Melting, Transposing, and Stacking Data

We have shown a lot of ways to manipulate a dataframe. But we are not done yet. This chapter will show some of the more complicated operations you can do to a dataframe to bend the data to your will. You probably will not use these operations often but will be grateful they are around when needed.

Melting Data

Another transformation we can do to data is "melt" it. Before looking at the method to melt data, let's discuss the structure of data. Two ways to organize the same data are "wide" (also called *stacked* or *record* form) and "long" (sometimes called *tidy* form) data. (Note that this differs from "big data", which refers to the amount of data.)

An OLAP database is an analytical database optimized for reporting. In OLAP terms, there is a notion of a *fact* and a *dimension*. A fact is a value that is measured and reported on, and a dimension is a value that describes the conditions of the fact. There are often multiple dimensions for a fact. In a sales scenario, typical facts would be the number of sales of an item and the cost. The dimensions might include the store where the item was sold, the date, and the customer.

The dimensions can then be *sliced* to explore the data. We might want to view sales by store. A dimension may be hierarchical; a store could have a region, zip code, or state, and we could view sales by any of those dimensions.

Here is data that tracks students' ages and scores. The test columns are fact columns, and the other columns are dimensions:

name	age	test1	test2	teacher
Adam	15	95	80	Ashby
Bob	16	81	82	Ashby
Dave Fred	16 15	89	84 88	Jones Jones

The scores data is in a *wide format*. This is in contrast to a *long format*, where each row contains a single fact (with perhaps other variables describing the dimensions). If we consider test scores to be a fact, this wide-format has more than one fact in a row. Hence, it is wide.

Often, tools require that data be stored in a long format and only have one fact per row. This format is *denormalized* and repeats many of the dimensions but may make analysis easier.

One long version of our scores looks like this (note that we dropped teacher information):

name	age	test	score
Adam	15	test1	95
Bob	16	test1	81
Dave	16	test1	89
Fred	15	test1	NaN
Adam	15	test2	80
Bob	16	test2	82
Dave	16	test2	84
Fred	15	test2	88

Melting Data

scores

	name	age	test1	test2	teacher
0	Adam	15	95.00	80	Ashby
1	Bob	16	81.00	82	Ashby
2	Suzy	16	89.00	94	Jones
3	Fred	15	nan	88	Jones

```
pd.melt(scores, id_vars=['name', 'age'],
        value_vars=['test1', 'test2'])
```

	name	age	variable	value
0	Adam	15	test1	95.00
1	Bob	16	test1	81.00
2	Suzy	16	test1	89.00
3	Fred	15	test1	nan
4	Adam	15	test2	80.00
5	Bob	16	test2	82.00
6	Suzy	16	test2	94.00
7	Fred	15	test2	88.00

Melting data with pandas. Melting allows you to stack columns on top of each other.

Let's show how to convert wide data to long data. We will start by creating a dataframe with scores:

```
>>> import io
>>> data = '''name,age,test1,test2,teacher
... Adam,15,95.0,80,Ashby
... Bob,16,81.0,82,Ashby
... Dave,16,89.0,84,Jones
... Fred,15,,88,Jones'''
>>> scores = pd.read_csv(io.StringIO(data), dtype_backend='pyarrow')

>>> print(scores)
   name  age  test1  test2 teacher
0  Adam   15   95.0     80   Ashby
1   Bob   16   81.0     82   Ashby
2  Dave   16   89.0     84   Jones
3  Fred   15   <NA>     88   Jones
```

Right now, the score for each test is in its own column. If we wanted to calculate the average of all of the tests, it would require some work to pull out

all of the test score columns, stack them, and calculate the mean. Let's melt the data and put it into long form. Below, we keep the name and age as dimensions, and pull out the test scores as facts:

```
>>> print(scores.melt(id_vars=['name', 'age'],
...         value_vars=['test1', 'test2']))
   name  age variable  value
0  Adam   15    test1   95.0
1   Bob   16    test1   81.0
2  Dave   16    test1   89.0
3  Fred   15    test1   <NA>
4  Adam   15    test2   80.0
5   Bob   16    test2   82.0
6  Dave   16    test2   84.0
7  Fred   15    test2   88.0
```

Using techniques that we have learned, we can accomplish this by building up a chain. But the `.melt` method is a friendly convenience method. Here is the hand-rolled non-melt version:

```
>>> print(scores
...  .groupby(['name', 'age'])
...  .apply(lambda g: pd.concat([
...     g[['test1']].rename(columns={'test1':'val'}).assign(var='test1'),
...     g[['test2']].rename(columns={'test2':'val'}).assign(var='test2')]))
...  .reset_index()
...  .drop(columns='level_2')
... )
   name  age   val    var
0  Adam   15  95.0  test1
1  Adam   15  80.0  test2
2   Bob   16  81.0  test1
3   Bob   16  82.0  test2
4  Dave   16  89.0  test1
5  Dave   16  84.0  test2
6  Fred   15  <NA>  test1
7  Fred   15  88.0  test2
```

As you can see, the melt version is much easier to create.

