Thinking in Pandas

How to Use the Python Data Analysis Library the Right Way

Hannah Stepanek

Apress®

Thinking in Pandas

How to Use the Python Data Analysis Library the Right Way

Hannah Stepanek

Apress®

Thinking in Pandas

Hannah Stepanek
Portland, OR, USA

ISBN-13 (pbk): 978-1-4842-5838-5 ISBN-13 (electronic): 978-1-4842-5839-2
https://doi.org/10.1007/978-1-4842-5839-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Rita Fernando
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5838-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Hannah Stepanek is a software developer with a passion for performance and is an open source advocate. She has over seven years of industry experience programming in Python and spent about two of those years implementing a data analysis project using pandas.

Hannah was born and raised in Corvallis, OR, and graduated from Oregon State University with a major in Electrical Computer Engineering. She enjoys engaging with the software community, often giving talks at local meetups as well as larger conferences. In early 2019, she spoke at PyCon US about the pandas library and at OpenCon Cascadia about the benefits of open source software. In her spare time, she enjoys riding her horse Sophie and playing board games.

About the Technical Reviewer

Jaidev Deshpande is a senior data scientist at Gramener, where he works on automating insight generation from data. He has a decade of experience in delivering machine learning solutions with the scientific Python stack. His research interests lie at the intersection of machine learning and signal processing.

Introduction

Using the pandas Python library requires a shift in thinking that is not always intuitive for those who use it. For beginners, pandas' rich API can often be overwhelming and unclear when determining which solution is optimal. This book aims to give you an intuition for using pandas correctly by explaining how its operations work underneath. We will establish a foundation of knowledge covering information such as Python and NumPy data structures, computer architecture, and performance differences between Python and C. With this foundation, we will then be able to explain why certain pandas operations perform the way they do under certain circumstances. We'll learn when to use certain operations and when to use a more performant alternative. And near the end we'll cover what improvements can be and are being made to make pandas even more performant.

CHAPTER 1

Introduction

We live in a world full of data. In fact, there is so much data that it's nearly impossible to comprehend it all. We rely more heavily than ever on computers to assist us in making sense of this massive amount of information. Whether it's data discovery via search engines, presentation via graphical user interfaces, or aggregation via algorithms, we use software to process, extract, and present the data in ways that make sense to us. pandas has become an increasingly popular package for working with big data sets. Whether it's analyzing large amounts of data, presenting it, or normalizing it and re-storing it, pandas has a wide range of features that support big data needs. While pandas is not the most performant option available, it's written in Python, so it's easy for beginners to learn, quick to write, and has a rich API.

About pandas

pandas is the go-to package for working with big data sets in Python. It's made for working with data sets generally below or around 1 GB in size, but really this limit varies depending on the memory constraints of the device you run it on. A good rule of thumb is have at least five to ten times the amount of memory on the device as your data set. Once the data set starts to exceed the single-digit gigabyte range, it's generally recommended to use a different library such as Vaex.

H. Stepanek, *Thinking in Pandas*, https://doi.org/10.1007/978-1-4842-5839-2_1

The name pandas came from the term panel data referring to tabular data. The idea is that you can make panels out of a larger panel of the data, as shown in Figure 1-1.

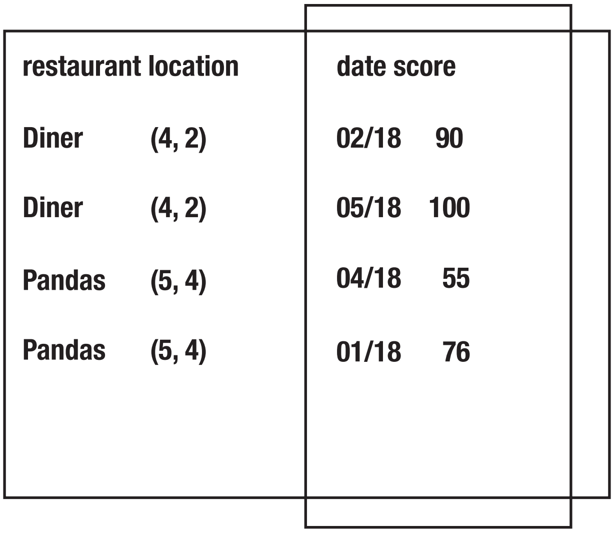


Figure 1-1. *Panel data*

When pandas was first implemented, it was tightly coupled to NumPy, a popular Python package for scientific computing providing an n-dimensional array object for performing efficient matrix math operations. Using the modern implementation of pandas today, you can still see evidence of its tight coupling in the exposition of the Not a Number (NaN) type and its API such as the dtype parameter.

pandas was a truly open source project from the start. The original author Wes McKinney in the Python Podcast.__init__ admitted, in order to foster an open source community and encourage contributions, pandas was tied perhaps a little too closely to the NumPy Python package, but looking back, he wouldn't have done it any different. NumPy was and still is a very popular and powerful Python library for efficient mathematical arithmetic. At the time of pandas inception, NumPy was the main data computation package of the scientific community, and in order to implement pandas quickly and simply in a way that was familiar to its existing user and contributor base, the NumPy package became the underlying data structure of the pandas DataFrame. NumPy is built on C extensions, and while it supplies a Python API, the main computation happens almost entirely in C, which is why it is so efficient. C is much faster than Python because it is a low-level language and thus doesn't consume the memory and CPU overhead that Python does in order to provide all the high-level niceties such as memory management. Even today, developers still rely heavily on NumPy and often perform exclusively NumPy-based operations in their pandas programs.

The difference in performance between Python and C is often not very significant to the average developer. Python is generally fast enough in most cases, and the nicety of Python's high-level language qualities (built-in memory management and pseudo-code like syntax, to name a few) generally outweighs the headaches of having to manage the memory yourself. However, when operating on huge data sets with thousands of rows, these subtle performance differences compound into a much more significant difference. For the average developer, this may seem absolutely outrageous, but it isn't unusual for the scientific research community to spend days waiting for big data computations to run. Sometimes the computations do really take this long; however, other times the programs are simply written in an inefficient way. There are many different ways to do the same thing in pandas which makes it flexible and powerful but also means it can lead developers down less efficient implementation paths that result in very slow data processing.

As developers, we live in an age where compute resources are considered cheap. If a program is CPU heavy, it's easier for us to simply upgrade our AWS instance to a larger machine and pay an extra couple bucks than it is to invest our time to root cause our program and address the overtaxing of the CPU. While it is wonderful to have such readily available compute resources, it also makes us lazy developers. We often forget that 50 years ago computers took up whole rooms and took several seconds just to add two numbers together. A lot of programs are simply fast enough and still meet performance requirements even though they are not written in the most optimal way. Compute resources for big data processing take up a significant amount of energy compared to a simple web service; they require large amounts of memory and CPU, often requiring large machines to run at their resource limits over multiple hours. These programs are taxing on the hardware, potentially resulting in faster aging, and require a large amount of energy both to keep the machines cool and also to keep the computation running. As developers we have a responsibility to write efficient programs, not just because they are faster and cost less but also because they will reduce compute resources which means less electricity, less hardware, and in general more sustainability.

It is the goal of this book in the coming chapters to assist developers in implementing performant pandas programs and to help them develop an intuition for choosing efficient data processing techniques. Before we deep dive into the underlying data structures that pandas is built on, let's take a look at how some existing impactful projects utilize pandas.

How pandas helped build an image of a black hole

pandas was used to normalize all the data collected from several large telescopes to construct the first image of a black hole. Since the black hole was so far away, it would have required a telescope as big as the Earth to capture an image of the black hole directly, so, instead, scientists came up with a way to piece one together using the largest telescopes we have today. In this international collaboration, the largest telescopes on Earth were used as a representative single mirror of a larger theoretical telescope that would be needed to capture the image of a black hole. Since the Earth turns, each telescope could act as more than one mirror, filling in a significant portion of the theoretical larger telescope image. Figure 1-2 demonstrates this technique. These pieces of the larger theoretical image were then passed through several different image prediction algorithms trained to recognize different types of images. The idea was if each of these different image reproduction techniques outputs the same image, then they could be confident that the image of the black hole was the real image (or reasonably close).

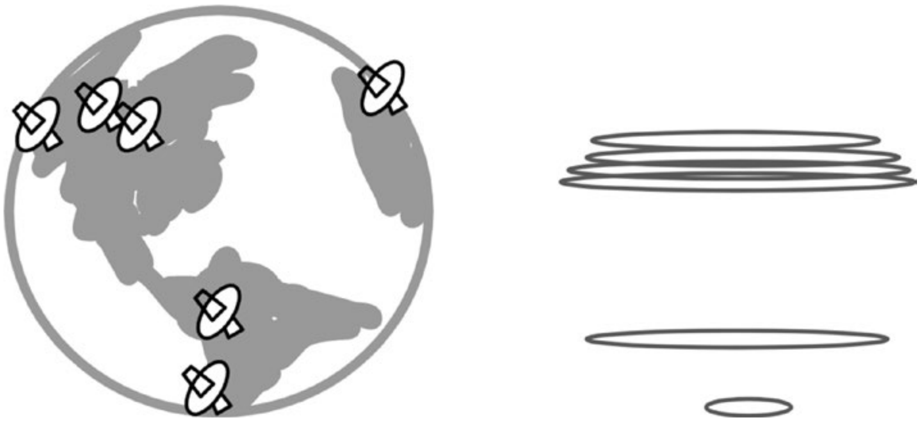


Figure 1-2. *Using the telescopes on Earth to represent pieces of a larger theoretical telescope*

The library is open source and posted on GitHub.[1] The images from radio telescopes were captured on hard disks and flown across the world to a lab at the Massachusetts Institute of Technology where they were loaded into pandas. The data was then normalized, synchronizing the captures from the telescopes in time, removing things like interference from the Earth's atmosphere, and calculating things like absolute phase of a single telescope over time. The data was then sent into the different image prediction algorithms, and finally the first image of a black hole was born.[2]

[1]https://github.com/achael/eht-imaging

[2]https://solarsystem.nasa.gov/resources/2319/first-image-of-a-black-hole/

How pandas helps financial institutions make more informed predictions about the future market

Financial advisors are always looking for an edge up on the competition. Many financial institutions use pandas along with machine learning libraries to determine whether new data points may be relevant in helping financial advisors make better investment decisions. New data sets are often loaded into pandas, normalized, and then evaluated against historical market data to see if the data correlates to trends in the market. If it does, the data is then passed along to the advisors to be used in making financial investment decisions. It may also be passed along to their customers so they can make more informed decisions as well.

Financial institutions also use pandas to monitor their systems. They look for outages or slowness in servers that might impact their trade performance.

How pandas helps improve discoverability of content

Companies collect tons of data on users every day. For broadcast companies' viewership, data is particularly relevant both for showing relevant advertisements and for bringing the right content in front of interested users. Typically, the data collected about users is loaded into pandas and analyzed for viewership patterns in the content they watch. They may look for patterns such as when they watch certain content, what content they watch, and when they are finished watching certain content and looking for something new. Then, new content or relevant product advertisements are recommended based on those patterns. There has been a lot of work recently to also improve business models so that users don’t get put into a bubble (i.e., recommended content isn't just the same type of content they've been watching before or presenting the same opinions). Often this is done by avoiding content silos from the business side.

Now that we've looked at some interesting use cases for pandas, in Chapter 2 we'll take a look at how to use pandas to access and merge data.

CHAPTER 2

Basic Data Access and Merging

There are many ways of accessing and merging DataFrames with pandas. This chapter will go over the basic methods for getting data out of a DataFrame, creating a sub-DataFrame, and merging DataFrames together.

DataFrame creation and access

pandas has a dictionary-like syntax that is very intuitive for those familiar with Python but not with pandas. Each column name is treated as a key, and the row values are returned as the value. The DataFrame object constructor also accepts a dictionary as a way of creating a DataFrame. Note when you get the column from a DataFrame, it points back to the original DataFrame, and this is what allows us to make modifications to the original. This happens despite the syntax that implies we are storing it into a subset of the original as demonstrated near the bottom of Listing 2-1. This is great for memory-based performance since we aren't constantly creating copies of the data.

H. Stepanek, *Thinking in Pandas*, https://doi.org/10.1007/978-1-4842-5839-2_2

Listing 2-1. Example of dictionary syntax

```
>> import pandas as pd
>> account_info = pd.DataFrame({
     "name": ["Bob", "Mary", "Mita"],
     "account": [123846, 123972, 347209],
     "balance": [123, 3972, 7209],
})
>> account_info["name"]
     0    Bob
     1    Mary
     2    Mita
Name: name, dtype: object
>> account_info["name"] = ["Smith", "Jane", "Patel"]
>> account_info
          name  account     balance
     0    Smith 123846      123
     1    Jane  123972      3972
     2    Patel 347209      7209
```

Similarly, a sub-DataFrame can be created by passing in a list of columns as in Listing 2-2.

Listing 2-2. Example of creating a sub-DataFrame

```
>> import pandas as pd
>> account_info = pd.DataFrame({
     "name": ["Bob", "Mary", "Mita"],
     "account": [123846, 123972, 347209],
     "balance": [123, 3972, 7209],
})
>> account_info[["name", "balance"]]
```

```
        name   balance
  0     Bob    123
  1     Mary   3972
  2     Mita   7209
```

The dictionary syntax can lead to confusion later on if you create a sub-DataFrame from the original DataFrame and modify the sub-DataFrame expecting the original to be untouched. pandas makes no guarantees outside of the simple cases presented in Listings 2-1 and 2-2 about whether the resulting object returned by the dictionary syntax is a view or a copy. This is why the loc method is preferred over the dictionary syntax for DataFrames that have multi-indexes or multi-level columns. The loc method, which we'll discuss in the next section, guarantees that you are operating on the original DataFrame and not a copy. Similarly, if you truly want a copy of the DataFrame, you should explicitly create one.

The iloc method

A DataFrame's rows can be accessed via the iloc method which uses a list-like syntax. Listing 2-3 demonstrates this.

Listing 2-3. Example of accessing rows in a DataFrame using iloc

```
>> import pandas as pd
>> account_info = pd.DataFrame({
     "name": ["Bob", "Mary", "Mita"],
     "account": [123846, 123972, 347209],
     "balance": [123, 3972, 7209],
})
>> account_info.iloc[1]
     name        Mary
     account     123972
     balance     3972
```

```
>> account_info.iloc[0:2]
        name   account  balance
    0   Bob    123846   123
    1   Mary   123972   3972
>> account_info.iloc[:]
        name   account  balance
    0   Bob    123846   123
    1   Mary   123972   3972
    2   Mita   347209   7209
```

iloc is used to index a DataFrame via integer position-based indexing. The first position in the iloc function specifies the row indexes, while the second position specifies the column indexes. This means we can select rows as well as columns like in Listing 2-4.

Listing 2-4. Example of accessing rows and columns in a DataFrame using iloc

```
>> import pandas as pd
>> account_info = pd.DataFrame({
    "name": ["Bob", "Mary", "Mita"],
    "account": [123846, 123972, 347209],
    "balance": [123, 3972, 7209],
})
>> account_info.iloc[1, 2]
    3972
>> account_info.iloc[1, 2] = 3975
>> account_info.iloc[1, 2]
    3975
>> account_info.iloc[:, [0, 2]]
        name   balance
    0   Bob    123
    1   Mary   3975
    2   Mita   7209
```

iloc also accepts a Boolean array. In Listing 2-5, we grab all odd rows by taking the modulus of each row index and converting it to a Boolean.

Listing 2-5. Example of accessing rows and columns in a DataFrame using iloc

```
>> import pandas as pd
>> account_info = pd.DataFrame({
     "name": ["Bob", "Mary", "Mita"],
     "account": [123846, 123972, 347209],
     "balance": [123, 3972, 7209],
})
>> account_info.iloc[account_info.index % 2 == 1]
           name  account    balance
     1     Mary  123972     3972
```

iloc also accepts a function; however, this function is called once with the entire DataFrame, and there's little difference between passing it in and simply calling the function beforehand so we won't go over that here.

iloc can come in quite handy when working with multi-indexed and multi-level column DataFrames since levels are integer values. Let's review an example and break it down. Here we specify the rows we want to grab as ":" meaning we want all rows, and we use a Boolean array to specify the columns. We grab the values for the multi-level column "data" which are ["score", "date", "score", "date"] and then create a Boolean array by specifying that the value must equal "score". This is broken down into stages in Listing 2-6 so it is easier to follow.

Listing 2-6. Extracting a sub-DataFrame from a multi-indexed multi-level column DataFrame using iloc

```
>> restaurant_inspections
     inspection             0                 1
     data                 score  date      score  date
     restaurant location
     Diner      (4, 2)    90     02/18     100    05/18
     Pandas     (5, 4)    55     04/18     76     01/18

>> score_columns = (
     restaurant_inspections.columns.get_level_values("data")
     == "score")
>> score_columns
     [True, False, True, False]
>> restaurant_inspections.iloc[:, score_columns]
     inspection                0          1
     data                      score      score
     restaurant location
     Diner      (4, 2)         90         100
     Pandas     (5, 4)         55         76
```

The loc method

loc is similar to iloc, but it allows you to index into a DataFrame via column names or labels. Listing 2-7 shows the loc equivalents to Listing 2-4.

Listing 2-7. Example of accessing rows and columns in a DataFrame using loc

```
>> import pandas as pd
>> account_info = pd.DataFrame({
     "name": ["Bob", "Mary", "Mita"],
     "account": [123846, 123972, 347209],
     "balance": [123, 3972, 7209],
})
>> account_info.loc[1, "balance"]
     3972
```

```
>> account_info.loc[:, ["name", "balance"]]
         name   balance
    0    Bob    123
    1    Mary   3972
    2    Mita   7209
```

loc can also be used on multi-indexed multi-level column DataFrames and just like iloc supports Boolean arrays. Listing 2-8 demonstrates this.

Listing 2-8. Example of extracting a sub-DataFrame from a multi-indexed multi-level column DataFrame using loc

```
>> import pandas as pd
>> account_info
    account           0                     1
    account_info      number   balance      number   balance
    name   username
    Bob    smithb     123846   123          123847   450
    Mary   mj100      123972   3972         123973   222
    Mita   patelm     347209   7209

>> account_info.loc[
    ("Mary", "mj100"), pd.IndexSlice[:, "balance"]
]
    0    balance    3972
    1    balance    222
```

At the end of the dictionary syntax section, it was mentioned that the loc method is preferred over the dictionary syntax for complex DataFrames. Let's look at what's happening underneath when we use each syntax to explain why that is. Listing 2-9 shows what each access method translates into underneath when operating on a more complex DataFrame. Note in the second half of Listing 2-9 where the dictionary syntax is used, the code underneath uses the __getitem__ method and then calls __setitem__ on it.

This is in opposition to the loc method which calls __setitem__ directly. It is the __getitem__ that cannot be trusted here and makes no guarantees about whether it returns a copy or what's called a view that points back to the original DataFrame. In simple cases where there are not multiple levels of columns, the code underneath in both these cases would look the same, but in the more complex case seen here, the dictionary syntax results in chained indexing and calls the unpredictable __getitem__.

Listing 2-9. A comparison of using loc vs. dictionary syntax to extract a sub-DataFrame

```
"""
The code below is equivalent to:

account_info.__setitem__(
      (slice(None), (0, 'balance')),
      NEW_BALANCE,
)
"""
account_info.loc[:, (0, "balance")] = NEW_BALANCE

"""
The code below is equivalent to:

account_info.__getitem__(0).__setitem__('balance', NEW_BALANCE )
"""
account_info[0]["balance"] = NEW_BALANCE
```

Quite often you may have data from multiple sources that you need to combine into a single DataFrame. Now that you know how to do some basic data access, we'll look at different methods for combining data from different DataFrames together.

Combining DataFrames using the merge method

Merge works the same way as a relational database join and even has the familiar options: outer, inner, left, and right. Merge right is essentially the same as merge left, but the DataFrames are simply passed in in reverse order so this chapter won't provide an explicit example for merge right.

Inner merge is used when you want to find the intersection between two pandas DataFrames (Figure 2-1). In Listing 2-10, for example, we are trying to find the data that is present in both data sets or in this case the buildings that were standing in 1844 that are still standing today.

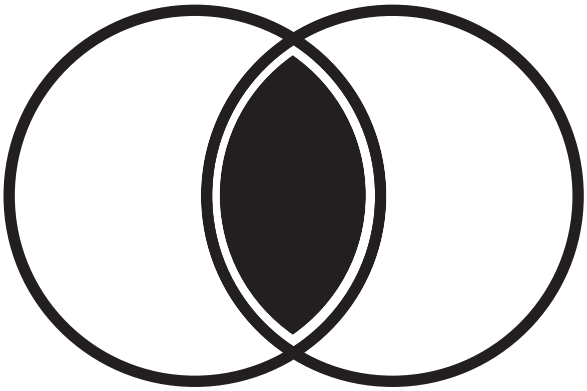


Figure 2-1. *Venn diagram of inner merge*

Listing 2-10. Finding 1844 buildings that are still standing in 2020 using an inner merge

```
>> import pandas as pd
>> building_records_1844
                       established
     building
     Grande Hotel      1830
     Jone's Farm       1842
     Public Library    1836
     Marietta House    1823
>> building_records_2020
                          established
     building
     Sam's Bakery         1962
     Grande Hotel         1830
     Public Library       1836
     Mayberry's Factory   1924
>> cols = building_records_2020.columns.difference(
               building_records_1844.columns
)
>> pd.merge(
     building_records_1844,
     building_records_2020[cols],
     how='inner',
     on=["building"],
)
                          established
     building
     Grande Hotel         1830
     Public Library       1836
```

In Listing 2-11, we are merging two data sets of gene samplings together, meaning we want all the data from both in the same data set without duplication. We can achieve this by doing an outer merge (Figure 2-2).

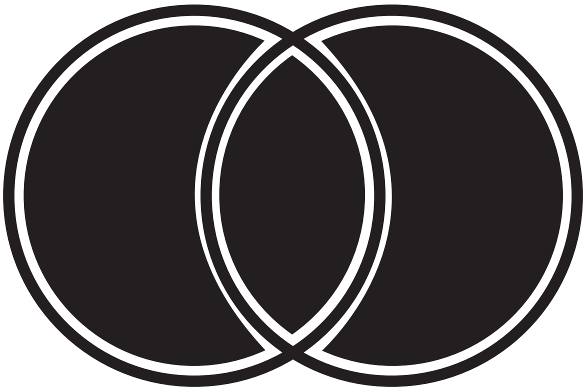


Figure 2-2. *Venn diagram of outer merge*

Listing 2-11. Merging two gene samplings together without duplicating the data in common using outer merge

```
>> import pandas as pd
>> gene_group1
                  FC1           P1
     id
     Myc          2             0.05
     BRCA1        3             0.01
     BRCA2        8             0.02
>> gene_group2
                  FC2           P2
     id
     Myc          2             0.05
     BRCA1        3             0.01
     Notch1       2             0.03
     BRCA2        8             0.02
```

```
>> pd.merge(
     gene_group1,
     gene_group2,
     how='outer',
     on=["id"],
)
                FC1         P2          FC2         P2
     id
     Myc        2           0.05        2           0.05
     BRCA1      3           0.01        3           0.01
     BRCA2      8           0.02        8           0.02
     Notch1     NaN         NaN         2           0.03
```

In Listing 2-12, we are updating the modern building records with more accurate historical data. The historical record contains the exact established date that we would like to use to update the modern record which contains only an estimate. First, we use a merge left to add the new more accurate established date as a new column in the data set (Figure 2-3). The merge left is useful in this case since we only care about updating the established date in the modern record for buildings that still exist. Note we are also taking advantage of the suffixes parameter to provide names for the column names. This is advantageous so we don't have to rename the column to be the same as the original column after we're done performing the operation. Once the merge is complete, then we need to merge the data from the two established columns together. This is done by replacing all the missing values (i.e., NaNs) in the old established column with the values in the modern record. So, if the historical record has an established date, then we use that; otherwise, we fall back on the modern record's established date. Finally, the modern record's original established column is deleted in favor of the new column that contains the merged values from the modern record and the historical record.

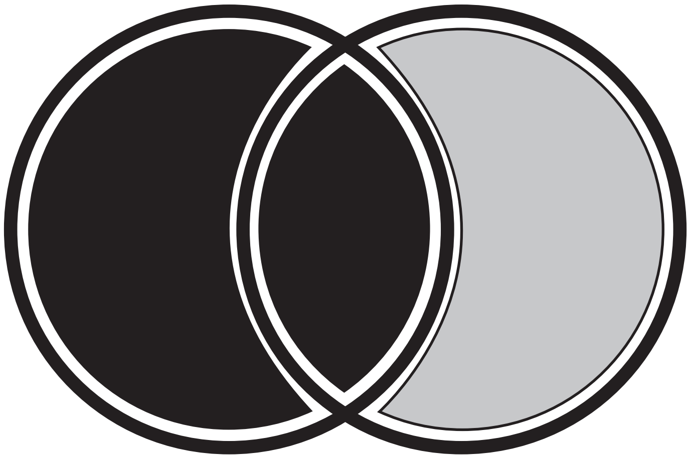


Figure 2-3. *Venn diagram of merge left*

Listing 2-12. Updating modern records with more accurate historical data using a merge left

```
>> import pandas as pd
>> building_records_1844
    building          established
    Grande Hotel      1832
    Jone's Farm       1842
    Public Library    1836
    Marietta House    1823
>> building_records_2020
    building             established
    Sam's Bakery         1962
    Grande Hotel         1830
    Public Library       1836
    Mayberry's Factory   1924
```

```
>> merged_records = pd.merge(
     building_records_2020,
     building_records_1844,
     how='left',
     right_on="building",
     left_on="building",
     suffixes=("_2000", ""),
)
>> merged_records
     building              established_2000     established
     Sam’s Bakery          1962                 NaN
     Grande Hotel          1830                 1832
     Public Library        1835                 1836
     Mayberry’s Factory    1924                 NaN
>> merged_records["established"].fillna(
     merged_records["established_2000"],
     inplace=True,
)
>> del merged_records["established_2000"]
>> merged_records
     building              established
     Sam's Bakery          1962
     Grande Hotel          1832
     Public Library        1836
     Mayberry's Factory    1924
```

In Listing 2-13, we are planning on running a third medical trial, and we want to generate a list of participants that are eligible. Only participants who have participated in the previous trial a or b, but not both, will be eligible for the third trial. In order to generate a list of eligible patients, we need to use an anti-join method when performing a merge of trial a and trial b patients (Figure 2-4). pandas merge method provides a parameter called indicator that adds an additional column called _merge into the resulting DataFrame that reports whether the key is present in left_only, right_only, or both DataFrames. This comes in handy in this particular case as we wish to do a somewhat unconventional merge. Using the query method, we are able to select rows where the _merge value is not both and then drop the _merge column. This can be done all in one line as shown at the end of Listing 2-13 but is broken up into two steps beforehand so you can see how it works underneath.

