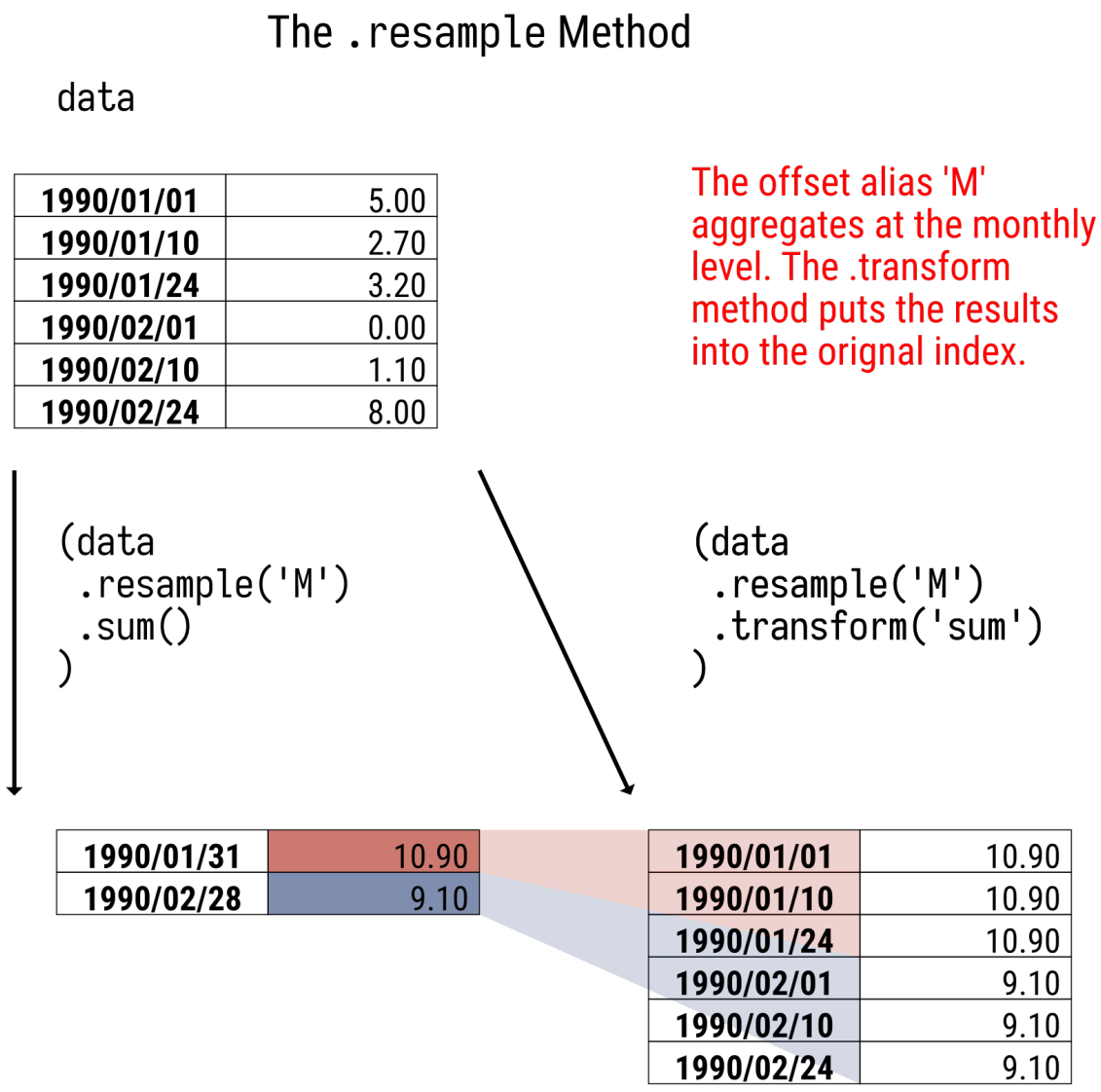


Figure 13.4: If you have dates in the index, you can use the `.resample` method to aggregate at date frequencies. The `.transform` method will take the resulting aggregates and place them back in the cell that contributed to the value (with the original index).

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.

13.10 Cumulative Operations

There are also a handful of cumulative methods that 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
2016-10-04      4.9
2016-10-05      5.5
               ...
2017-09-26    524.0
2017-09-27    524.0
2017-09-28    524.0
2017-09-29    524.0
2017-09-30    524.0
Name: SNOW, Length: 364, dtype: float64
```

Alternatively, it we wanted to do this calculation for every year, we can combine `.resample` with `.transform` and `'cumsum'`:

```
>>> (snow
...     .resample('A-SEP')
...     .transform('cumsum')
... )
1980-01-01      2.0
1980-01-02      5.0
1980-01-03      6.0
1980-01-04      6.0
1980-01-05      6.0
               ...
2019-09-03    504.5
2019-09-04    504.5
2019-09-05    504.5
2019-09-06    504.5
2019-09-07    504.5
Name: SNOW, Length: 14160, dtype: float64
```

Method	*Description*
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=None, format=None, exact=True, unit='ns', infer_datetime_format=False, origin='unix', cache=True)`	Convert `arg` to date index, series, or timestamp for list, series, or scalar. Set `errors` to `'coerce'` to have invalid be `NaT`, `'ignore'` to leave. Specify strftime format with `format` or set `infer_datetime_format` to `True` if only one format type.

`.isna()`	Return boolean array (series) indicating where values are missing.
`.fillna(value=None, method=None, limit=None, downcast=None)`	Return series with missing values set to `value` (scalar, dict, series). Use `method` to fill additional holes (`'bfill'` or `'ffill'`) only `limit` times. Provide `downcast='infer'` to convert float to int if possible.
`.loc`	If index is datetime, can use partial string indexing. `'2010'` to select all of 2010. `'2010-10'` to select Oct 2010. Stop index includes that stopping period. Indexing with `Timestamp` and `datetime` objects is not partial.
`.ffill(limit=None)`	Forward fill the missing values.
`.bfill(limit=None)`	Forward fill the missing values.
`.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)`	Return a series with interpolated values.
`.where(cond, other=nan, level=None, errors='raise', try_cast=False)`	Return a series with values replaced with `other` where `cond` is `False`. `cond` can be boolean array or function (series passed in, return boolean array). `other` can be scalar, series, or function (series passed in, return scalar or series).
`.dropna()`	Return a series with missing values removed.
`.shift(periods=1, freq=None, fill_value=None)`	Return a series with data shifted forward by `periods` (can be negative). If time series and `freq` is offset alias, index values are shifted to offset alias. Fill in empty values with `fill_value`.
`.rolling(window, min_periods=None, center=False, win_type=None, closed='right')`	Return a `Window` or `Rolling` class to aggregate. `window` is number windows, offset alias (for time series), or `BaseIndexer`. Set `center=True` to label at center of window. To use non-evenly weighted window, set `win_type` to string with Scipy window type.
`.resample(rule, closed='left', label='left', convention='start', kind=None, level=None, origin='start_day', offset=None)`	Return `Resampler` object to aggregate on. Use `rule` to specify `DateOffset`, `TimeDelta`, or offset alias string.
`.transform(func)`	Return a series with same index but with transformed values. Best when used on a `.groupby` or `.resample` result. `func` may be an aggregation function or string when called on groupby or resample.
`.groupby(by=None, level=None, sort=True, group_keys=True, observed=False, dropna=True)`	Return a groupby object to aggregate on. `by` may be a function (pass the index, return label), mapping (dict or series that maps index to label), or a sequence of labels. Use `observed=True` to limit combinatoric explosion with categorical series.
`.cummax(skipna=True)`	Return cumulative maximum of series
`.cummin(skipna=True)`	Return cumulative minimum of series
`.cumprod(skipna=True)`	Return cumulative product of series

`.cumsum(skipna=True)`	Return cumulative sum of series

Table 13.3: Date Manipulation Methods

13.11 Summary

In this chapter, we 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.

13.12 Exercises

With a dataset of your choice:

1. Convert a column with date information to a date.
2. Put the date information into the index for a numeric column.
3. Calculate the average value of the column for each month.
4. Calculate the average value of the column for every 2 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 3 months of a year.
8. Using `.loc` pull out the last 4 months of a year.

Chapter 14

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 huge fan of plotting, as it 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 huge benefit to you.

In this chapter, we will explore how to create plots from series with pandas.

14.1 Plotting in Jupyter

Pandas has native integration with Matplotlib. To leverage it in Jupyter, make sure you include the following cell magic to tell Jupyter to display the plots in the browser:

```
%matplotlib inline
```

14.2 The `.plot` Attribute

A series object has a `.plot` attribute. This attribute is interesting as you can call it directly to create plots, or access sub-attributes of it. Let's load the snow data and create some plots:

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/'\
...     'data/alta-noaa-1980-2019.csv'
>>> alta_df = pd.read_csv(url)
>>> 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
1980-01-04    0.0
1980-01-05    0.0
              ...
2019-09-03    0.0
2019-09-04    0.0
2019-09-05    0.0
2019-09-06    0.0
2019-09-07    0.0
Name: SNOW, Length: 14160, dtype: float64
```

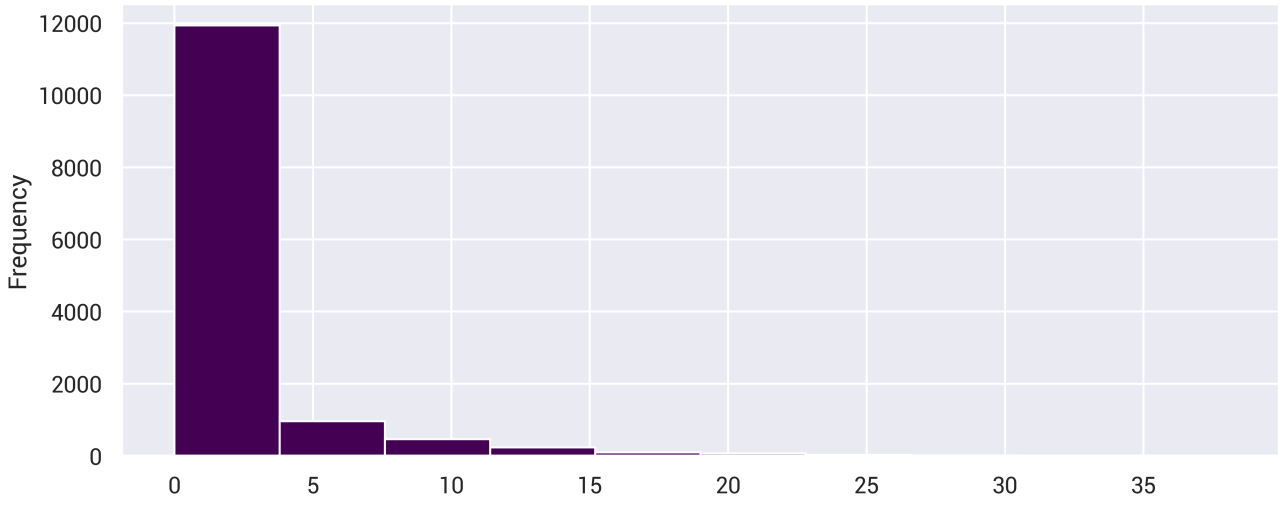


Figure 14.1: Basic histogram.

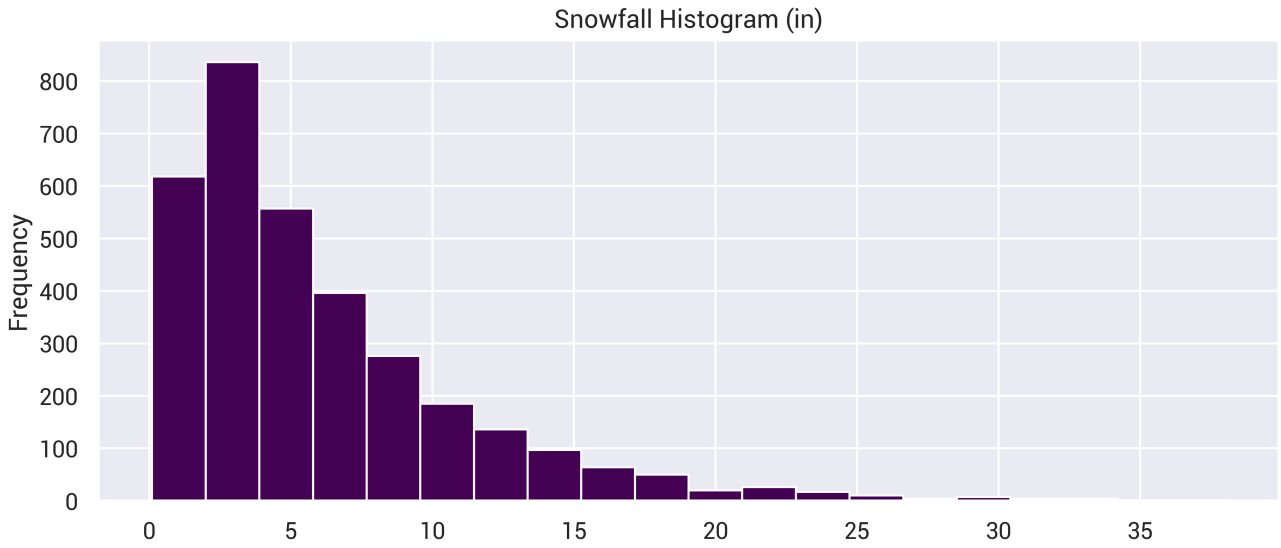


Figure 14.2: Histogram with zero values filtered out and 20 bins.