If we want to change the description of the fact column to a more descriptive name, pass that as the `var_name` parameter. We can change the name of the value of the column (it defaults to `value`) by providing a `value_name` parameter. Here, we change the description to *test* and the value to *score*:

```
>>> print(scores.melt(id_vars=['name', 'age'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score'))
   name  age   test  score
0  Adam   15  test1   95.0
1   Bob   16  test1   81.0
2  Dave   16  test1   89.0
3  Fred   15  test1   <NA>
4  Adam   15  test2   80.0
5   Bob   16  test2   82.0
6  Dave   16  test2   84.0
7  Fred   15  test2   88.0
```

If we want to preserve the teacher information, we would need to include it in the `id_vars` parameter:

```
>>> print(scores.melt(id_vars=['name', 'age', 'teacher'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score'))
   name  age teacher   test  score
0  Adam   15   Ashby  test1   95.0
1   Bob   16   Ashby  test1   81.0
2  Dave   16   Jones  test1   89.0
3  Fred   15   Jones  test1   <NA>
4  Adam   15   Ashby  test2   80.0
5   Bob   16   Ashby  test2   82.0
6  Dave   16   Jones  test2   84.0
7  Fred   15   Jones  test2   88.0
```

> Note
>
> Long data is also referred to as *tidy* data. See the Tidy Data paper[1] by Hadley Wickham.

Un-melting Data

We can go from a long to a wide format using a pivot table. Here is our melted data from the previous section:

```
>>> melted = scores.melt(id_vars=['name', 'age', 'teacher'],
...         value_vars=['test1', 'test2'],
...         var_name='test', value_name='score')
```

```
>>> print(melted)
   name  age teacher   test  score
0  Adam   15   Ashby  test1   95.0
1   Bob   16   Ashby  test1   81.0
2  Dave   16   Jones  test1   89.0
3  Fred   15   Jones  test1   <NA>
4  Adam   15   Ashby  test2   80.0
5   Bob   16   Ashby  test2   82.0
6  Dave   16   Jones  test2   84.0
7  Fred   15   Jones  test2   88.0
```

It is a little more involved going in the reverse direction because we will put the id variables that we kept from the original data in a hierarchical index. I generally flatten hierarchical indices with the `.reset_index` method. You can use `.pivot_table` or `.groupby` to do this:

```
>>> print(melted
...  .pivot_table(index=['name','age', 'teacher'],
...               columns='test', values='score')
...  .reset_index())
test  name  age teacher  test1  test2
0     Adam   15   Ashby   95.0   80.0
1      Bob   16   Ashby   81.0   82.0
2     Dave   16   Jones   89.0   84.0
3     Fred   15   Jones   <NA>   88.0

>>> print(melted
...  .groupby(['name', 'age', 'teacher', 'test'])
...  .score
...  .mean()
...  .unstack()
...  .reset_index()
... )
test  name  age teacher  test1  test2
0     Adam   15   Ashby   95.0   80.0
1      Bob   16   Ashby   81.0   82.0
2     Dave   16   Jones   89.0   84.0
3     Fred   15   Jones   <NA>   88.0
```

Undoing Melting

```
melted
```

	name	age	variable	value
0	Adam	15	test1	95.00
1	Bob	16	test1	81.00
2	Suzy	16	test1	89.00
3	Fred	15	test1	nan
4	Adam	15	test2	80.00
5	Bob	16	test2	82.00
6	Suzy	16	test2	94.00
7	Fred	15	test2	88.00

```
(melted
 .pivot_table(index=['name', 'age'],
              columns='variable', values='value')
 .reset_index()
)
```

	name	age	test1	test2
0	Adam	15	95.00	80.00
1	Bob	16	81.00	82.00
2	Fred	15	nan	88.00
3	Suzy	16	89.00	94.00

Unmelting data with pandas. By pivoting the data, you can specify the label column (`columns`) for the stacked columns (`values`).