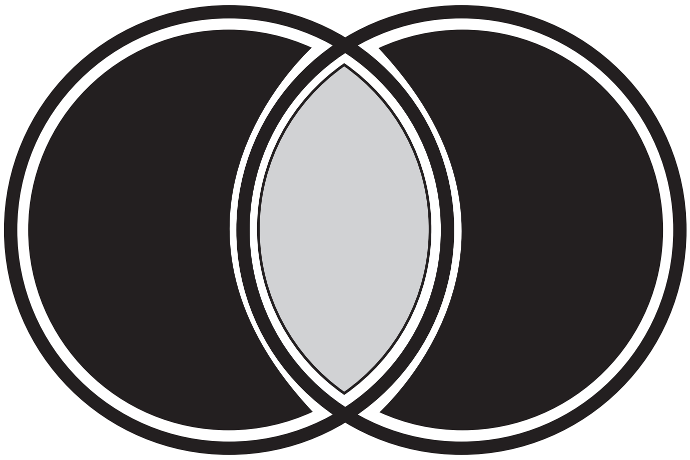


Figure 2-4. *Venn diagram of anti-join*

Listing 2-13. Eliminating patients who participated in both trials using anti-join merge method

```
>> import pandas as pd
>> trial_a_records
              name
    patient
    230858     John
    237340     May
    240932     Catherine
    124093     Ahmed
```

```
>> trial_b_records
                name
     patient
     210858      Abi
     237340      May
     240932      Catherine
     154093      Julia

>> both_trials = pd.merge(
     trial_a_records,
     trial_b_records,
     how='outer',
     indicator=True,
     right_index=True,
     left_index=True,
     on="name",
)
                name            _merge
     patient
     230858      John            left_only
     237340      May             both
     240932      Catherine       both
     124093      Ahmed           left_only
     210858      Abi             right_only
     154093      Julia           right_only
>> both_trials.query('_merge != "both"').drop('_merge', 1)
                name
     patient
     230858      John
     124093      Ahmed
     210858      Abi
     154093      Julia
```

```
>> both_trials = pd.merge(
     trial_a_records,
     trial_b_records,
     how='outer',
     indicator=True,
     right_index=True,
     left_index=True,
     on="name",
).query('_merge != "both"').drop('_merge', 1)
```

Combining DataFrames using the join method

The pandas join method is just a wrapper around merge, and it provides the same basic merging methods: left, right, outer, and inner. It allows you to perform merge operations on multi-index DataFrames automatically without needing to specify the indexes to merge on. When doing a left join, it automatically uses the indexes from the left DataFrame to join on, and the same is true for the right DataFrame. Since join is performed on a DataFrame as opposed to merge where you pass in both DataFrames explicitly, join's default is to merge on the "right" DataFrame's indexes. In this case, "right" is the passed in DataFrame. This is in opposition to merge's default which is an inner join.

Using merge, it is possible to not specify an explicit key to merge on. In cases where you are merging two DataFrames with the same data and do not wish to have duplicated columns for the left and right DataFrames but simply merge the two data sets together, merge is preferable over join. Because join calls merge underneath, it explicitly specifies the keys to merge on, thus eliminating the possibility of not outputting duplicated columns for DataFrames that share common column names. A basic rule to follow here is use merge if you are not joining on the index.

Listing 2-14 plays off of a previous inner merge example in Listing 2-10, but unlike the previous example where the records of the buildings in common matched, this time there are discrepancies. A join is desirable for a couple reasons in this scenario. Firstly, the data has already been indexed according to the unique building and join will automatically pick up the indexes and use those to join the two sets of data. Secondly, there are discrepancies in the data, and thus we wish to see columns from both DataFrames side by side in the output DataFrame so we can compare them.

Listing 2-14. Highlighting discrepancies in established date of 1844 buildings that are still standing in 2020 using an inner join

```
>> import pandas as pd
>> building_records_1844
                                  established
     building         location
     Grande Hotel     (4,5)       1831
     Jone's Farm      (1,2)       1842
     Public Library   (6,4)       1836
     Marietta House   (1,7)       1823
>> building_records_2020
                                     established
     building           location
     Sam's Bakery        (5,1)       1962
     Grande Hotel        (4,5)       1830
     Public Library      (6,4)       1835
     Mayberry's Factory  (3,2)       1924
>> building_records_1844.join(
     building_records_2020,
     how='inner',
     rsuffix="_2000",
)
```

```
                                established   established_2000
     building          location
     Grande Hotel      (4,5)    1831          1830
     Public Library    (6,4)    1836          1835
```

Combining DataFrames using the concat method

Concatenate is a simple way of combining two DataFrames together. Listing 2-15 demonstrates a simple concatenation of the same data from multiple sources. Concatenate has many options including the option join which specifies whether to use an outer or inner merge and axis which specifies whether to merge across columns with axis=1 or rows with axis=0. By default, concatenate performs an outer merge across rows. Note in Listing 2-15, location (6,4) is present in both county_a and county_b data, and in the concatenated result, it is repeated in the index.

Listing 2-15. Joining two DataFrames together using concat

```
>> import pandas as pd
>> temp_county_a
                  temp
     location
     (4,5)         35.6
     (1,2)         37.4
     (6,4)         36.3
     (1,7)         40.2
```

```
>> temp_county_b
                  temp
    location
    (6,4)         34.2
    (0,4)         33.7
    (3,8)         38.1
    (1,5)         37.0

>> pd.concat([temp_county_a, temp_county_b])
                  temp
    location
    (4,5)         35.6
    (1,2)         37.4
    (6,4)         36.3
    (1,7)         40.2
    (6,4)         34.2
    (0,4)         33.7
    (3,8)         38.1
    (1,5)         37.0
```

Concatenate can also be used in a more complicated manner to create multi-level columns or indexes. Listing 2-16 demonstrates a concatenation where each DataFrame being concatenated is a value in a multi-level column. Note in Listing 2-16, device_a and device_b data are temperature measurements at the same locations. Here we specify axis=1 so that the two DataFrames are outer merged across the columns. The key parameter in Listing 2-16 tells concatenate to treat the two temperature columns as different columns even though they are named the same and as a bi-product creates a multi-level column. Performing an outer merge across the columns and putting the temperatures of each device into separate columns means the concatenated result has no repeated location index values. This is in opposition to Listing 2-15 where the resulting index values were repeated and the two DataFrames were simply stacked on top of each other.

Listing 2-16. Joining two DataFrames together using a multi-level column concat

```
>> import pandas as pd
>> temp_device_a
                  temp
     location
     (4,5)         35.6
     (1,2)         37.4
     (6,4)         36.3
     (1,7)         40.2
>> temp_device_b
                   temp
     location
     (4,5)         34.2
     (1,2)         36.7
     (6,4)         37.1
     (1,7)         39.0

>> pd.concat(
     [temp_device_a, temp_device_b],
     keys=["device_a", "device_b"],
     axis=1,
)
                  device_a       device_b
                  temp           temp
     location
     (4,5)        35.6           34.2
     (1,2)        37.4           36.7
     (6,4)        36.3           37.1
     (1,7)        40.2           39.0
```

There are many different ways to combine DataFrames together and extract a sub-DataFrame in pandas. Which method you use really depends on your particular use case. The examples presented here are a representative sample, but you should consult the documentation of each method as there are some parameters that were not explicitly covered in this chapter, such as sorting.[1]

[1]https://pandas.pydata.org/pandas-docs/version/0.25/user_guide/merging.html

CHAPTER 3

How pandas Works Under the Hood

As with any program language, it's important to understand what is going on underneath because it helps you write more explicit, simpler, performant, and correct code. The building blocks of a language (its data structures and API) when used correctly can make an operation trivial and when used incorrectly can make an operation overly complex if not impossible. Python packages are no different.

A programming language is simply text that is easily readable and writable by humans that can be translated into CPU instructions that are understood by machines. As programming languages have become increasingly high level (farther removed from the machine code that computers understand), the necessity for developers to understand the translation has become less essential. A byproduct of this, however, is that software can be written in a non-performant and atypical way without developers being forced to address the underlying issues. Non-performant solutions on modern computing platforms typically are not visibly non-performant until they are scaled to handle much more data. Big data software typically operates at a scale where the performance impact is visible because it processes huge data sets, often repeating a small quick operation so many times that its performance becomes significant. When working at this scale, it's important to understand the data structures and performance optimizations available to you, in order to get the most out of your machine with the least amount of effort. This starts with understanding the performance of data structures in the language you are using.

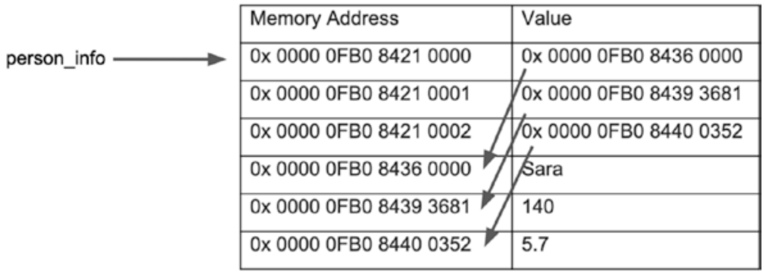
Python data structures

Python's data structures are its building blocks. Choosing the right data structure for the problem you are trying to solve is essential to writing correct and performant code.

First, we'll look at tuples. They are in many ways comparable to a C array and in fact are an array underneath. They are an iterable, meaning you can loop over them and look at each value, though they are immutable, meaning the values cannot be changed once a tuple has been created. They are great at storing static chunks of related information such as metadata. When a small tuple is no longer referenced in a program and its memory can be freed, Python keeps it around and adds it to the tuple free list so that it can be used again. This saves time in the interpreter as it does not have to re-allocate the memory for a new tuple. Underneath, tuples translate into a fixed-size array, meaning an array of pointers whose size cannot be changed. Listing 3-1 shows a code example of a tuple, and Figure 3-1 illustrates its representation in memory. Each index is represented as a memory addresses 0x0000 0FB0 8421 0000 through 0002 in Listing 3-1. The value at each index is a memory address or pointer to the actual value in memory. This is how Python is able to store non-like types into the same underlying array object. Each address or index of the tuple contains a pointer and only has to make room for the pointer rather than the actual value.

Listing 3-1. Example tuple

```
person_info = ("Sara", 140, 5.7)
```



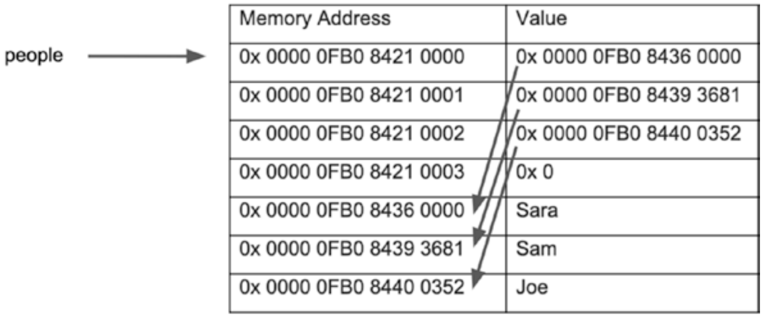
Memory Address	Value
0x 0000 0FB0 8421 0000	0x 0000 0FB0 8436 0000
0x 0000 0FB0 8421 0001	0x 0000 0FB0 8439 3681
0x 0000 0FB0 8421 0002	0x 0000 0FB0 8440 0352
0x 0000 0FB0 8436 0000	Sara
0x 0000 0FB0 8439 3681	140
0x 0000 0FB0 8440 0352	5.7

***Figure 3-1.** A representation of Listing 3-1 in memory*

A list, simply put, is a mutable tuple. A list is an array of a fixed size underneath, but when the number of elements exceeds the size that Python originally allocated, it creates a new fixed-size array with space for more elements and copies the elements from the old array into the new array. The allocated size is base 2, so if you initialize a list with five values, underneath Python will allocate the fixed-size array to hold eight references. If you then append four more values, on the fourth append, the fixed-size array will be re-allocated to be double the size (16) and the previous value references will be copied into the new array including the new fourth value reference that didn't fit in the previous size 8 array. Unlike tuples, they do not have any behind-the-scenes performance optimizations for reusing freed memory. This is due to the fact that lists are mutable—meaning their values can be changed after creation and they are not of fixed size. Listing 3-2 shows an example of a list, and Figure 3-2 illustrates its representation in memory. Just like a tuple, the list contains references to the values rather than the values themselves. Note since the list was initialized with three values, the fixed-size array underneath is of length 4, so at index 3 (element 4), there is an empty placeholder value of 0x0.

Listing 3-2. Example list

```
people = ["Sara", "Sam", "Joe"]
```



Memory Address	Value
0x 0000 0FB0 8421 0000	0x 0000 0FB0 8436 0000
0x 0000 0FB0 8421 0001	0x 0000 0FB0 8439 3681
0x 0000 0FB0 8421 0002	0x 0000 0FB0 8440 0352
0x 0000 0FB0 8421 0003	0x 0
0x 0000 0FB0 8436 0000	Sara
0x 0000 0FB0 8439 3681	Sam
0x 0000 0FB0 8440 0352	Joe

Figure 3-2. *A representation of Listing 3-2 in memory*

A dictionary is a hash table. The keys are hashed to a memory address or a particular index in an array. Depending on the number of keys in the dictionary, a certain number of bits of the hash are used to determine the index. Listing 3-3 shows an example dictionary and illustrates its representation in memory. In the example in Listing 3-3, 2 bits are used as there are only two elements. The values in the hashed array are the indexes into a second array that contains the complete hash, the key, and the value of the key in the dictionary. Note each element in the hashed index array only takes up 64 bytes (the size of a pointer) compared to the array of data which takes up much more than that because it includes the hash, the key, and the value. By keeping a separate array of indexes to the data array, the dictionary implementation is able to save space in memory by using a smaller hashed index array to be a placeholder for the unused keys rather than the larger data array. It is also able to grow the hashed index array as it needs to, similar to how a list grows as more elements are inserted, as opposed to allocating a bunch of memory for hash indexes that don't exist. Note the hashed index array can be completely re-initialized when the number of hashed bits increases, independent of the data array. Also note because of this implementation, the dictionary keys are now sorted by insertion time in the data array, and as of Python 3.7, dictionary keys are now guaranteed to be in insertion order.

Listing 3-3. Example dictionary and its representation in memory

```
word_alphabet = {"a": "apple", "b": "banana"}

Hash-Index          Data
None                hash("a"), "a", "apple"
0                   hash("b"), "b", " banana"
1
None
```

Sets are basically the same implementation as dictionaries but without a value. They are a data structure for tracking membership and perform almost all the operations in mathematical set theory such as union and intersection. Listing 3-4 shows an example set and how it is represented in memory.

Listing 3-4. Example set and its representation in memory

```
alphabet = {"a", "b"}

Hash-Index          Data
None                hash("a"), "a"
0                   hash("b"), "b"
1
None
```

There are also many other data structures including integers, floats, Booleans, and strings. These pretty much directly translate into their c-type equivalents underneath and aren't really worth going over here. Something that is worth mentioning though is some of these have special built-in caching in Python.

Python has a string and integer cache. Take, for example, str1 and str2 in Listing 3-5. They are both set to the value "foo" but underneath they are pointing at the same memory location. This means that rather than creating a new string that is an exact copy of str1 and duplicating the memory, the new string will simply point to the existing string value. This is demonstrated here by the assertion line where the "is" property is used to compare the references or pointers of the two strings for equality.

Listing 3-5. str1 and str2 are pointing to the same memory location

```
str1 = "foo"
str2 = "foo"
assert(str1 is str2)
```

The string cache, however, only works on strings containing letters, numbers, and underscore. This is advantageous to know when working with large data sets that may contain other letters. You can save a lot of memory by eliminating characters from string values in the data set that prevent string caching. See Listing 3-6.

Listing 3-6. str1 and str2 are not pointing to the same memory location

```
str1 = "foo bar"
str2 = "foo bar"
assert(str1 is not str2)
```

The integer cache works in a similar way; it only caches integers between and including –5 and 256. See Listing 3-7.

Listing 3-7. int1 and int2 are pointing to the same memory location but int3 and int4 are not

```
int1 = 22
int2 = 22
int3 = 257
int4 = 257
assert(int1 is int2)
assert(int3 is not int4)
```

Again, this can be advantageous to know as breaking a single column of large numbers into two columns representing the number in scientific notation, for example, may lead to memory savings.

The performance of the CPython interpreter, Python, and NumPy

The Python interpreter that most developers typically install is called CPython. It is the interpreter that is recommended for use with pandas as pandas is highly dependent on C for performance optimizations. CPython is implemented in C and translates Python code into what's called bytecode, an intermediate low-level format that is run on the Python Virtual Machine. There are many different Python interpreters including Jython, IronPython, and PyPy which are implemented in Java, C#, and RPython (a restricted subset of Python), respectively. PyPy is a Just-In-Time or JIT compiler, which means it compiles the Python code into machine code as it runs. This is in opposition to CPython that runs the bytecode on the Python Virtual Machine and calls into pre-compiled C extensions. PyPy is generally faster than CPython because it runs low-level optimized machine code as opposed to parsing the bytecode on the Python Virtual Machine. Unfortunately, PyPy does not fully support pandas at this time.

Python is a high-level language, which means it's easy to read and fast to implement. This also means it is slow compared to some lower-level languages because of all these self-managing niceties. These niceties include the garbage collector, the global interpreter lock, and dynamic typing, but they don't come for free. The garbage collector is responsible for freeing memory that is no longer in use so that it can be used again. The global interpreter lock, also known as the GIL, protects objects from being accessed by multiple threads at the same time. Dynamic typing allows the same variable to hold different types of values. Because CPython is implemented in C, it is C compatible and thus allows Python to call into more performant extensions written in C. But why is C so much more performant than Python? There are a several reasons why Python is relatively slower than C; it is interpreted rather than compiled, it has a global interpreter lock, it allows dynamic typing, and it has a built-in garbage collector.

Interpreting Python into bytecode is like compiling C into object files only bytecode is run on the Python Virtual Machine and machine code is run on the CPU. The interpretation of the Python code at runtime adds extra overhead that makes Python generally run slower than C. There are several phases of interpretation: a tokenizer that converts Python code to a token stream, a lexical analyzer that runs syntax analysis, a bytecode generator that optimizes and converts the Python code to bytecode (.pyc files), and the bytecode interpreter that interprets the bytecode stream and maintains the state of the bytecode interpreter.

If you've ever edited a Python file and re-run your program only to notice that it didn't run with the change you just made, you'll understand that the Python interpreter caches the bytecode in .pyc files. Deleting the .pyc files before re-running forces the interpreter to re-interpret the Python code into bytecode—in essence, forcing the Python interpreter to clear its bytecode cache. The bytecode cache decreases the overhead of runtime interpretation by not re-interpreting the Python code into bytecode unless the Python code has changed. On versions of Python that pre-dated 3.3, however, the method used to obtain the timestamp on the .py file did not match the windows operating system timestamp which resulted in a timestamp that was younger than the .pyc file. This meant that the interpreter did not re-interpret the .py file into bytecode. Similarly, if you remove a .py file but still import it in your code, the imports may continue to work because the .pyc file is still present on your system. While these two issues may sway you from wanting to use the bytecode cache, the cache plays an important role in improving interpretation performance and typically operates in the background without developers being the wiser. The bytecode cache is particularly advantageous for third-party libraries where the code is installed and not expected to change or the library owners do not want to expose the Python source code.

Python has a global interpreter lock or what's known to most as the GIL. In order to understand why the GIL exists, you really have to take a step back in time and explore what was happening in computer science at the time the GIL was invented. It began with the invention of threads. Anticipating the future of computing, software introduced the concept of multi-threading prior to multi-core CPUs. Threading enabled a program to run processes in parallel that operated on the same memory space. It was a fantastic way to improve performance of CPU-intensive computation.

For example, say we want to calculate how many times the name Tiffany appears in a list. You could do this by counting how many times Tiffany appears in the list all by yourself, or you could break up the list into sub-lists and give one to each of your friends, and each time one of you sees the word Tiffany, increase the running total on the whiteboard by one. In this example, you and your friends are the threads and the count on the whiteboard is the shared memory. Generally, breaking up the problem into smaller chunks and using threads to parallelize the computation is faster than computing the whole thing on one thread. The problem you may encounter here is when one of your friends has erased the total on the whiteboard and is updating it at the same time you wish to update it. Fortunately, you are smart enough to realize this is happening and wait until your friend has finish updating the total before you try to do so. Computers, on the other hand, need to be told how to handle this or need to be what's called thread safe. If a piece of software encounters this same scenario, it's going to simply squash the value. This is what's known as a race condition.

In Figure 3-3, time is represented on the y axis as t, and there are two threads that each wish to increment the total at nearly the same time. Total in this example is located in shared memory, meaning both threads can access that value. Here you can see that Thread 1 increments the counter first, followed by Thread 2. However, Thread 2 effectively does not increment the counter because at the time that it grabbed the total to increment, Thread 1's increment had not taken effect yet. This has an end result of the total being one less than it should be (6 instead of 7).

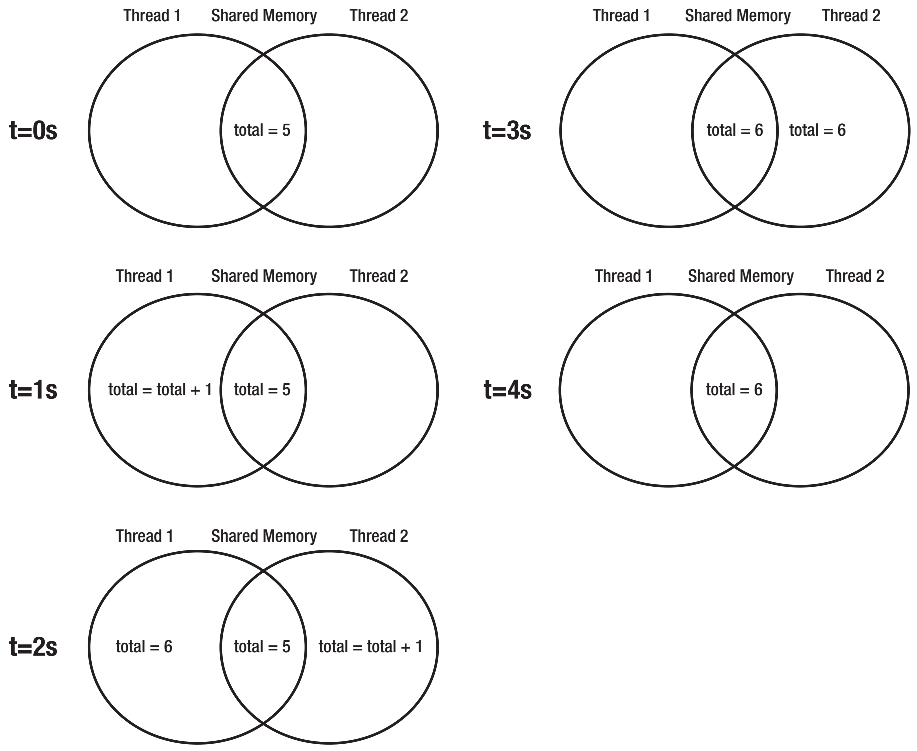


Figure 3-3. *A demonstration of a race condition on increment of total between Thread 1 and Thread 2*

Similar to this running total example, the CPython interpreter uses a reference garbage collector technique where it keeps track of the number of places that refer to an object. The garbage collector is responsible for keeping track of allocated memory and deallocating it when it is no longer used. It does this by keeping a running total of all the places that reference each object in the program. When there are no more references, meaning the reference count is zero, then the memory is deallocated, meaning it is freed and available to store something else. In Listing 3-8, string foo's reference count would be two because there are two variables that reference it.

Listing 3-8. Example of creating two references to the string foo

```
ref1 = "foo"
ref2 = "foo"
```

Recall that the string cache is at play here, and because of that, both ref1 and ref2 point to the same value underneath.

When we delete ref2, string foo's reference count is 1, and when we delete ref1, string foo's reference count is 0 and the memory can be freed. This is demonstrated in Listing 3-9.

Listing 3-9. Example of deleting references to foo that were created in Listing 3-8

```
delete(ref2)       # reference count = 1 after this line executes
delete(ref1)       # reference count = 0 after this line executes
```

Not all objects are freed when their references reach 0 though because some never reach 0. Take, for example, the scenario presented in Listing 3-10 which tends to happen quite often when working with classes and objects in Python. In this scenario, exec_info is a tuple and the value at the third index is the traceback object. The traceback object contains a reference to the frame, but the frame also contains a reference to the exc_info variable. This is what's known as a circular reference, and since there is no way to delete one without breaking the other, these two objects must be garbage collected. Periodically the garbage collector will run, identify, and delete circular referenced objects like this.

Listing 3-10. Example of creating a circular reference

```
import sys

try:
      raise Exception("Something went wrong.")
except Exception as e:
      exc_info = sys.exc_info()
      frame = exc_info[2].tb_frame # create a third reference

assert(sys.getrefcount(frame) == 3)
del(exc_info)
assert(sys.getrefcount(frame) == 3)
```

Keeping track of these references does not come for free. Each object has an associated reference counter which takes up space, and each reference made in the code takes up CPU cycles to compute the appropriate increment or decrement of the object's reference count. This is partially why, if you compare the size of an object in Python to the size of an object in C, the sizes are so much larger in Python and also why Python is slower to execute than C. Part of those extra bytes and extra CPU cycles are due to the reference count tracking. While the garbage collector does have performance implications, it also makes Python a simple language to program in. As a developer, you don't have to worry about keeping track of memory allocation and deallocation; the Python garbage collector does that for you.

In a multi-threaded application, reference counts have the same problem as the total has in Figure 3-3. A thread may create a new reference to an object in the shared memory space at the same time as another thread and a race condition occurs where the reference count ends up only being incremented once instead of twice. When this happens, it can ultimately lead to the object being freed from memory before it should be (because the race condition leads to the object's reference count being incremented by one instead of two). In other cases when there is a race condition on a decrement of the reference counter, this can lead to a memory leak where the memory is never destroyed because the reference count is one larger than it should be. So, as you can see here, running a multi-threaded application can not only have consequences on the data that the program is operating on but also within the Python interpreter itself.