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

14.3 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)')
```

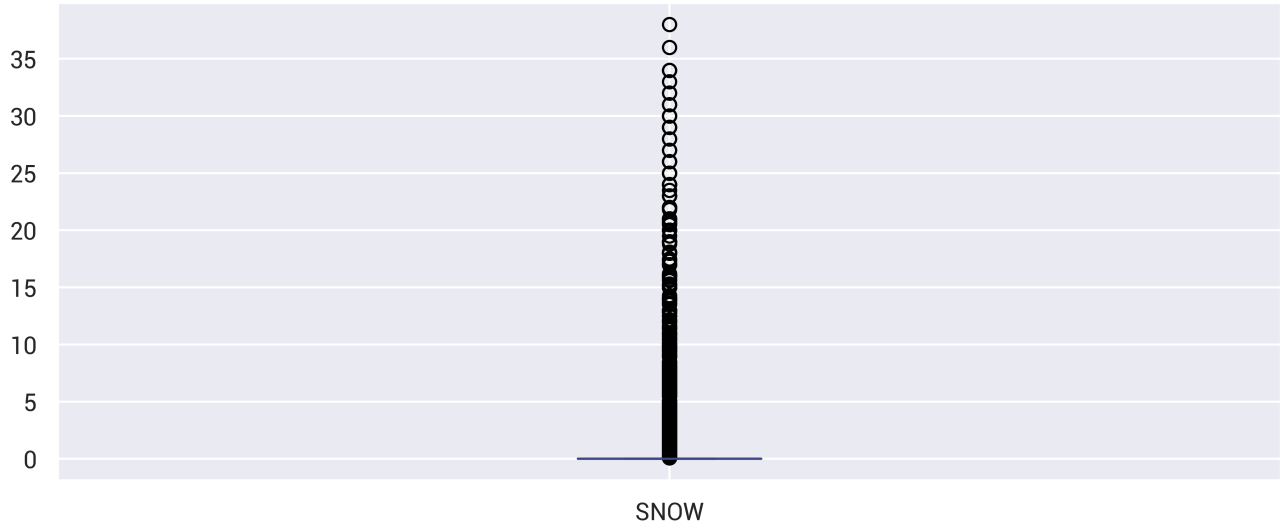


Figure 14.3: Basic boxplot.

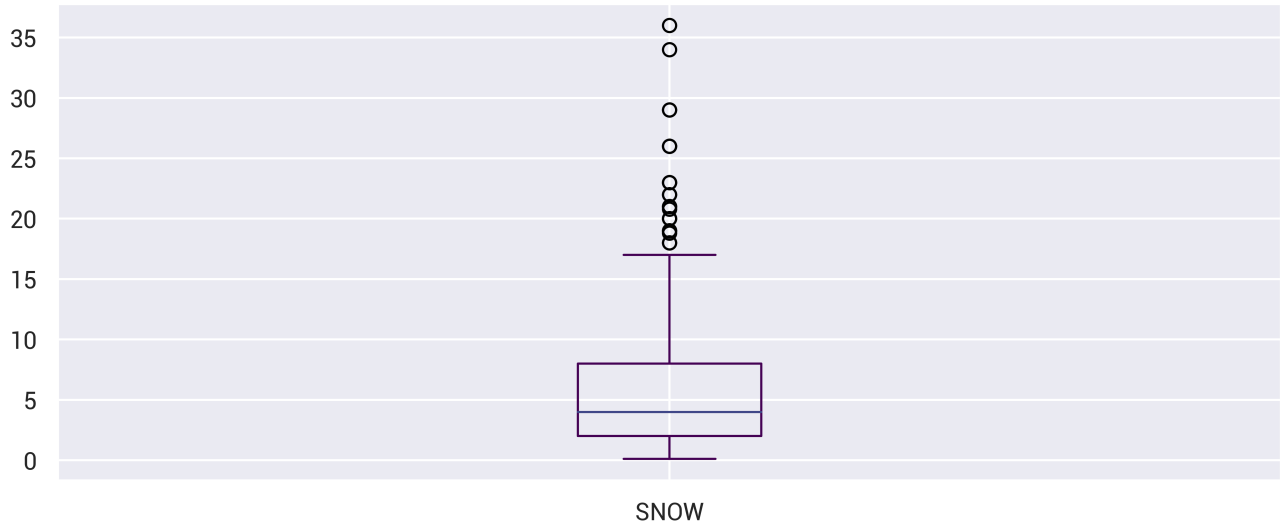


Figure 14.4: A better basic boxplot with snowfall levels for each January.

14.4 Box Plot

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

```
>>> snow.plot.box()
```

It looks more boxy if we limit it to snow amounts during January (ignoring zero):

```
>>> (snow
...     [lambda s:(s.index.month == 1) & (s>0)]
...     .plot.box()
... )
```

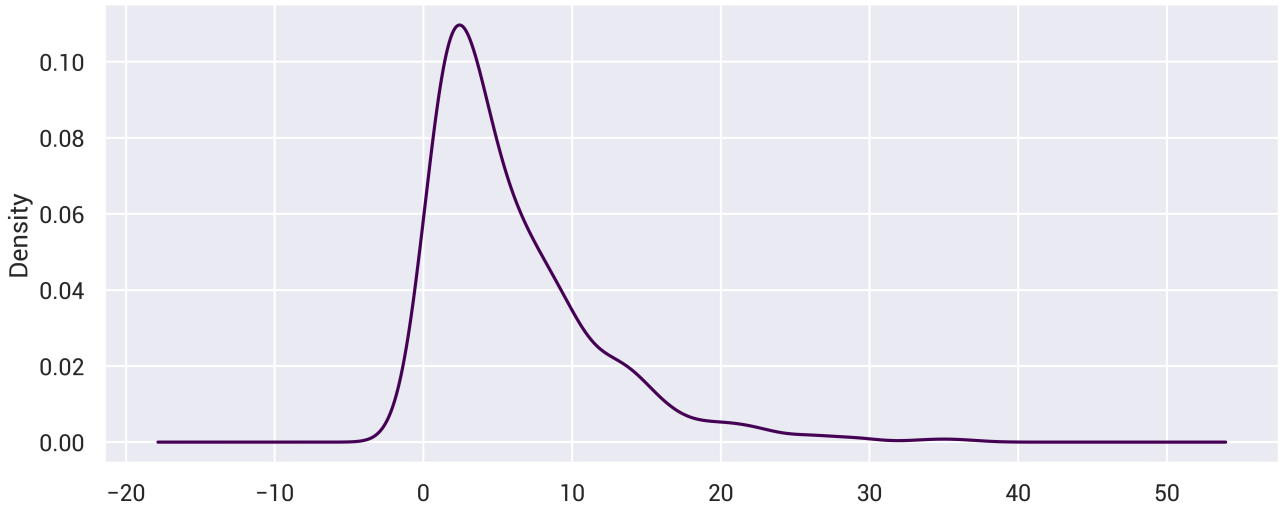


Figure 14.5: A basic kernel density estimate plot.

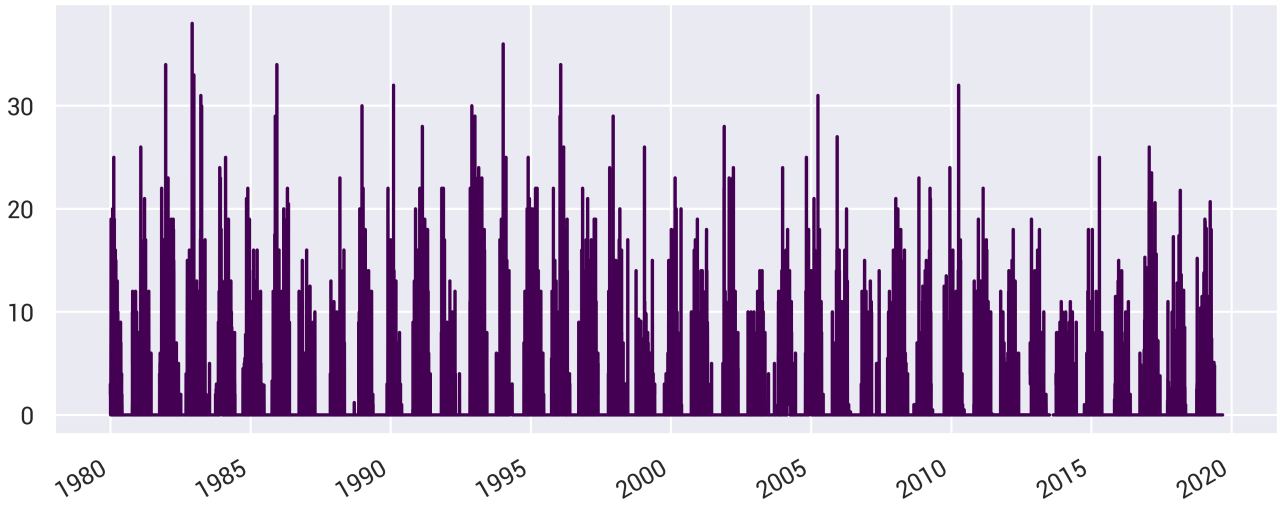


Figure 14.6: Basic line plot.

14.5 Kernel Density Estimation Plot

Another option to view the kernel density estimation (KDE). This is essentially a smoothed histogram:

```
>>> (snow
...     [lambda s:(s.index.month == 1) & (s>0)]
...     .plot.kde()
... )
```

14.6 Line Plots

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

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

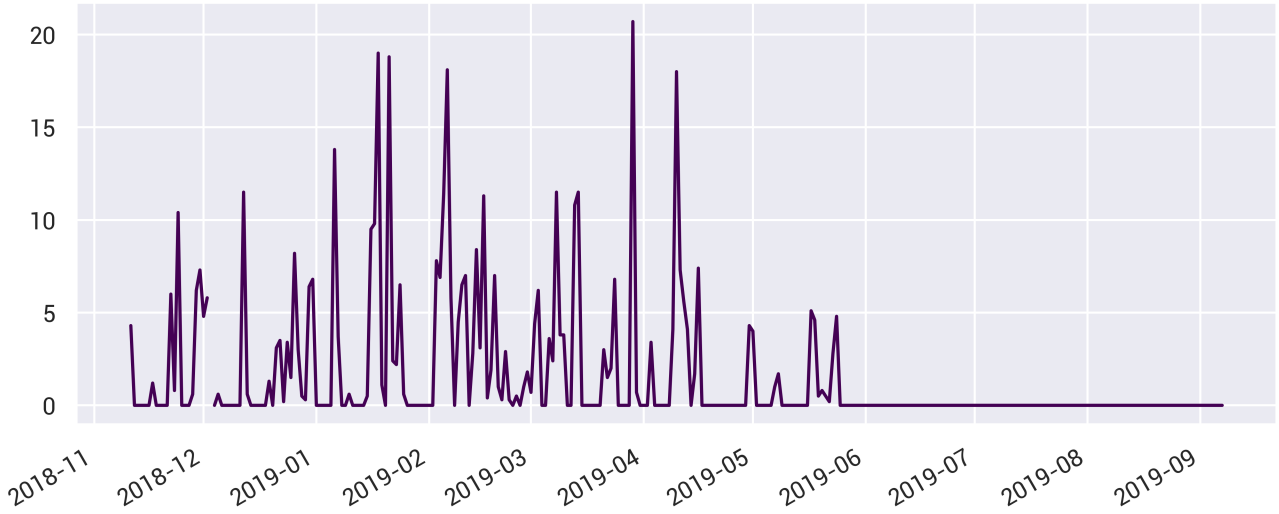


Figure 14.7: Last few values of basic line plot.

A line plot in pandas plots the values in the series in the y-axis and the index in the x-axis. This plot is a little crowded as we are packing 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
...    .iloc[-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.

Here I'm going to aggregate at the monthly level and look at the mean snowfall using `.resample` with the `'M'` offset alias and the `.mean` aggregation method:

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

14.7 Line Plots with Multiple Aggregations

Plotting can be even more powerful with dataframes. To give you an idea, we will use the `.quantile` method to pull out the 50%, 90%, and 99% values. This returns a series with multiindex (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('Q')
...  .quantile([.5, .9, .99])
...  .unstack()
...  .iloc[-100:]
...  .plot.line()
... )
```

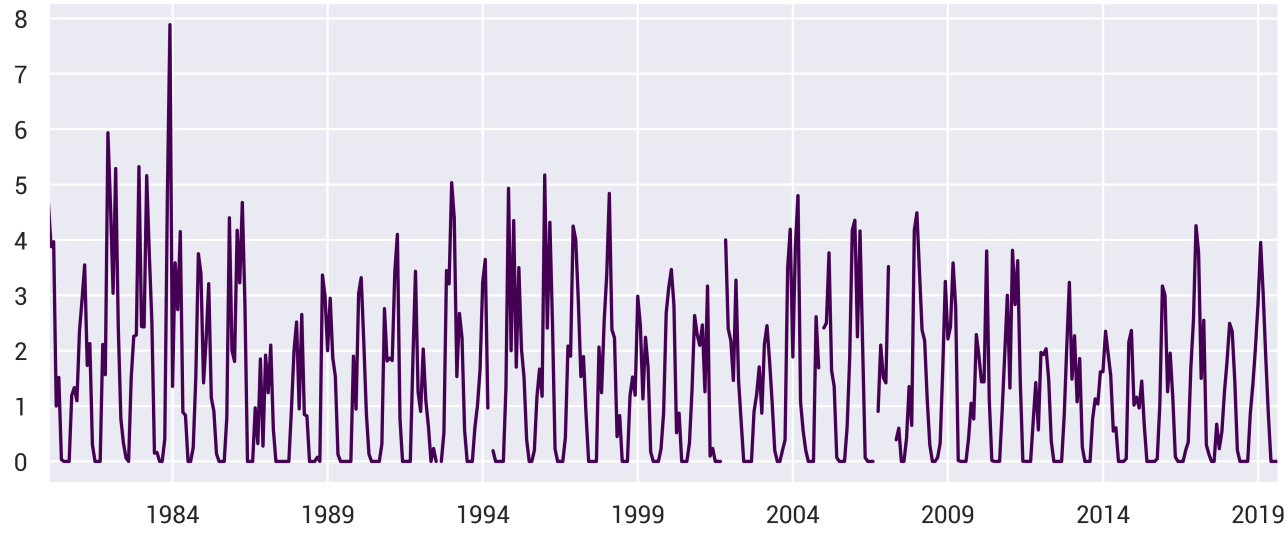


Figure 14.8: Resampled line plot.