Pulling Out Categorical Values into Columns

Suppose we wanted to create a dataframe where every column was the scores for a teacher. This might be useful if we wanted to plot a histogram of the scores for each teacher. We can use the `.pivot` method to do this. The `.pivot` method differs from the `.pivot_table` method in that it requires the index and column values to be unique. The `.pivot_table` method is more flexible and can handle aggregation of duplicate values.

Here is the `scores` dataframe.

```
>>> print(scores)
   name  age  test1  test2 teacher
0  Adam   15   95.0     80   Ashby
```

```
1   Bob   16   81.0     82   Ashby
2  Dave   16   89.0     84   Jones
3  Fred   15   <NA>     88   Jones
```

We can create a new dataframe where the columns are the teachers, and the values are the scores. We will use the `.pivot` method and pass the `teacher` column as the columns and the test score columns as the values:

```
>>> print(scores
...  .pivot(columns='teacher', values=['test1', 'test2'])
... )
        test1       test2
teacher Ashby Jones Ashby Jones
0        95.0   NaN    80   NaN
1        81.0   NaN    82   NaN
2         NaN  89.0   NaN    84
3         NaN  <NA>   NaN    88
```

Note that this returns a sparse dataframe. Also note that as of pandas 2.2, this returns `object` columns [2]. We will address this using the `.apply` method to pack each column and convert the types.

```
>>> print(scores
...  .pivot(columns='teacher', values=['test1', 'test2'])
...  .apply(lambda ser: ser
...             [~ser.isna()]
...             .reset_index(drop=True)
...             .astype('int64[pyarrow]')
...         )
... )
        test1       test2
teacher Ashby Jones Ashby Jones
0          95    89    80    84
1          81  <NA>    82    88
```

If we want to combine all of the scores for each teacher, we can use the same technique with the pivoted data.

```
>>> def pack_as_int(ser):
...     return (ser[~ser.isna()].reset_index(drop=True)
...              .astype('int64[pyarrow]'))

>>> print(melted
...  .pivot(columns='teacher', values='score')
...  .apply(pack_as_int)
... )
```

```
teacher  Ashby  Jones
0           95     89
1           81     84
2           80     88
3           82   <NA>
```

Transposing Data

We have been exploring reshaping data. We have already seen and used a common method to reshape data, the `.transpose` method or the `.T` property. Remember, this flips rows and columns.

I find that I use transposition mostly in two places:

- Viewing more data in Jupyter
- Swapping axis for plotting

Transposition often works for viewing more data because pandas uses numeric index values by default. When the numeric index goes into the column, it takes up less horizontal space, and you can see more data without having to scroll around.

I have some thoughts on viewing data. Often, when I teach, a student will ask how to turn off the default behavior of pandas in Jupyter to only show a limited number of rows and columns. (You can change `pd.options.display.max_columns` and `pd.options.display.min_rows` to modify these if you really want to.) I generally try to dissuade them from changing these settings.

In [189]:
```
jb2
```

Out[189]:

	age	are_you_datascientist	company_size	country_live	employment_status	first_learn_about_main_ide	how_often_use_main_ide	ide_main	is_python_main	job_team	mai
1	21.0	True	5000.0	India	Fully employed by a company / organization	School / University	Daily	VS Code	Yes	Work in a team	Bc a
2	30.0	False	5000.0	United States	Fully employed by a company / organization	Friend / Colleague	Daily	Vim	Yes	Work on your own project(s) independently	Bc a
10	21.0	False	51.0	Other country	Fully employed by a company / organization	School / University	Daily	IntelliJ IDEA	Yes	Work in a team	Bc a
11	21.0	True	51.0	United States	Fully employed by a company / organization	Online learning platform / Online course	Daily	PyCharm Community Edition	Yes	Work in a team	Bc a
13	30.0	True	5000.0	Belgium	Fully employed by a company / organization	Social network	Daily	VS Code	Yes	Work in a team	Bc a
...	...	...	...	...	...	...	...	...	...	...	
54456	30.0	False	1001.0	Turkey	Fully employed by a company / organization	Friend / Colleague	Daily	PyCharm Community Edition	Yes	Work on your own project(s) independently	Bc a
54457	21.0	False	2.0	Russian Federation	Fully employed by a company / organization	School / University	Daily	Vim	Yes	Work on your own project(s) independently	Bc a
54459	21.0	False	1.0	Russian Federation	Self-employed (a person earning income directl...	Friend / Colleague	Daily	PyCharm Professional Edition	Yes	Work in a team	Bc a
54460	30.0	True	51.0	Spain	Fully employed by a company / organization	Search engines	Daily	Other	Yes	Work on your own project(s) independently	Bc a
54461	21.0	False	11.0	Algeria	Fully employed by a company / organization	Online learning platform / Online course	Daily	VS Code	Yes	Work in a team	Bc a