Traditionally, race conditions are solved through locks. This is effectively what you were doing subconsciously—that is, waiting for your friend to finish updating the total before you updated it yourself. In software, this is done through a shared memory lock. When a thread needs to update the total, it acquires the lock, updates the total, and then releases the lock. Meanwhile, the other thread waits until the lock is free, acquires the lock, and then also updates the total. This interaction is demonstrated in Figure 3-4.

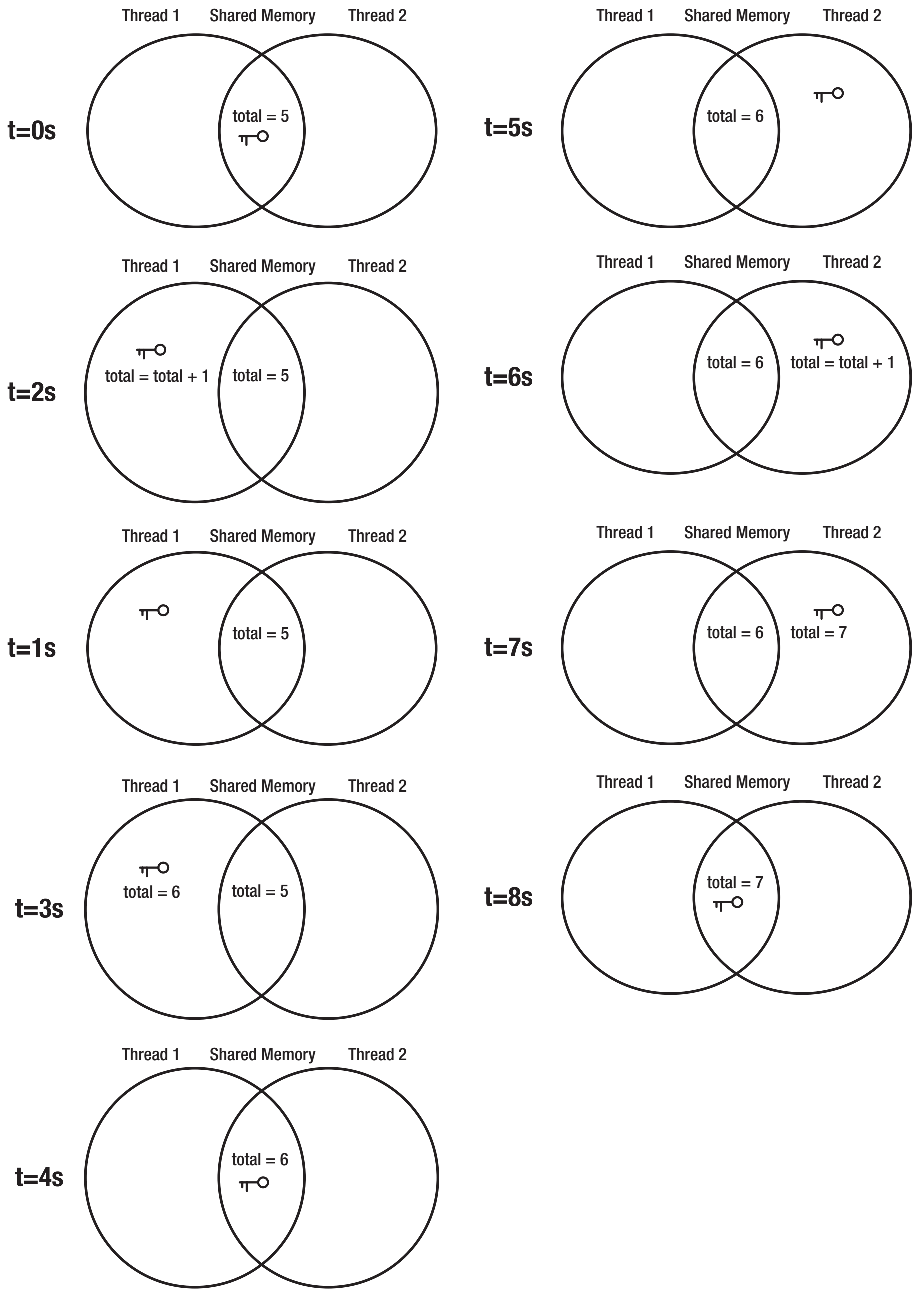


Figure 3-4. *A demonstration of a lock on total shared between Thread 1 and Thread 2*

This is great! We've solved the problem! Or have we? Consider the scenario in Figure 3-5 where there are instead two locks and two totals.

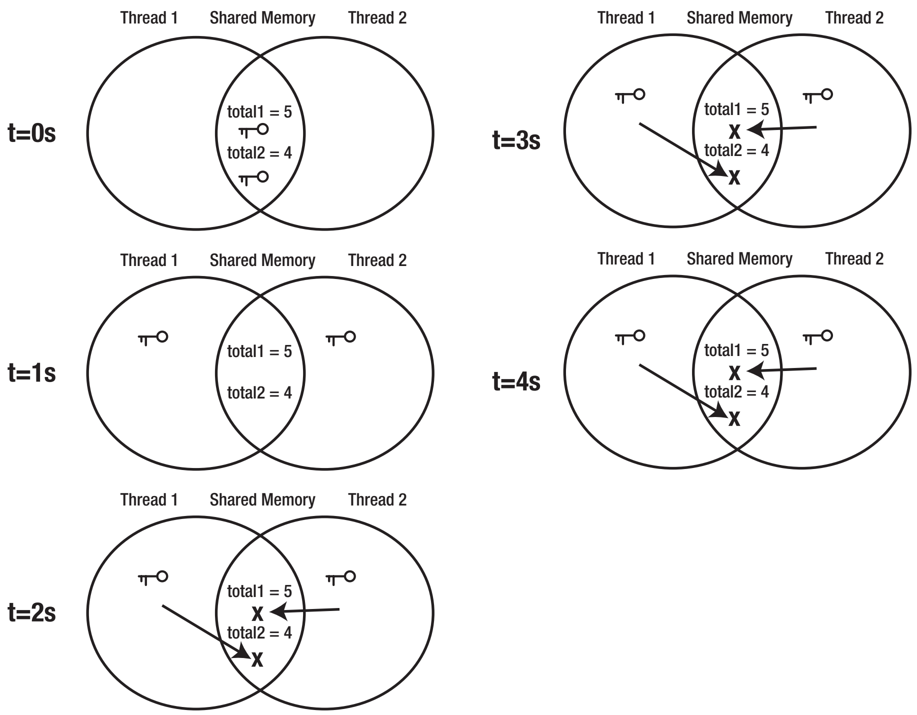


***Figure 3-5.** A demonstration of a deadlock between Thread 1 and Thread 2*

Figure 3-5 is what's known as deadlock. This happens when two threads require multiple pieces of data to execute, but they request them in different orders. In order to avoid these kinds of issues altogether, the author of Python implemented a lock at the thread level which only allowed one thread to run at any given time. This was a simple and elegant way to solve this problem. At the time, since multi-core CPUs were quite uncommon, it didn't really impact performance since in the CPU, these threads' instructions would be run serially anyway. However, as computers have become more advanced and computations have become more intensive, multi-core CPUs have become the standard in pretty much all modern computing platforms. When it comes to big data which involves running large CPU-intensive computations, not taking advantage of multi-core CPUs means in some cases not being able to run the computation at all (at least within a reasonable amount of time). So how can we break out of the GIL and truly run a multi-core big data computation?

C extensions, being written in C and not in Python, are not subject to the GIL. pandas is built on NumPy which is a Python wrapper around C extensions that do all the heavy lifting for the computation. This means that all the intensive computation when using pandas is done in C, where the computations are just generally faster due to the properties of the language. This also means that pandas is able to break out of the GIL and truly run multi-core computation on multiple cores simultaneously.

Since NumPy runs the computations in C, it must translate all the Python objects into C-compatible types. According to the NumPy documentation, as long as an array's types are translatable to C types, the GIL is released prior to the calculation. This means that when using NumPy, it's important to operate on types that are translatable to C types. If you operate on Python objects that NumPy is unable to translate to C, the operation cannot compute the result in C. It must instead stay in Python where the GIL cannot be released until the computation has finished.

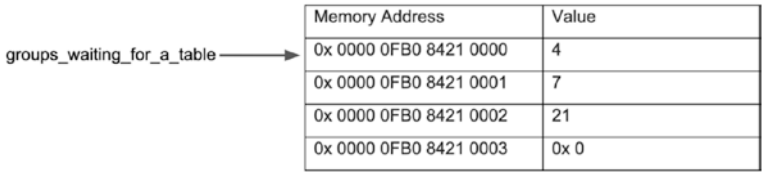
The Appendix shows a detailed mapping of the types in Python to the types NumPy uses in its C extensions known as scalars. All of the NumPy types are referred to as dtypes within the NumPy API and are available as attributes of the NumPy library.

NumPy's main data structure is an N-dimensional array or ndarray. It is a special array structure that handles this Python-C boundary. It's important to note that ndarrays in NumPy are homogeneous, meaning all the elements have the same type. This again has to do with Python vs. C. C is a lower-level language where the developer must manage the memory allocation and deallocation themselves, and it does not permit dynamic typing. In the case of an array in C, all the elements must be of consistent and known size or type in order to allocate the appropriate amount of memory for the input data arrays and the output data array in C. Listing 3-11 shows an example of how an array of floats is created in C. Note the memory must be explicitly allocated using malloc, and it is of fixed size 100—having only enough room for 100 floats.

Listing 3-11. Allocating memory for an array of 100 floats in C

```
float* array = (float*) malloc(100 * sizeof(float));
```

NumPy uses ndarray as shown in Listing 3-12 to build a fixed-size array where dtype is used to specify the type of each element in the array or how much memory each element in the array will take up. Recall from the previous section that Python's list implementation is a dynamic array underneath. Python's list type, in contrast to an ndarray, handles elements of any size by allocating space for the pointer to that data as opposed to the data itself. This leads to a Python list taking up more memory than an ndarray because there is a layer of indirection due to the list holding the pointers to the data rather than the data itself. In summary, ndarrays have a fixed length and each element has the same type, whereas Python's list type has a dynamic length and elements can be many different types. This is an important distinction and why each column in pandas is assigned a particular dtype. This is also why it's important to make sure pandas has the correct type for a particular column and why it's important to normalize your data so that a type other than the all-encompassing object type is assigned. Figure 3-6 shows how the ndarray created in Listing 3-12 is represented in memory.



Memory Address	Value
0x 0000 0FB0 8421 0000	4
0x 0000 0FB0 8421 0001	7
0x 0000 0FB0 8421 0002	21
0x 0000 0FB0 8421 0003	0x 0

Figure 3-6. *A representation of Listing 3-12 in memory. Compare this to Figure 3-2*

Listing 3-12. ndarray of three unsigned 8-bit ints

```
import numpy as np
groups_waiting_for_a_table = np.ndarray(
     (3,0),
     buffer=np.array([4, 7, 21], dtype=np.uint8),
     dtype=np.uint8,
)
```

The CPython interpreter provides a C-API that exposes the ability to acquire and release the GIL. NumPy uses the macros NPY_BEGIN_THREADS and NPY_END_THREADS to denote when C code is able to run without the GIL. NumPy mathematical operations are instances of universal functions, known as ufunc's, implemented in C, and all of them call these macros. See the Appendix for a list of common ufuncs.

Recall that running without the GIL means that the program can now execute instructions on multiple cores simultaneously. This means that intensive mathematical operations when using NumPy are able to break up the example of counting how many times Tiffany appears in a list, using multi-threading at the C level. While NumPy itself doesn't typically implement multi-threaded computations (simply running the computation in C is enough of a performance improvement itself), there are other libraries which will do so when used in combination with NumPy.

When NumPy is compiled to use the Basic Linear Algebra Subroutines, known as BLAS, or a Linear Algebra Package, known as LAPACK, it runs operations according to the size of the memory cache and the number of cores available on the system. By optimizing the calculation according to the resources on the machine, NumPy is able to run the computations much faster than it otherwise would. There are several different implementations of BLAS/LAPACK including OpenBLAS, ATLAS, and Intel MKL. We'll explore how these libraries work to improve performance in more detail in later chapters, but for now, just know they exist and for large computations they can make a huge difference in performance.

An introduction to pandas performance

pandas is a wrapper around NumPy and NumPy is a wrapper around C; thus, pandas gets its performance from running things in C and not in Python. This concept is fundamental to everything you do in pandas. When you are in C, you are fast, and when you are in Python, you are slow.[1]

The same requirements present for working with NumPy arrays hold true when working with pandas DataFrames—namely, the Python code must be translatable to C code; this includes the types that hold the data and the operations performed on the data. Table 3-1 is a table of pandas types to NumPy types. Note that datetimes and timedeltas don't translate into NumPy types. This is because C does not have a datetime data structure, and so in cases where operations must be made on datetime data, it is more performant to, instead, convert the datetimes to an integer type of seconds since the epoch.

Table 3-1. *pandas to NumPy types*

pandas type	NumPy type
object	string_, unicode_
int64	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64
float64	float_, float16, float32, float64
bool	bool_
datetime64	datetime64[ns]
timedelta[ns]	NA
category	NA

[1] www.youtube.com/watch?v=ObUcgEO4N8w

Note that category is also not translatable into C. category is similar to a tuple in that it is intended to hold a collection of categorical variables, meaning metadata with a fixed unique set of values. Because it's not translatable into C, it should never be used to hold data that needs to be analyzed. Its advantage mainly comes in its ability to sort things in a custom sort order efficiently and simply. Underneath it looks like a data array of indexes where the indexes correspond to a unique value in an array of categories. The documentation claims that it can result in a huge memory savings when using string categories. Of course, we know from the previous section that Python already has a built-in string cache that does that for us automatically for certain strings so this would really only make a difference if the strings contained characters other than alphanumeric and underscore. Listing 3-13 shows an example of a category and its representation in memory. Note that it uses integers to represent the value and those integers map to an index in the category array. This is a common method of conserving memory in pandas. We'll run into this again later when we look at multi-indexing.

Listing 3-13. pandas category example and its representation in memory

```
import pandas as pd
produce = pd.Series(
     ["apple", "banana", "carrot", "apple"], dtype="category"
)
```

```
Data        Categories
0           apple
1           banana
2           carrot
0
```

Operations must also be translatable into C in order to take advantage of NumPy's performance optimizations. This means custom functions like the one in Listing 3-14 will not be performant because they will run in Python and not in C. We'll dig more into this example and the apply function specifically in Chapter 6.

Listing 3-14. pandas custom Python operation that isn't translatable to C

```
import pandas as pd

def grade(values):
    if 70 <= values["score"] < 80:
        values["score"] = "C"
    elif 80 <= values["score"] < 90:
        values["score"] = "B"
    elif 90 <= values["score"]:
        values["score"] = "A"
    else:
        values["score"] = "F"
    return values

scores = pd.DataFrame(
     {"score": [89, 70, 71, 65, 30, 93, 100, 75]}
)
scores.apply(grade, axis=1)
```

Since pandas is built on NumPy, it uses NumPy arrays as the building blocks for a pandas DataFrame, which ultimately translate into ndarrays deep down during computations.

Listing 3-15. pandas single-index DataFrame and its representation in memory

```
import pandas as pd
restaurant_inspections = pd.DataFrame({
  "restaurant": ["Diner","Diner","Pandas","Pandas"],
  "location": [(4,2),(4,2),(5,4),(5,4)],
  "date": ["02/18","05/18","04/18","01/18"],
  "score": [90,100,55,60]})
>> restaurant_inspections
     restaurant  location   date      score
     Diner       (4, 2)     02/18     90
     Diner       (4, 2)     05/18     100
     Pandas      (5, 4)     04/18     55
     Pandas      (5, 4)     01/18     76

Index           Blocks
restaurant      Diner      Diner      Pandas     Pandas
location        (4, 2)     (4, 2)     (5, 4)     (5, 4)
date            02/18      05/18      04/18      01/18
score           90         100        55         76
```

Listing 3-15 is an example of the simplest form of a pandas DataFrame. The data is restaurant health inspection data. It has four columns: restaurant, location, date, and score. Each column has four rows worth of data. Note that some of the data is repeated as there can be multiple inspections of the same restaurant over time. Underneath, this DataFrame is represented as a NumPy array called Index that contains the column names and a two-dimensional NumPy array called Blocks that contains the data.

This same data could be represented in a more expressive way using a multi-index DataFrame, where each index is a unique restaurant. This is done in two parts. First, we create the index, then the index is attached to the data as shown in Listing 3-16. The data is represented the same as in the previous example, but note there are only two data columns instead of four.

Listing 3-16. pandas multi-index DataFrame and its representation in memory

```
import pandas as pd
restaurants = pd.MultiIndex.from_tuples(
   (
         ("Diner", (4,2)),
        ("Diner", (4,2)),
        ("Pandas", (5,4)),
        ("Pandas", (5,4)),
    ),
    names = ["restaurant", "location"]
)
restaurant_inspections = pd.DataFrame(
  {
      "date": ["02/18", "05/18", "04/18", "01/18"],
      "score": [90, 100, 55, 76],
    },
    index=restaurants,
)
>> restaurant_inspections
                          date    score
restaurant      location
Diner           (4, 2)    02/18   90
                          05/18   100
Pandas          (5, 4)    04/18   55
                          01/18   76

Levels      Names                 Labels
restaurant  Diner   Pandas        0    0
location    (4, 2)  (5, 4)        0    0
                                  1    1
                                  1    1
Index       Blocks
date        02/18     05/18    04/18    01/18
score       90        100      55       76
```

Something special happens when we create a multi-index. Underneath the index doesn't look the same as it did in the single-index example.

There is still a NumPy array called Levels that holds the index names; however, instead of a simple two-dimensional NumPy array of data, the data undergoes a form of compression. The Names is a two-dimensional NumPy array that keeps track of the unique values within the index, and Labels is a two-dimensional NumPy array of integers whose values are the indexes of the unique index values in the Names NumPy array. This is the same memory saving technique used by the pandas category data type, and in fact, since category came later, they probably copied this technique from the pandas multi-index.

The DataFrame in Listing 3-16 ends up being about two-thirds the size of the single-index DataFrame in Listing 3-15 due to the data compression incurred by the use of the multi-index. pandas is able to save memory by using an integer type instead of another larger type to keep track of and represent index data. This of course is advantageous when there is a lot of repeated data in the index and less advantageous when there is little to no repeated data in the index. This is also why it is important to normalize the data. If, for example, there were multiple representations for the same restaurant name (DINER, Diner, diner), we would not be able to take advantage of the compression as we have done here. We would also not be able to take as large of an advantage of the Python string cache either.

Similar to multi-level indexes, pandas also permits multi-level columns. The multi-level columns are implemented the same as the multi-level indexes with the same data compression technique. Listing 3-17 shows an example of how to create a multi-index multi-level column DataFrame.

Listing 3-17. pandas multi-index multi-level column DataFrame

```
import pandas as pd
restaurants = pd.MultiIndex.from_tuples(
  (
        ("Diner", (4,2)),
        ("Pandas", (5,4)),
  ),
  names = ["restaurant", "location"]
)
inspections = pd.MultiIndex.from_tuples(
  (
        (0, "score"),
        (0, "date"),
        (1, "score"),
        (1, "date"),
  ),
  names=["inspection", None],
)
restaurant_inspections = pd.DataFrame(
  [[90, "02/18", 100, "05/18"], [55, "04/18", 76, "01/18"]],
  index=restaurants,
  columns=inspections,
)
>> restaurant_inspections
     inspection             0            1
                            score date   score date
     restaurant location
     Diner      (4, 2)      90    02/18  100   05/18
     Pandas     (5, 4)      55    04/18  76    01/18
```

Choosing the right DataFrame

Choosing the orientation of a pandas DataFrame is a decision that takes a lot of consideration and planning. Considerations include

- What kind of data processing will you be doing with the data?
- Do you need to run aggregated calculations over the data or group it?

- Are all the data types translatable to C types and what can you do to make them so?
- Can you separate the data from the metadata?
- Is there a particular DataFrame orientation(s) that would make the processing simpler and more performant?

Consider the following example of health inspection data. Each restaurant can have multiple inspections, and as part of the data processing, we would like to count how many inspections each restaurant has had.

The simplest form of a pandas DataFrame looks like the DataFrame in Listing 3-18. In order to calculate the number of inspections, the data must be aggregated uniquely by restaurant and then the number of inspections for each restaurant must be counted. This DataFrame takes up approximately 1,120 bits underneath.

Listing 3-18. Storing and operating on restaurant health inspection data in a single-index DataFrame

```
import pandas as pd
restaurant_inspections = pd.DataFrame({
  "restaurant": ["Diner","Diner","Pandas","Pandas"],
  "location": [(4,2),(4,2),(5,4),(5,4)],
  "date": ["02/18","05/18","02/18","05/18"],
  "score": [90,100,55,60]})
>> restaurant_inspections
     restaurant  location     date      score
     Diner       (4, 2)       02/18     90
     Diner       (4, 2)       05/18     100
     Pandas      (5, 4)       02/18     55
     Pandas      (5, 4)       05/18     76
```

```
>> total_inspections = restaurant_inspections.groupby(
     ["restaurant", "location"], as_index=False,
)["score"].count()
>> total_inspections.rename(
     columns={"score": "total"}, inplace=True
)
>> total_inspections
     restaurant  location    total
     Diner       (4, 2)      2
     Diner       (4, 2)      2
>> restaurant_inspections = pd.merge(
     restaurant_inspections,
     total_inspections,
     how="outer",
)
>> restaurant_inspections
     restaurant  location    date     score    total
     Diner       (4, 2)      02/18    90       2
     Diner       (4, 2)      05/18    100      2
     Pandas      (5, 4)      02/18    55       2
     Pandas      (5, 4)      05/18    76       2
```

Using a single-index DataFrame is less than ideal for this type of calculation for several reasons. First, we need to run an aggregated calculation so we need to group the data by unique restaurant. This grouping can be quite time-consuming if there are many groups. After running the calculation on each group, you'll notice the resulting total_inspections is not the same dimensions as the original restaurant_inspections DataFrame. The dimension mismatch requires us to do some finagling to get the new data back into the original DataFrame. We end up using a merge to do it which builds an entirely new DataFrame. This means we will be doubling our memory during the merge, and if the original DataFrame is very large, this could cause a slowdown or even a memory crash if we are very close to our max memory usage.

If instead we represent the data as a multi-index DataFrame as shown in Listing 3-19, the data is already grouped uniquely by restaurant. This means the groupby will be faster since the data is already grouped in the index. It also means the DataFrame will take up less memory since, as you recall from the previous section, the data in the index is compressed. Most significantly, however, we don't have to do the kind of finagling that we had to do when using a single-index DataFrame. We are able to run the calculation and put it back into the original DataFrame without creating a copy which is a huge time and memory saver. The code you'll notice is also simpler and easier to follow. This DataFrame takes up approximately 880 bits underneath. Recall that when we create a multi-index, the index data is compressed, which is why this multi-index DataFrame is smaller than its single-index counterpart.

Listing 3-19. Storing and operating on restaurant health inspection data in a multi-index DataFrame

```
import pandas as pd
restaurants = pd.MultiIndex.from_tuples(
  (
      ("Diner", (4,2)),
      ("Diner", (4,2)),
      ("Pandas", (5,4)),
      ("Pandas", (5,4)),
  ),
  names = ["restaurant", "location"],
)
```

```
restaurant_inspections = pd.DataFrame(
  {
      "date": ["02/18", "05/18", "02/18", "05/18"],
      "score": [90, 100, 55, 76],
  },
  index=restaurants,
)
>> restaurant_inspections
                           date         score
  restaurant  location
  Diner       (4, 2)       02/18        90
                           05/18        100
  Pandas      (5, 4)       02/18        55
                           05/18        76
>> restaurant_inspections["total"] = \
     restaurant_inspections["score"].groupby(
         ["restaurant","location"],
     ).count()
>> restaurant_inspections.set_index(
     ["total"],
     append=True,
     inplace=True,
    )
                                             date         score
     restaurant  location     total
     Diner       (4, 2)       2              02/18        90
                                             05/18        100
     Pandas      (5, 4)       2              02/18        55
                                             05/18        76
```

What if we take this one step further? If we make the dates the column names, then all the scores will be on the same row and the calculation becomes trivial. Here the unique restaurants are indexes, and the unique inspection dates are columns. Note the score is now the only data. This makes each row a unique restaurant, and thus the count can simply be performed across each row. See Listing 3-20.

Listing 3-20. Storing and operating on restaurant health inspection data in a multi-index date column DataFrame

```
import pandas as pd
restaurants = pd.MultiIndex.from_tuples(
  (
      ("Diner", (4,2)),
      ("Pandas", (5,4)),
  ),
  names = ["restaurant", "location"],
)
restaurant_inspections = pd.DataFrame(
  {
      "02/18": [90, 55],
      "05/18": [100, 76],
  },
  index=restaurants,
)
>> restaurant_inspections
     date                    02/18    05/18
     restaurant  location
     Diner       (4, 2)      90       100
     Pandas      (5, 4)      55       76

>> restaurant_inspections["total"] = \
     restaurant_inspections.count(axis=1)
```

```
>> restaurant_inspections.set_index(
     ["total"],
     append=True,
     inplace=True,
    )
     date                                     02/18    05/18
     restaurant  location    total
     Diner       (4, 2)      2                90       100
     Pandas      (5, 4)      2                55       76
```

This DataFrame takes up approximately 660 bits underneath. Note this takes up less memory because we no longer are tracking the date and score column names and the date values are no longer being repeated. This is pretty much as compressed as we can get with this data, and it allows us to perform a very efficient aggregated calculation over each unique restaurant. Let's see if we can identify any holes in using this format on a larger data set.

Currently each row is a unique restaurant, but what if there were multiple restaurants with the same name at different locations? This would still mean that there is a unique restaurant per row so no issues there.

What if the restaurants were not all inspected on the same days? In a large city, it would be near impossible for an inspector to inspect all the restaurants on the same day. This means then that there would be holes in the data as shown in Listing 3-21.

Listing 3-21. A representation of Listing 3-20 if not all restaurants were inspected on the same day

```
date                          02/18    05/18    06/18    07/18
restaurant  location  total
Diner       (4, 2)    2       90       100      NaN      NaN
Pandas      (5, 4)    2       NaN      NaN      55       76
```

These holes are potentially a big problem. Recall that the score data was represented as an unsigned 8-bit integer, now because there are NaNs in the data, the type must accommodate the NaN type size which forces the type to be a 32-bit float. That's four times more memory for each score. Not only that, but now we have a bunch of gaps in our data that wasted space and ultimately wasted memory. The fewer dates in common there are between the restaurants, the worse this problem becomes. Multi-level column index to the rescue! See Listing 3-22.