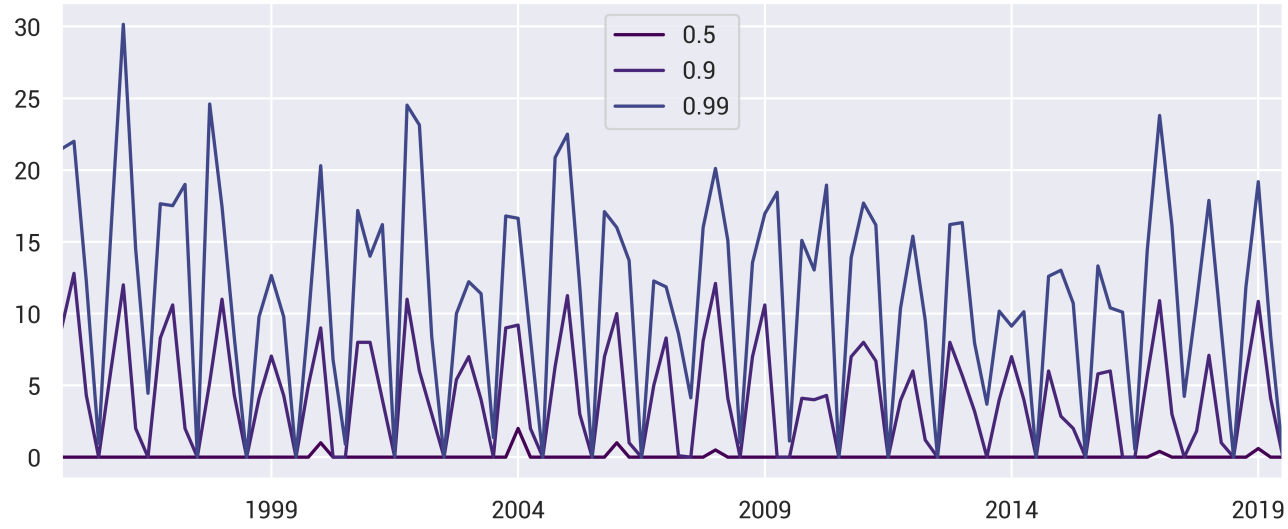


Figure 14.9: Resampled line plot from dataframe.

14.8 Bar Plots

You can also create bar plots. These are useful for comparing values. In the previous section, we looked at the percent of snow that fell during each month:

```
>>> season2017 = (snow.loc['2016-10':'2017-05'])
>>> (season2017
...    .resample('M')
...    .sum()
...    .div(season2017.sum())
...    .mul(100)
...    .rename(lambda idx: idx.month_name())
... )
October      2.153969
November     9.772637
```

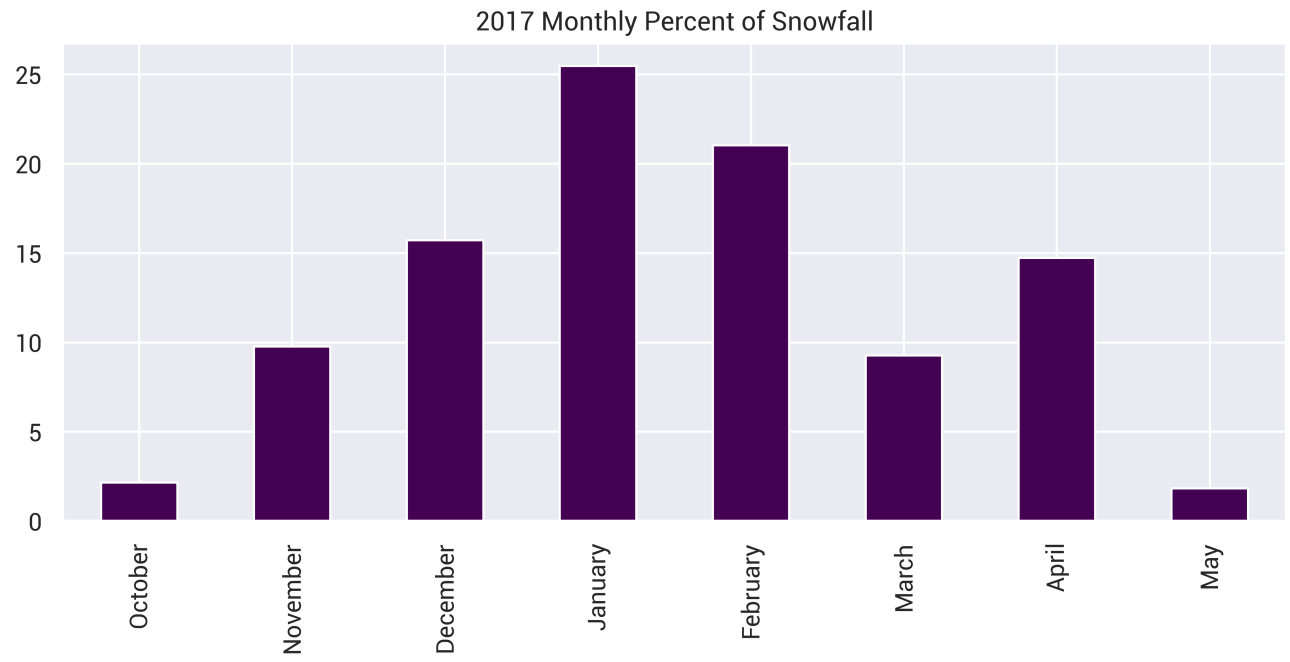


Figure 14.10: Basic series bar plot.

```
December    15.715995
January     25.468688
February    21.041085
March        9.274033
April       14.738732
May          1.834862
Name: SNOW, dtype: float64
```

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('M')
...   .sum()
...   .div(season2017.sum())
...   .mul(100)
...   .rename(lambda idx: idx.month_name())
...   .plot.bar(title='2017 Monthly Percent of Snowfall')
... )
```

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

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

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'
```

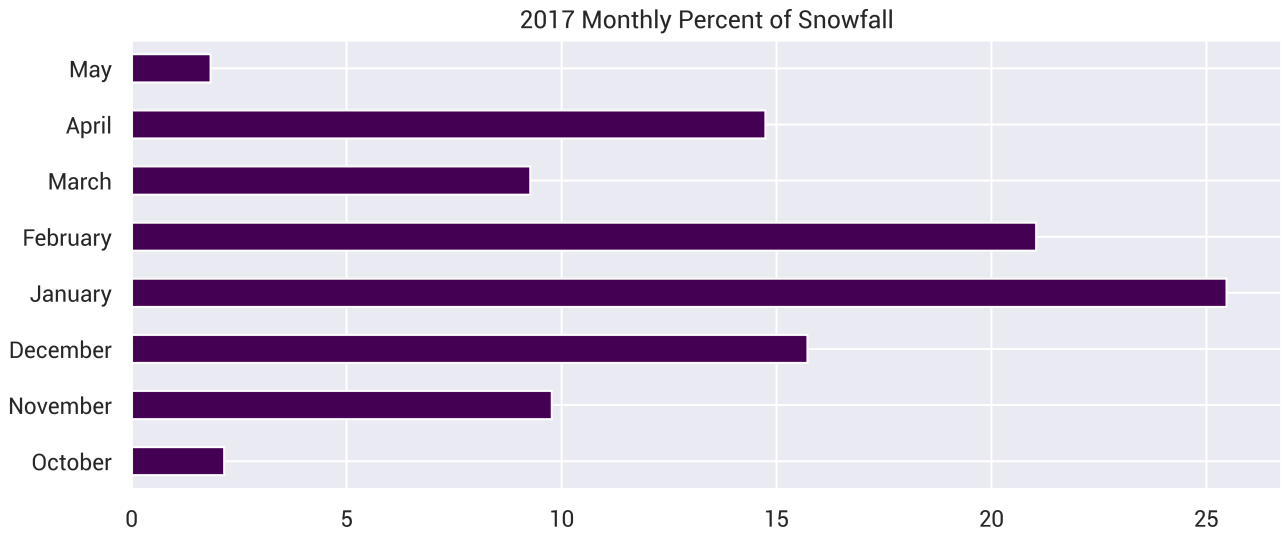


Figure 14.11: Basic series horizontal bar plot.

```
>>> df = pd.read_csv(url)
>>> 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()
Chevrolet                 4003
Ford                      3371
Dodge                     2583
GMC                       2494
Toyota                    2071
                          ...
E. P. Dutton, Inc.           1
Mahindra                     1
London Taxi                  1
Panos                        1
Lambda Control Systems       1
Name: make, Length: 136, dtype: int64
```