13711 rows × 19 columns

Jupyter showing default view of dataframe. We have ten rows but need to scroll to see all of the data.

```
In [190]: (jb2
           .head(10)
           .T
          )
```

Out[190]:

	1	2	10	11	13	14	15	17	22	25
age	21.0	30.0	21.0	21.0	30.0	30.0	50.0	30.0	40.0	50.0
are_you_datascientist	True	False	False	True	True	True	False	True	False	True
company_size	5000.0	5000.0	51.0	51.0	5000.0	501.0	1001.0	2.0	51.0	11.0
country_live	India	United States	Other country	United States	Belgium	Ecuador	Germany	Chile	Australia	United States
employment_status	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization	Fully employed by a company / organization
first_learn_about_main_ide	School / University	Friend / Colleague	School / University	Online learning platform / Online course	Social network	Other	Friend / Colleague	Social network	Technical review / Forum / Blog	Search engines
how_often_use_main_ide	Daily	Daily	Daily	Daily	Daily	Weekly	Daily	Daily	Daily	Daily
ide_main	VS Code	Vim	IntelliJ IDEA	PyCharm Community Edition	VS Code	VS Code	Vim	VS Code	VS Code	PyCharm Professional Edition
is_python_main	Yes	Yes	Yes	Yes	Yes	Yes	No, I use Python as a secondary language	Yes	No, I use Python as a secondary language	Yes
job_team	Work in a team	Work on your own project(s) independently	Work in a team	Work in a team	Work in a team	Work on your own project(s) independently	Work in a team	Work on your own project(s) independently	Work in a team	Work on your own project(s) independently
main_purposes	Both for work and personal	Both for work and personal	Both for work and personal	Both for work and personal	Both for work and personal	For work	For work	Both for work and personal	Both for work and personal	Both for work and personal
missing_features_main_ide	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	No, it has all the features I need	Yes – Please list:	No, it has all the features I need	No, it has all the features I need	Yes – Please list:
nps_main_ide	8.0	10.0	10.0	9.0	10.0	10.0	5.0	10.0	10.0	9.0
python_years	3.0	3.0	1.0	3.0	6.0	3.0	1.0	1.0	6.0	11.0
python3_version_most	3.6	3.6	3.8	3.9	3.7	3.8	3.6	3.8	3.7	3.8
several_projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on many different projects	Yes, I work on one main and several side projects	Yes, I work on one main and several side projects
team_size	2	5	2	2	2	5	2	0	2	2
use_python_most	Software prototyping	DevOps / System administration / Writing autom...	Web development	Data analysis	Data analysis	Programming of web parsers / scrapers / crawlers	Web development	Machine learning	Software prototyping	Data analysis
years_of_coding	3.0	3.0	1.0	3.0	3.0	3.0	11.0	1.0	11.0	11.0

Jupyter showing a transposed view of dataframe. Notice that we see ten complete samples of data showing on the screen without scrolling.

However, if you change these settings to view more data and find yourself scrolling through a million rows of data, your spidey sense should go off, telling you that you are doing things incorrectly. Humans are not made to look for interesting data by scrolling through rows of data. It is better to use a computer (which is optimized to search through data) to find rows you might be interested in. My two favorite methods of leveraging a computer to search for us are visualization and filtering the data.

On that note, if you use the `.transpose` method to view more data on your screen, you might not want to transpose your whole data set. Remember that pandas stores and optimizes data by column types. If you make a row that contains different data types (strings, dates, numbers) into a column that can

be a slow and memory-loving operation. It is better to pull off the head and tail or take a sample of the data and then transpose it.

When we explored line plots in the plotting section, we showed an example of transposing the data. We had a presidential data set with the president's names in the index and ratings for various skills in the columns. When we did a line plot of this data, each characteristic was its own line. Instead, we wanted each president to be its own line, so we transposed the data.

Stacking & Unstacking

I have previously used the `.unstack` method but have not discussed it. It (along with its complement, `.stack`) is a powerful method for reshaping your data.

At a high level, `.unstack` moves an index level into the columns. Usually, we use this operation on multi-index data, moving one of the indices into the columns (creating hierarchical columns). The `.stack` method does the reverse, moving a multi-level column into the index.