Listing 3-22. Storing and operating on restaurant health inspection data in a multi-index multi-level column DataFrame

```
import pandas as pd
restaurants = pd.MultiIndex.from_tuples(
  (
      ("Diner", (4,2)),
      ("Pandas", (5,4)),
  ),
  names = ["restaurant", "location"]
)
inspections = pd.MultiIndex.from_tuples(
  (
      (0, "score"),
      (0, "date"),
      (1, "score"),
      (1, "date"),
  ),
  names=["inspection", "data"],
)
```

```
restaurant_inspections = pd.DataFrame(
   [[90, "02/18", 100 "05/18",], [55, "04/18", 76 "01/18",]],
   index=restaurants,
   columns=inspections,
)
>> restaurant_inspections
     inspection                 0                1
                                score date       score date
     restaurant  location
     Diner       (4, 2)         90   02/18       100   05/18
     Pandas      (5, 4)         55   04/18       76    01/18

>> total = \
     restaurant_inspections.iloc[
         :,
         restaurant_inspections.columns.get_level_values("data") \
             == "score"
     ].count()
>> new_index = pd.DataFrame(
     total.values,
     columns=["total"],
     index=restaurant_inspections.index,
)
>> new_index.set_index("total", append=True, inplace=True)
>> restaurant_inspections.index = new_index.index
>> restaurant_inspections
inspection                      0                1
                                score date       score date
restaurant  location  total
Diner       (4, 2)    2         90   02/18       100   05/18
Pandas      (5, 4)    2         55   04/18       76    01/18
```

This is probably the most optimal we can get with this DataFrame format for this particular use case. We have compressed our data as much as possible taking advantage of both multi-level indexes and multi-level columns and organized the DataFrame in such a way as to achieve the fastest calculation possible. Note the main disadvantage of this particular format is it requires a bit of finagling to get the total back onto the index, and for that reason, this solution is less readable. If this was the solution you were going to go with, you might consider making two custom functions: one that puts data onto the index and another that puts data onto the columns. These functions would improve code readability by hiding the finer details of appending level data to the DataFrame.

Once you have decided on a DataFrame format that makes sense, you will likely need to load your raw data into pandas, normalize it, and convert it to that particular DataFrame format. In Chapter 4, we'll dive into some common pandas data loading methods and discuss the normalization options they provide in more detail.

CHAPTER 4

Loading and Normalizing Data

Raw data comes in many forms: CSV, JSON, SQL, HTML, and so on. pandas provides data input and output functions for loading data into a pandas DataFrame and outputting data from a pandas DataFrame into various common formats. In this chapter, we'll deep dive into some of these input functions and explore the various loading and normalization options they provide.

The functions that load data into pandas provide a wide range of normalization and optimization capabilities that can improve the performance of a program, even to the point where it means the difference between being able to load the data into pandas and running out of memory. Each input function is different however, so it really depends on the input/output format that you are working with and it's always worthwhile to check the documentation of the particular functions you are using. Table 4-1 lists the various input and output functions that pandas supports.

H. Stepanek, *Thinking in Pandas*, https://doi.org/10.1007/978-1-4842-5839-2_4

Table 4-1. *IO pandas data functions*

Input	Output
read_csv	to_csv
read_excel	to_excel
read_hdf	to_hdf
read_sql	to_sql
read_json	to_JSON
read_html	to_html
read_stata	to_stata
read_clipboard	to_clipboard
read_pickle	to_pickle

Chapter 3 mentioned several very good reasons for normalizing data; it can save memory and optimize data analysis. Normalizing data can gain you the benefit of utilizing Python's string cache or run computations in C rather than Python by choosing a C-compatible data type. Many of the preceding input functions provide options for various ways of normalizing data as part of the load process. Instead of loading the data with a large memory foot print and then removing unnecessary columns or casting columns to a smaller data type to reduce the memory foot print after loading, many of the input functions allow you to remove and specify the types of columns during load. This means you can load more data without running out of memory, and the data load and normalization process is faster since you are doing two things at once rather than consecutively loading and then normalizing.

Operations that result in the creation and elimination of data can be expensive because they require large chunks of memory to be allocated and deallocated. Converting (more commonly referred to as casting) data from one type to another is also expensive as it requires large chunks of memory to be allocated and deallocated. Working with large chunks of memory often means cache misses and a lot of time is spent in IO moving things from memory that is farther away from the CPU to memory that is closer to the CPU (such as from main memory to the first level cache, for example). Thus, while you may think that memory has nothing to do with processing speed, it can actually have a huge impact on the runtime.

In these input functions, pandas typically infers the data type upon loading the data. While this can be quite nice at first and seem like a fantastic feature that you should surely always take advantage of, it also has a large and often negative impact on performance. Often the data being loaded has not yet been normalized, and numeric columns may contain non-numeric values, for example, that force the inferred data type to be an object, the largest data type it can be. Many of the data load functions allow you to specify the type of the columns and convert place holder values to NaNs which can prevent pandas from inferring the wrong data type.

pd.read_csv

The pandas CSV loader pd.read_csv is the most widely used of the loaders and by far the most complete in terms of data normalization options. Because the Python standard library has a built-in CSV loader and the pandas loader has some fairly fancy Pythonic options, it has two different parsing engines: the C engine and the Python engine. As you can probably guess by now, the C engine is more performant than the Python engine, but depending on what options you specify, you may have no choice but to use the Python engine for parsing. Thus, it’s advisable to be careful which options you use and the values you provide to those options so that you guarantee you are using the C parsing engine and get the best load performance possible. The CSV loader has an explicit engine parameter that lets you force the parsing engine to be Python or C. Explicitly always specifying this parameter when loading is an easy way to guarantee the CSV loader uses the C engine for parsing. If you specify another Python parser–specific option while engine is explicitly set to 'c', the CSV loader will throw an exception informing you that particular setting is not compatible with the C parsing engine as shown in Listing 4-1.

Listing 4-1. read_csv will raise a ValueError when engine is set to 'c' and other settings are not compatible

```
>> data = io.StringIO(
    """
    id,age,height,weight
    129237,32,5.4,126
    123083,20,6.1,145
    """
)
>> df = pd.read_csv(data, sep=None, engine='c')
     ValueError: the 'c' engine does not support sep=None
     with delim_whitespace=False
```

Another reason to use the C parsing engine is that it supports a higher precision of floating points via the float_precision parameter. Generally, the Python engine uses double floating point precision, and the C engine uses its own low-level string to decimal parser that is comparable to the Python engine. Both can result in floating point rounding errors such as cases where –15.361 and –15.3610 are not equal. However, the C parsing engine supports additional options high precision and round-trip precision. If you want your floats to be as accurate as possible, use the round-trip precision option. This is a common problem with floating points, and so, an alternative approach, often used when handling financial data, is to split a float into two integers: one integer represents the number above the decimal and the other represents the number below the decimal.

The first parameter to the read_csv function is filepath_or_buffer. Typically, the path to the CSV file is passed here, but note for unit testing purposes and for example purposes in the rest of this chapter, a StringIO object can be passed in its place. It also accepts a URL path if the CSV file is hosted by a third-party application for example. The documentation officially reads[1]

By file-like object, we refer to objects with a read() method, such as a file handler (e.g. via builtin open function) or StringIO.

This file-like object is another Python-ism and is commonly known as duck typing. This term is born of the idiom "If it walks like a duck and quacks like a duck, it's a duck." In this case, if it has a read method, it is a file-like object. This is why StringIO can also be substituted for a file handler since it also has a read method. StringIO is a nice substitute in unit tests since it allows you to pretend it's a file without actually having to include a test.csv for validating your loader works as expected.

read_csv provides a sep parameter which specifies the character(s) used to delineate the data. The default is a comma. Note sep treats any values longer than one character with the exception of \s+ as regular expressions. The use of complex delimiters here can force the use of the Python parsing engine instead of C, and for that reason, it is advisable to use single-character delimiters when possible and not specify complex regular expressions. The parameter delim_whitespace may also be set to True as an alternative to setting sep= "\s+", specifically to denote whitespace file delineation. The sep parameter can also be set to None, in which case the Python parsing engine will be used and it will automatically detect the delimiter. The skipinitialspace parameter can be used to ignore spaces surrounding the delimiter. By default, this is disabled, so if there are spaces between the delimiters in your file, you will need to set this to True. Listing 4-2 demonstrates how you might use sep in combination with skipinitialspace to configure loading of data that is not comma delimited.

[1] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_hdf.html

Listing 4-2. Loading non-comma-delimited data

```
>> data = io.StringIO(
     """
     id| age| height| weight
     129237| 32| 5.4| 126
     123083| 20| 6.1| 145
     """
)
>> pd.read_csv(data, sep="|", skipinitialspace=True)
     idageheightweight
     0129237325.4126
     1123083206.1145
```

The parameter usecols narrows down the list of columns to load. It's possible to have columns within the CSV file that you don't care about, and thus this can be an efficient way of eliminating them upon load as opposed to loading all the data and removing them after. Note that usecols can also be a function where the column name is an input and the output is a Boolean indicating whether to include that column or discard it upon loading. A function however is less ideal as it requires calls between the C parsing engine and the custom function which will slow the loader down. Listing 4-3 shows an example of using use_cols to eliminate columns id and age during load.

Listing 4-3. Eliminating columns during load

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,145
     """
)
```

```
>> pd.read_csv(data, usecols=["height", "age"])
     heightweight
     05.4126
     16.1145
```

The skiprows parameter allows you to skip certain rows in the file. In its simplest form, it can be used to skip the first n number of rows in a file; however, it can also be used to skip particular rows by specifying a list of indexes to skip. It can also be a function that accepts a row index and returns True if that row should be skipped. Note if a function is passed here, it will have the unfortunate consequence of jumping between the C parsing engine and the skiprows Python function which may lead to a substantial slowdown when parsing large data sets. For this reason, it's recommended to keep the skiprows a simple integer or list value.

The skipfooter parameter lets you specify the number of lines at the end of the file to skip. The documentation notes that this is unsupported with the C parsing engine. Since the Python engine uses the Python CSV parser, the CSV parser runs and then the last lines of the file are dropped. This makes sense if you think about this problem a little more deeply: How would the parser know which lines to skip without knowing how many lines there are in the file first (which would require first parsing the file)? This behavior can be somewhat surprising for some users when, for example, they are actively trying to avoid lines in the file because they break the parser and find that the parser is still trying to parse those lines they configured the parser to skip. If you run into this situation in your own program, nrows is a nice alternative. Listing 4-4 demonstrates an example of running into a parsing error even though the loader was configured to skip that line.

Listing 4-4. Encountering a parsing error when using skipfooter

```
import pandas as pd
data = io.StringIO(
  """
  student_id, grade
  1045,"a"
  2391,"b"
  8723,"c"
  1092,"a"
  """
)
try:
  grades = pd.read_csv(
    data,
    skipfooter=1,
  )
except pd.errors.ParserError as e:
    pass
```

The parameter comment lets you specify a character that denotes a comment, and the rest of the line is ignored after the character. This can be a good strategy for manually filtering out certain lines prior to parsing. If you comment out the line, then it will not be included in the data set. You may also consider setting error_bad_lines to False. Note by default pandas will still raise a warning on each bad line, so if you wish to disable the warning as well, you may set warn_bad_lines to False.

By default, the header or column names are inferred by read_csv, and the first row of the data is treated as the header. Using the parameter header, you can specify which row numbers are to be treated as columns if the data contains multi-level columns. Similarly, using index_col, you can specify which columns via column index are to be treated as part of the multi-index.

In Listing 4-5, a multi-index multi-level column DataFrame was dumped to a CSV file using the pandas to_csv function. The data contains information on family and species of nightshades. The indexes contain the id of the family and species, while the columns contain the actual names. Here, the first two rows contain multi-level column data so header is set to [0,1], and the first two columns are the multi-index so index_col is set to [0,1].

Listing 4-5. Example of loading a multi-index multi-level column DataFrame

```
>> data = io.StringIO(
     """
     family,,nightshade,nightshade,nightshade
     species,,tomatoe,deadly-nightshade,potato
     family_id,species_id,,,
     61248,129237,1,0,0
     61248,123083,0,1,0
     61248,123729,0,0,1
     """
)
>> df = pd.read_csv(data, header=[0,1], index_col=[0,1])
     family                 nightshade
     species                tomatoes     deadly-nightshade potato
     family_id  species_id
     61248      12937       1            0                  0
     61248      123083      0            1                  0
     61248      123729      0            0                  1
```

If the squeeze parameter is enabled, read_csv returns a Series instead of a DataFrame if there is only one column in the CSV file. This can be useful when you need to load data from multiple sources and combine it into a single DataFrame. If the data is loaded into a series, you can simply add it to an existing DataFrame as a new column as demonstrated in Listing 4-6.

Listing 4-6. Example of using squeeze

```
import pandas as pd
site_data = pd.read_csv('site1.csv')
site_data['site2'] = pd.read_csv('site2.csv', squeeze=True)
```

The dtype parameter allows you to specify a type for each column in the data. If this is not specified, read_csv will attempt to infer the data type which typically results in the inferred type being an object which is the largest size that a data type can be. Specifying the dtype during load can be a huge performance improvement, but that also means you have to have some knowledge at load time about the columns in the data set. If you don't know exactly what to expect until you look at the data, you might consider loading the header of the data first or the first couple rows using nrows, identifying the column types, and then loading the whole data file with the appropriate types specified.

Listing 4-7. Example of not specifying the types of the columns when loading

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,145
     """
)
>> df = pd.read_csv(data, index_col=[0])
                    age    height     weight
     id
     129237         32     5.398438   126
     123083         20     6.101562   145
>> df.memory_usage(deep=True)
```

```
     Index     16
     age       16
     height    16
     weight    16
>> df.dtypes
     age          int64
     height       float64
     weight       int64
>> df.index.dtype
     dtype('int64')
```

Listing 4-7 shows an example of loading the data without specifying the type of each column. Note that all the types take up 8 bytes and are defaulted to the largest int and float possible, whereas in Listing 4-8, they take up much less memory. This data only has two rows worth of data, but just in those two rows, we've decreased the memory footprint of the DataFrame by more than half just by specifying the types of the columns.

Listing 4-8. Example of specifying the types of the columns when loading

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,145
     """
)
>> df = pd.read_csv(
     data,
     dtype={
         'id': np.int32,
         'age': np.int8,
         'height': np.float16,
         'weight': np.int16},
     index_col=[0],
)
                    age    height    weight
     id
     129237         32     5.398438  126
     123083         20     6.101562  145
>> df.memory_usage(deep=True)
     Index     16
     age       2
     height    4
     weight    4
>> df.dtypes
     age       int8
     height    float16
     weight    int16
>> df.index.dtype
     dtype('int64')
```

The converters parameter allows you to specify a function to convert values in a particular column such as in Listing 4-9. This is a nice normalization feature if, for example, there are multiple values that represent the same value in a column and you wish to normalize it to a single value. This, however, comes at a cost. Because these functions are written in Python, the C engine must make calls between C and Python to convert each of the values which can be very time-consuming when working with large data sets. So, while the data is being normalized at load time, it is also going to load more slowly because it will be jumping between C and Python for each value to convert in each column. In this scenario, it would be more performant to convert the column values after using an Apply-Cython implementation so that the conversion happens quickly all in C and avoids this jumping back and forth. See Chapter 6 for how to implement an apply in Cython.

Listing 4-9. Standardizing values at load time with converters

```
import pandas as pd

MEDICATIONS_MAPPER = {"atg": "atg", "aftg": "atg", "bta": "bta"}
def medication_converter(value):
     return MEDICATIONS_MAPPER[value.lower()]

data = io.StringIO(
     """
     id,age,height,weight,med
     129237,32,5.4,126,bta
     123083,20,6.1,145,aftg
     """
)
>> treatments = pd.read_csv(
     data,
     converters={'med': medication_converter},
)
     id        age   height    weight    med
     129237    32    5.4       126       bta
     123083    20    6.1       145       atg
```

The nrows parameter allows you to specify the number of rows to read from the file. Something that may be unintuitive here is that nrows doesn't actually skip reading the rows when using the Python parsing engine. This is because the Python parsing engine reads the whole file first. This means that if there are lines after the number of rows you intended to read from the file that result in parsing errors, when running with the Python parsing engine, you will not be able to avoid them by using nrows. Since the Python parsing engine reads the whole file first, it will still throw a parsing error on those lines, even though you told the CSV loader not to read those rows. So, this is yet another reason to avoid the Python parsing engine, particularly when using this setting. Note that skipfooter, on the other hand, even in the C parsing engine does in fact read the footer row. This is simply because in order to identify it as the footer of the file, it has to read it and reach the end of the file to identify it as the footer. Listing 4-10 shows an example of how to avoid lines that would otherwise cause parsing errors using nrows and the C parsing engine.

Listing 4-10. Avoiding a parsing error by using nrows

```
import pandas as pd
data = io.StringIO(
     """
     student_id, grade
     1045,"a"
     2391,"b"
     8723,"c"
     1092,"a"
     """
)
grades = pd.read_csv(
     data,
     nrows=3,
)
```

The nrows parameter in combination with skiprows and header can also be useful for reading a file into memory in pieces, processing it, and then reading the next chunk. This is particularly useful with huge sets of data that you may otherwise be unable to read all at once due to memory constraints. Listing 4-9 shows an example of this. Note process is a function that is wrapping the read_csv function. It takes the loaded data from read_csv and does some processing on it to reduce the memory footprint and/or normalize it beyond the capabilities of read_csv and returns it to be concatenated with the rest of the data. In Listing 4-11, we load the first 1000 rows, process them, and use those first 1000 rows to initialize data. Then we continue reading in rows, processing 1000 at a time until we read in less than 1000 rows. Once we read in less than 1000 rows, we know we've read the entire file and exit the loop.

Listing 4-11. Reading a file and processing it nrows at a time to reduce memory overhead

```
import pandas as pd

ROWS_PER_CHUNK = 1000
data = process(pd.read_csv(
     'data.csv',
     nrows=ROWS_PER_CHUNK,
))
read_rows = len(data)
chunk = 1
while chunk * ROWS_PER_CHUNK == read_rows:
     chunk_data = process(pd.read_csv(
          'data.csv',
          skiprows=chunk * ROWS_PER_CHUNK,
          nrows=ROWS_PER_CHUNK,
          header=None,
          names=data.columns,
     ))
     read_rows += len(chunk_data)
     data = data.append(process(chunk_data), ignore_index=True)
```

The parameter iterator used in combination with chunksize also lets you read the data in chunks similar to Listing 4-11. Again, this may be necessary for performance reasons. Maybe the data you are reading cannot be read in with a smaller memory footprint using read_csv, and some normalization must take place after loading that results in a smaller memory footprint, meaning while you aren't able to read all the data into memory all at once using read_csv because the resulting DataFrame would be too large, you are able to read it in a chunk at a time and reduce the memory footprint on each chunk such that the resulting DataFrame will fit in memory. Note using iterator and chunksize is a better alternative if you are reading the whole file chunks at a time than using nrows and skiprows as it keeps the file open at the correct location instead of constantly re-opening it and scrolling to the next position. Listing 4-12 shows an example of this.

Listing 4-12. Reading a file in chunks to reduce memory overhead

```
import pandas as pd

ROWS_PER_CHUNK = 1000
data = pd.DataFrame({})
reader = pd.read_csv(
     'data.csv',
     chunksize=ROWS_PER_CHUNK,
     iterator=True
)
for data_chunk in reader:
     processed_data_chunk = process(data_chunk)
     data = data.append(processed_data_chunk)
```

The parameter low_memory which defaults to True actually processes the file in chunks already when using the C parsing engine in order to save memory. However, it is limited in the processing it can do of each chunk by the options of read_csv, and thus custom processing and iterating over the chunks manually may be better in certain scenarios.

pandas read_csv function also provides an option called memory_map. When set to True and if a filepath is provided, it will map the file directly into virtual memory and access the data directly from there. Using this option can improve performance because there is no longer any IO overhead waiting for the next chunk of the file to be loaded into memory. Generally, accessing memory mapped files is faster because the memory is local to the program and the memory mapped is already in the page cache so there is no need to load it on the fly. In practice, memory mapping the file generally doesn't provide much of a performance advantage in the typical use case of loading a file serially from beginning to end. If you are experiencing a lot of cache misses, meaning the file data that would normally be loaded into cache (memory closer to the CPU) is not present and must be loaded from main memory, this may hold a performance improvement. Cache misses may happen if other programs are running concurrently which add their memory into the cache and consequently knock your file data out of the cache. See Chapter 8 for a more detailed explanation of the memory hierarchy and cache misses. This might also hold a performance advantage if you are reading this file many times over the course of your program or your program runs periodically and you don't want to keep having to load the same file into memory each time it runs. So, while this feature sounds like it can provide you with a substantial speedup, the reality is unless you are working outside of the standard read a file from start to finish workflow, it's unlikely to do so.

The na_values parameter allows you to specify values to interpret as Not a Number, also known as NaNs. This type comes from NumPy which, if you recall from Chapter 2, is a dependency of pandas. It's commonly used in NumPy as a placeholder for a value resulting from a computation that is invalid such as divide by 0. Note by default pandas interprets any string Nan or nan as a NaN type automatically. This automatic conversion may be problematic if you are working with data where Nan or nan may actually be a valid name, for example. This is where keep_default_na comes in handy. Setting the parameter keep_default_na to False turns off pandas automatic interpretation of certain values to NaNs. For a complete list of values that pandas automatically converts to the NaN type, see the Appendix.

The parameter na_filter when set to False disables checking for NaNs altogether, and the documentation notes this can lead to a performance improvement when you know for certain there are no NaNs in the data. The parameter na_values, on the other hand, lets you specify additional values other than the default set that you would also like to be converted to NaNs.

The parameter verbose outputs the number of NaN values in each column that contains NaNs when the Python parsing engine is used and parsing performance metrics when the C parsing engine is used. The pandas documentation states it outputs NaN values explicitly for non-numeric columns. This can be somewhat deceiving however, as the non-numeric determination is made at the time the parsing engine runs and not based on the final type of the column in the resulting DataFrame. Any column with a NaN in it at parsing time is considered a non-numeric column, even if the type of that column ultimately ends up being a numeric type (such as a float64 in the following example). The Python parser must parse all the values in the column and convert them appropriately to NaNs before assigning the final type. This means the NaN values are counted during parsing before the final type of the column has been assigned. Listing 4-13 demonstrates this behavior.

Listing 4-13. Unexpectedly counting NaNs in numeric columns

```
>> import pandas as pd
>> data = io.StringIO(
     """
     student_id,grade
     1045,"a"
     2391,"b"
     ,"c"
     1092,"a"
     """
)
```

```
>> grades = pd.read_csv(
     data,
     verbose=True,
     index_col="student_id",
     engine='python',
)
     Filled 1 NA values in column student_id
>> grades
               grade
     student_id
     1045        a
     2391        b
     NaN         c
     1092        a
>> grades.index.dtype
     dtype('float64')
```

A limitation of how pandas read_csv handles placeholder types is that you cannot specify a converter to convert a NaN to a 0, for example, and also cast a column to a particular dtype. Listing 4-14 illustrates a case where you might wish to do this. The weight column in the data set does not always have a value, nor is it consistent in the way a non-value is entered. Sometimes it is left as empty; other times, it is "unknown". If we do not specify a type for this column and let pandas infer the type, pandas stores the column values as objects, meaning some of them are integers, some of them are NaNs, and some of them are strings. Note that an object in this example takes up 32 bytes per element with some additional overhead. This is much more than the desired type of an int16 which takes up 2 bytes per element. Not only does the resulting DataFrame take up much more memory, but it is also unusable in its state to run computations over. Since some of the values are objects, summing all the weights in the column, for example, might result in string addition rather than integer addition. Thus, leaving pandas to infer the data type in this scenario is less than ideal.

Listing 4-14. Example of how pandas handles NaNs in the data by default

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,
     123087,25,4.5,unknown
     """
)
>> df = pd.read_csv(
     data,
     dtype={
         'id': np.int32,
         'age': np.int8,
         'height': np.float16},
     index_col=[0],
)
                 age    height      weight
     id
     129237      32     5.398438    126
     123083      20     6.101562    NaN
     123083      20     6.101562    unknown
>> df.memory_usage(deep=True)
     Index      24
     age        3
     height     6
     weight     155
>> df.dtypes
     age          int8
     height       float16
     weight       object
>> df.index.dtype
     dtype('int64')
```

Instead of letting pandas infer the data type, let's convert all the placeholder values to NaNs using na_values. Although ideally we would like them to be int16s, float16s take up the same amount of memory, and pandas supports NaNs being stored as floats whereas it does not support them being stored as integers during loading, so we set the dtype of the weight column to be float16. Note if we do not specify the dtype, it will be a float64. If we really need them to be integers, we can replace the NaNs with zeros using fillna and convert them using astype after loading as shown in Listing 4-15.