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()
... )
```

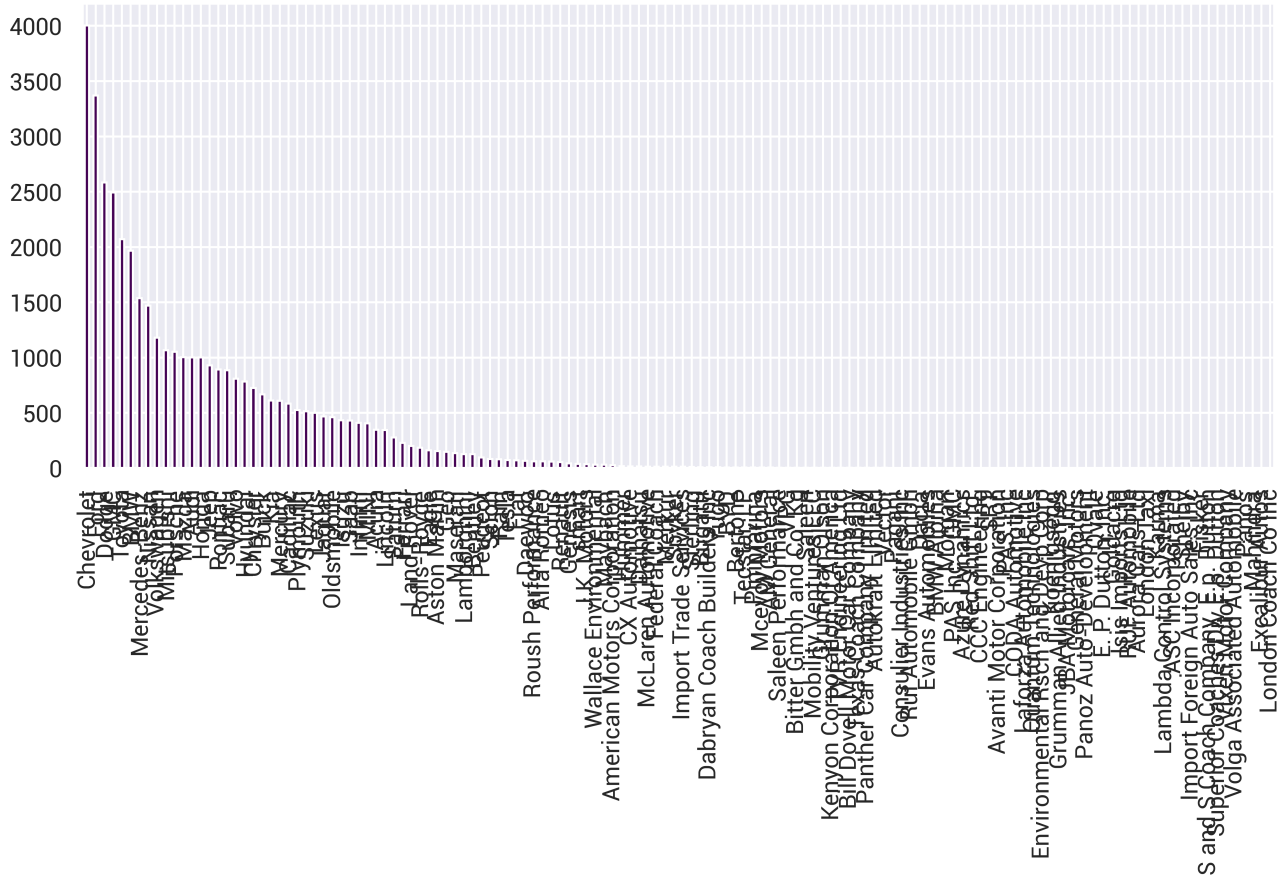


Figure 14.12: Crowded bar plot.

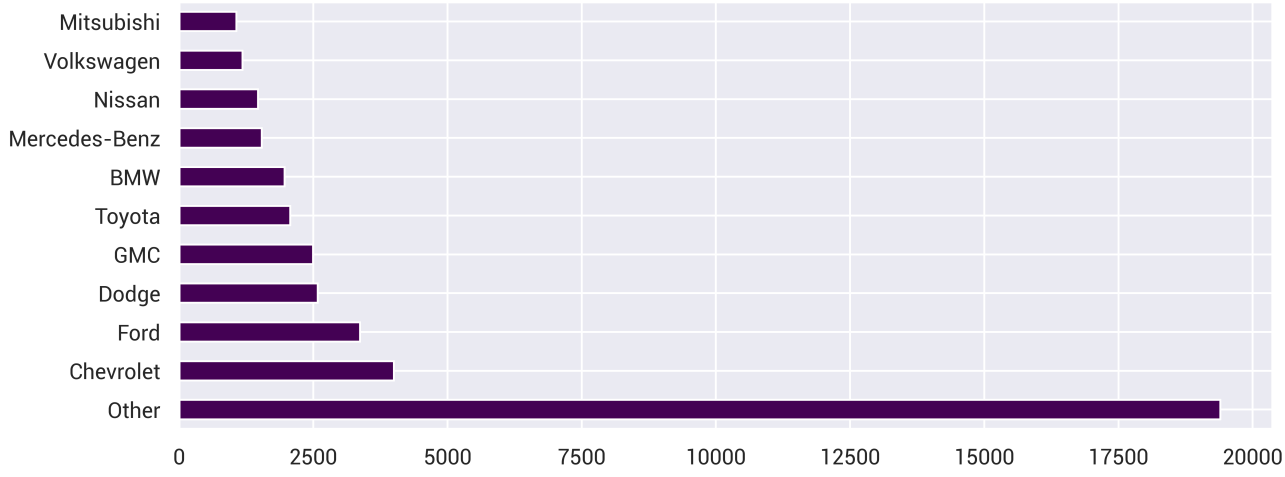


Figure 14.13: Grouping long-tail members together for legible bar plot.

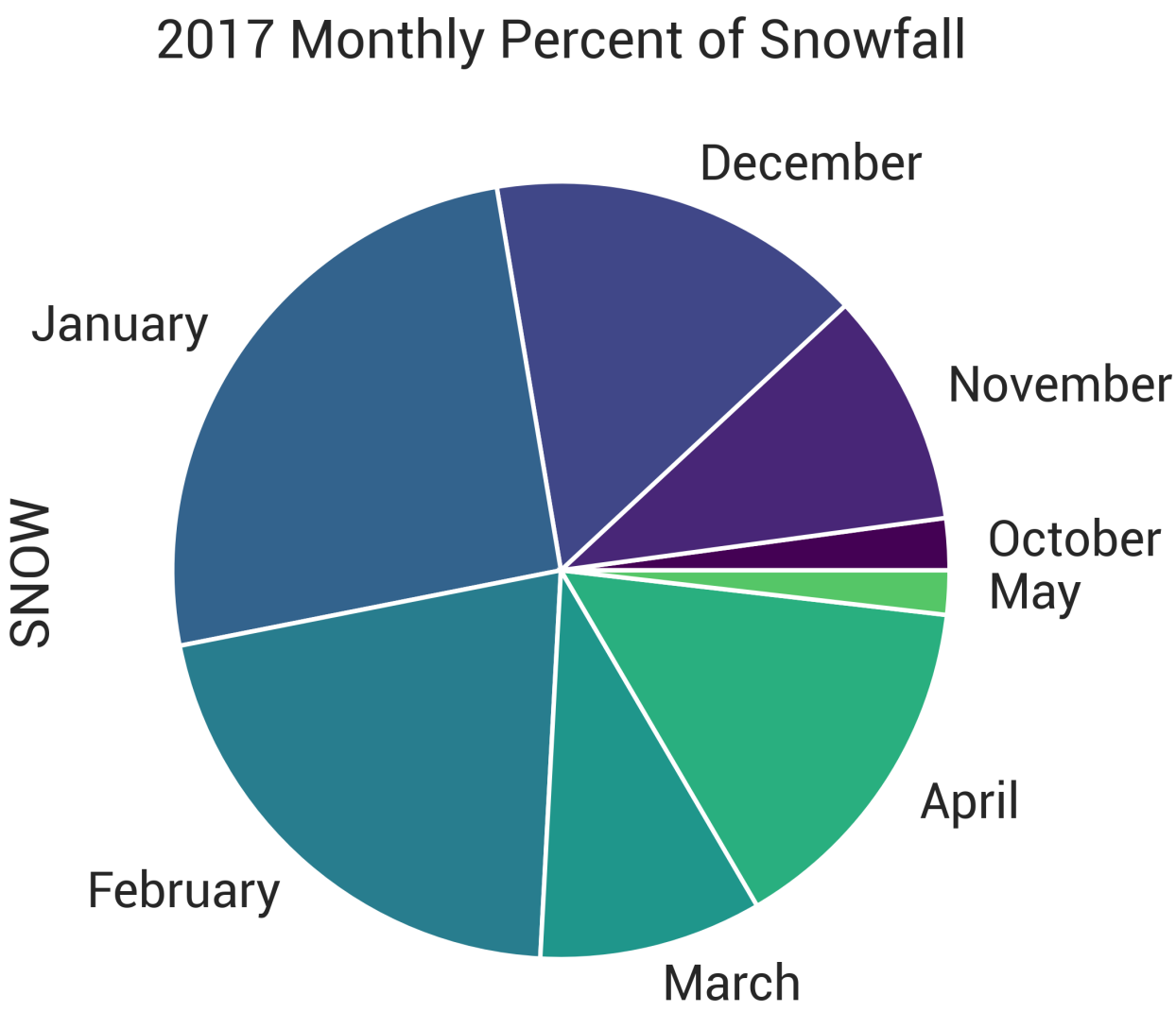


Figure 14.14: Basic series pie plot.

14.9 Pie Plots

If you are the type that prefers pie plots, you can create those as well:

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

14.10 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. To do similar, you could use code like this:

```
import matplotlib
import seaborn as sns
color_palette = ["#440154", "#482677", "#404788", "#33638d", "#287d8e",
   "#1f968b", '#29af7f', '#55c667', '#73d055', '#b8de29', '#fde725']
fp = matplotlib.font_manager.FontProperties(
    fname='/Fonts/roboto/Roboto-Condensed.ttf')
with sns.plotting_context(rc=dict(font='Roboto', palette=color_palette)):
    fig, ax = plt.subplots(dpi=600, figsize=(10,4))
    snow.plot.hist()
    fig.savefig('snowhist.png', dpi=600, bbox_inches='tight')
```

Method	*Description*
`s.plot(ax=None, style=None, logx=False, logy=False, xticks=None, yticks=None, xlim=None, ylim=None, xlabel=None, ylabel=None, rot=None, fontsize=None, colormap=None, table=False, **kwargs)`	Common plot parameters. Use `ax` to use existing Matplotlib axes, `style` for color and marker style (see `matplotlib.marker`), `_ticks` to specify tick locations, `_lim` to specify tick limits, `_label` to specify x/y label (default to index/column name), `rot` to rotate labels, `fontsize` for tick label size, `colormap` for coloring, `position`, `table` to create table with data. Additional arguments are passed to `plt.plot`
`s.plot.bar(position=.5, color=None)`	Create a bar plot. Use `position` to specify label alignment (0-left, 1-right). Use `color` (string, list) to specify line color.
`s.plot.barh(x=None, y=None, color=None)`	Create a horizontal bar plot. Use `position` to specify label alignment (0-left, 1-right). Use `color` (string, list) to specify line color.
`s.plot.hist(bins=10)`	Create a histogram. Use `bins` to change the number of bins.
`s.plot.box()`	Create a boxplot.
`s.plot.kde(bw_method='scott', ind=None)`	Create a Kernel Density Estimate plot. Use `bw_method` to calculate estimator bandwidth (see `scipy.stats.gaussian_kde`). Use `ind` to specify evaluation points for PDF estimation (NumPy array of points, or integer with equally spaced points).
`s.plot.line(color=None)`	Create a line plot. Use `color` to specify line color.
`s.plot.pie()`	Create a pie plot.

Table 14.1: Series Plotting Methods

14.11 Summary

In this chapter, we explored basic plotting functionality with series objects. We showed a little bit of the functionality that you get 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.

14.12 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 a frequency count of a categorical column.
6. Create a pie plot from a frequency count of a categorical column.

Chapter 15

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.

15.1 Categorical Data

Categories are labels that describe data. Generally, there are repeated values, and if they have an intrinsic order, they are referred to as *ordinal* values. One example is shirt sizes: small, medium, and large. Underordered values such as colors are called *nominal* values. In addition, you can convert numerical data to categories by binning them.

We will start by looking at the categorical values found in the fuel economy data set. The *make* column has categorical information:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/' \
...       'data/vehicles.csv.zip'
>>> df = pd.read_csv(url)
>>> make = df.make
>>> 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: object
```

15.2 Frequency Counts

I like to use the `.value_counts` method to determine the *cardinality* of the values. The frequency of values will tell you if a column is categorical. If every value was unique or free form text, it is not categorical:

```
>>> make.value_counts()
Chevrolet              4003
Ford                   3371
Dodge                  2583
GMC                    2494
Toyota                 2071
                       ...
London Taxi               1
General Motors            1
E. P. Dutton, Inc.        1
RUF Automobile            1
JBA Motorcars, Inc.       1
Name: make, Length: 136, dtype: int64
```

We can also inspect the size and the number of unique items to infer the cardinality:

```
>>> make.shape, make.nunique()
((41144,), 136)
```

15.3 Benefits of Categories

The first benefit of categorical values is that they use less memory:

```
>>> cat_make = make.astype('category')

>>> make.memory_usage(deep=True)
2606395

>>> cat_make.memory_usage(deep=True)
95888
```

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()
1.41 ms ± 37.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

>>> %%timeit
make.str.upper()
11.5 ms ± 45.7 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. 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.

15.4 Conversion to Ordinal Categories

If we wanted to make an ordinal categorical (say alphabetic order) from the makes, we 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, object): [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]
```

15.5 The .cat Accessor

In addition, there are a few methods attached to the `.cat` attribute of categorical series. If you need to rename the categories, you can use the `.rename_categories` method. You need to pass in a list with the same length as the current categories or a dictionary mapping old values to new values. Here we will lowercase the categories using both methods:

```
>>> cat_make.cat.rename_categories(
...    [c.lower() for c in cat_make.cat.categories])
0        alfa romeo
1           ferrari
2             dodge
```

```
3             dodge
4            subaru
            ...
41139        subaru
41140        subaru
41141        subaru
41142        subaru
41143        subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): [am general, asc incorporated, acura, alfa
    romeo, ..., volvo, wallace environmental, yugo, smart]

>>> ordered_make.cat.rename_categories(
...    {c:c.lower() for c in ordered_make.cat.categories})
0        alfa romeo
1           ferrari
2             dodge
3             dodge
4            subaru
            ...
41139        subaru
41140        subaru
41141        subaru
41142        subaru
41143        subaru
Name: make, Length: 41144, dtype: category
Categories (136, object): [am general < asc incorporated < acura
     < alfa romeo ... volvo < wallace environmental < yugo < smart]
```

The `.cat` attribute also allows you to add or remove categories and change the order of nominal categories.

Here we change the ordering. Previously *smart* was the maximum value because it was lowercased. 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']
```

15.6 Category Gotchas

Here are a few oddities to be aware of with categorical data. Applying the `.value_counts` method or `.groupby` to categorical data uses all of the categories even if there were no values for them. 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.iloc[:100].value_counts()
Dodge                          17
Oldsmobile                      8
Ford                            8
Buick                           7
Mazda                           5
                               ..
Panos                           0
Panoz Auto-Development          0
Panther Car Company Limited     0
Peugeot                         0
AM General                      0
Name: make, 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
...  .iloc[:100]
...  .groupby(cat_make.iloc[:100])
...  .first()
... )
make
AM General                           NaN
ASC Incorporated                     NaN
Acura                                NaN
Alfa Romeo                     Alfa Romeo
American Motors Corporation          NaN
                                   ...
Volkswagen                     Volkswagen
Volvo                               Volvo
Wallace Environmental                NaN
Yugo                                 NaN
smart                                NaN
Name: make, Length: 136, dtype: category
Categories (136, object): ['AM General', 'ASC Incorporated', ...
                           'Wallace Environmental', 'Yugo', 'smart']
```

Compare this with just the result from the string series:

```
>>> (make
...  .iloc[:100]
...  .groupby(make.iloc[:100])
...  .first()
... )
make
Alfa Romeo          Alfa Romeo
Audi                      Audi
BMW                        BMW
Buick                    Buick
CX Automotive    CX Automotive
                       ...
Rolls-Royce        Rolls-Royce
Subaru                  Subaru
Toyota                  Toyota
Volkswagen          Volkswagen
```

```
Volvo                    Volvo
Name: make, Length: 25, dtype: object
```

There is an optional parameter, `observed`, for `.groupby` to tell it to only include results for which there are values:

```
>>> (cat_make
...  .iloc[:100]
...  .groupby(cat_make.iloc[:100], observed=True)
...  .first()
... )
make
Alfa Romeo      Alfa Romeo
Ferrari            Ferrari
Dodge                Dodge
Subaru              Subaru
Toyota              Toyota
                   ...
Mazda                Mazda
Oldsmobile      Oldsmobile
Plymouth          Plymouth
Pontiac            Pontiac
Rolls-Royce    Rolls-Royce
Name: make, Length: 25, dtype: object
```

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]
```