Let's look at an example using the JetBrains data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)

>>> print(jb2)
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...              3.0         3.6
5       21                 False  ...              3.0         3.8
10      21                 False  ...              1.0         3.8
11      21                  True  ...              3.0         3.9
13      30                  True  ...              3.0         3.7
...    ...                   ...  ...              ...         ...
54432   21                  True  ...              0.5         3.8
54433   30                 False  ...             11.0         3.6
54442   50                  True  ...             11.0         3.6
54447   30                 False  ...              3.0         3.6
54450   30                 False  ...             11.0         3.8

[6980 rows x 20 columns]
```

We will create a hierarchical or multi-index by grouping with multiple columns. Let's take the size of responses to *are_you_datascientist* column by country:

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
... )
country_live  are_you_datascientist
Algeria       False                   5
              True                    3
Argentina     False                  48
              True                   10
Armenia       False                   8
                                     ..
Uzbekistan    True                    1
Venezuela     False                   8
              True                    4
Viet Nam      False                  13
              True                    8
Length: 152, dtype: int64
```

Notice that the result is a series with a multi-index. This result is useful but a little hard to scan through. It would be easier if we had countries in the index and each of the responses to *are_you_datascientist* as their own column. We can do that by unstacking the inner index into a column (note that you could also do this operation with `pd.crosstab`):

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack()
... )
are_you_datascientist  False  True
country_live
Algeria                    5      3
Argentina                 48     10
Armenia                    8      2
Australia                104     35
Austria                   42     19
...                      ...    ...
United States            976    346
Uruguay                    8      6
Uzbekistan                 2      1
Venezuela                  8      4
Viet Nam                  13      8
```

```
[76 rows x 2 columns]
```

By default, `.unstack` moves the inner index up to the columns. Because this operation was performed on a series, it is changed to a dataframe. (If we perform `.unstack` on a dataframe, we will get a dataframe with nested columns.)

If we wanted to pull up the country index (the outer index), we could specify it by name or position. The position is 0 for the outer index, *country_live*, and 1 for *are_you_datascientist*:

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack(0)
... )
country_live           Algeria  Argentina  ...  Venezuela  Viet Nam
are_you_datascientist                      ...
False                        5         48  ...          8        13
True                         3         10  ...          4         8

[2 rows x 76 columns]
```

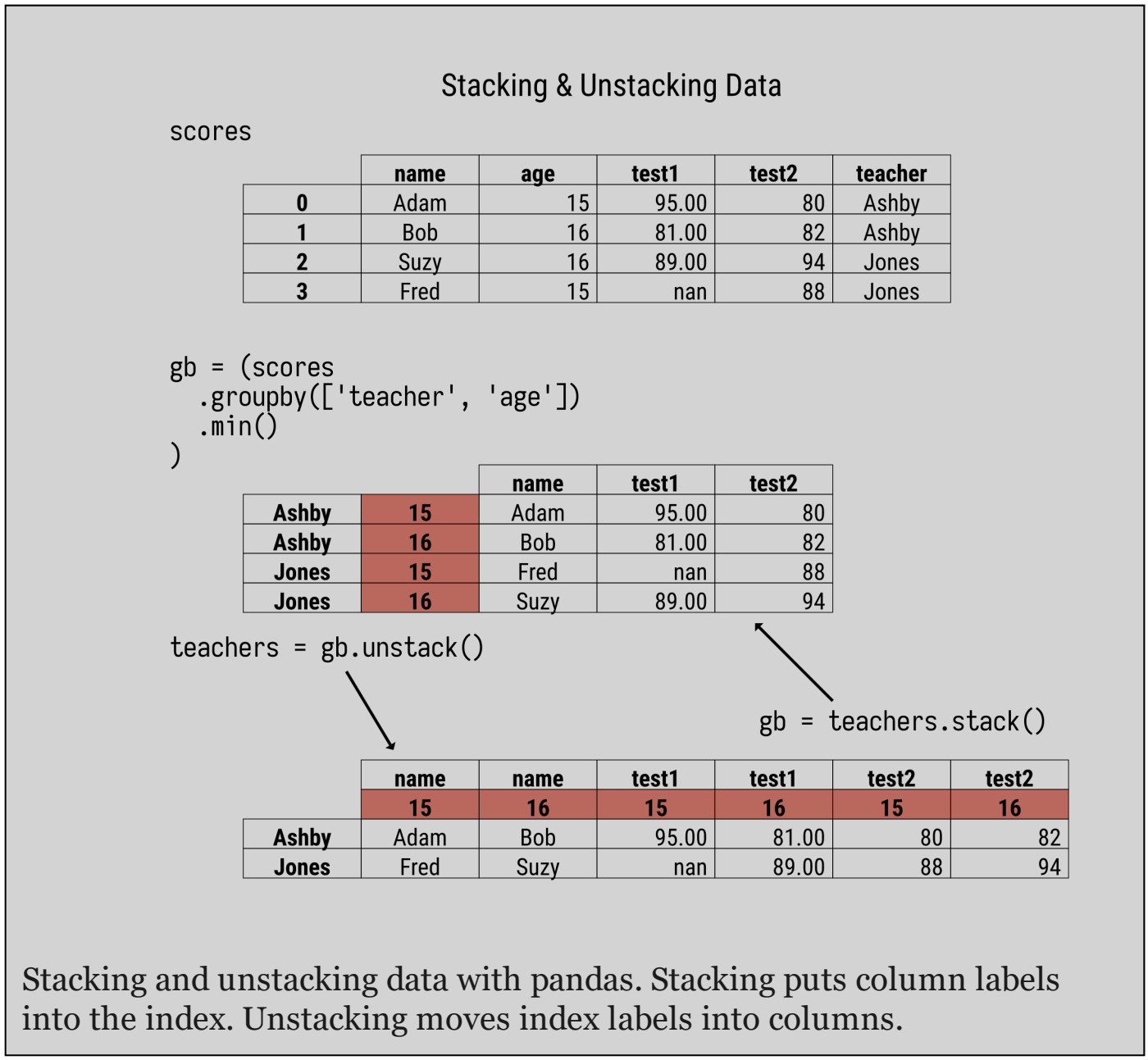
I would prefer to use the index name (rather than the index position) in this case as it is easier to understand (and one less thing you need to memorize):

```
>>> print(jb2
...   .groupby(['country_live', 'are_you_datascientist'], observed=True)
...   .size()
...   .unstack('country_live')
... )
country_live           Algeria  Argentina  ...  Venezuela  Viet Nam
are_you_datascientist                      ...
False                        5         48  ...          8        13
True                         3         10  ...          4         8

[2 rows x 76 columns]
```