Listing 4-15. Example of using na_values and dtype to convert placeholder values to float16 NaNs during load

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,
     123087,25,4.5,unknown
     """
)
>> df = pd.read_csv(
     data,
     dtype={
         'id': np.int32,
         'age': np.int8,
         'height': np.float16,
         'weight': np.float16},
     na_values={"unknown"},
     index_col=[0],
)
                    age     height      weight
     id
     129237         32      5.398438    126
```

```
    123083        20      6.101562   NaN
    123083        20      6.101562   NaN
>> df.memory_usage(deep=True)
    Index      16
    age        3
    height     6
    weight     6
>> df.dtypes
    age       int8
    height    float16
    weight    float16
>> df.index.dtype
    dtype('int64')
>> df["weight"].fillna(0, inplace=True)
>> df["weight"] = df["weight"].astype(np.int16)
>> df
                age    height    weight
    id
    129237      32     5.398438  126
    123083      20     6.101562  0
    123083      20     6.101562  0
>> df.memory_usage(deep=True)
    Index      16
    age        3
    height     6
    weight     6
>> df.dtypes
    age       int8
    height    float16
    weight    int16
```

The parsing performance metrics output in verbose mode when using the C parsing engine can be useful in determining where the parsing engine is spending its time. Listing 4-16 shows an example output. Tokenization is the parser breaking up the data into individual values, Type conversion is converting each column to a particular type whether that is inferred by pandas or explicitly specified, and Parser memory cleanup is the time it took to free all the no longer needed memory after the data was read. Depending on these values, they may point to areas for improvement, for example, if Tokenization is taking a long time, you may be able to speed up performance by specifying additional options like how to interpret quotes, spaces, bad lines, and so on. If a lot of time is spent in type conversion, it may indicate you have some custom converters that are slowing down the process or you may need to specify the dtypes rather than letting pandas infer them. If you find a lot of time is spent in memory cleanup, you may need to not parse the whole file at once or you may be doing too many conversions of the data that lead to a lot of memory duplication and thus a lot of memory cleanup.

Listing 4-16. Example output when running in verbose mode with C parsing engine

```
>> grades = pd.read_csv(verbose=True, engine='c')
     Tokenization took: 0.01 ms
     Type conversion took: 0.45 ms
     Parser memory cleanup took: 0.01 ms
```

The parse_dates parameter if set to True will attempt to automatically detect and convert columns with date formatted strings to datetime objects. Rather than just setting it to True however, it's far better to explicitly list which columns should be converted to datetime objects. This parameter lets you explicitly specify which columns to convert in the form of a list and even combine multiple columns into a single datetime object when it is specified as a list of lists of columns. Listing 4-17 shows an example of explicitly specifying which columns to convert. Note each datetime object takes up 8 bytes, but this is still far less memory than if we didn't convert it at all.

Listing 4-17. Explicitly converting certain columns to datetime objects

```
>> data = io.StringIO(
     """
     id,birth,height,weight
     129237,04/10/1999,5.4,126
     123083,07/03/2000,6.1,150
     123087,11/23/1989,4.5,111
     """
)
>> df = pd.read_csv(
     data,
     dtype={
         'id': np.int32,
         'height': np.float16,
         'weight': np.int16},
     parse_dates = ["birth"],
     index_col=[0],
)
                    birth        height      weight
     id
     129237         1999-04-10   5.398438    26
     123083         2000-07-03   6.101562    150
     123083         1989-11-23   6.101562    111
>> df.memory_usage(deep=True)
     Index      24
     birth      24
     height     6
     weight     6
```

```
>> df.dtypes
    age       int8
    height    float16
    weight    datetime64[ns]
>> df.index.dtype
    dtype('int64')
```

Note Listing 4-17 assumes that there are no NaNs or placeholder values in the column. If there are, like in Listing 4-18, na_values must be specified to convert all the placeholder values to NaNs; otherwise, the column will be an object rather than a datetime because the placeholder values will be left as strings.

Listing 4-18. Explicitly converting certain columns to datetime objects and handling NaNs

```
>> data = io.StringIO(
    """
    id,birth,height,weight
    129237,04/10/1999,5.4,126
    123083,unknown,6.1,150
    123087,11/23/1989,4.5,111
    """
)
>> df = pd.read_csv(
    data,
    dtype={
        'id': np.int32,
        'height': np.float16,
        'weight': np.int16},
    parse_dates=["birth"],
    na_values=["unknown"],
    index_col=[0],
)
```

```
                 birth        height      weight
    id
    129237       1999-04-10   5.398438    26
    123083       NaT          6.101562    150
    123083       1989-11-23   6.101562    111
>> df.memory_usage(deep=True)
    Index      24
    birth      24
    height     6
    weight     6
>> df.dtypes
    age            int8
    height         float16
    weight         datetime64[ns]
>> df.index.dtype
    dtype('int64')
```

Generally, parse_dates while it's convenient is counterproductive from a performance perspective. It takes time to convert the dates, and once they are converted, the data type is not translatable to C. For this reason, it's recommended to convert datetimes to time since the epoch or some simple numeric C-translatable value if possible. If you need to work with particular days, months, and years, it might even make sense to store those in separate columns.

There are many other date-specific parameters included in read_csv. The parameter infer_datetime_format is enabled by default, so whenever possible, pandas attempts to infer the format of any datetime values automatically. The documentation says that in some cases this can increase the parsing speed by five to ten times when it's able to detect and use a particular date parsing format.[2] When set to True, the parameter

[2]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

keep_date_col keeps both the combined column and the original separate date columns if parse_dates specified that multiple date columns should be combined together. The date_parser parameter lets you specify a date parsing function. The documentation notes this function may be called in several different ways ranging from calling it once on each row or passing in all rows and columns at once. Since this function is jumping between C and Python, it's advantageous to call it the least amount of times as possible. This means it's best to implement this function to operate on all datetime rows and columns and output an array of datetime instances. This function could be an existing parser (the default is dateutil.parser.parser), or it could be a custom function. This might be useful if you need to do some special timezone handling or the data is stored in a special datetime format. Not all countries specify the day before the month so pandas provides a dayfirst parameter so you can specify whether the day comes first in the dates you are parsing. The parameter cache_dates which is enabled by default keeps a cache lookup of the converted dates, so that if the same date appears multiple times in the data set, it does not have to run the conversion again and can just use the cached value.

The parameter escapechar lets you escape certain characters. For example, in most programing languages, a commonly used escape character is a backslash (\) so it may be desirable to escape certain quote characters inside of a quote with a \" or element delimiter characters with \,. Listing 4-19 illustrates this use case. If the temperature recordings were recorded by a country that uses commas as a decimal point delimiter and also uses commas as a CSV element delimiter, read_csv will not be able to parse this file with its default configuration and will raise a parsing error, "pandas.errors.ParserError: Error tokenizing data. C error: Expected 2 fields in line 5, saw 3". If, instead, the backslash character is used to escape all the commas delimiting decimal places (\,), then read_csv can be configured in such a way to correctly parse the data.

Listing 4-19. Using commas as a delimiter and a decimal point

```
>> import pandas as pd
>> data = io.StringIO(
     """
     temp, location
     35,234unf923
     32,2340inf012
     33,2340inf351
     33\,1,2340abe045
     """
)
>> grades = pd.read_csv(
     data,
     decimal=",",
     escapechar="\\",
     index_col="location",
)
                 temp
     location
     234unf923   35.000000
     2340inf012  32.000000
     2340inf351  33.000000
     2340abe045  33.100000
```

pd.read_json

The read_json loader parses entirely in C unlike read_csv which may use the Python parser under certain conditions.

The parameter orient defines how the JSON format will be converted into a pandas DataFrame. There are six different options: split, records, index, columns, values, and table. If the JSON is formatted such that there are columns, data, and an index already defined as keys, as is the case in Listing 4-20, the split option should be used. It's also worth noting that the JSON parser is particularly picky about spacing including whitespace.

Listing 4-20. Using orient split

```
>> data = io.StringIO(
     """
     {
         "columns": ["temp"],
         "index": ["234unf923", "340inf351", "234abe045"],
         "data": [[35.2],[32.5],[33.1]],
     }
     """
)
>> temperatures = pd.read_json(
     data,
     orient="split",
)
                 temp
     234unf923   35.200000
     340inf351   32.500000
     234abe045   33.100000
```

If the JSON is formatted such that each value is a row in the data with the column names as keys, as is the case in Listing 4-21, the records option should be used.

Listing 4-21. Using orient records

```
>> data = io.StringIO(
     """
     [
         {"location": "234unf923", "temp": 35.2},
         {"location": "340inf351", "temp": 32.5},
         {"location": "234abe045", "temp": 33.1},
     ]
     """
)
>> temperatures = pd.read_json(
     data,
     orient="records",
)
     location     temp
     234unf923    35.200000
     340inf351    32.500000
     234abe045    33.100000
```

If the JSON is formatted such that each key is the index value and the value of each key is a dictionary of the columns and values for the row, as is the case in Listing 4-22, the index option should be used.

Listing 4-22. Using orient index

```
>> data = io.StringIO(
     """
     {
         "234unf923": {"temp": 35.2},
         "340inf351": {"temp": 32.5},
         "234abe045": {"temp": 33.1},
     }
     """
)
>> temperatures = pd.read_json(
     data,
     orient="index",
)
                temp
     234unf923   35.200000
     340inf351   32.500000
     234abe045   33.100000
```

If the JSON is formatted such that each key is the column and each value is a dictionary where the key is the index and the value is the column value, as is the case in Listing 4-23, the columns option should be used.

Listing 4-23. Using orient columns

```
>> data = io.StringIO(
     """
     {
         "temp": {
             "234unf923": 35.2,
             "340inf351": 32.5,
             "234abe045": 33.1,
         },
     }
     """
)
>> temperatures = pd.read_json(
     data,
     orient="columns",
)
```

```
                 temp
     234unf923   35.200000
     340inf351   32.500000
     234abe045   33.100000
```

If the JSON is formatted such that each row is simply represented as a list of values, as is the case in Listing 4-24, the values option should be used.

Listing 4-24. Using orient values

```
>> data = io.StringIO(
     """
     [
         ["234unf923", 35.2],
         ["340inf351", 32.5],
         ["234abe045", 33.1],
     ]
     """
)
>> temperatures = pd.read_json(
     data,
     orient="values",
)
             0           1
     0       234unf923   35.200000
     1       340inf351   32.500000
     2       234abe045   33.100000
```

If the JSON is formatted such that it provides a detailed data schema, as is the case in Listing 4-25, the table option should be used.

Listing 4-25. Using orient table

```
>> data = io.StringIO(
     """
     {
         "schema": {
             "fields": [
                 {"name": "location", "type": "string"},
                 {"name": "temp", "type": "string"},
             ],
             "primaryKey": "location",
         },
         "data": [
                 {"location": "234unf923", "temp": 35.2},
                 {"location": "340inf351", "temp": 32.5},
                 {"location": "234abe045", "temp": 33.1},
         ]
     }
     """
)
>> temperatures = pd.read_json(
     data,
     orient="table",
)
                 temp
     location
     234unf923    35.200000
     340inf351    32.500000
     234abe045    33.100000
```

Similar to read_csv, read_json has a chunksize that lets you read the files in chunks at a time. This only is permitted however if the lines option is also set to True, meaning the JSON format is oriented as records without the list brackets. Listing 4-26 demonstrates this.

Listing 4-26. Loading the file in chunks

```
>> data = io.StringIO(
     """
     {"location": "234unf923", "temp": 35.2}
     {"location": "340inf351", "temp": 32.5}
     {"location": "234abe045", "temp": 33.1}
     """
)
>> temperatures = pd.DataFrame({})
>> reader = pd.read_json(
     data,
     lines=True,
     chunksize=2,
)
>> for chunk in reader:
     temperatures = temperatures.append(process(chunk))
>> temperatures
     location       temp
     234unf923      35.200000
     340inf351      32.500000
     234abe045      33.100000
```

By default, the JSON loader determines whether certain columns are date-like based on the column name unlike other readers that look at the values. It accepts a convert_dates parameter which can be a list of column names or a Boolean. If it's set to True, it converts columns that end with _at or _time, begin with timestamp, or are named modified or date into datetimes. To disable automatic detection of date columns, you can set keep_default_dates to False.

By default, the JSON loader will try to infer the type of each column and axis, unless orient is set to table, in which case the type is provided as part of the JSON schema. Just like reading a CSV file, if the types are not specified in the JSON file, it saves a lot of memory to provide them. Listing 4-27 shows what the memory footprint might be of a JSON file being loaded without types specified vs. Listing 4-28 which shows the memory footprint of the same JSON file with types specified. Note in Listing 4-28 where types are explicitly specified, the memory of the resulting DataFrame decreased by about 40%.

Listing 4-27. Loading a JSON with pandas type inference

```
>> data = io.StringIO(
     """
     {
         "birth": {
             "129237": "04/10/1999",
             "123083": "05/18/1989",
         },
         "height": {
             "129237": 5.4,
             "123083": 6.1,
         },
         "weight": {
             "129237": 126,
             "123083": 130,
         },
     }
     """
)
>> patient_info = pd.read_json(
     data,
     orient="columns",
)
```

```
                birth         height  weight
     129237     04/10/1999    5.4     126
     123083     05/18/1989    6.1     130
>> df.dtypes
     birth          object
     height         float64
     weight         int64
>> df.index.dtype
     dtype('int64')
>> df.memory_usage()
     Index          16
     birth          16
     height         16
     weight         16
```

Listing 4-28. Explicitly specifying the type when loading a JSON

```
>> data = io.StringIO(
     """
    {
          "birth": {
              "129237": "04/10/1999",
              "123083": "05/18/1989",
          },
          "height": {
              "129237": 5.4,
              "123083": 6.1,
          },
          "weight": {
              "129237": 126,
              "123083": 130,
            },
        }
        """
)
>> patient_info = pd.read_json(
        data,
        orient="columns",
        convert_dates=["birth"],
        dtype={"height": np.float16, "weight": np.int16},
)
                  birth           height    weight
        129237    1999-04-10      5.4       126
        123083    1989-05-18      6.1       130
>> df.dtypes
        birth           datetime64[ns]
        height          float16
        weight          int16
>> df.index.dtype
        dtype('int64')
>> df.memory_usage()
        Index           16
        birth           16
        height          4
        weight          4
```

pd.read_sql, pd.read_sql_table, and pd.read_sql_query

The pandas read_sql loader is a wrapper around read_sql_table and read_sql_query. Depending on the parameters passed to it, it calls one of those two functions underneath. It also has built-in support for SQLAlchemy.

SQLAlchemy is a very popular Object Relational Mapper library, also known as an ORM. The pandas SQL reader also supports talking directly to DBAPI which is a lower-level database library that SQLAlchemy depends on. While DBAPI is limited within pandas to SQLite3, SQLAlchemy can talk to all kinds of relational databases so there's no need to rewrite all your SQL queries when switching databases. ORMs are quite popular in the application development space as they allow you to map your database tables to objects or classes in Python. This is nice because you can track your database table definitions inside your codebase. You can define your database tables as classes and then with a simple command add them to your database. You can also modify existing database tables via migration scripts using migration libraries like Alembic, for example, which let you roll forward and roll back database changes with little risk of unrecoverable production database mishaps. When building out table definitions, you can also specify things like the conversion of column types between Python and the database. SQLAlchemy is also nice because it abstracts the SQL query into more human-readable query language with easily parameterized expressions, like in Listing 4-29.

Listing 4-29. Using a raw SQL string query vs. the SQLAlchemy ORM to generate a query

```
cur.execute(
    """
    SELECT * FROM temperature_readings
    WHERE temperature_readings.temp > 45
    """
)
session.query(TemperatureReadings).filter(
    temp > 45
)
```

Listing 4-30 shows an example of how you might build a database and insert data into it. In this example, we are creating a user table with two columns, id and name, and inserting a new user with an id of zero and name Eric into that table. Note two different URLs are defined in the code, the one in use connects to sqlite and the other connects to a local Postgres database instance.

Listing 4-30. Creating tables in a postgres database and adding data to it using SQLAlchemy

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()

SQLITE_URL = "sqlite://"
POSTGRES_URL = "postgresql://postgres@localhost:5432"

class User(Base):
     __tablename__ = 'user'
     id = Column(Integer, primary_key=True)
     name =  Column(String(50))

engine = create_engine(SQLITE_URL)
Session = sessionmaker(bind=engine)

def create_tables():
     Base.metadata.create_all(engine)

def add_user():
     session = Session()
     user = User(id=0, name="Eric")
     session.add(user)
     session.commit()
     session.close()

>> create_tables()
>> add_user()
```

Listing 4-31 provides a simple docker-compose.yml file which will spin up a local Postgres database on your machine using the command in Listing 4-32. Note this database is configured to write the data to disk so you can kill the docker container at any time and the data will persist in the postgres-data directory and be there the next time you spin up the docker container.

Listing 4-31. The contents of the docker-compose.yml that will create a local Postgres database

```
version: '3'
services:
     postgres:
          image: postgres:9.4-alpine
          ports:
               - '127.0.0.1:5432:5432'
          volumes:
              - ./postgres-data:/var/lib/postgresql/data
```

Listing 4-32. Starting the Postgres database

```
>> docker-compose up -d
```

The SQL loaders generally can either load the whole table or load parts of the table based on a query. Although the SQL loaders accept a SQLAlchemy engine, they only accept a select statement rather than a query object. This means while you can use SQLAlchemy's query API, you must convert it to a selectable before passing it into the loader as shown in Listing 4-34. A selectable is essentially the raw SQL query string. Listing 4-33 shows an example of how you would load all the user data from the database into a DataFrame, while Listing 4-34 shows how you might load the user with id=0 into a DataFrame.

Listing 4-33. Loading all the users into a DataFrame using read_sql

```
>> pd.read_sql(
     sql=User.__tablename__,
     con=engine,
     columns=["id", "name"],
)
     id   name
0    0    Eric
```

Listing 4-34. Loading the user with id=0 into a DataFrame using read_sql

```
>> select_user0 = session.query(Patient).filter_by(id=0).
     selectable
>> pd.read_sql(
     sql=select_user0,
     con=engine,
     columns=["id", "name"],
)
     id   name
0    0    Eric
```

The SQL loaders have similar options as the other loaders we've looked at, for example, loading the data chunks at a time or datetime conversion. However, there are differences as well. Unlike some of the other loaders we've looked at, the SQL loaders do not have an option for data type specification. This often poses a problem for pandas users working with databases as they may store a normalized version of the data in a database and then wish to load it back out only to find the data types are not the same. If you run into this situation, SQLAlchemy and some custom loading code can help. SQLAlchemy provides a custom types option which lets you convert between the database type and the Python type. As we've seen with other loaders where the types are not explicitly specified, pandas will store the id column as an int64. Listing 4-35 shows an example of how we might specify the Python type for the integer id column as an int32 instead of a more generic and larger integer type. Using this table definition, now when we add a user, the id will be stored as an integer inside the database, but when we read it out, it will be a NumPy int32 type.

Listing 4-35. Using SQLAlchemy TypeDecorator to specify a data type for pandas

```
from sqlalchemy import Column, String
from sqlalchemy.ext.declarative import declarative_base
import sqlalchemy.types as types
import numpy as np
Base = declarative_base()

class Int32(types.TypeDecorator):
     impl = types.Integer

     def process_bind_param(self, value, dialect):
         return value

     def process_result_value(self, value, dialect):
         return np.int32(value)

class User(Base):
     __tablename__ = 'user'
     id = Column(Int32, primary_key=True)
     name =  Column(String(50))
```

Listing 4-36 shows a code snippet of the internal implementation of read_sql where self.pd_sql is the SQLAlchemy engine object, sql_select is the selectable passed in as the SQL parameter, and self.frame is the resulting DataFrame that is returned. Here you can see exactly how pandas is loading the data from the database and converting it into a DataFrame.

The fetchall function returns the data as a list of tuples, for example, [(0, 'Eric')]. This implementation is relevant for the next step in how we will get pandas to use the correct data types we defined in Listing 4-35.

Listing 4-36. Part of the pandas implementation of read_sql

```
result = self.pd_sql.execute(sql_select)
column_names = result.keys()

data = result.fetchall()
self.frame = DataFrame.from_records(
     data, columns=column_names, coerce_float=coerce_float
)
```

Instead of relying on the pandas read_sql implementation, we are going to write our own custom SQL loading code that will maintain the data types we defined in the SQLAlchemy user table when creating the DataFrame. The custom SQL loading code is shown in Listing 4-37 and is faster and consumes less memory than using astype to convert the types after loading.

Listing 4-37. Custom SQL loader code that maintains the data types defined on the SQLAlchemy table in Listing 4-35

```
>> sql = session.query(User).selectable
>> results = engine.execute(sql).fetchall()
>> data = {
     columns[col]: np.array(
           [row[col] for row in results],
           dtype=type(results[0][col]))
     for col, v in enumerate(results[0])}
>> df = pd.DataFrame(data)
>> df.dtypes
     0    int32
     1    object
```

We've covered several of the most popular loaders and their options in this chapter, but there are still many more. Be sure to read the documentation for the particular loader you are using and see what kinds of normalization during load features are at your disposal, and if not, you may have to write some custom code yourself. Keep in mind performance savings can come from reducing memory overhead and reducing steps during the load and normalization process. pandas provides many ways of improving normalization and load performance depending on the bottlenecks you have in your particular situation. In Chapter 5, we'll explore how to reshape the data into the desired DataFrame format once it's loaded and normalized.

CHAPTER 5

Basic Data Transformation in pandas

The pandas library has a huge API that provides many ways of transforming data. In this chapter, we'll cover some of the most powerful and most popular ways to transform data in pandas.

Pivot and pivot table

Pivot and pivot table are very popular and particularly attractive to beginners because they are so powerful. However, their powerfulness comes at a performance cost. While pivot is a great tool for initially transforming the DataFrame as part of a data normalization step, it should not be used frequently throughout the data analysis phase. Listing 5-1 shows an example of using pivot table to convert the raw inspection data into an aggregated format of restaurants and their average inspection score. Note in Listing 5-1 the aggregation function is explicitly specified as np.mean though this is unnecessary since np.mean is the default. Pivot is essentially doing a groupby, applying the aggregation function as needed, and reorganizing the results into a new table format.

Listing 5-1. Calculating the average inspection score per restaurant with pivot_table

```
>> df
     restaurant  location    date       score
     Diner       (4, 2)      02/18      90
     Pandas      (5, 4)      04/18      55
     Diner       (4, 2)      05/18      100
     Pandas      (5, 4)      01/18      76
>> df = df.pivot_table(
     values=['score'],
     index=["restaurant","location"],
     aggfunc=np.mean
)
>> df
                               score
     restaurant  location
     Diner       (4, 2)        95
     Pandas      (5, 4)        66
```

There are a couple performance issues in Listing 5-1. Pivot table does not have an option to limit memory duplication so it creates an entirely new DataFrame each time it is used. If your DataFrame is quite large, this can be a big performance hit to your program. Internally, pivot table is grouping the data by unique restaurant and location combinations which takes time, particularly with a large amount of combinations. If this was being used as part of a data normalization step, it would be far better than if it was used many times throughout a program as part of data analysis. This is because the performance hit of uniquely grouping and copying all that memory would happen only once compared to it happening many times throughout the program. It is far better to normalize and orient a DataFrame once in such a way that it optimizes all the analysis you plan to perform on it than leave it in a somewhat unoptimized raw form and have to re-orient it at each analysis step. Note, if instead the DataFrame was already uniquely grouped, we could run a groupby to calculate the average score like in Listing 5-2 which would be twice as fast. We'll discuss the performance of groupby in depth in Chapter 7. It is very likely that other analysis needs the data grouped by unique restaurant, so the grouping in this example at the very least should be part of the normalization step, in which case it becomes unnecessary to use pivot table at all.

Listing 5-2. Calculating the average inspection score per restaurant with groupby

```
>> df
                          date        score
     restaurant  location
     Diner       (4, 2)   02/18       90
                          04/18       55
     Diner       (4, 2)   05/18       100
                          01/18       76
>> df = df[["score"]].groupby(["restaurant","location"]).mean()
>> df
     restaurant  location    score
     Diner       (4, 2)      95
     Pandas      (5, 4)      66
```

Pivot does the same thing as pivot table, but it does not allow you to aggregate data. Any columns and index value combinations that result in multiple values must be aggregated together when using pivot table. Pivot, on the other hand, simply throws a ValueError if it runs into such a scenario. Note in Listing 5-3 no combination of drug and date results in multiple values; however, in Listing 5-4, there are or would be multiple rows for the same drug and date; thus, Listing 5-4 throws a ValueError. So, a regrettable limitation of both pivot and pivot table is they do not output data where there are multiple values for an index and column combination. Pivot table forces you to aggregate the multiple values together or select one and pivot simply throws a ValueError.

Listing 5-3. Re-orienting a DataFrame using pivot

```
>> df
    date          tumor_size     drug        dose
    02/18         90             01384       10
    02/25         80             01384       10
    03/07         65             01384       10
    03/21         60             01384       10
    02/18         30             01389       7
    02/25         20             01389       7
    03/07         25             01384       10
    03/21         25             01389       7
>> df.pivot(
    index="drug",
    columns="date",
    values="tumor_size"
)
    date    02/18    02/25    03/07    03/21
    drug
    01384   90        80       65        60
    01389   30        20       25        25
```

Listing 5-4. Pivot throws a ValueError when there are multiple values for the same column-index combination

```
>> df
    date          tumor_size    drug     dose
    02/18         90            01384    10
    02/25         80            01384    10
    03/07         65            01384    10
    03/21         60            01384    10
    02/18         30            01389    7
    02/25         20            01389    7
    03/07         25            01389    7
    03/21         25            01389    7
```

```
>> df.pivot(
     index="drug",
     columns="date",
     values="tumor_size",
)
ValueError: Index contains duplicate entries, cannot reshape
```

Pivot is more performant than pivot table because it does not allow specification and generation of multi-level columns and multi-indexes. Thus, it does not have the overhead of generating and handling this more complex DataFrame format. Regardless of whether the resulting DataFrame is a multi-index or multi-level column DataFrame, pivot table still runs the various computations as if it is multi-level which adds a fair amount of overhead, up to six times more than pivot in some cases. While pivot will allow you to specify multiple values and create a multi-level column for them, it does not allow you to provide an explicit list of columns to generate multi-level columns or provide a list of indexes to generate multi-level indexes. Pivot table, on the other hand, supports this type of multi-level DataFrame. It also has some other nicety options like adding a subtotal of all rows and columns and dropping columns with NaNs. In summary, if you can get away with using pivot, you should, as it's more performant than using pivot table.

Stack and unstack

Stack and unstack reshape a DataFrame's column level into an innermost index and vice versa. An example of this is shown in Listing 5-5 where each column is a restaurant health inspection, the value is the health inspection score, and the index represents the restaurant that was inspected. Stack is used to reshape the data so that the health inspection scores for each restaurant occur across each row rather than each column. Note stack converts the column names across the top into column values which then are ultimately dropped from the DataFrame.

Listing 5-5. Reshaping a DataFrame so that each row represents an inspection using stack

```
>> df
                                        score
     inspection                         0    1
     restaurant  location
     Diner       (4, 2)                 90   100
     Pandas      (5, 4)                 55   76
>> df = df.stack().reset_index()
>> df
     restaurant  location   inspection  score
     Diner       (4, 2)      0           90
                             1           100
     Pandas      (5, 4)      0           55
                             1           76
>> df.drop(column=["inspection"], inplace=True)
>> df.set_index(["restaurant", "location"], inplace=True)
>> df
                             score
     restaurant  location
     Diner       (4, 2)       90
                              100
     Pandas      (5, 4)       55
                              76
```

You might recognize the shape of the original DataFrame in Listing 5-5 from Listing 3-22. The shape of the DataFrame before it's reshaped in Listing 5-5 is the orientation that was deemed the most optimal in the

“Choosing the right DataFrame” section at the end of Chapter 3. Listing 5-5 shows how to convert from that optimal shape to the original non-optimal shape. Now let’s look at how we might take the original non-optimal shape and turn it into the optimal shape. Listing 5-6 adds a new column called inspection to the DataFrame whose values become the column names for the new DataFrame. We also are making use of a handy groupby aggregation function called cumcount that creates a row number for each row in each group.