15.7 Generalization

In the manipulation methods chapter, we 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
3         Dodge
4        Subaru
          ...
41139    Subaru
```

```
41140    Subaru
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 do 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
3            US
4         Other
          ...
41139     Other
41140     Other
41141     Other
41142     Other
41143     Other
Name: make, Length: 41144, dtype: category
Categories (3, object): ['German', 'Other', 'US']
```

Method	*Description*
`.astype(dtype)`	Return a series converted to categories. Set `dtype` to `'category'` for unordered category, `CategoricalDType` for ordered category.
`pd.CategoricalDtype( categories, ordered=False)`	Create categorical type. Set `categories` to a list of categories.
`pd.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates='raise', ordered=True)`	Bin values from `x` (a series). If `bins` is an integer, use equal-width bins. If `bins` is a list of numbers (defining minimum and maximum positions) use those for the edges. `right` defines whether the right edge is open or closed. `labels` allows us to specify bin names. Out of bounds values will be missing.

`pd.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')`	Bin values from `x` (a series) into `q` equal-sized bins (`10` for decile quantiles, `4` for quartile quantiles). Alternatively, we can pass in a list of quantile edges. Out of bounds values will be missing.
`.cat.add_categories( new_categories)`	Return a series with the new categories added. If it is ordinal, the new values are added at the end (highest).
`.cat.as_ordered()`	Convert categorical series to an ordered series. Use `.reorder_categories` or `CategoricalDtype` to specify the order.
`.cat.categories`	Property with the index of categories.
`.cat.codes`	Property with a series with category codes (index into a category).
`.cat.ordered`	Boolean property if series is ordered.
`.cat.remove_categories( removals)`	Return a series with the categories removed (replace with `NaN`).
`.cat.remove_unused_categories()`	Return a series with the categories removed that are being used.
`.cat.rename_categories( new_categories)`	Return a series with the categories replaced by a list (with new values) or a dictionary (mapping old to new values).
`.cat.reorder_categories( new_categories)`	Return a series with the categories replaced by a list.
`.cat.set_categories( new_categories, ordered=False, rename=False)`	Return a series with the categories replaced by a list.

Table 15.1: Category Attributes and Methods

15.8 Summary

If you are dealing with text data, it is worth considering whether converting the text data to categorical data makes sense. You can save a lot of memory and speed up many operations by doing so. A categorical series has a `.cat` attribute that will allow you to manipulate the categories.

15.9 Exercises

With a dataset of your choice:

1. Convert a text column into a categorical column. How much memory did you save?
2. Convert a numeric column into a categorical column by binning it (`pd.cut`). How much memory did you save?
3. Use the `generalize_topn` function to limit the amounts of categories in your column. How much memory did you save?

Chapter 16

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.

16.1 Database and Spreadsheet Analogues

If you think of a dataframe as row-oriented, the interface will feel wrong. Many tabular data structures are row-oriented. Perhaps this is due to spreadsheets and CSV files dealt with on a row by row basis. Perhaps it is due to the many OLTP[9] databases that are row-oriented out of the box. A `DataFrame`, is often used for analytical purposes and is better understood when thought of as 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.

16.2 A Simple Python Version

Below is a simple attempt to create a tabular Python data structure that is column-oriented. It has a 0-based integer index, but that is not required, 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] },
```

[9] *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. In contrast with *OLAP* (On-line Analytical Processing), which is optimized for complex querying and aggregation. Typically, reporting systems use these types of databases, which might store data in a denormalized form to speed up access.

```
...     { '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']
```

16.3 Dataframes

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

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

>>> df
   growth    Name
0     0.5    Paul
1     0.7  George
2     1.2   Ringo
```

The leftmost values, 0, 1, and 2, are the index. 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
```

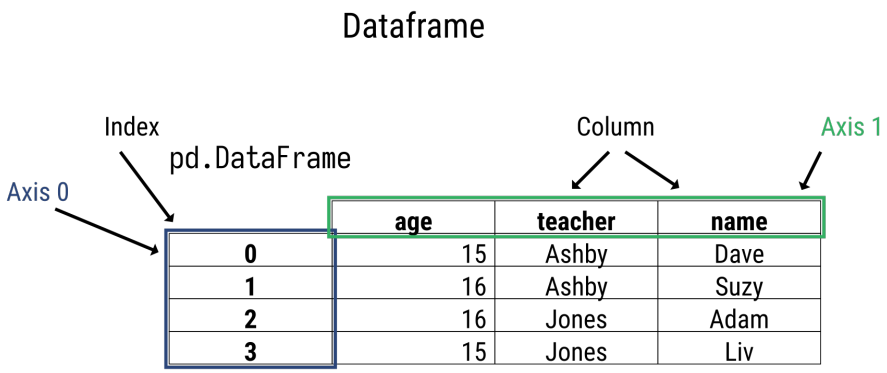


Figure 16.1: Figure showing column-oriented nature of Dataframe. (Note that a column can be pulled off as a `Series`)

```
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'])
<class '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 find that 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 provide clues as to 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.

16.4 Construction

dataframes can be created from many types of input:

- columns (dicts of lists)
- rows (list of dicts)
- CSV files (`pd.read_csv`)
- 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:

```
>>> pd.DataFrame([
...   {'growth':.5, 'Name':'Paul'},
...   {'growth':.7, 'Name':'George'},
...   {'growth':1.2, 'Name':'Ringo'}])
     Name  growth
0    Paul     0.5
1  George     0.7
2   Ringo     1.2
```

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""")

>>> pd.read_csv(csv_file)
   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).

After parsing the CSV file, pandas makes a best-effort to give a type to each column. A "best-effort" means it will convert numerics to `int64` if the column is whole numbers and not missing values. Other numeric columns are converted to `float64` (if they have decimals or are missing values). If there are non-numeric values, pandas will use the `object` type. Usually `object` means that the column has string type data, though it might be mixed-typed column that has string data and `nan` values stored as floats.

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:

Type	*Description*
`float64`	Floating point. Can specify different sizes, ie: `float16`, `float32` or `float64`.
`int64`	Integer number. Can put `u` in front for unsigned. Can specify size, ie: `int8`, `int16`, `int32`, or `int64`. Does not support missing values.
`Int64`	Nullable integer number. Supports `<NA>` for integer columns. Can put `U` in front for unsigned. Can specify size, ie: `Int16`, `Int32`, or `Int64`.
`datetime64[ns]`	Datetime number
`datetime64[ns, tz]`	Datetime number with timezone
`timedelta[ns]`	A difference between datetimes
`category`	Used to specify categorical columns
`object`	Used for other columns such as strings, or Python objects
`string`	Used for text data. Supports `<NA>` for missing values.

Figure 16.2: Data types in pandas

Tip

Having said this, my experience with the `dtype` parameter is that it is easier to convert many types after they are loaded into a dataframe. 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)
>>> 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
```

16.5 Dataframe Axis

Unlike a series, which has 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 along the index or along the columns using the labels 0 and 1:

```
>>> df.sum(axis=0)
growth                 2.4
Name      PaulGeorgeRingo
dtype: object

>>> df.sum(axis=1)
0    0.5
1    0.7
2    1.2
dtype: float64
```

We can also spell out the axis. This is my preferred method because it is easier to read:

```
>>> df.sum(axis='index')
growth                 2.4
Name      PaulGeorgeRingo
dtype: object

>>> df.sum(axis='columns')
0    0.5
1    0.7
2    1.2
dtype: float64
```

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')
```

Tip

Here is a clue to help 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]})
>>> 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=1)
0    85
1    90
dtype: int64
```

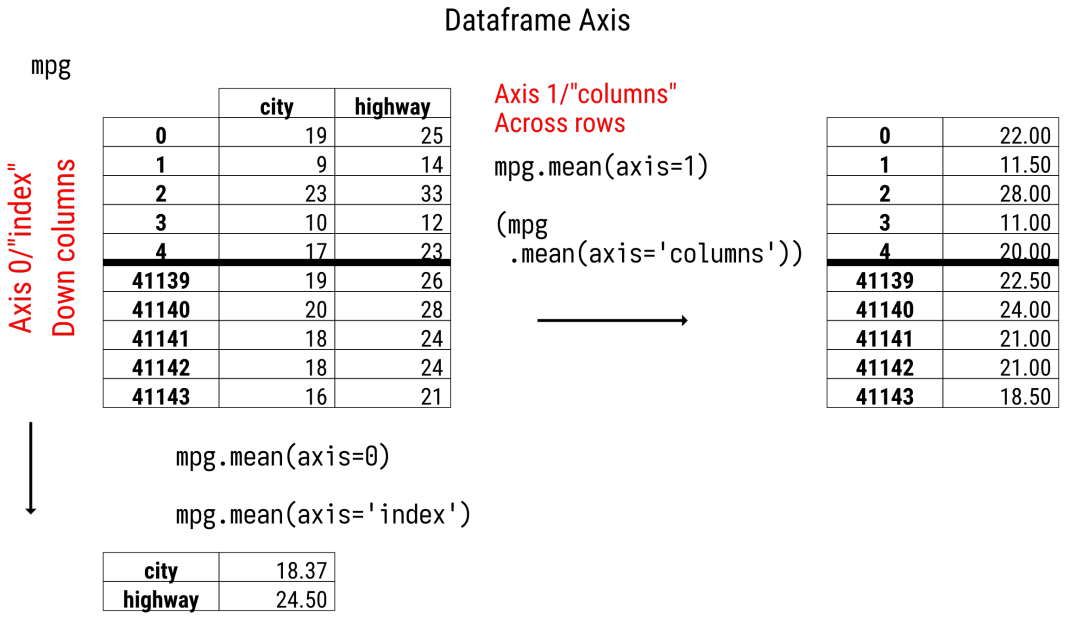


Figure 16.3: Figure showing the relation between axis 0 and axis 1. Note that when an operation is applied along axis 0, it is applied down the column. Likewise, operations along axis 1 operate across the values in the row.

Code	*Description*
`pd.DataFrame(data=None, index=None, columns=None)`	Create a dataframe from scalar, sequence, dict, ndarry or dataframe.
`.axes`	Tuple of index and columns.

Table 16.1: Dataframe creation

16.6 Summary

In this chapter, we introduced a Python data structure that is 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 really 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.

16.7 Exercises

1. Create a dataframe with the names of your colleagues, their age (or an estimate), and their title.
2. Capitalize the values in the name column.
3. Sum up the values of the age column.

Chapter 17

Similarities with `Series` and `DataFrame`

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 series objects are directly applicable to dataframes.

In the next few chapters, we will explore the similarities between the two classes, before diving into 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'm going to pull the raw data and show how to recreate the visualization first. Then we will demonstrate more features of dataframes with the presidential data.

17.1 Getting the Data

Wikipedia has the data[10] from Siena College. I scraped the data using the following commands. (Given that Wikipedia can change at any time, there is no guarantee that this code will work for you.):

```
url = 'https://en.wikipedia.org/wiki/'\
      'Historical_rankings_of_presidents_of_the_United_States'
pres_dfs = pd.read_html(url)
df = pres_dfs[-4]
```

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:

```
(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

[10] https://en.wikipedia.org/wiki/Historical_rankings_of_presidents_of_the_United_States

- Im = Imagination
- Int = Integrity
- IQ = Intelligence
- L = Luck
- WR = Willing to take risks
- AC = Ability to compromise
- EAb = Executive ability
- LA = Leadership ability
- CAb = Communication ability
- OA = Overall ability
- PL = Party leadership
- RC = Relations with Congress
- CAp = Court appointments
- HE = Handling of economy
- EAp = Executive appointments
- DA = Domestic accomplishments
- FPA = Foreign policy accomplishments
- AM = Avoid crucial mistakes
- EV = Experts' view
- O = Overall

At this point, I exported my data and saved it to a CSV (to avoid possible future changes at Wikipedia). You can load the data from my GitHub account:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0)

>>> df
              President                   Party  Bg  ...  AM  EV   O
Seq.                                              ...
1     George Washington             Independent   7  ...   1   2   1
2            John Adams              Federalist   3  ...  16  10  14
3      Thomas Jefferson  Democratic-Republican   2  ...   7   5   5
4         James Madison  Democratic-Republican   4  ...  11   8   7
5          James Monroe  Democratic-Republican   9  ...   6   9   8
...                 ...                     ...  ..  ...  ..  ..  ..
41    George H. W. Bush              Republican  10  ...  17  21  21
```

```
42         Bill Clinton             Democratic  21  ...  30  14  15
43       George W. Bush             Republican  17  ...  36  34  33
44         Barack Obama             Democratic  24  ...  10  11  17
45         Donald Trump             Republican  43  ...  41  42  42

[44 rows x 23 columns]
```

Note that we lose fancy pandas types when we load from CSV, so I will need to set those up again:

```
>>> df.dtypes
President    object
Party        object
Bg            int64
Im            int64
Int           int64
              ...
DA            int64
FPA           int64
AM            int64
EV            int64
O             int64
Length: 23, 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' 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')),
...              Quartile=lambda df_:pd.qcut(df_.Average_rank, 4,
...                  labels='1st 2nd 3rd 4th'.split())
...             )
...     )
```

We will go over all of the functionality exposed in the `tweak_siena_pres` function in detail in later chapters. I will briefly explain the chained operations.