Stacking

Let's look at stacking. Previously, we saw that we could specify multiple aggregation functions with the `.pivot_table` method. The result is a dataframe with hierarchical columns:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'company_size': ['min', 'max']})
... )
              age     company_size
              max min          max min
country_live
Algeria        30  21           11   1
Argentina      50  21         5000   1
Armenia        60  18         5000   1
Australia      60  18         5000   1
Austria        50  21         5000   1
...            ..  ..          ...  ..
United States  60  18         5000   1
Uruguay        30  21         5000   2
Uzbekistan     21  21           51   1
Venezuela      50  21           51   1
Viet Nam       60  18         1001   1

[76 rows x 4 columns]
```



Stacking and unstacking data with pandas. Stacking puts column labels into the index. Unstacking moves index labels into columns.

In a previous example, we saw that we could unstack the index by the name of the index (the name of the column before it was put in the index) or by the position. In this example, we want to stack one of the hierarchical columns into the index. The columns do not have names, so we must use the position. The outermost column level is 0. Stacking by this level will move *age* and *company_size* into the index:

```
>>> print(jb2
...   .pivot_table(index='country_live',
...                aggfunc={'age': ['min', 'max'],
...                         'company_size': ['min', 'max']})
...   .stack(0)
... )
```

```
                           max  min
country_live
Algeria      age             30   21
             company_size    11    1
Argentina    age             50   21
             company_size  5000    1
Armenia      age             60   18
...                         ...  ...
Uzbekistan   company_size    51    1
Venezuela    age             50   21
             company_size    51    1
Viet Nam     age             60   18
             company_size  1001    1

[152 rows x 2 columns]
```

If we want to move the inner columns, *max* and *min*, into the index, this is the default behavior. Alternatively, we can specify level 1 as an argument for `.stack`:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'company_size': ['min', 'max']})
...  .stack(1)
... )
                  age  company_size
country_live
Algeria      max   30            11
             min   21             1
Argentina    max   50          5000
             min   21             1
Armenia      max   60          5000
...               ...           ...
Uzbekistan   min   21             1
Venezuela    max   50            51
             min   21             1
Viet Nam     max   60          1001
             min   18             1

[152 rows x 2 columns]
```

Finally, if you want to change the order of the levels in a hierarchical index or columns, you can use the `.swaplevel` method:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
```

```
...                          'company_size': ['min', 'max']})
...  .stack(1)
...  .swaplevel()
... )
                   age  company_size
    country_live
max Algeria         30            11
min Algeria         21             1
max Argentina       50          5000
min Argentina       21             1
max Armenia         60          5000
...                ...           ...
min Uzbekistan      21             1
max Venezuela       50            51
min Venezuela       21             1
max Viet Nam        60          1001
min Viet Nam        18             1