Listing 5-6. Reshaping a DataFrame so that each column is an inspection using unstack

```
>> df
                              score
     restaurant  location
     Diner       (4, 2)       90
                              100
     Pandas      (5, 4)       55
                              76
>> df["inspection"] = df.groupby(
     ["restaurant", "location"]).cumcount()
>> df
                           inspection  score
     restaurant  location
     Diner       (4, 2)     0           90
                            1           100
     Pandas      (5, 4)     0           55
                            1           76
>> df.set_index("inspection", append=True, inplace=True)
>> df
```

```
                                            score
     restaurant  location    inspection
     Diner         (4, 2)    0              90
                             1              100
     Pandas          (5, 4)  0              55
                             1              76
>> df = df.unstack()
>> df
                                            score
     inspection                             0    1
     restaurant  location
     Diner       (4, 2)                     90   100
     Pandas      (5, 4)                     55   76
```

So how performant are stack and unstack? They both duplicate memory as they are not inplace operations which can be costly and thus should really only be used in data normalization. They are, however, very unique in the way they can transform the data, so it is difficult to find a more performant alternative other than melt which is what we'll explore in the next section.

Melt

An example of using melt is shown in Listing 5-7. Note that this is very similar to the stack example. We are essentially doing what we did in approximately four lines with stack in one line in this example. While melt does the same thing as stack and a bit more, it does it in a slightly more performant way. This is mainly due to the slight overhead advantage it has in not calling into all the various data transformations at a high level, meaning rather than calling stack underneath, melt performs the lower-level data manipulations underneath stack directly, thus avoiding the middle code layers. If you compare a raw stack to melt, stack is about four times faster. The drawback of using stack is that it often requires other manipulation such as setting an index, renaming columns, converting it back to a DataFrame, and so on. This means in some cases it's more performant to just use melt.

Listing 5-7. Reshaping a DataFrame so that each row represents an inspection using melt

```
>> df
     restaurant  location    0    1
     Diner       (4, 2)      90   100
     Pandas      (5, 4)      55   76
>> df = df.melt(
     id_vars=["restaurant","location"],
     value_vars=[0,1],
     value_name="score").drop(columns="variable")

>> df
     restaurant  location      score
     Diner       (4, 2)        90
                               100
     Pandas      (5, 4)        55
                               76
```

Transpose

Transpose is a useful trick. It simply turns the columns into rows and the rows into columns. In Listing 5-8, there is a list of patients who need to be treated for a certain disease and a table that provides a list of drugs used to treat the disease based on blood type. We need to add the list of drugs that can be used to treat the given patient into the patient table based on the patient's blood type. The first step is to index both the patient list and the drug table by blood type, and then we can do a simple join to add the drug data into the patient list. Because the drug table is oriented such that the blood types are across the columns instead of the rows, we first do a transpose. Note when we do this, the index which is provided by default when creating the DataFrame turns into the columns and the columns turn directly into the index. This means in Listing 5-8 we don't explicitly have to set the index as the transpose already sets the index to the blood type for us.

Listing 5-8. Using a transpose to reshape a DataFrame

```
>> patient_list
               id      history
  blood_type
  O+           02394   hbp
  B+           02312   NaN
  O-           23409   lbp
>> drug_table
  index        O+   O-   A+   A-   B+   B-
  0            ADF  ADF  ACB  DCB  ACE  BAB
  1            GCB  RAB  DF   EFR       HEF
  2            RAB
>> drug_table = drug_table.transpose(copy=False)
>> drug_table
  blood_type  0     1      2
  O+          ADF   GCB    RAB
  O-          ADF   RAB
  A+          ACB   DF
  A-          DCB   EFR
  B+          ACE
  B-          BAB   HEF
>> patient_list.join(drug_table)

                id       history    0      1      2
  blood_type
  O+             02394   hbp        ADF    GCB    RAB
  B+             02312   NaN               ACE
  O-             23409   lbp               ADF    RAB
```

Transpose is one of the few DataFrame reshaping functions that has an option of not duplicating data if possible named copy. That being said, copy=False doesn't necessarily mean the data is not duplicated, as we'll explain in more detail in Chapter 9. Whether the data is duplicated or not depends on a multitude of factors which ultimately boil down to whether the new shape of the data can reuse the underlying NumPy arrays as is or whether new NumPy arrays must be created. Recall that NumPy arrays must all be the same type. This means if the DataFrame you are transposing has the same types for the rows and the columns, then it will likely be able to reuse the existing NumPy arrays. If not, it will have to duplicate memory and re-build them. This means transpose should really be used only when it's absolutely necessary and ideally as part of a data normalization step.

The takeaway from looking at all these data transformation functions is data transformation is quite costly and in an ideal program should happen only during the normalization phase. It should be used sparingly during data analysis. Chances are that if you find yourself having to do a lot of transformations at each of the data analysis steps, you should re-think the orientation of your normalized DataFrame. In the next chapter, we'll look at the apply method and explore when it should and should not be used as well as more performant alternatives.

CHAPTER 6

The apply Method

apply is one of the most incorrectly used functions in pandas. Chances are if you are using it, you shouldn't be. This is because apply "applies" the function to each row or each column in the data set effectively breaking one of the cardinal rules of using pandas: do not iterate over the data set. In this chapter, we'll explore when apply is the right choice and present alternative solutions for when it's not.

When not to use apply

For those comfortable with basic programing features, iteration is a familiar way to manipulate data. We think to ourselves: I would like to run this operation on every row or every column, and thus apply looks very friendly. However, that way of organizing a problem is completely wrong in pandas. Much of the same principles used in working with relational databases can also be used when working in pandas. When you perform an operation on data in a database, you don't do it one row at a time but rather define a range; the same is true in pandas. When operating on a data set, you define all the elements you wish to operate on and then provide the operation. In the simplest form, this might look like df["col 1"] + df["col 2"], and in a more complex case, this might look like df.where(100 > df >= 90, "A").

pandas has many built-in functions for performing data computational operations. A comprehensive list is provided in the Appendix. These computations often directly translate to a NumPy function, operating in C, which makes these much more performant than their apply equivalents. They are accessible directly off the pandas DataFrame and also the pandas Series object (a column or row of a pandas DataFrame).

A simple example of apply is illustrated in Listing 6-1. We pass it the sum function, specify the axis we want to apply the function to, and we get the sum of each row.

Listing 6-1. Example of using apply

```
>> df = pd.DataFrame([[4, 9],[6, 7]], columns=['A', 'B'])
>> df
        A   B
     0  4   9
     1  6   7
>> df.apply(np.sum, axis=1)
     0    13
     1    13
```

While the example in Listing 6-1 is simple and illustrates *how* to use apply, the use case *in which* it is used is very wrong. It's a textbook example of when to *not* use apply as the np.sum function is a built-in off the DataFrame itself and thus the built-in should be used as it's much more performant. But why is it so much more performant? Let's explore that in more detail.

The answer to the question of why the built-in pandas sum is so much more performant than applying the NumPy sum to each row lies in where the iteration over the rows takes place. The following loop in Listing 6-2 is the underlying implementation of the pandas apply method.

Listing 6-2. Main loop in the pandas apply implementation

```
for i, v in enumerate(series_gen):
     results[i] = self.f(v)
     keys.append(v.name)
```

As you can see in Listing 6-2, the looping over the rows takes place in Python. Here you can see the series_gen which is either the columns or the rows that the function to be applied (held in self.f) will be applied to. This is in opposition to the built-in pandas sum function that simply passes an ndarray to be operated on to the NumPy sum function, which then iterates and sums the data in C and returns the resulting ndarray back to Python. This process of running the operation on the data in C instead of Python is known as vectorization. Essentially, vectorization is able to achieve a huge speedup over the alternative of running the operation in Python. For all the reasons covered in Chapter 3, looping and performing operations in C is much more performant than Python. However, the speedup doesn't always come from just looping in C.

Vectorized operations allow you to apply a mathematical operation to a sequence of numbers. For example, if you want to add 4 to each element in an ndarray, you specify that using the syntax arr + 4. In the case of NumPy ufuncs (see the Appendix for a comprehensive list), they actually make use of specialized vector registers in the CPU itself. Vector registers are registers that can contain a series of values, and when an operation is performed on them, it is performed on each value in the register at once. So, what would have been a loop over an array of eight values and eight consecutive add instructions in the CPU becomes one add instruction operating on eight values in the CPU. As you can imagine, this leads to a huge speedup. Vectorization will also pad arrays of mismatched dimensions in order to make the dimensions match such that an operation can run. This process is known as broadcasting. When you add a new column in pandas via df["new_col"] = 4, 4 is broadcast to have the same number of rows as all the other columns in the DataFrame. Similarly, aggregation functions like sum operate over a sequence of numbers using vectorization. What all of this boils down to is apply is not a vectorized operation—it loops in Python and should be avoided whenever possible. It becomes effectively the same thing as iterating over the rows and applying the function yourself as illustrated in Listing 6-3.

Listing 6-3. Equivalent of apply implemented manually

```
results = [0]*len(df)
for i, v in df.itterrows():
     results[i] = v.sum()
df["sum"] = results
```

In fact, the performance of this custom implementation of apply illustrated in Listing 6-3 vs. Listing 6-1 where apply is used directly yields slightly better performance simply because there is less overhead to get to the guts of the operation when implementing this simple custom alternative.

How much slower is apply vs. using a built-in pandas operation though? Let's look at some concrete examples and compare the performance. Comparing the performance of the apply example in Listing 6-1 to the alternative method in Listing 6-4 when performed on 100,000 rows, the apply function averages about 8.5 seconds compared to running the sum function directly off the pandas DataFrame which averages about 0.4 milliseconds.

Listing 6-4. Alternative implementation of Listing 6-1

```
df.sum(axis=1)
```

Let's look at another example. Say you have a data set with one column named A but that column has incomplete data and you wish to replace the values that are missing with the max of columns B and C. This could be implemented using apply as demonstrated in Listing 6-5, or this could be implemented in a much more performant way using the where method demonstrated in Listing 6-6.

Listing 6-5. Replacing missing data using apply

```
def replace_missing(series):
     if np.isnan(series["A"]):
         series["A"] = max(series["B"], series["C"])
     return series

df = df.apply(replace_missing, axis=1)
```

Listing 6-6. Replacing missing data using the where method

```
df["A"].where(
     ~df["A"].isna(),
     df[["B", "C"]].max(axis=1),
     inplace=True,
)
```

The where method replaces falsey values with the value provided in the second parameter. This means, in Listing 6-6, all the NaN values are being replaced with the max of columns B and C. Note we are also specifying inplace=True so that this replace happens on the current DataFrame as opposed to creating a new DataFrame that would result in duplicated memory.

Let's look at a trickier example in Listings 6-7 and 6-8. Suppose you have a DataFrame with two columns, fruit and order, and you want to drop all the data where the fruit is not present in the order for each row. pandas does have string operations including Series.str.find that will return True if a substring is present in a string for each value in a Series. However, it will only allow you to pass in a constant. In other words, you cannot specify a Series of substrings but only a single string value, so find will not work in this case. There is also no "in" check built into pandas that operates on two series objects, so although this is exactly what we want, pandas does not support it. This means we must implement some kind of customized solution ourselves, so let's explore the performance of various options.

Listing 6-7. Dropping rows whose order column does not contain the substring in the fruit column using apply

```
def test_fruit_in_order(series):
     if (series["fruit"].lower() in
          series["order"].lower()
     ):
          return series
     return np.nan

>> data = pd.DataFrame({
     "fruit": ["orange", "lemon", "mango"],
     "order": [
         "I'd like an orange",
         "Mango please.",
         "May I have a mango?",
     ],
})
          fruit     order
     0    orange    I'd like an orange
     1    lemon     Mango please.
     2    mango     May I have a mango?

>> data.apply(
     test_fruit_in_order,
     axis=1,
     result_type="reduce",
).dropna()
          fruit     order
     0    orange    I'd like an orange
     2    mango     May I have a mango?
```

Listing 6-8. Solving Listing 6-7 using a list comprehension

```
mask = [fruit.lower() in order.lower()
     for (fruit, order) in data[["fruit", "order"]].values]
data = data[mask]
```

Using apply to solve this problem as in Listing 6-7 takes about 14 seconds on 100,000 rows, whereas using a list comprehension as in Listing 6-8 takes about 100 milliseconds. But why is a list comprehension so much faster than apply? Don't they both loop in Python? List comprehensions are specially optimized loops within the Python interpreter. The bytecode that they translate into more closely resembles a loop written in C as they do not load a bunch of specialized Python list attributes. What follows is the bytecode for a for loop (Listing 6-9) vs. a list comprehension (Listing 6-10). Notice how much simpler and smaller the bytecode is for a list comprehension than for a for loop even though they are doing the same thing.

Listing 6-9. Bytecode translation of a simple for loop

```
def for_loop():
     l = []
     for x in range(5):
          l.append( x % 2 )

     0        0 BUILD_LIST            0
              2 STORE_FAST            0 (l)
     1        4 SETUP_LOOP            30 (to 36)
              6 LOAD_GLOBAL           0 (range)
              8 LOAD_CONST            1 (5)
             10 CALL_FUNCTION         1
             12 GET_ITER
       >>    14 FOR_ITER              18 (to 34)
             16 STORE_FAST            1 (x)
     2       18 LOAD_FAST             0 (l)
             20 LOAD_METHOD           1 (append)
```

```
            22 LOAD_FAST                1 (x)
            24 LOAD_CONST               2 (2)
            26 BINARY_MODULO
            28 CALL_METHOD              1
            30 POP_TOP
            32 JUMP_ABSOLUTE            14
       >>   34 POP_BLOCK
       >>   36 LOAD_CONST               0 (None)
            38 RETURN_VALUE             None
```

Listing 6-10. Bytecode translation of a list comprehension

```
def list_comprehension():
    l = [x % 2 for x in range(5)]

    0         0 LOAD_CONST              1
              2 LOAD_CONST              2
              4 MAKE_FUNCTION           0
              6 LOAD_GLOBAL             0 (range)
              8 LOAD_CONST              3 (5)
              10 CALL_FUNCTION          1
              12 GET_ITER
              14 CALL_FUNCTION          1
              16 STORE_FAST             0 (l)
              18 LOAD_CONST             0 (None)
              20 RETURN_VALUE           None
```

When to use apply

So far, we've looked at some examples that don't warrant the use of apply. Let's take a look at one that does. Often the reality of working with data in the wild results in some much more complex scenarios. Say you want to calculate the percentile of score for each element in a DataFrame, the implementation of which is provided in Listing 6-11.

Listing 6-11. Implementation of scipy.stats.percentileofscore

```
def percentileofscore(a, score):
     """
     Three-quarters of the given values lie below a given score:
     >>> stats.percentileofscore([1, 2, 3, 4], 3)
     75.0

     With multiple matches, note how the scores of the two
     matches, 0.6 and 0.8 respectively, are averaged:
     >>> stats.percentileofscore([1, 2, 3, 3, 4], 3)
     70.0
     """
     n = len(a)
     left = np.count_nonzero(a < score)
     right = np.count_nonzero(a <= score)
     pct = (right + left + (1 if right > left else 0)) * 50.0/n
     return pct
```

This means if we had the following input DataFrame, we would see the following output DataFrame after applying scipy.stats.percentileofscore using the pandas apply function (Listing 6-12).

Listing 6-12. Applying scipy.percentileofscore to each element in a DataFrame

```
>> from scipy import stats
>> data = pd.DataFrame(np.arange(20).reshape(4,5))
              0    1    2    3    4
         0    0    1    2    3    4
         1    5    6    7    8    9
         2    10   11   12   13   14
         3    15   16   17   18   19
```

```
>> def apply_percentileofscore(series):
     return series.apply(
         lambda x:stats.percentileofscore(series,x)
     )
>> data.apply(apply_percentileofscore, axis=1)
             0     1     2     3      4
        0   20.0  40.0  60.0  80.0  100.0
        1   20.0  40.0  60.0  80.0  100.0
        2   20.0  40.0  60.0  80.0  100.0
        3   20.0  40.0  60.0  80.0  100.0
```

This is a fairly complicated use case and a very non-performant implementation since it calls the apply method twice for each row. Unfortunately, there is no built-in pandas function that matches the behavior of SciPy's percentileofscore function being applied to each element like we need to do in this example. While we could do this calculation individually on the DataFrame one column at a time and piece the results back together, that would be a very cumbersome implementation. Listing 6-13 demonstrates this approach.

Listing 6-13. A more performant implementation of Listing 6-12

```
def percentileofscore(df):
     res_df = pd.DataFrame({})
     for col in df.columns:
         score = pd.DataFrame([df[col]]*5, index=df.columns).T
         left = df[df < score].count(axis=1)
         right = df[df <= score].count(axis=1)
         right_is_greater = (
             df[df <= score].count(axis=1)
             > df[df < score].count(axis=1)
         ).astype(int)
```

```
        res_df[f'res{col}'] = (
            left + right + right_is_greater
            ) * 50.0 / len(df.columns)
    return res_df

percentileofscore(data)
```

Listing 6-13's implementation results in better performance since we have eliminated one of the loops in Python (the loop over the rows) by doing pandas operations on all the rows at once. However, we were also forced to create a duplicate DataFrame where all columns are populated with the score in order to achieve these row-wise operations which results in undesirable memory overhead. Note we are also not reusing the implementation from SciPy but have re-implemented it using pandas operations, which is less than ideal for readability and increases the complexity and possibly the fragility of implementation. Fortunately, there is yet another way to implement this as we'll discuss in the next section.

Improving performance of apply using Cython

Taking a lesson from the previous example of a simple summation, if the pandas developers had provided this function for us off the DataFrame, it would have been implemented in C. So why don't we just implement it in C ourselves? You might be saying to yourself, "I don't know C—that sounds hard." But, in fact, the Cython library makes it quite easy, and you don't need to know C syntax to do it! Cython lets you write Python and compile it into C to be used as a C extension. First, we need to write our percentileofscore function that will operate on the entire DataFrame as shown in Listing 6-14. Then, compile it as shown in Listing 6-15, and finally we can use it as shown in Listing 6-16.

Listing 6-14. A more performant implementation of Listing 6-12 using Cython

```
from scipy.stats import percentileofscore as pctofscore
from copy import deepcopy

def percentileofscore(values):
     percentiles = [0]*len(values[0])
     num_rows = len(values)
     for row_index in range(num_rows):
         row_vals = values[row_index]
         for col_index, col_val in enumerate(row_vals):
             percentiles[col_index] = \
                 pctofscore(row_vals, col_val)
         values[row_index] = percentiles
```

Listing 6-15. setup.py for compiling Cython in Listing 6-14

```
import pyximport; pyximport.install(language_level=3)
from distutils.core import setup
from Cython.Build import cythonize

setup(
     ext_modules = cythonize("percentileofscore.pyx")
)

>> python setup.py build_ext --inplace
```

Listing 6-16. Using the compiled Cython function implemented in Listing 6-14

```
from percentileofscore import percentileofscore
percentileofscore(data.values)
```

Note that the Cython function accepts values and not the full pandas DataFrame; this is because values is a two-dimensional array and something that is easily translatable into C, whereas the pandas DataFrame is a Python object and is not. Also note that the function modifies the data in place as opposed to returning a whole new two-dimensional array. This is a performance benefit as we do not have to allocate new memory for the new array, and once the data has been converted, we no longer need the original data set (at least in this particular case).

So how different is the performance of these approaches when run over 100,000 rows? Using apply in Listing 6-12 averages around 58 seconds. Using pandas operations to effectively re-implement the SciPy equivalent as in Listing 6-13 averages around 24 seconds. The third approach of building a custom Cython function averages around 4 seconds. There are also other advantages of going with the Cython approach other than performance. The SciPy function could be used as is and did not have to be re-implemented, so from an effort of implementation and readability perspective, it looks very appealing as well.

In conclusion, only when all other options have been exhausted should apply be used. It is equivalent in performance to iterrows and itercolumns and should be treated with the same level of precaution. In cases where apply needs to be used over a large data set and is causing a second or more slowdown, a customized Cython apply equivalent should be implemented instead so as to not degrade data analysis performance.

CHAPTER 7

Groupby

Chances are at some point when working with data in pandas, you will need to do some sort of grouping and aggregation of data. This is what Groupby is for. It allows you to cluster your data into groups and run aggregated calculations on those groups.

Using groupby correctly

When starting out, you may be inclined to do something like Listing 7-1 where you cluster your data into groups, then loop over each group, and run some aggregate. This however results in terrible performance because just as we saw in Chapter 6, we are looping in Python and not in C. If instead you call the aggregate function directly off the groupby as in Listing 7-2, the groups will be passed into the aggregate function and the looping will occur in C.

Listing 7-1. Calculating total number of arrivals to each destination per year by looping over groups

```
>> arrivals_by_destination
                    number
     date    place
     2015    LON    10
     2015    BER    20
     2015    LON    5
     2016    LON    10
     2016    BER    15
     2016    BER    10
>> groups = arrivals_by_destination.groupby(["date","place"])
>> for idx, grp in groups:
     arrivals_by_destination.loc[idx, "total"] = \
          grp["number"].sum()
>> arrivals_by_destination
                   number total
     date    place
     2015    LON    10      15
     2015    BER    20      20
     2015    LON    5       15
     2016    LON    10      10
     2016    BER    15      25
     2016    BER    10      25
```

Listing 7-2. Calculating total number of arrivals to each destination per year using groupby

```
>> arrivals_by_destination
                   number
     date    place
     2015    LON    10
     2015    BER    20
     2015    LON    5
     2016    LON    10
     2016    BER    15
     2016    BER    10
>> arrivals_by_destination["total"] = \
     arrivals_by_destination.groupby(["date","place"]).sum()
>> arrivals_by_destination
```

```
                    number total
date      place
2015      LON       10      15
2015      BER       20      20
2015      LON       5       15
2016      LON       10      10
2016      BER       15      25
2016      BER       10      25
```

The difference in performance between Listing 7-1 and Listing 7-2 is proportional to the number of groups. With just 8 groups, the performance of Listing 7-1 is twice as slow as Listing 7-2, and with 16 groups, it's four times as slow. Note in both these examples, we are starting with a pre-indexed DataFrame. This means the groups have already been pre-computed so the groupby does not have to calculate all the groups again but just reuse the existing groups in the index. This is a huge timesaver, particularly if you are going to be doing a lot of groupby operations over the columns in the index.

You may encounter a groupby scenario where you need a custom function that isn't built-in off the groupby object. This, however, is not the time to give in to looping. What you are implementing when you loop over the groups is the same performance as if you called apply on the groupby object itself and passed in your custom function. If you find yourself in a situation like this, consult Chapter 6 on apply and implement your custom function in Cython.

Indexing

Working with a sorted index provides a substantial speedup when there are many different values in each index. You may encounter the warning "PerformanceWarning: indexing past lexsort depth may impact performance." This is referring to the number of levels in an index that are sorted lexically or alphabetically.

When accessing an unsorted index, pandas has O(n) performance since it has to search the entire index for the index value as is demonstrated in Figure 7-1. When accessing a sorted index, pandas has O(log(n)) performance as it uses binary search to find the index value such as in Figure 7-2. When the index is unique, it uses a hash lookup which has O(1) performance as in Figure 7-3. This can make a huge difference when n, the number of values in the index, is very large which is why unique indexes result in the fastest performance. It's worth noting that just as is the case in Listing 7-2, a unique index cannot always be achieved, so the best performance we can get in those scenarios is with a sorted multi-index.

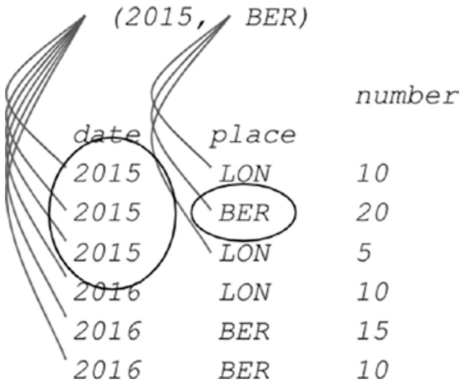


Figure 7-1. *Unsorted index access O(n)*

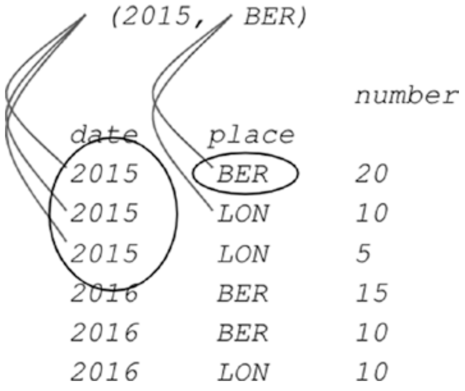
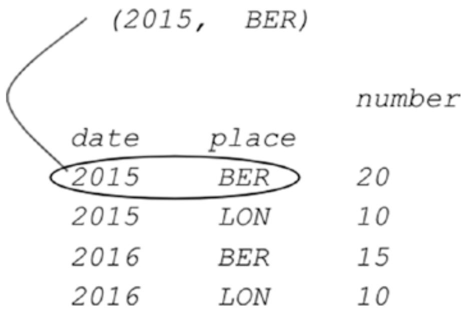


Figure 7-2. *Sorted index access O(log(n))*



```
(2015,  BER)

                        number
date     place
2015      BER           20
2015      LON           10
2016      BER           15
2016      LON           10
```

Figure 7-3. *Unique index access O(1)*

Avoiding groupby

So far, we've explored how to get the best performance when running a groupby operation. Sometimes, however, the most performant option is to not use a groupby at all. If you find yourself having to do a lot of groupby operations on your DataFrame, you may consider re-orienting your DataFrame so that you don't need to use groupby. Since groupby groups the data and then runs an aggregate function on each group of data, it is essentially doing a loop over the number of groups. Even though in the most performant case the groups are already pre-computed, the indexes are fast to access, and the looping is run at the C level, all of that still takes time. It's much more performant in pandas to run simple row-wise or column-wise operations.

Let's take a look at how we can reformat the DataFrame in Listing 7-3 so that we can avoid using groupby. If we keep the index columns where they are but instead break out the multiple values for each index across the row, we can do two things to optimize this sum by groups operation. The first thing this does is eliminate the groupby sum operation and turn it into a simple sum across the columns. The second thing this does is make the indexes unique. Note we are taking on some additional memory overhead by doing this as the gaps in the data will be filled with zeros. Integers, however, take up little space even in a very large DataFrame so the overall performance speedup is worth the additional memory usage.