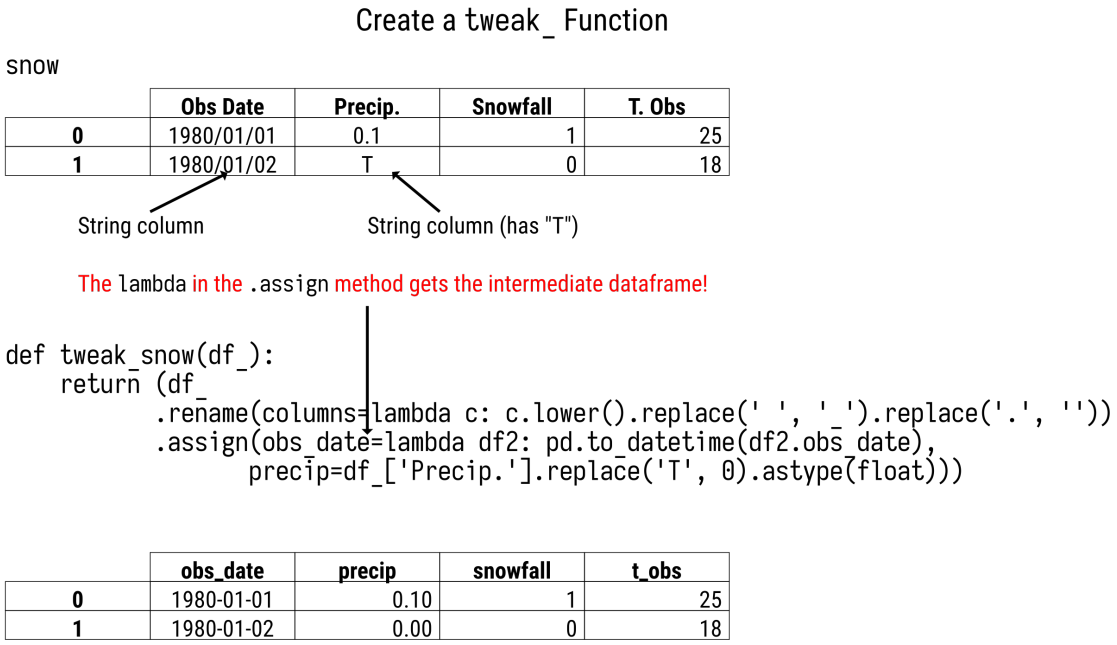


Figure 17.1: A tweak function is useful for maintaining order and sanity when working in Jupyter.

The first call to `.rename` (#1) removes the period from the column named *Seq.*. 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 of 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 all of the numeric values of a row and then taking the *dense* rank of the resulting values. It also creates a *Quartile* column by binning the *Average_rank* column into four bins.

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 that, 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, then 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 come back to your notebook, it can be hard to get back to the state where your data is in the form that you want it to be. If you follow this pattern, it 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 big 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:

```
>>> 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')
...    .iloc[:,2:-1]
...   ),annot=True, cmap='viridis', ax=ax)
>>> g.set_xticklabels(g.get_xticklabels(), rotation=45, fontsize=8,
...     ha='right')
>>> _ = plt.title('Presidential Ranking')
>>> fig.savefig('img/pandas2/20-pres.png', bbox_inches='tight')
```

But the purpose of this chapter is not to look at visualizations, rather to see that most of what you can do with a series you can do with a dataframe. Let's start comparing.

17.2 Viewing Data

Dataframes have `.head` and `.tail` methods to view the first or last few rows of the data. 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)
>>> pres.head(3)
              President                   Party  ...  Average_rank  Quartile
Seq.                                             ...
1     George Washington             Independent  ...             1       1st
2            John Adams              Federalist  ...            13       2nd
3      Thomas Jefferson  Democratic-Republican  ...             5       1st

[3 rows x 25 columns]

>>> pres.sample(3)
              President       Party  ...  Average_rank  Quartile
Seq.                                 ...
18     Ulysses S. Grant  Republican  ...            24       3rd
36    Lyndon B. Johnson  Democratic  ...            16       2nd
21    Chester A. Arthur  Republican  ...            34       4th

[3 rows x 25 columns]
```

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

Table 17.1: Dataframe viewing Methods

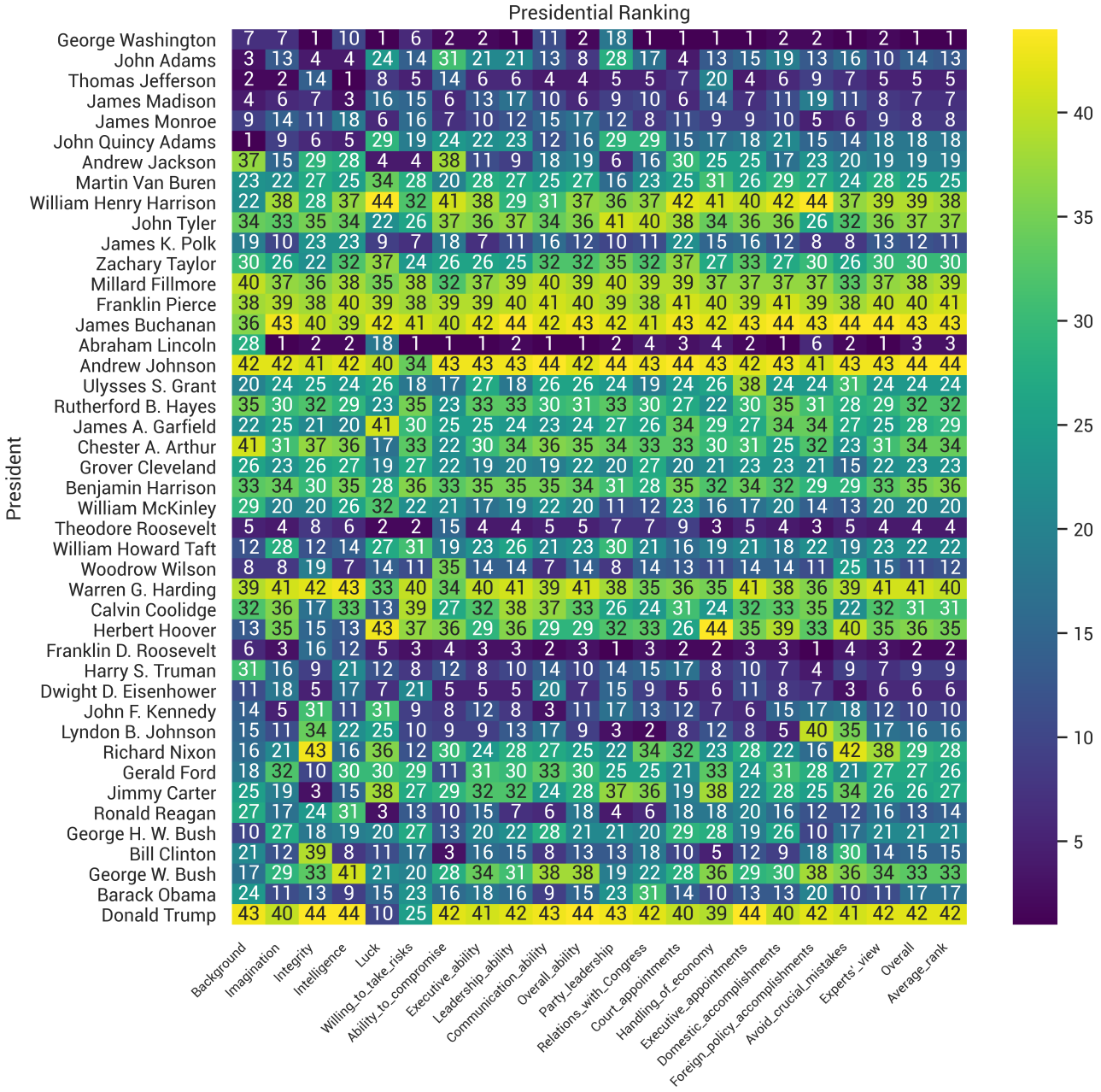


Figure 17.2: Visualization of United States presidential attributes.

17.3 Summary

This chapter demonstrated loading data from Wikipedia and then cleaned 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.

17.4 Exercises

With a tabular dataset of your choice:

1. Create a dataframe from the data.
2. View the first 20 rows of data.
3. Sample 30 rows from your data.

Chapter 18

Math Methods in DataFrames

We have seen that you can perform math operations on Series objects in pandas. In this chapter, we will show that you can also do math on dataframes.

We will begin by looking at the basic math operations. We will use a cleaned up version of the data:

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0)

>>> pres = tweak_siena_pres(df)
```

18.1 Index Alignment

We can perform math operations of 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 then to the index values from rows 1-5 and 0-4:

```
>>> scores = (pres
...  .loc[:,'Background':'Average_rank']
... )
>>> scores
      Background  Imagination  ...  Overall  Average_rank
Seq.                           ...
1              7            7  ...        1             1
2              3           13  ...       14            13
3              2            2  ...        5             5
4              4            6  ...        7             7
5              9           14  ...        8             8
...          ...          ...  ...      ...           ...
41            10           27  ...       21            21
42            21           12  ...       15            15
43            17           29  ...       33            33
44            24           11  ...       17            17
45            43           40  ...       42            42

[44 rows x 22 columns]
```

We will pull out two sections of the data:

```
>>> s1 = scores.iloc[:3, :4]
>>> s1
      Background  Imagination  Integrity  Intelligence
Seq.
1              7            7          1            10
2              3           13          4             4
3              2            2         14             1

>>> s2 = scores.iloc[1:6, :5]
>>> s2
      Background  Imagination  Integrity  Intelligence  Luck
Seq.
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.

```
>>> s1 + s2
      Background  Imagination  Integrity  Intelligence  Luck
Seq.
1            NaN          NaN        NaN           NaN   NaN
2            6.0         26.0        8.0           8.0   NaN
3            4.0          4.0       28.0           2.0   NaN
4            NaN          NaN        NaN           NaN   NaN
5            NaN          NaN        NaN           NaN   NaN
6            NaN          NaN        NaN           NaN   NaN
```

Only the overlapping rows (rows 2 and 3) and columns (*Background* through *Intelligence*) get added together. The other values are missing!

18.2 Duplicate Index Entries

If you have duplicate index values, each index value in the left dataframe will match up with the index in the right dataframe. You should be aware if you have repeated index values before performing operations that align the index.

Lets add a dataframe that has duplicated values in the index (created by concatenating the dataframe with itself):

```
>>> scores.iloc[:3, :4] + pd.concat([scores.iloc[1:6, :5]]*2)
      Background  Imagination  Integrity  Intelligence  Luck
Seq.
1            NaN          NaN        NaN           NaN   NaN
2            6.0         26.0        8.0           8.0   NaN
2            6.0         26.0        8.0           8.0   NaN
3            4.0          4.0       28.0           2.0   NaN
3            4.0          4.0       28.0           2.0   NaN
...          ...          ...        ...           ...   ...
4            NaN          NaN        NaN           NaN   NaN
5            NaN          NaN        NaN           NaN   NaN
5            NaN          NaN        NaN           NaN   NaN
6            NaN          NaN        NaN           NaN   NaN
6            NaN          NaN        NaN           NaN   NaN
```

```
[11 rows x 5 columns]

>>> pd.concat([scores.iloc[1:6, :5]]*2).index.duplicated().any()
True
```

Method	*Description*
`.add(other, axis='columns', level=None, fill_value=None)`	Add other to dataframe across `axis`. Unlike operator, can specify `fill_value`.
`.sub(other, axis='columns', level=None, fill_value=None)`	Subtract other from dataframe across `axis`. Unlike operator, can specify `fill_value`.
`.mul(other, axis='columns', level=None, fill_value=None)`	Multiply other with dataframe across `axis`. Unlike operator, can specify `fill_value`.
`.div(other, axis='columns', level=None, fill_value=None)`	Divide dataframe by other across `axis`. Unlike operator, 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 operator, can specify `fill_value`.
`.mod(other, axis='columns', level=None, fill_value=None)`	Perform modulo operation with other across `axis`. Unlike operator, can specify `fill_value`.
`.pow(other, axis='columns', level=None, fill_value=None)`	Raise to other power across `axis`. Unlike operator, can specify `fill_value`.

Table 18.1: Dataframe Math Methods

18.3 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 demonstrated index alignment.

18.4 Exercises

With a tabular dataset of your choice:

1. Create a dataframe from the data and add it to itself.
2. Create a dataframe from the data and multiply it by two.
3. Are the results from the previous exercises equivalent?

Chapter 19

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.

19.1 For Loops

You can use a `for` loop with a dataframe, though you generally want to avoid for loops when doing numerical manipulation. When I see a for loop with pandas code, it means this is a slow operation, and you are not able to 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, here are three methods for doing it. The `.iteritems` method gives you a tuple with the column name and the column (a series). The `.iterrows` method gives you a tuple with the index value and the row (converted into a series). Finally, the `.itertuples` method gives you a row represented as a named tuple (with the index in position 0):

```
>>> # iteration over columns (col_name, series) tuple
>>> for col_name, col in pres.iteritems():
...     print(col_name, type(col))
...     break
Seq <class 'pandas.core.series.Series'>

>>> # iteration over rows (index, row(as a series)) tuple
>>> for idx, row in pres.iterrows():
...     print(idx, type(row))
...     break
1 <class 'pandas.core.series.Series'>

>>> # iteration over rows as namedtuple (index as first item)
>>> for tup in pres.itertuples():
...     print(tup[0], tup.Party)
...     break
1 Independent
```

19.2 Aggregations

The aggregations that are found in a series are also applicable to a dataframe. You need to 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)
Seq.
1      3.681818
2     14.454545
3      6.545455
4      9.636364
5     10.454545
        ...
41    20.818182
42    14.636364
43    30.363636
44    15.818182
45    39.772727
Length: 44, dtype: float64
```