[152 rows x 2 columns]
```

Flattening Hierarchical Indexes and Columns

When you start applying grouping operations, you can end up with a hierarchical index or columns. In practice, I find these nested structures challenging to deal with and often want to remove (or flatten them).

Let's start by discussing removing the hierarchical index, which is simple. We use the `.reset_index` method. Here is a dataframe with a hierarchical index:

```
>>> print(jb2
...  .groupby(['country_live', 'age'])
...  .mean(numeric_only=True)
... )
                  company_size  nps_main_ide  python_years  \
country_live age
Algeria      18            <NA>          <NA>          <NA>
             21            3.75          7.25           1.0
             30             1.5          10.0          2.75
             40            <NA>          <NA>          <NA>
             50            <NA>          <NA>          <NA>
...                         ...           ...           ...
Viet Nam     21      133.142857      8.928571           2.5
             30        7.666667      9.333333      2.333333
             40            51.0           9.0           3.0
             50            <NA>          <NA>          <NA>
             60             1.0           8.0           3.0
```

```
                  team_size  years_of_coding
country_live age
Algeria      18         <NA>             <NA>
             21          1.5            0.875
             30          1.5            2.625
             40         <NA>             <NA>
             50         <NA>             <NA>
...                      ...              ...
Viet Nam     21     5.571429         1.464286
             30     1.666667         4.166667
             40          2.0              6.0
             50         <NA>             <NA>
             60          1.0              1.0

[456 rows x 5 columns]
```

We can use `.reset_index` to push each index level into a column:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
...  .mean(numeric_only=True)
...  .reset_index()
... )
    country_live  age  ...  team_size  years_of_coding
0         Algeria   21  ...        1.5            0.875
1         Algeria   30  ...        1.5            2.625
2       Argentina   21  ...   5.333333         2.916667
3       Argentina   30  ...   3.814815              5.0
4       Argentina   40  ...        3.6             8.15
..            ...  ...  ...        ...              ...
308      Viet Nam   18  ...        7.5             0.75
309      Viet Nam   21  ...   5.571429         1.464286
310      Viet Nam   30  ...   1.666667         4.166667
311      Viet Nam   40  ...        2.0              6.0
312      Viet Nam   60  ...        1.0              1.0

[313 rows x 7 columns]
```

Alternatively, when using `.groupby`, you can set the `as_index` parameter to `False`, and the result does not insert the grouping columns in the index, they will stay as columns:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], as_index=False, observed=True)
...  .mean(numeric_only=True)
... )
    country_live  age  ...  team_size  years_of_coding
```

```
0        Algeria   21  ...        1.5           0.875
1        Algeria   30  ...        1.5           2.625
2      Argentina   21  ...   5.333333        2.916667
3      Argentina   30  ...   3.814815             5.0
4      Argentina   40  ...        3.6            8.15
..           ...  ...  ...        ...             ...
308     Viet Nam   18  ...        7.5            0.75
309     Viet Nam   21  ...   5.571429        1.464286
310     Viet Nam   30  ...   1.666667        4.166667
311     Viet Nam   40  ...        2.0             6.0
312     Viet Nam   60  ...        1.0             1.0

[313 rows x 7 columns]
```

Now, let's explore flattening hierarchical columns. Sadly, the `.reset_index` method won't work for the column names. Generally, we don't want to push the column names into a row, but we want to combine them into a single level of column names. And there is no convenient method to do that in pandas.

Here is an example of data with a hierarchical column. For every country we have the mean values for each numeric column broken down by age:

```
>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
...  .mean(numeric_only=True)
...  .unstack()
... )
              company_size                ... years_of_coding  \
age                     18           21   ...              50
country_live                               ...
Algeria               <NA>         3.75  ...            <NA>
Argentina             <NA>   566.055556  ...            11.0
Armenia               11.0        761.0  ...            <NA>
Australia              6.0   752.862069  ...       10.583333
Austria               <NA>   222.294118  ...            11.0
...                    ...          ...  ...             ...
United States        693.0  1745.711688  ...       10.557692
Uruguay               <NA>         21.0  ...            <NA>
Uzbekistan            <NA>    17.666667  ...            <NA>
Venezuela             <NA>         35.0  ...             6.0
Viet Nam              51.0   133.142857  ...            <NA>


age                     60
country_live
Algeria               <NA>
```

```
Argentina           <NA>
Armenia              0.5
Australia       9.333333
Austria             <NA>
...                  ...
United States  10.381579
Uruguay             <NA>
Uzbekistan          <NA>
Venezuela           <NA>
Viet Nam             1.0