Listing 7-3. Calculating total number of arrivals to each destination without using groupby

```
>> arrivals_by_destination
     number          0      1
     date    place
     2015    BER    20      0
     2015    LON    10      5
     2016    BER    15     10
     2016    LON    10      0
>> arrivals_by_destination["total"] = \
     arrivals_by_destination.sum()
```

Listing 7-2 is approximately 8 times slower than Listing 7-3, whereas Listing 7-1 is 25 times slower. This is why it's so important to carefully select the DataFrame format that best suits the operations you plan to perform on it. It can literally save you minutes of execution time.

We've looked at how to use groupby most efficiently, with pre-indexing and using an aggregate function directly off the groupby object instead of looping. We've also covered how to reformat your DataFrame so that you don't need to use a groupby and can use a more performant row- or column-wise operation. Unfortunately, there isn't one easy catch-all DataFrame format or groupby method that can be applied to all use cases. But you should now have an understanding and a repertoire of methods to help you solve your particular groupby problem in the most efficient and simplest way possible.

CHAPTER 8

Performance Improvements Beyond pandas

You may have heard another pandas user mention using eval and query to speed up evaluation of expressions in pandas. While use of these functions can speed up evaluation of expressions, it cannot do it without the help of a very important library: NumExpr. Use of these functions without installing NumExpr can actually cause a performance hit. In order to understand how NumExpr is able to speed up calculations however, we need to take a deep dive into the architecture of a computer.

Computer architecture

CPUs are broken up into multiple cores where each core has a dedicated cache. Each core evaluates one instruction at a time. These instructions are very basic compared to what you might see in a Python program. One line of Python is often broken up into many CPU instructions. Some examples include loading data such as storing an array value into a temporary variable when looping, jumping to a new instruction location such as when calling a function, and an evaluation expression such as adding two values together.

H. Stepanek, *Thinking in Pandas*, https://doi.org/10.1007/978-1-4842-5839-2_8

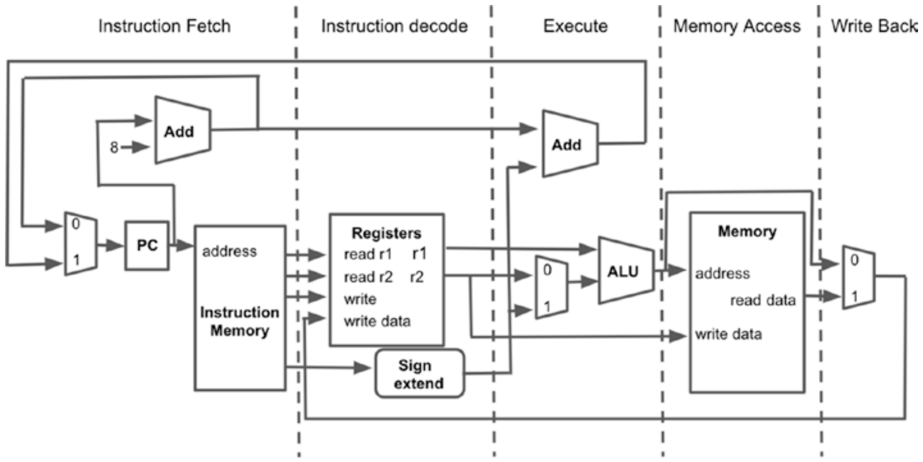


Figure 8-1. *Five-phase pipeline architecture*

In modern cores, the evaluation is split up into many phases. These multiple phases are called a pipeline, where each instruction evaluation is piped through a series of phases until the evaluation is complete. Modern Intel CPUs are typically broken up into about 15 pipeline phases. Figure 8-1 shows an example of a simple five-phase pipeline processor. First, the instruction is fetched typically from a dedicated instruction cache, or if not present, it must be fetched from a farther cache or main memory. Then the instruction is decoded. In the decoding phase, each instruction has a particular numerical code that decodes to a certain type of instruction and results in certain behavior so the decode phase is responsible for decoding the instruction and gathering the data from the registers (you can think of these as a very small dedicated memory cache) to pass to the execution phase. In the execution phase, the instruction is actually run; this may mean two values are added together, or if it's a load instruction, simply a memory address is passed to the next phase. Since an instruction may be a jump to a different memory location rather than a sequential location, part of the evaluation phase is also determining the next instruction to send through the pipeline. In the memory access phase, any data that needs to be loaded from memory into a register for an upcoming instruction is fetched and then loaded into a register in the writeback phase. This means that if you wish to add two values together, those values must first be loaded with a load instruction into two different registers before an add instruction can be run. So, the line of Python code in Listing 8-1 consists of three instructions inside the CPU.

Listing 8-1. Converting a line of Python code into pseudo-code CPU instructions

```
a = b + c        load b
                 load c
                 add a, b, c
```

While the CPU instructions in Listing 8-1 may look similar to Python bytecode, it's important to note that they are not the same. Remember that bytecode is run on the Python Virtual Machine, whereas CPU instructions are run on the CPU. While you can use the dis module (dis standing for disassembly) to output the bytecode and it may give you some idea of what the machine code might look like, it is not machine code. The Python Virtual Machine contains a giant switch statement that translates a bytecode instruction into a function call which then executes CPU instructions. So, while we may think of Python as being an interpreted language that runs bytecode instructions in a software virtual machine, the fact is at some point that add instruction makes its way to the CPU. Eventually that add becomes a series of CPU instructions that are shown in Listing 8-1.

It's very common for the memory access phase of the instruction pipeline to take longer than all the other pipeline phases. Rather than making all the other pipeline phases as slow as the memory access phase or inserting NOPs (commonly called no ops or no operations), other instructions not dependent on the data being loaded are used to fill the time. This enables the processor to keep evaluating instructions even though one phase of the instruction may take hundreds of cycles to complete. Compilers also play a part in keeping the processor busy during long instruction loads by re-ordering instructions so that instructions not dependent on a memory load occur between a memory load instruction and the next instruction that is dependent on that memory having loaded. In modern Intel processors, re-ordering can occur inside the CPU as well. Of course, sometimes there are no instructions to fill the gap and so NOPs are used as a last resort. This ensures that the core's instruction throughput is still as close to one instruction per clock cycle (phase) as possible so that you don't have to wait hundreds of cycles for a memory load to complete.

The takeaway here is in order to make efficient use of your CPU and get the highest instruction throughput possible you must make sure that your data is loaded before you use it and that you give it enough computations to do in between the data you are operating on and the next data you need.

So far, we've covered how the CPU operates on a low level to evaluate instructions and how it works to achieve the best performance possible; now let's take a deeper look at the memory access phase and why it's often a bottleneck. Figure 8-2 shows a typical cache architecture of a modern Intel CPU. Each core has a dedicated level 1 data, level 1 instruction, and a level 2 cache. All cores share the level 3 cache and main memory. Each of these caches is placed farther and farther from the core on the board itself. While core speeds are closely correlated to transistor size and speed, memory speeds are correlated to how physically close they are to the core on the board.

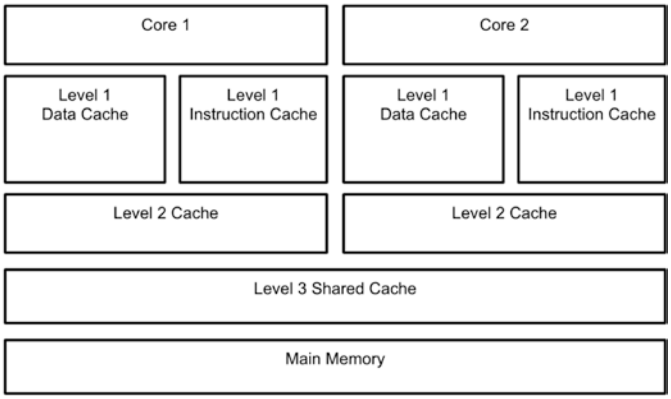


Figure 8-2. *Cache architecture*

When working with large data sets as is the expected use case in pandas, all of that data cannot be stored inside the cache. It typically takes about three clock cycles or instruction phases to access level 1 cache and at each level exponentially increases in latency. To access the level 3 cache, it takes about 21 clock cycles, and if the data we wish to load is not in any of the caches and it has to go all the way to main memory to load it, it takes anywhere from 150 to 400 clock cycles. At around 21 clock cycles, the performance hit incurred by accessing the level 3 cache will likely exceed the number of pipelines in our core. If we have to delay the instructions in our pipeline until the data is retrieved without re-ordering to pad the delay, that could stall our entire program for 21 clock cycles. 21 clock cycles of delay on a 4 GHz processor is about 5.25 ns. This might seem insignificant and it is if we only incur this delay a couple times in our program. However, keep in mind we are typically operating on megabytes of data in pandas, and since not all of that is going to fit in the caches, we will likely incur many performance hits like this. In fact, we're even more likely to incur larger performance hits all the way out to main memory if running an operation over the entire data set.

Caches are generally designed for the best performance of the average case. In software this means things like looping over arrays which are sequential data structures. Because of this, when they have to load something into the cache, they load sequential blocks of memory at a time called cache lines. This helps to offset the performance hit of loading something into the cache. The idea is typically programs operate on sequential memory, so by loading the memory that follows the memory the core needs right now, it will save needing to load that memory later.

In order to make the most effective use of caching, data that is located sequentially or close together in memory should be repeatedly referred to in a short time span. Sequential data will result in less cache loads. Repeatedly referring to the same data in a short time span will prevent new data from bumping the older data out of the cache and causing a cache miss that will require the same data to be loaded into the cache again. Arrays, as we learned in Chapter 3, are sequential data types, meaning the first element occurs at address A and the last element occurs at A plus the length of the array. When you create a bunch of objects in memory with many attributes that point to other objects and reference those attributes, each object has an address that is not sequential, and thus you will not be able to utilize your cache as you will be loading a bunch of different cache lines from a bunch of different memory locations. Figure 8-3 demonstrates these two types of memory accesses.

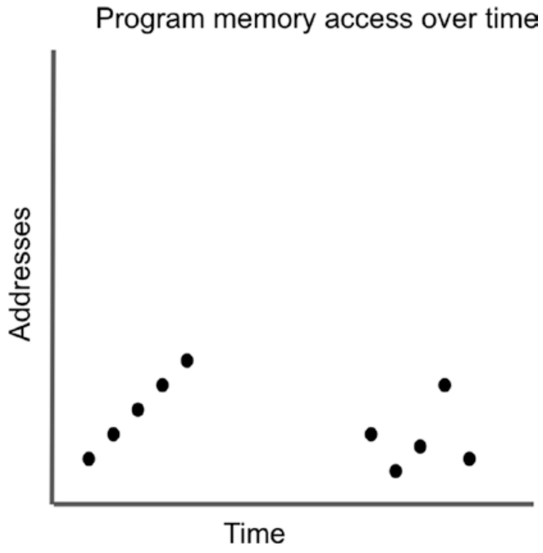


Figure 8-3. *Sequential memory access vs. object attribute access over time*

How NumExpr improves performance

NumExpr improves performance of pandas by running calculations on subsets of the pandas DataFrame that are the size of the cache. Take the example (A + B) ∗ 3 where A and B are pandas DataFrames. Without NumExpr, each row of A + B would be added together, stored into a temporary structure, and then multiplied by 3. With NumExpr, the first n number of rows that fit inside the cache are added together and multiplied by 3 before moving to the next n rows. In this way, NumExpr is able to reduce the number of memory loads and stores which we learned in the "Computer architecture" section were the bottlenecks of the CPU and thus a computer program. Listing 8-2 demonstrates this.

Note the cache in Listing 8-2 is three cache lines (enough to hold 64 rows of A, B, and C data). While this is much smaller than a real-world level 1 cache which can generally hold around 128 cache lines, it simplifies the example. This means that after computing the result of A + B for 64 rows, the result must be stored back out to memory in order to make room for the next 64 rows of A and B and their result. Note that by using NumExpr's method of running the computation over cache-sized chunks, we have reduced the number of loads and the number of stores. Also note the example in Listing 8-2 is written as in-order pseudo-code CPU instructions (i.e., these are not the actual instructions that would execute inside the core, and they would most likely be re-ordered in the real world in order to offset the memory load delay as discussed in the "Computer architecture" section).

Listing 8-2. CPU pseudo-code instructions during evaluation of a pandas expression with and without NumExpr

```
# NumExpr is not installed.          # NumExpr is installed.
C = ( A + B ) * 3                    C = pd.eval("( A + B ) * 3")

load A[0:64]                         load A[0:64]
load B[0:64]                         load B[0:64]
add C[0], A[0], B[0]                 add C[0], A[0], B[0]
                                     mult C[0], C[0], 3
add C[1], A[1], B[1]                 add C[1], A[1], B[1]
                                     mult C[1], C[1], 3

•
•
```

```
add C[63], A[63], B[63]           add C[63], A[63], B[63]
                                  mult C[63], C[63], 3
store C[0:64]                     store C[0:64]
load A[64:128]                    load A[64:128]
load B[64:128]                    load B[64:128]
add C[64], A[64], B[64]           add C[64], A[64], B[64]
                                  mult C[64], C[64], 3
add C[65], A[65], B[65]           add C[65], A[65], B[65]
                                  mult C[65], C[65], 3
.
.
add C[127], A[127], B[127]        add C[127], A[127], B[127]
                                  mult C[127], C[127], 3
store C[64:128]                   store C[64:128]
load C[0:64]
mult C[0], C[0], 3
mult C[1], C[1], 3
.
.
mult C[3], C[63], 3
store C[0:64]
load C[64:128]
mult C[64], C[64], 3
.
.
mult [127], C[127], 3
store C[64:128]
```

Note that in order to run an evaluation like this all at once on chunks of the pandas DataFrame(s), we must communicate to NumExpr the whole expression prior to computation. (A + B) ∗ 3 must be specified in such a way that NumExpr knows it can be combined together.

This is where query and eval come in.eval allows you to specify a complex expression as a string to signal NumExpr that it can be run on a chunk of the DataFrame(s) at a time. query is effectively another form of eval as it calls eval underneath.

Depending on the computation, the shape and size of the data, the operating system, and the hardware you are using, you may find that using NumExpr and eval actually results in a significant performance degradation. It's always good to run a performance comparison before blindly combining computations into an eval or query. NumExpr really only works well for computations that exceed the size of the level 3 cache. Typically, this is greater than 256,000 array elements. As we've seen with other pandas functions, it also requires the data type and computation be easily translatable into C. So, for example, datetimes will not yield a performance improvement as they cannot be evaluated in NumExpr. It's also worth noting that using NumExpr directly can be much more performant than using eval or query in pandas. Listing 8-3 demonstrates such an example.

Listing 8-3. An example where eval is slower than the typical pandas syntax with NumExpr

```
import pandas as pd
import numpy as np
import numexpr as ne

nrows, ncols = 1000000, 1
df1, df2, df3, df4 = [pd.DataFrame(
     np.random.randn(nrows, ncols)) for _ in range(4)]

# Calculate the sum using normal syntax.
df_sum1 = df1 + df2 + df3 + df4

# Calculate the sum using eval so that NumExpr optimizes it.
df_sum2 = pd.eval("df1 + df2 + df3 + df4")
```

```
# Calculate the sum using NumExpr directly.
a1, a2, a3, a4 = (
     df1.to_numpy(), df2.to_numpy(),
     df3.to_numpy(), df4.to_numpy()
)
df_sum3 = ne.evaluate("a1 + a2 + a3 + a4")
```

The calculation of df_sum1 is twice as fast as df_sum2. This is generally the opposite of what we would expect as df_sum2 is being calculated using NumExpr. However, if we use NumExpr directly instead of going through pandas eval, we find that df_sum3 is about four times faster than df_sum1. This is due to the slowdown incurred inside the pandas eval function itself. Inside eval, it wraps up the environment into a dictionary of local and global variables and makes them accessible to NumExpr. This includes converting the DataFrames to NumPy arrays. All of this takes a significant amount of overhead. So much so that it actually ends up being slower than not using eval and NumExpr. Very often, as is the case here, it's much faster to convert the DataFrames explicitly to NumPy arrays and call NumExpr explicitly on those converted DataFrames.

Now that we've looked at how NumExpr improves pandas performance of combined computations at the hardware level, let's look at how another one of NumPy and NumExpr's dependencies, BLAS, takes advantage of the hardware to optimize its computations.

BLAS and LAPACK

NumPy uses Basic Linear Algebra Subprograms (BLAS) underneath to implement very performant linear algebra operations such as matrix multiplication and vector addition. These subprograms are typically written in assembly—a very low-level and performant language that closely resembles CPU instructions. The Linear Algebra Package (LAPACK) provides routines for solving linear equations. It is typically written in Fortran and, just like NumPy, calls into BLAS underneath. There are many implementations of BLAS and LAPACK available such as the Netlib BLAS and LAPACK, OpenBLAS, Intel MKL, Atlas, BLIS, and so on, each with their own pros and cons. But, let's explore more deeply how BLAS improves the performance of pandas operations.

BLAS optimizes matrix operations by using Single Instruction Multiple Data (SIMD) instructions that make use of vector registers in the hardware. All CPUs have registers that hold the data that a CPU instruction needs to operate on. Vector registers are just a special type of those registers. They allow storage of multiple pieces of data in a single register, and when an operation is run on the data, it is run on each piece of data in the register at once. The advantage of SIMD instructions is that you can load a bunch of data into a register and run the same operation on it concurrently rather than having to run the same operation on each element consecutively. By using a SIMD instruction, we reduce the number of CPU clock cycles that it takes to complete the computation. For example, if a vector register is able to hold four data elements, then we have reduced the number of clock cycles from four to one. This means if you have a dot product of y and x as in Listing 8-4, then you can specify it as a series of SIMD instructions as in Listing 8-5. Note the y and x data is first loaded into the vector registers r1 and r2, and then the dot product is computed and stored in register r1.

Listing 8-4. Dot product

$$\begin{pmatrix} Y1 & Y2 & Y3 & Y4 \end{pmatrix} \begin{pmatrix} X1 \\ X2 \\ X3 \\ X4 \end{pmatrix} = Y1X1 + Y2X2 + Y3X3 + Y4X4$$

Listing 8-5. Dot product as SIMD pseudo-instruction

```
load vr2, Y1, Y2, Y3, Y4
load vr1, X1, X2, X3, X4
dot r1, vr2, vr1
```

We've left out one important detail here which is that typically data can only be loaded into vector registers if it is sequential in memory. This poses a slight problem for most complex vector operations which typically happen on rows of one matrix and columns of another matrix or vice versa. BLAS is the opposite of Python in that its arrays are column majored instead of row majored. BLAS also does not have two-dimensional arrays—they are stored as a single-dimensional array. Listing 8-6 shows an example of a Python array and how it would be stored in BLAS.

Listing 8-6. Comparison of a matrix representation in Python vs. BLAS

```
Python                    BLAS
y = [[1, 2], [3, 4]]      y = [1, 3, 2, 4]
y[row][col]               y[col * num_cols + row]
```

So, going back to the dot product example in Listing 8-5, because these arrays are both represented as a single-dimensional array, despite one being a bunch of rows and the other being a bunch of columns, they both have contiguous memory addresses and so they both can be loaded into the vector registers. This issue of consecutive memory addresses only comes into play when working with more complex matrices and more complex operations, so let's look at a more complex example.

There are many ways to perform a matrix multiply. One way, using dot product is shown in Figure 8-4. Taking the dot product of the first matrix's row with the second matrix's column will yield the value of a single element in the result of the matrix multiply.

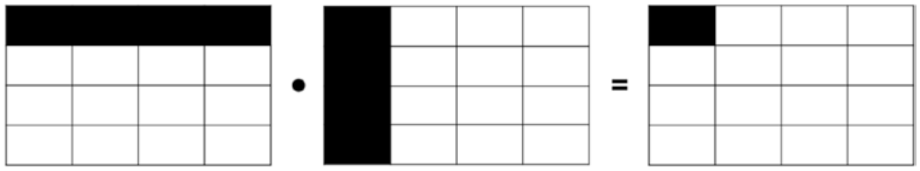


***Figure 8-4.** Matrix multiplication using dot product*

In Figure 8-4, we now run into a situation where loading a row of the first matrix into a vector register is not possible as the memory is not contiguous. Note, however, if we transpose the matrix so that the rows become the columns, then the memory is contiguous and we can load the row of the first matrix into a vector register.

But what happens if the matrix is very large? If we transposed a 1000 by 1000 matrix, it would not all fit in the cache and it would result in huge delays when it had to go out to main memory to write and read the data. BLAS optimizes for this by breaking up the matrix data into blocks just like NumExpr. An example is shown in Figure 8-5. By doing this, BLAS is able to keep the transposed matrix all in the cache and also reuse that placeholder transpose buffer on each block. This is advantageous not just for keeping the transposed buffer in the cache but also because it doesn't have to keep re-allocating a new buffer for each block. It simply overwrites the buffer of the previous block with the new transposed data for the current block.

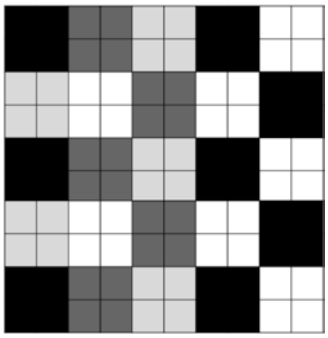


***Figure 8-5.** Breaking up a large matrix into blocks*

By breaking up the problem into blocks, BLAS is also able to take advantage of multiple cores. It can run each of the blocks on a different core which also saves time.

Another technique BLAS uses to speed up the computation is loop unrolling. This is when you convert a loop into a series of repeated instructions. Loop unrolling removes the need to predict branches and potentially incur a branch prediction penalty for mispredicting a branch. Recall that in the data pipeline instructions are loaded and processed potentially before the result of a conditional instruction check occurs. So, the hardware tries its best to correctly predict the result of that conditional and which branch will be taken before it knows for certain. Loop unrolling also avoids having to jump the instruction pointer to a new location in instruction memory which potentially avoids cache misses and having to go out to main memory to load instructions that aren't in the instruction cache. It also avoids the conditional instruction check at the beginning of each iteration which saves CPU cycles. By unrolling a loop, you may also be able to re-order computations such that the computations that use the same memory are placed close together, thus reducing register load instructions.

In summary, BLAS uses a lot of techniques to improve performance of linear algebra operations: SIMD instructions, blocking, loop unrolling, threading, and so on. There are also many implementations of BLAS available, and choosing a more performant option for certain types of pandas programs may have a huge impact.

If you find yourself doing a lot of linear algebra operations with pandas, you may consider switching to a more performant BLAS implementation. np.config_show() will show you what BLAS implementation NumPy is currently using. The Netlib BLAS implementation does not fully support multiple cores and tends to be much less performant than the alternatives. Other implementations like OpenBLAS fully support multiple cores and are open source and freely available. As of Anaconda 2.5 or later, Intel MKL is the default BLAS library and, though proprietary and large, is highly optimized and available for free.

By ensuring NumPy is using optimized dependencies NumExpr and BLAS, you can significantly improve performance of certain pandas operations. These libraries optimize operations to the particular hardware you are running on to ensure you are getting the best performance possible. But be mindful of when they will and will not boost performance. In the final chapter, we'll look at the future of pandas 1.0 and beyond.

CHAPTER 9

The Future of pandas

There are an increasing number of packages that are built on top of or compatible with pandas. Some of these like sklearn-pandas integrate with other packages like scikit-learn to utilize DataFrames for machine learning. Others, like Plotly, provide interactive plotting capabilities and online collaboration. pandas has recently made the push in the last couple years to branch out into other languages. There is now a pandas.js package and a ruby wrapper that enables ruby users to call into the Python pandas API. There is also a push to optimize data analysis at a more global scale using an up-and-coming LLVM called Weld. It takes an approach similar to NumExpr but on an even larger scale. The idea is to combine all the data analysis operations together lazily and only run them when an actual result is needed. This allows the operations to be optimized for parallel compute and loading of memory on a much grander scale.

pandas 1.0

The pandas community has been feverishly working on pandas 1.0, the first big upgrade since the initial release of pandas. It addresses a lot of the shortcomings in previous versions.

pandas 1.0 adds a new pandas specific NA type. This new type will make null values consistent across all types of columns. As you may recall from Chapter 4, NaNs in pandas 0.25 must be stored as floats; they cannot be Booleans, integers, or strings. Previously, it was not possible to load a column with NaNs in it as an integer type—you had to convert it to an integer after it was loaded. Now with pandas 1.0, it's possible to load a column with NaNs as an integer type. Listing 9-1 is the same example presented earlier in the text in Listing 4-15; only now it's making use of the new nullable integer type available in pandas 1.0. Note the memory usage of this new type takes up one more byte than is indicated by the data type. So, while the type is set to an Int16Dtype, each element actually takes up three bytes instead of two. The extra byte corresponds to a Boolean mask in the IntegerArray implementation which marks which values are NA.

Listing 9-1. Example of how pandas handles NaNs in the data in 1.0

```
>> data = io.StringIO(
     """
     id,age,height,weight
     129237,32,5.4,126
     123083,20,6.1,
     123087,25,4.5,unknown
     """
)
>> df = pd.read_csv(
     data.
     dtype={
         'id': np.int32,
         'age': np.int8,
         'height': np.float16,
         'weight': pd.Int16Dtype()},
     na_values=["unknown"],
     index_col=[0],
)
              age    height  weight
     id
     129237   32     5.4     126
```

```
    123083   20     6.1     <NA>
    123087   25     4.5     <NA>
>> df.memory_usage(deep=True)
    Index     24
    age        3
    height     6
    weight     9
>> df.dtypes
    age       int8
    height    float16
    weight    Int16
>> df.index.dtype
    dtype('int64')
```

With the introduction of pd.NA and the addition of nullable Boolean arrays and a dedicated string data type, the null values can be consistently represented using the same type across all data types. This seemingly simple change also improves subtle inconsistencies across the API due to having an inconsistent null type, for example, Categorical.min now returns the expected minimum value instead of NaN as shown in Listing 9-2.

Listing 9-2. Behavior change of pandas 1.0 when computing the minimum of a Categorical

```
>> pd.Categorical([1, 3, 5, np.nan], ordered=True).min()
NaN
>> pd.Categorical([1, 3, 5, pd.NA], ordered=True).min() # 1.0
1
```

The introduction of a dedicated string data type (StringDtype) is also huge in itself. In pandas 0.25, strings were stored as objects which take up a lot of space and do not guarantee data consistency because an object can hold any type. With the new explicit StringDtype, it will take up less space, guarantee consistency within the column, and also identify as a text type rather than lumping text values in with all values that are of the generic object container type. Listing 9-3 demonstrates how much less memory the new pandas string type uses. When using the new string type, each value takes up only 8 bytes, which is a huge decrease in memory compared to previous versions where each object value took up about 60 bytes.