(Note we could also use `.mean(axis=1)` 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 just returns the value for index *1*):

```
>>> pres.agg(['count', 'size', 'sum', lambda col: col.loc[1]])
                                                          Seq  ... Quartile
count                                                      44  ...       44
size                                                       44  ...       44
sum       1234567891011121314151617181920212224232526 27...  ...      NaN
<lambda>                                                    1  ...      1st

[4 rows x 26 columns]
```

We can pass in a dictionary to perform multiple aggregations on a column:

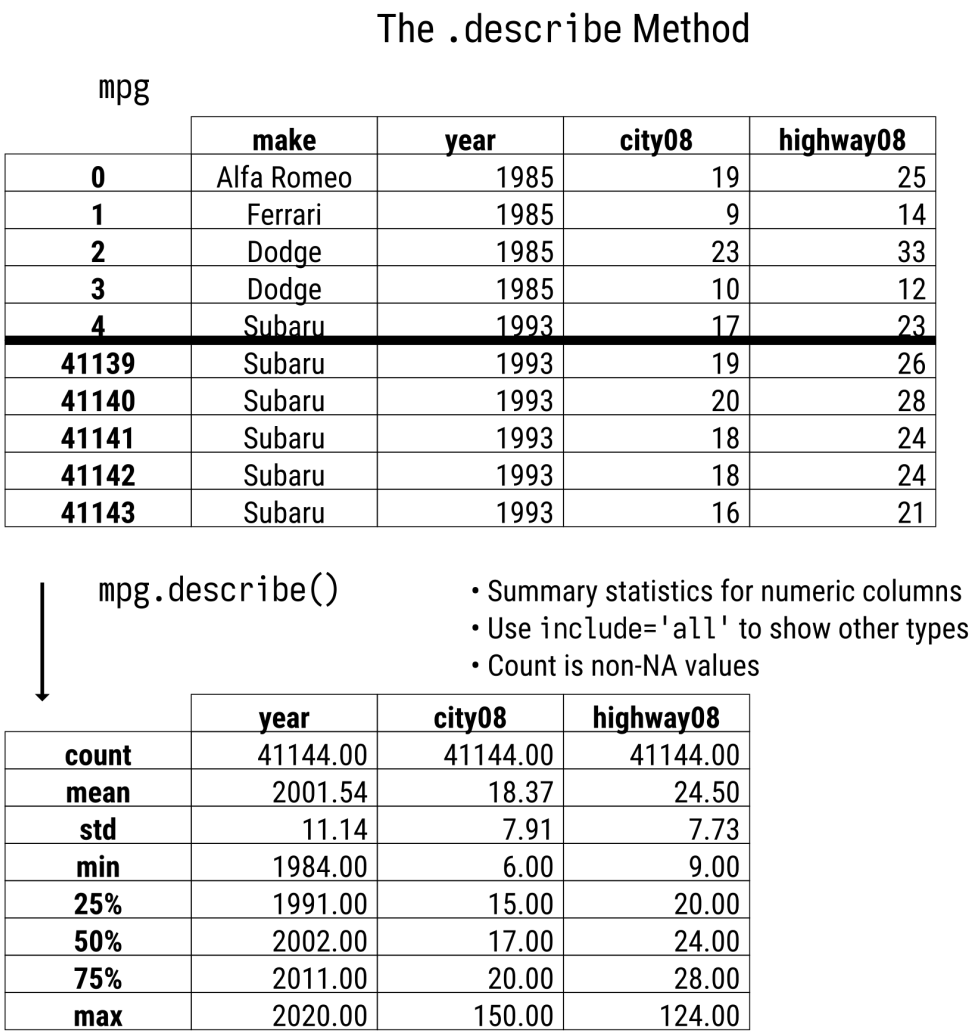
```
>>> pres.agg({'Luck': ['count', 'size'], 'Overall': ['count', 'max']})
       Luck  Overall
count  44.0     44.0
size   44.0      NaN
max     NaN     44.0
```

You can use a keyword argument with a tuple to specify the index value of the resultant aggregation:

```
>>> 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 columns:

```
>>> pres.describe()
       Background  Imagination  ...    Overall  Average_rank
count   44.000000    44.000000  ...  44.000000     44.000000
```



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	year	city08	highway08
count	41144.00	41144.00	41144.00
mean	2001.54	18.37	24.50
std	11.14	7.91	7.73
min	1984.00	6.00	9.00
25%	1991.00	15.00	20.00
50%	2002.00	17.00	24.00
75%	2011.00	20.00	28.00
max	2020.00	150.00	124.00

Figure 19.1: The `.describe` method provides the count of non-missing values, the mean, standard deviation, minimum, maximum, and quartiles.

```
mean    22.000000    21.750000  ...  22.500000     22.500000
std     12.409674    12.519984  ...  12.845233     12.845233
min      1.000000     1.000000  ...   1.000000      1.000000
25%     11.750000    11.000000  ...  11.750000     11.750000
50%     22.000000    21.500000  ...  22.500000     22.500000
75%     32.250000    32.250000  ...  33.250000     33.250000
max     43.000000    43.000000  ...  44.000000     44.000000

[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, rather it is the count of the non-missing (not `na`) rows.

19.3 The `.apply` Method

Like the series, the dataframe has an `.apply` method. Like the series method, you should be wary of using the dataframe 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 a whole 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 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'))
... )
Seq.
1     17
2     28
3     19
4     16
5     13
      ..
41    19
42    36
43    24
44    22
45    34
Length: 44, dtype: uint8
```

The `.apply` version looks like this:

```
>>> (pres
...  .select_dtypes('number')
...  .apply(lambda row: row.max()-row.min(), axis='columns')
... )
Seq.
1     17
2     28
3     19
4     16
5     13
      ..
41    19
42    36
43    24
44    22
```

```
45    34
Length: 44, dtype: int8
```

They look pretty similar but the former does an optimized max and min 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: int64
```

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: int64
```

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),
...     parse_dates=['cancel_date', 'period_start', 'start_date',
...                  'end_date'])

>>> bill_df
```

```
  cancel_date period_start start_date   end_date  rev  sum_payments
0  2019-12-01   2020-01-01 2019-12-15 2020-05-15  999            50
1         NaT   2020-01-01 2019-12-15 2020-05-15  999            50
2         NaT   2020-01-01 2019-12-15 2020-05-15  999          1950
3  2020-01-20   2020-01-01 2019-12-15 2020-05-15  499             0
4         NaT   2020-01-01 2019-12-24 2020-05-24  699           100
5         NaT   2020-01-01 2019-11-29 2020-04-29  799           250
6         NaT   2020-01-01 2020-01-15 2020-04-29  799           250
```

Here is some logic. If the start and end dates bound the period start date, we calculate if the revenue is greater than the sum of the payments:

```
>>> def calc_unbilled_rec(vals):
...     cancel_date, period_start, start_date, end_date, rev, \
...          sum_payments = vals
...     if cancel_date < period_start:
...         return
...     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 `np.select`. Sadly this runs about twice as slow on my machine on this small dataset. However, if the dataset has a hundred thousand rows, it runs about 200 times faster!

```
>>> import numpy as np
>>> pd.Series(np.select([
...        (bill_df.cancel_date < bill_df.period_start),   # 1
...        ((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.start_date < bill_df.period_start) &  # 3
...         (bill_df.end_date > bill_df.period_start) &
...         (bill_df.rev <= bill_df.sum_payments))
...       ],
...       [np.nan, bill_df.rev - bill_df.sum_payments, 0],  # 1, 2, 3
...       np.nan))  # default
0      NaN
1    949.0
2      0.0
3    499.0
4    599.0
5    549.0
6      NaN
```

```
dtype: float64
```

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!

Method	*Description*
`.iteritems()`	Iterate over a tuple of column name and series.
`.iterrows()`	Iterate over tuple of index name and row (presented as a series). Type information not preserved.
`.itertuples(index=True, name="Pandas")`	Iterate over a namedtuples of rows. Include `index` by default. Use `name` to specify the classname of the namedtuple (or set to `None` to return normal tuples).
`.sum(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return sum over axis. Default of empty sequence is 0, set `min_count=1` to return `nan`.
`.min(axis=0, skipna=True, level=None, numeric_only=None`	Return minimum over the axis.
`.max(axis=0, skipna=True, level=None, numeric_only=None`	Return maximum over the axis.
`.idxmin(axis=0, skipna=True)`	Return the index of first minimum value over the axis.
`.idxmax(axis=0, skipna=True)`	Return the index of first maximum value over the axis.
`.agg(func=None, axis=0, *args, **kwargs)`	Aggregate using `func` over the axis. The `func` can be a function that collapses a column (or row), string, list of functions (or strings), dictionary of axis to function, list, or string.
`.describe(percentiles=[.25, .5, .75], include=None, exclude=None, datetime_is_numeric=False)`	Return summary statistics for dataframe.
`.apply(func=None, axis=0, raw=False, result_type=None, *args, **kwargs)`	Apply `func` over the axis. If `func` returns a sequence then return a dataframe. If `func` returns a scalar, then return a series.
`np.select(condlist, choicelist, default=0`	Simulate an if then statement. Pass in a list of boolean arrays to `condlist` and the corresponding value in the list `choicelist`.

Table 19.1: Dataframe Looping Methods

19.4 Summary

In this chapter, we demonstrated looping and aggregation methods of dataframes. We also demonstrated the `.apply` method.

19.5 Exercises

With a tabular dataset of your choice:

1. Loop over each of the rows and calculate the maximum and minimum value.

2. Calculate the maximum and minimum value of each row and column using the `.agg` method.
3. Calculate the maximum and minimum value of each row and column using the `.apply` method.

Chapter 20

Columns Types, `.assign`, and Memory Usage

In this chapter, we will explore updating columns, creating columns, and changing the types of columns. We will show how to see how this impacts memory usage.

20.1 Conversion Methods

There are various methods and functions for changing the types of a series in pandas. We can all `.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 values of the column. 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')),
            )
    )
```

20.2 Memory Usage

One thing to be aware of is memory usage. You can often shrink the memory usage of a dataframe by changing the type while not losing any data.

Here is the original column sizes of the presidential data:

```
>>> df.memory_usage(deep=True)
Index        3662
President    3175
Party        2976
Bg            352
Im            352
             ...
DA            352
FPA           352
AM            352
EV            352
O             352
Length: 24, dtype: int64
```

Here are the sizes of the columns where the numeric values have been optimized:

```
>>> pres.memory_usage(deep=True)
Index                     3662
President                 3175
Party                      624
Background                  44
Imagination                 44
                           ...
Avoid_crucial_mistakes      44
Experts'_view               44
Overall                     44
Average_rank                44
Quartile                   456
Length: 26, dtype: int64
```

You can see that the ranking columns use less memory because they are stored as `uint8` values instead of `int64`.

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 45
Data columns (total 25 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   President                       44 non-null     object
 1   Party                           44 non-null     category
 2   Background                      44 non-null     uint8
 3   Imagination                     44 non-null     uint8
 4   Integrity                       44 non-null     uint8
 5   Intelligence                    44 non-null     uint8
 6   Luck                            44 non-null     uint8
 7   Willing_to_take_risks           44 non-null     uint8
 8   Ability_to_compromise           44 non-null     uint8
 9   Executive_ability               44 non-null     uint8
 10  Leadership_ability              44 non-null     uint8
 11  Communication_ability           44 non-null     uint8
```

```
 12  Overall_ability                 44 non-null     uint8
 13  Party_leadership                44 non-null     uint8
 14  Relations_with_Congress         44 non-null     uint8
 15  Court_appointments              44 non-null     uint8
 16  Handling_of_economy             44 non-null     uint8
 17  Executive_appointments          44 non-null     uint8
 18  Domestic_accomplishments        44 non-null     uint8
 19  Foreign_policy_accomplishments  44 non-null     uint8
 20  Avoid_crucial_mistakes          44 non-null     uint8
 21  Experts'_view                   44 non-null     uint8
 22  Overall                         44 non-null     uint8
 23  Average_rank                    44 non-null     uint8
 24  Quartile                        44 non-null     category
dtypes: category(2), object(1), uint8(22)
memory usage: 8.7 KB
```

Method	*Description*
`.astype(dtype, copy=True, errors='raise')`	Cast dataframe into `dtype`. (More common to use this on series.)
`.assign(**kwargs)`	Return a new dataframe with updated or new columns. `kwargs` maps column name to function, scalar, or series. If using a function, it is passed in the current state of the dataframe and should return a scalar or series. Subsequent columns may reference earlier columns in `kwargs` if you use a function.
`.memory_usage(index=True, deep=False)`	Return a series with the memory usage of each column in bytes. By default includes `index`. Use `deep=True` to show how much space `object` columns consume.
`.info(verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None)`	Print summary of dataframe to stdout. Use `memory_usage='deep'` to show object column memory usage.

Table 20.1: Dataframe Methods from this Chapter

20.3 Summary

The series chapters showed how to convert the type of a series from one type to another. With a dataframe, we want to optimize the types of each column. To create a dataframe with the newer columns, we use the `.assign` method. If you master this method you will eliminate many bugs that pandas users encounter when they try to change columns using other methods.

20.4 Exercises

With a tabular dataset of your choice:

1. Find a numeric column and change the type of it. Did you save memory? Did you lose precision?
2. Find a string column and convert it to a category. What happened to the memory usage? Time a few string operations. Are they faster on the categorical column or string column?

Chapter 21

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.

21.1 Loading the Data

We will be looking at a dataset of Python users from JetBrains[11].

Let's load the data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url)
>>> jb
      is.python.main other.lang.None  ...    age        country.live
0                Yes             NaN  ...  30-39                 NaN
1                Yes             NaN  ...  21-29               India
2                Yes             NaN  ...  30-39       United States
3                Yes             NaN  ...    NaN                 NaN
4                Yes             NaN  ...  21-29               Italy
...              ...             ...  ...    ...                 ...
54457            Yes             NaN  ...  21-29  Russian Federation
54458            Yes             NaN  ...    NaN                 NaN
54459            Yes             NaN  ...  21-29  Russian Federation
54460            Yes             NaN  ...  30-39               Spain
54461            Yes             NaN  ...  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 columns starting with *database.* end with a database name. In the values for those columns, the database name is included. Because a user might use multiple databases, that is a mechanism to encode this. However, it also creates many columns, one per database. To keep the data for the book manageable, I'm going to filter out columns like the database columns.

[11] https://www.jetbrains.com/lp/python-developers-survey-2020/

Below is code that determines whether a feature can have multiple values (like database) and removes those:

```
>>> import collections
>>> counter = collections.defaultdict(list)
>>> for col in sorted(jb.columns):
...     period_count = col.count('.')
...     if period_count >= 2:
...         part_end = 2
...     else:
...         part_end = 1
...     parts = col.split('.')[:part_end]
...     counter['.'.join(parts)].append(col)
>>> uniq_cols = []
>>> for cols in counter.values():
...     if len(cols) == 1:
...         uniq_cols.extend(cols)

>>> uniq_cols
['age', 'are.you.datascientist', 'company.size',
 'country.live', 'employment.status', 'first.learn.about.main.ide',
 'how.often.use.main.ide', 'ide.main', 'is.python.main', 'job.team',
 'main.purposes', 'missing.features.main.ide', 'nps.main.ide',
 'python.years', 'python2.version.most', 'python3.version.most',
 'several.projects', 'team.size', 'use.python.most', 'years.of.coding']
```

Note that these column names have a period in them. I'm going to replace those with an underscore as it will allow us to access the names of the columns via attributes (with a period).

Let's look at the age column:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .value_counts(dropna=False)
... )
NaN            29701
21-29           9710
30-39           7512
40-49           3010
18-20           2567
50-59           1374
60 or older      588
Name: age, dtype: int64
```

I'm going to pull out the first two characters from the *age* column and convert it to numbers. We will have to convert to `float` because there are missing values:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .str.slice(0,2)
...  .astype(float)
... )
0        30.0
1        21.0
2        30.0
3         NaN
```

```
4         21.0
          ...
54457     21.0
54458      NaN
54459     21.0
54460     30.0
54461     21.0
Name: age, Length: 54462, dtype: float64
```

Note

You can also write `.str.slice(0,2)` as `.str[0:2]`.

Note that currently, pandas (here is the bug[12]) can't convert strings directly to `'Int64'`, you need to convert to `float` first.

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .str.slice(0,2)
...  .astype('Int64')
... )
Traceback (most recent call last):
  ...
TypeError: object cannot be converted to an IntegerDtype

>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .age
...  .str.slice(0,2)
...  .astype(float)
...  .astype('Int64')
... )
0           30
1           21
2           30
3         <NA>
4           21
          ...
54457       21
54458     <NA>
54459       21
54460       30
54461       21
Name: age, Length: 54462, dtype: Int64
```

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 new series or dataframes. 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:

[12]https://github.com/pandas-dev/pandas/issues/33254

```
>>> 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 completely.

Also note that line `jb2['age'] = age_int` does not return anything. You cannot chain on it! The `.assign` method will let you get update or add a column and will also return a dataframe for chaining:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age
...     .str.slice(0,2)
...     .astype(float)
...     .astype('Int64'))
... )
        age  ...   years_of_coding
0        30  ...          1-2 years
1        21  ...          3-5 years
2        30  ...          3-5 years
3      <NA>  ...         11+ years
4        21  ...  Less than 1 year
...     ...  ...               ...
54457    21  ...          1-2 years
54458  <NA>  ...          1-2 years
54459    21  ...         6-10 years
54460    30  ...          3-5 years
54461    21  ...          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 normal function, but often we use a lambda to have the logic inline) has an unseen benefit. This function will accept the *current state* of the dataframe. If you have done any filtering or manipulation in the chain before calling `.assign`, it will be represented in this dataframe.

In the example above, my `lambda` looked like this:

```
.assign(age=lambda df_:df_.age
```

I could have gotten away without a `lambda` in this case 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 style of programming, you will start to think 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 huge 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 chain would enable this.

I'll get off my `.assign` soapbox here. It appears that many have an almost allergic reaction to this style of coding. Yet, they aren't able to present anything better, but the spaghetti code found everywhere else.

21.2 More Column Cleanup

The *are_you_datascientist* column can be converted to a boolean column with the `.replace` method:

```
>>> import numpy as np
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype(float).astype('Int64'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': True, 'No': False, np.nan: False})
...         )
...  .are_you_datascientist
... )
0        False
1         True
2        False
3        False
4        False
         ...
54457    False
54458    False
54459    False
54460     True
54461    False
Name: are_you_datascientist, Length: 54462, dtype: object
```

On to the next column. Let's look at *company_size*. I'll use the `.value_counts` method to see unique values:

```
>>> (jb
...  [uniq_cols]
...  .rename(columns=lambda c: c.replace('.', '_'))
...  .assign(age=lambda df_:df_.age.str.slice(0,2)
...                            .astype(float).astype('Int64'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...              .replace({'Yes': True, 'No': False, np.nan: False})
...          )
...  .company_size
...  .value_counts(dropna=False)
... )
NaN                35037
51–500              4608
More than 5,000     3635
11–50               3507
2–10                2558
1,001–5,000         1934
Just me             1492
501–1,000           1165
Not sure             526
Name: company_size, dtype: int64
```

I'm going to do replacements here as well. It would be possible to split or use a regular expression to pull out these values. I'm going to pull off the left value of the interval. The code will look like this:

```
company_size=lambda df_:df_.company_size.replace({'Just me': 1,
    'Not sure': np.nan, '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'),
```

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(float)
...              .astype('Int64'),
...          are_you_datascientist=lambda df_:df_.are_you_datascientist
...              .replace({'Yes': True, 'No': False, np.nan: False}),
...          company_size=lambda df_:df_.company_size.replace({
...              'Just me': 1, 'Not sure': np.nan,
...              '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'),
...          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).astype(float),
...          years_of_coding=lambda df_:df_.years_of_coding
...              .replace('Less than 1 year', .5).str.extract(r'(\d+)')
...              .astype(float),
...          python_years=lambda df_:df_.python_years
...              .replace('Less than 1 year', .5).str.extract(r'(\d+)')
...              .astype(float),
...          python3_ver=lambda df_:df_.python3_version_most
...               .str.replace('_', '.').str.extract(r'(\d\.\d)')
...               .astype(float),
...          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:

```
>>> jb2
        age are_you_datascientist  ...  years_of_coding python3_ver
0        30                 False  ...              1.0         3.7
1        21                  True  ...              3.0         3.6
2        30                 False  ...              3.0         3.6
3      <NA>                 False  ...             11.0         3.8
4        21                 False  ...              NaN         3.8
...     ...                   ...  ...              ...         ...
54457    21                 False  ...              1.0         3.6
54458  <NA>                 False  ...              1.0         3.7
54459    21                 False  ...              6.0         3.7
54460    30                  True  ...              3.0         3.7
54461    21                 False  ...              1.0         3.8

[54462 rows x 20 columns]
```

Upon inspection, the *team_size* column is still missing quite a few entries. It looks like there are over 5,000 respondents that are employed but neglected to enter a team size:

```
>>> (jb2
...  .query('team_size.isna()')
...  .employment_status
...  .value_counts(dropna=False)
... )
Fully employed by a company / organization        5279
Working student                                    696
Partially employed by a company / organization     482
Self-employed                                      430
Freelancer                                           0
Other                                                0
Retired                                              0
Student                                              0
Name: employment_status, dtype: int64
```

I'm going to use another call to `.assign` to use machine learning to predict the missing values for that column. I will leverage the CatBoost[13] library to do that. A nice feature of this library is that it will accept missing values and also accept 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.

While CatBoost works with data from pandas dataframes, 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.

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 simpler method like `.fillna` to impute these missing values. (You can see I'm kind of 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(**{col:df[col].astype(float)
...                 for col in df.select_dtypes('number')},
...              **{col:df[col].astype(str).fillna('')
...                 for col in df.select_dtypes(['object', 'category'])})
...     )

>>> 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)
```

[13] https://catboost.ai/

```
...     model.fit(X,y, cat_features=cat_idx)
...     pred = model.predict(df.drop(columns=[col]))
...     return df[col].where(~df[col].isna(), pred)
```

With the function to predict the missing values ready let's give it a try:

```
>>> jb2 = (jb
...   [uniq_cols]
...   .rename(columns=lambda c: c.replace('.', '_'))
...   .assign(age=lambda df_:df_.age.str.slice(0,2).astype(float)
...               .astype('Int64'),
...           are_you_datascientist=lambda df_:df_.are_you_datascientist
...               .replace({'Yes': True, 'No': False, np.nan: False}),
...           company_size=lambda df_:df_.company_size.replace({
...               'Just me': 1, 'Not sure': np.nan,
...               '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'),
...           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).astype(float),
...           years_of_coding=lambda df_:df_.years_of_coding
...               .replace('Less than 1 year', .5).str.extract(r'(\d+)')
...               .astype(float),
...           python_years=lambda df_:df_.python_years
...               .replace('Less than 1 year', .5).str.extract(r'(\d+)')
...               .astype(float),
...           python3_ver=lambda df_:df_.python3_version_most
...                .str.replace('_', '.').str.extract(r'(\d\.\d)')
...                .astype(float),
...           use_python_most=lambda df_:df_.use_python_most
...                .fillna('Unknown')
...         )
...   .assign(team_size=lambda df_:predict_col(df_, 'team_size')
...            .astype(int))
...   .drop(columns=['python2_version_most'])
...   .dropna()
... )
>>> jb2
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...              3.0         3.6
2       30                 False  ...              3.0         3.6
10      21                 False  ...              1.0         3.8
11      21                  True  ...              3.0         3.9
13      30                  True  ...              3.0         3.7
...    ...                   ...  ...              ...         ...
54456   30                 False  ...              6.0         3.6
54457   21                 False  ...              1.0         3.6
54459   21                 False  ...              6.0         3.7
54460   30                  True  ...              3.0         3.7
54461   21                 False  ...              1.0         3.8
```

```
[13711 rows x 20 columns]
```

At this point, I'm pretty satisfied with my chain (I would generally develop and debug this chain link by link using Jupyter). What I like to do is 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 find that I need to modify my dataframe further, I will update the tweak function, so all of my changes can be found in one place. This 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!

Here is my what my cleaned up code will look like:

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

>>> import collections

>>> def get_uniq_cols(jb):
...     counter = collections.defaultdict(list)
...     for col in sorted(jb.columns):
...         period_count = col.count('.')
...         if period_count >= 2:
...             part_end = 2
...         else:
...             part_end = 1
...         parts = col.split('.')[:part_end]
...         counter['.'.join(parts)].append(col)
...     uniq_cols = []
...     for cols in counter.values():
...         if len(cols) == 1:
...             uniq_cols.extend(cols)
...     return uniq_cols

>>> def prep_for_ml(df):
...     # remove pandas types
...     return (df
...      .assign(**{col:df[col].astype(float)
...                 for col in df.select_dtypes('number')},
...              **{col:df[col].astype(str).fillna('')
...                 for col in df.select_dtypes(['object', 'category'])})
...     )

>>> def predict_col(df, col):
...     df = prep_for_ml(df)
...     missing = df.query(f'~{col}.isna()')
...     cat_idx = []
...     for i,typ in enumerate(df.drop(columns=[col]).dtypes):
...         if str(typ) == 'object':
...             cat_idx.append(i)
...     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(float)
...                      .astype('Int64'),
...                  are_you_datascientist=lambda df_:df_
...                      .are_you_datascientist
...                      .replace({'Yes': True, 'No': False, np.nan: False}),
...                  company_size=lambda df_:df_.company_size.replace({
...                      'Just me': 1, 'Not sure': np.nan,
...                      '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'),
...                  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).astype(float),
...                  years_of_coding=lambda df_:df_.years_of_coding
...                      .replace('Less than 1 year', .5)
...                      .str.extract(r'(\d+)').astype(float),
...                  python_years=lambda df_:df_.python_years
...                      .replace('Less than 1 year', .5)
...                      .str.extract(r'(\d+)').astype(float),
...                  python3_ver=lambda df_:df_.python3_version_most
...                       .str.replace('_', '.').str.extract(r'(\d\.\d)')
...                       .astype(float),
...                  use_python_most=lambda df_:df_.use_python_most
...                       .fillna('Unknown')
...                 )
...          .assign(team_size=lambda df_:predict_col(df_, 'team_size')
...               .astype(int))
...          .drop(columns=['python2_version_most'])
...          .dropna()
...      )
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url)
>>> jb2 = tweak_jb(jb)
```

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

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

Table 21.1: Dataframe Chapter Methods

21.3 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 my data, so I can load the raw data and then clean it up in two steps.

21.4 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 data and the tweaked data.