Listing 9-3. Memory usage of the pandas 1.0 string type compared to using object in previous versions

```
>> data = io.StringIO(
     """
     id,name
     129237,Mary
     123083,Lacey
     123087,Bob
     """
)

>> # Load the data with pandas 0.25.3.
>> df = pd.read_csv(
     data,
     dtype={'id': np.int32},
     index_col=[0],
)
               name
     id
     129237   Mary
     123083   Lacy
     123083   Bob
>> df.memory_usage(deep=True)
     Index     24
     name      197
```

```
>> df.dtypes
    name         object

>> # Load the data with pandas 1.0.
>> df = pd.read_csv(
    data,
    dtype={
        'id': np.int32,
        'name': pd.StringDtype()},
        index_col=[0],
)
             name
    id
    129237   Mary
    123083   Lacy
    123083   Bob
>> df.memory_usage(deep=True)
    Index     24
    name      24
>> df.dtypes
    name      string
```

Nullable Booleans are also a huge win for pandas users. Previously, Boolean columns could not have a nullable state; only True and False were allowed. This meant users had to use an integer representation or an object to represent a Boolean with a third NaN state, but now they can use the pandas BooleanArray type.

The introduction of the new types in pandas, namely, the nullable Boolean, pandas NA type, and dedicated string type, yields marked improvements to the pandas type casting in pandas 1.0. Now, integers, Booleans, and strings will be recognized and stored as smaller data types even when they contain null values. This is a huge win for performance and saving memory on load. Note that while these new types exist and are inferred when creating pandas arrays, they are not inferred when creating DataFrames. You must explicitly specify the types for pandas to use them when creating pandas DataFrames. This is why in Listings 9-1 through 9-3 the new pandas types were explicitly specified when loading data using read_csv. If the types were not explicitly specified, they would be inferred to be the same types as in previous versions of pandas.

Rolling apply methods also now support an engine argument that gives the option of using Numba instead of Cython. Numba converts the custom apply function into optimized compiled machine code similar to Cython, but for data sets with millions of rows and custom functions that operate on NumPy ndarrays, the pandas team found Numba to produce more optimized code than Cython. It only makes sense to use Numba, of course, when you are running the calculation a lot of times over and over again since Numba has the overhead of compiling the first time it is used.

There has been a lot of work to clean up the Categorical data type in pandas 1.0. As you may recall from Chapter 2, the Categorical data type is used to hold metadata with a unique set of values. Deprecations within the API have been removed, previous operations on the data type that did not return back a Categorical now do, and there is improved handling of null values. There are also performance improvements, for example, now, all the values passed into searchsorted are converted to the same data type before running a comparison. Listing 9-4 shows an example of using searchsorted on a Categorical. This operation in pandas 1.0 is about 24 times faster than in previous versions.

Listing 9-4. Using searchsorted on Categorical

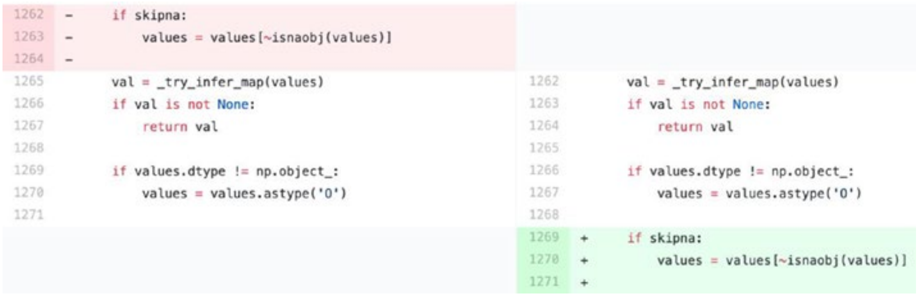
```
import pandas as pd
metadata = pd.Categorical(
     ['Mary'] * 100000 + ['Boby'] * 100000 + ['Joe'] * 100000
)
metadata.searchsorted(['Mary', 'Joe'])
```

There have also been a lot of refactors and bug fixes made to groupby. This used to be a complex bit of code to look at with a fair number of bugs, but there have been many improvements in pandas 1.0 including improved handling of null values, offering a selection by column name when axis is one, allowing multiple custom aggregate functions for the same column to match series groupby behavior, and many more.

The support of load and dump options for reading CSV data in pandas far exceeds options for other loaders. While supporting so many options leads to complicated code for developers, it is very nice for users. Some of the loaders have a nice balance of options, but some fall short in load and dump capabilities that could lead to performance speedup for users. As we saw in Chapter 3, read_sql is missing the ability to specify data types during load which can be fairly critical for performance. The CSV loader on the other hand has so many options, some of which can result in a performance slowdown if you aren't careful. A lot of work has been done to address this and standardize the options for input and output data methods in pandas 1.0. For example, both read_json and read_csv are now able to parse and interpret Infinity, –Infinity, +Infinity, and NaNs as expected. In previous versions, read_json didn't handle NaNs or Infinity strings, and read_csv didn't cast Infinity strings as floats. The usecols parameter in read_excel has also been standardized to behave more like read_csv's usecols parameter. Previously, usecols was allowed to be a single integer value, whereas now it's a list of integer values just like read_csv.

There have been a lot of other subtle performance improvements to pandas 1.0 as well. We'll look at a couple of them here just to give you some idea of what methods are being used to improve performance.

A regression in performance of the infer_type method was fixed in pandas 1.0. An if statement was moved down in the implementation to avoid a performance slowdown introduced by converting data types to objects when running an isnaobj comparison prematurely as shown in Figure 9-1.



```
-     if skipna:
-         values = values[~isnaobj(values)]
-
      val = _try_infer_map(values)
      if val is not None:
          return val

      if values.dtype != np.object_:
          values = values.astype('O')

```

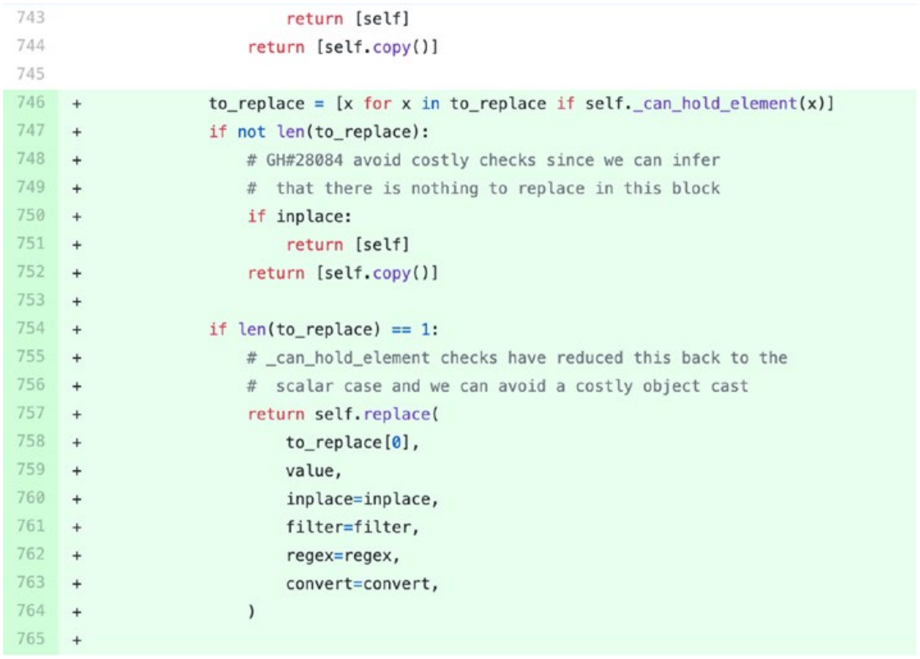
```
      val = _try_infer_map(values)
      if val is not None:
          return val

      if values.dtype != np.object_:
          values = values.astype('O')

+     if skipna:
+         values = values[~isnaobj(values)]
+
```

Figure 9-1. *A diff of the performance fix for infer_type*

Another performance fix was made to the replace method which is used to replace values with a different value. Here, some additional code was inserted above the original to take advantage of some early exit conditions. If the list of values to replace is empty, simply return the original values or a copy of the original values if inplace is False. If there is only one valid value, replace that single value with the new value. The values were also converted to a list of valid values as opposed to being left as a list of values that may or may not even be legal for the given column. Note while it is not explicitly shown in Figure 9-2, the new to_replace list was also used in the final replace call. By doing so, this reduced the number of replaces that were needed and improved the overall performance over large data sets where several columns did not contain any of the values that were to be replaced.

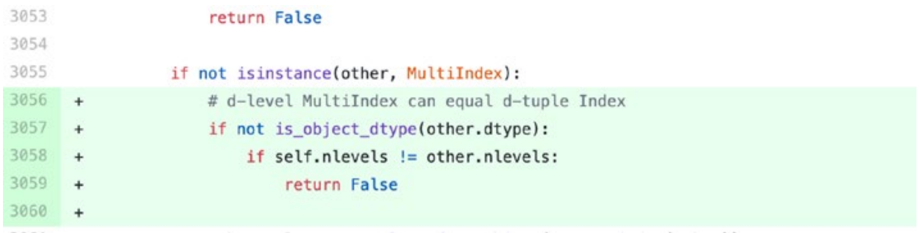


```
                    return [self]
                return [self.copy()]

+           to_replace = [x for x in to_replace if self._can_hold_element(x)]
+           if not len(to_replace):
+               # GH#28084 avoid costly checks since we can infer
+               #  that there is nothing to replace in this block
+               if inplace:
+                   return [self]
+               return [self.copy()]
+
+           if len(to_replace) == 1:
+               # _can_hold_element checks have reduced this back to the
+               #  scalar case and we can avoid a costly object cast
+               return self.replace(
+                   to_replace[0],
+                   value,
+                   inplace=inplace,
+                   filter=filter,
+                   regex=regex,
+                   convert=convert,
+               )
+
```

Figure 9-2. *Additional code inserted above original replace logic to take advantage of early exit conditions*

The performance of comparing indexes for equality was also improved. This is another performance improvement that was made by adding the early exit condition shown in Figure 9-3. If the dimensions are not equivalent, then the indexes are determined to be not equivalent and the MultiIndex equality check exits early.

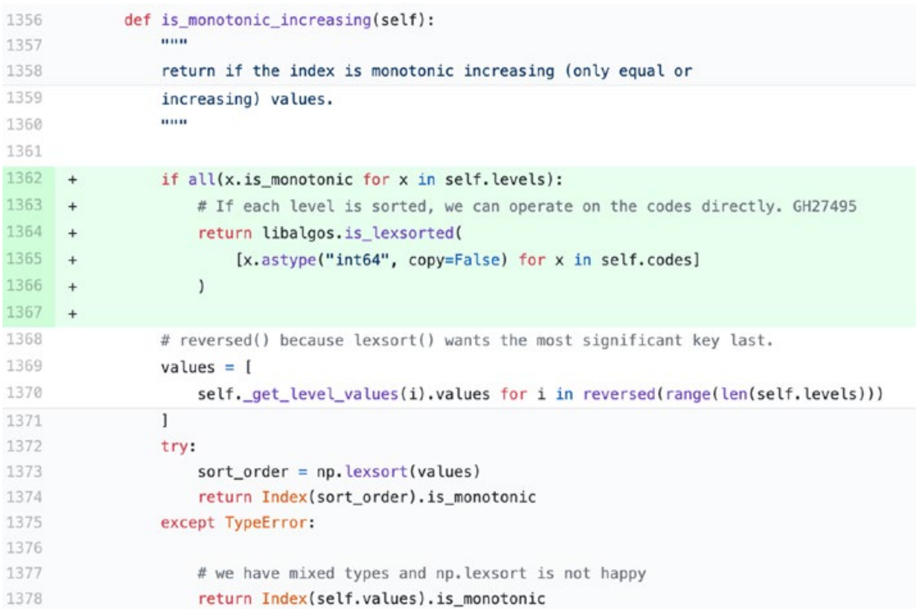


```
                return False

            if not isinstance(other, MultiIndex):
+               # d-level MultiIndex can equal d-tuple Index
+               if not is_object_dtype(other.dtype):
+                   if self.nlevels != other.nlevels:
+                       return False
+
```

Figure 9-3. *Additional code inserted into is MultiIndex check to take advantage of early exit conditions*

Improvements were also made to the is_monotonic check on an index. Previously, the result relied on the generation of cached values, but when the levels of an index are already individually sorted, is_lexsorted of codes can be used to determine monotonicity instead. Recall from Chapter 3 that levels are the unique list of values within the index and codes hold the position of those values within the index. codes represent each value as an integer, and this integer is the index location of the value in the list of values. So, putting that all together, is_lexsorted is an O(n) algorithm which is operating on integers that represent the values, whereas the previous implementation was always operating on the index values directly in an O(nlog(n)) check, first sorting them using NumPy's lexsort and then determining based on the sort result whether any of them were not in monotonic order. By using the already sorted integer representations of the values, we are able to more quickly determine monotonicity. Figure 9-4 shows the change in bold.



```
    def is_monotonic_increasing(self):
        """
        return if the index is monotonic increasing (only equal or
        increasing) values.
        """

+       if all(x.is_monotonic for x in self.levels):
+           # If each level is sorted, we can operate on the codes directly. GH27495
+           return libalgos.is_lexsorted(
+               [x.astype("int64", copy=False) for x in self.codes]
+           )
+
        # reversed() because lexsort() wants the most significant key last.
        values = [
            self._get_level_values(i).values for i in reversed(range(len(self.levels)))
        ]
        try:
            sort_order = np.lexsort(values)
            return Index(sort_order).is_monotonic
        except TypeError:

            # we have mixed types and np.lexsort is not happy
            return Index(self.values).is_monotonic
```

Figure 9-4. *Additional code inserted to improve performance of is_monotonic check*

There is talk from the pandas team of removing the inplace option from all pandas methods, and for that reason, they have generally recommended to not use it. The inplace option, contrary to what its name suggests, does not always operate inplace without duplicating memory. This typically happens as a result of pandas type inference where the operation results in a data type change, and thus the data has to be reconstructed with the new type. Listing 9-5 illustrates this example. When the NaN value is replaced with 0.0, the type is still a float and the value can be directly replaced in the NumPy array without having to create a new one and copy memory. When 0.0 is replaced with the string null, the float64 type cannot hold a string and so the NumPy array must be rebuilt and the memory must be copied into a new array of type object. Both these operations were specified with inplace=True, yet the latter resulted in a memory copy because the type of the underlying data structure had to change.

Listing 9-5. An example of inplace=True copying rather than modifying the data

```
>> data = pd.DataFrame({"size":[np.nan,1.0,3.5]})
>> data.dtypes
    size     float64
>> id(data._data.blocks[0].values)
    4757583472
>> data.fillna(0.0, inplace=True)
>> data.dtypes
    size     float64
>> id(data._data.blocks[0].values)
    4757583472
>> data.replace(0.0, "null", inplace=True)
>> data.dtypes
    size     object
>> id(data._data.blocks[0].values)
    4757572464
```

While attempting to not duplicate data is arguably better performance than always duplicating it, in an effort to shift users with as little pain as possible over to a world where inplace does not exist, the pandas team has recommended to not use inplace=True.

In the last couple years, the original author of pandas, Wes McKinney, has started working on a new project called Apache Arrow[1] which he hopes will one day be the back end of pandas. It aims to fix a lot of the core issues of pandas including reducing the memory overhead and enabling lazy evaluations.

Conclusion

Because of pandas' diverse user base, it supports many different options and many different methods for doing the same thing. pandas' API has a large and ever-expanding set of features and options which can be incredibly overwhelming and often lead users to implement things in a suboptimal way. It's a difficult decision to make: limit the number of features and options such that users can't do the wrong thing or provide a set of features so that users can find a way to do whatever they want. pandas has certainly erred on the side of the latter which makes it a very powerful tool and applicable to many different types of big data problems. And for those users who don't care if their program takes a minute or an hour, it's not an issue that they have used a suboptimal implementation. However, for users that do, it can be difficult to reason about and understand. Hopefully, this book has left you with a better understanding of how pandas works underneath and an intuition for which method to use during certain scenarios.

[1] https://arrow.apache.org/

In general, we've covered a few basic rules-to-code-by throughout this book which you should keep in mind as you implement your future pandas projects:

- Normalize your data at the same time as you load it if possible
- Explicitly specify your data types
- Avoid looping in Python
- Carefully select a DataFrame orientation that optimizes analysis
- Avoid operations that duplicate data
- Take advantage of Cython or faster custom implementations as needed

By following these basic rules-to-code-by and now with an intuition for how a given pandas operation will perform underneath, you should be able to select the most optimal implementation for your next pandas project.

APPENDIX

Useful Reference Tables

Table A-1. *Conversion between NumPy and C types*[1]

NumPy type	C type	Description
`np.bool`	Bool	Boolean (True or False) stored as a byte
`np.byte`	signed char	Platform-defined
`np.ubyte`	unsigned char	Platform-defined
`np.short`	Short	Platform-defined
`np.ushort`	unsigned short	Platform-defined
`np.intc`	Int	Platform-defined
`np.uintc`	unsigned int	Platform-defined
`np.int_`	Long	Platform-defined
`np.uint`	unsigned long	Platform-defined
`np.longlong`	long long	Platform-defined
`np.ulonglong`	unsigned long long	Platform-defined

(*continued*)

[1]https://docs.scipy.org/doc/numpy/user/basics.types.html

Table A-1. (*continued*)

NumPy type	C type	Description
`np.half` / `np.float16`	N/A	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`np.single`	Float	Platform-defined single-precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
`np.double`	Double	Platform-defined double-precision float: typically sign bit, 11 bits exponent, 52 bits mantissa
`np.longdouble`	long double	Platform-defined extended-precision float
`np.csingle`	float complex	Complex number, represented by two single-precision floats (real and imaginary components)
`np.cdouble`	double complex	Complex number, represented by two double-precision floats (real and imaginary components)
`np.clongdouble`	long double complex	Complex number, represented by two extended-precision floats (real and imaginary components)
`np.int8`	int8_t	Byte (–128 to 127)
`np.int16`	int16_t	Integer (–32768 to 32767)
`np.int32`	int32_t	Integer (–2147483648 to 2147483647)
`np.int64`	int64_t	Integer (–9223372036854775808 to 9223372036854775807)
`np.uint8`	uint8_t	Unsigned integer (0 to 255)

(*continued*)

Table A-1. *(continued)*

NumPy type	C type	Description
np.uint16	uint16_t	Unsigned integer (0 to 65535)
np.uint32	uint32_t	Unsigned integer (0 to 4294967295)
np.uint64	uint64_t	Unsigned integer (0 to 18446744073709551615)
np.intp	intptr_t	Integer used for indexing, typically the same as ssize_t
np.uintp	uintptr_t	Integer large enough to hold a pointer
np.float32	Float	Platform-defined single-precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
np.float64 / np.float_	Double	Note that this matches the precision of the built-in Python *float*
np.complex64	float complex	Complex number, represented by two 32-bit floats (real and imaginary components)
np.complex128 / np.complex_	double complex	Note that this matches the precision of the built-in Python *complex*

Table A-2. *Common ufuncs for NumPy*[2]

Ufuncs	Description
add(x1, x2, /[, out, where, casting, order, ...])	Add arguments element-wise
subtract(x1, x2, /[, out, where, casting, ...])	Subtract arguments, element-wise
multiply(x1, x2, /[, out, where, casting, ...])	Multiply arguments element-wise

(continued)

[2]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Table A-2. (*continued*)

Ufuncs	Description
divide(x1, x2, /[, out, where, casting, …])	Return a true division of the inputs, element-wise
logaddexp(x1, x2, /[, out, where, casting, …])	Logarithm of the sum of exponentiations of the inputs
logaddexp2(x1, x2, /[, out, where, casting, …])	Logarithm of the sum of exponentiations of the inputs in base 2
true_divide(x1, x2, /[, out, where, …])	Return a true division of the inputs, element-wise
floor_divide(x1, x2, /[, out, where, …])	Return the largest integer smaller or equal to the division of the inputs
negative(x, /[, out, where, casting, order, …])	Numerical negative, element-wise
positive(x, /[, out, where, casting, order, …])	Numerical positive, element-wise
power(x1, x2, /[, out, where, casting, …])	First array elements raised to powers from second array, element-wise
remainder(x1, x2, /[, out, where, casting, …])	Return element-wise remainder of division
mod(x1, x2, /[, out, where, casting, order, …])	Return element-wise remainder of division
fmod(x1, x2, /[, out, where, casting, …])	Return element-wise remainder of division

(*continued*)

Table A-2. (*continued*)

Ufuncs	Description
divmod(x1, x2[, out1, out2], / [[, out, …])	Return element-wise quotient and remainder simultaneously
absolute(x, /[, out, where, casting, order, …])	Calculate the absolute value element-wise
fabs(x, /[, out, where, casting, order, …])	Compute the absolute value element-wise
rint(x, /[, out, where, casting, order, …])	Round elements of the array to the nearest integer
sign(x, /[, out, where, casting, order, …])	Return an element-wise indication of the sign of a number
heaviside(x1, x2, /[, out, where, casting, …])	Compute the Heaviside step function
conj(x, /[, out, where, casting, order, …])	Return the complex conjugate, element-wise
conjugate(x, /[, out, where, casting, …])	Return the complex conjugate, element-wise
exp(x, /[, out, where, casting, order, …])	Calculate the exponential of all elements in the input array
exp2(x, /[, out, where, casting, order, …])	Calculate 2**p for all p in the input array
log(x, /[, out, where, casting, order, …])	Natural logarithm, element-wise
log2(x, /[, out, where, casting, order, …])	Base 2 logarithm of x

(*continued*)

Table A-2. (*continued*)

Ufuncs	Description
log10(x, /[, out, where, casting, order, …])	Return the base 10 logarithm of the input array, element-wise
expm1(x, /[, out, where, casting, order, …])	Calculate exp(x) - 1 for all elements in the array
log1p(x, /[, out, where, casting, order, …])	Return the natural logarithm of one plus the input array, element-wise
sqrt(x, /[, out, where, casting, order, …])	Return the non-negative square root of an array, element-wise
square(x, /[, out, where, casting, order, …])	Return the element-wise square of the input
cbrt(x, /[, out, where, casting, order, …])	Return the cube root of an array, element-wise
reciprocal(x, /[, out, where, casting, …])	Return the reciprocal of the argument, element-wise
gcd(x1, x2, /[, out, where, casting, order, …])	Return the greatest common divisor of |x1| and |x2|
lcm(x1, x2, /[, out, where, casting, order, …])	Return the lowest common multiple of |x1| and |x2|

Table A-3. *Values that are automatically converted to NaNs by read_csv*[3]

‘’
NULL
-1.#IND
NaN
-NaN
#N/A
NA
#N/A N/A
n/a
#NA
1.#QNan
-1.#QNan
NaN
-NaN
Null
N/A
1.#IND

[3]https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats

***Table A-4.** Built-in DataFrame computation methods*

Computation	Description
DataFrame.abs(self)	Return a Series/DataFrame with absolute numeric value of each element
DataFrame.all(self[, axis, bool_only, …])	Return whether all elements are True, potentially over an axis
DataFrame.any(self[, axis, bool_only, …])	Return whether any element is True, potentially over an axis
DataFrame.clip(self[, lower, upper, axis])	Trim values at input threshold(s)
DataFrame.corr(self[, method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values
DataFrame.corrwith(self, other[, axis, …])	Compute pairwise correlation
DataFrame.count(self[, axis, level, …])	Count non-NA cells for each column or row
DataFrame.cov(self[, min_periods])	Compute pairwise covariance of columns, excluding NA/null values
DataFrame.cummax(self[, axis, skipna])	Return cumulative maximum over a DataFrame or Series axis
DataFrame.cummin(self[, axis, skipna])	Return cumulative minimum over a DataFrame or Series axis
DataFrame.cumprod(self[, axis, skipna])	Return cumulative product over a DataFrame or Series axis
DataFrame.cumsum(self[, axis, skipna])	Return cumulative sum over a DataFrame or Series axis
DataFrame.describe(self[, percentiles, …])	Generate descriptive statistics

(*continued*)

Table A-4. (*continued*)

Computation	Description
DataFrame.diff(self[, periods, axis])	First discrete difference of element
DataFrame.eval(self, expr[, inplace])	Evaluate a string describing operations on DataFrame columns
DataFrame.kurt(self[, axis, skipna, level, …])	Return unbiased kurtosis over requested axis
DataFrame.kurtosis(self[, axis, skipna, …])	Return unbiased kurtosis over requested axis
DataFrame.mad(self[, axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
DataFrame.max(self[, axis, skipna, level, …])	Return the maximum of the values for the requested axis
DataFrame.mean(self[, axis, skipna, level, …])	Return the mean of the values for the requested axis
DataFrame.median(self[, axis, skipna, …])	Return the median of the values for the requested axis
DataFrame.min(self[, axis, skipna, level, …])	Return the minimum of the values for the requested axis
DataFrame.mode(self[, axis, numeric_only, …])	Get the mode(s) of each element along the selected axis
DataFrame.pct_change(self[, periods, …])	Percentage change between the current and a prior element
DataFrame.prod(self[, axis, skipna, level, …])	Return the product of the values for the requested axis

(*continued*)

Table A-4. (*continued*)

Computation	**Description**
DataFrame.product(self[, axis, skipna, …])	Return the product of the values for the requested axis
DataFrame.quantile(self[, q, axis, …])	Return values at the given quantile over requested axis
DataFrame.rank(self[, axis])	Compute numerical data ranks (1 through n) along axis
DataFrame.round(self[, decimals])	Round a DataFrame to a variable number of decimal places
DataFrame.sem(self[, axis, skipna, level, …])	Return unbiased standard error of the mean over requested axis
DataFrame.skew(self[, axis, skipna, level, …])	Return unbiased skew over requested axis
DataFrame.sum(self[, axis, skipna, level, …])	Return the sum of the values for the requested axis
DataFrame.std(self[, axis, skipna, level, …])	Return sample standard deviation over requested axis
DataFrame.var(self[, axis, skipna, level, …])	Return unbiased variance over requested axis
DataFrame.nunique(self[, axis, dropna])	Count distinct observations over requested axis

Index