[76 rows x 30 columns]
```

Flattening Grouping Data with Multiple Aggregations

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
def flatten(df_):
    cols = ['_'.join(cs) for cs in df_.columns.to_flat_index()]
    df_.columns = cols
    return df_
(auto
 .groupby('make')
 .agg(['min', 'max'])
 .pipe(flatten))
```

	year_min	year_max	cylinders_min	cylinders_max
Acura	1986	2020	4.00	6.00
Audi	1984	2020	4.00	12.00
BMW	1984	2020	2.00	12.00
BYD	2012	2019	nan	nan
Bentley	1998	2019	8.00	12.00
VPG	2011	2013	8.00	8.00
Vector	1992	1997	8.00	12.00
Volvo	1984	2019	4.00	8.00
Yugo	1986	1990	4.00	4.00
smart	2008	2019	3.00	3.00

Grouping and then flattening hierarchical columns.

In addition to the lack of a convenient method to flatten columns being a gaping hole in the pandas API, to add insult to injury, you have to mutate the dataframe to update the columns. Remember, mutation generally throws a wrench in our chaining operations.

To get around this, I make a function that will flatten columns. The function joins each level of columns with an underscore. Then, I combine that function with the `.pipe` method. This lets me do a column flattening operation in a chain:

```
>>> def flatten_cols(df):
...     cols = ['_'.join(map(str, vals))
...             for vals in df.columns.to_flat_index()]
...     df.columns = cols
...     return df

>>> print(jb2
...  .groupby(['country_live', 'age'], observed=True)
...  .mean(numeric_only=True)
...  .unstack()
...  .pipe(flatten_cols)
... )
               company_size_18  company_size_21  ...  \
country_live                                     ...
Algeria                    <NA>             3.75  ...
Argentina                  <NA>       566.055556  ...
Armenia                    11.0            761.0  ...
Australia                   6.0       752.862069  ...
Austria                    <NA>       222.294118  ...
...                         ...              ...  ...
United States             693.0      1745.711688  ...
Uruguay                    <NA>             21.0  ...
Uzbekistan                 <NA>        17.666667  ...
Venezuela                  <NA>             35.0  ...
Viet Nam                   51.0       133.142857  ...

               years_of_coding_50  years_of_coding_60
country_live
Algeria                      <NA>                <NA>
Argentina                    11.0                <NA>
Armenia                      <NA>                 0.5
Australia               10.583333            9.333333
Austria                      11.0                <NA>
...                           ...                 ...
```

```
United States           10.557692           10.381579
Uruguay                      <NA>                <NA>
Uzbekistan                   <NA>                <NA>
Venezuela                     6.0                <NA>
Viet Nam                     <NA>                 1.0

[76 rows x 30 columns]
```

Chapter Methods

Method	Description
`.melt( id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None, ignore_index=True)`	Return an unpivoted dataframe. With each column in `value_vars` stack on top of each other. Keep the `id_vars` columns.
`g.transform(func, *args, **kwargs)`	Return a dataframe with original index. The function will get passed a group and should return a dataframe with same dimensions as group.
`pd.options .display.max_columns`	Property to set to configure pandas to show at most this amount of columns.
`pd.options .display.min_rows`	Property to set to configure pandas to show at most this amount of row.
`.stack(level=-1, dropna=True)`	Push a column level into an index level. Can specify the column `level` (`-1` is innermost).
`.unstack(level=-1, dropna=True)`	Push an index level into a column level. Can specify an index `level` (`-1` is innermost).
`.swaplevel(i=-2, j=-1, axis=0)`	Swap the levels of multi-indexed object (`0` is outermost, `-1` (or length of multi-index) is innermost). Can specify the name for `i` and `j`.

Method	Description
`.reset_index( level=None, drop=False, col_level=0, col_fill='')`	Return a dataframe with a new index (or new level). To remove a level, specify that with `level` (by position or name). Position 0 is the outermost level, and it goes up. Alternatively, -1 is the innermost level. Index values are moved to columns or dropped if `drop=True`. `col_level` determines where the index label goes with multiple column levels. Other levels will get the value of `col_fill`.
`.pipe(func, *args, **kwargs)`	Apply a function to a dataframe. Return the result of the function.

Summary

In this chapter, we showed how to melt and unmelt data. If you use the Seaborn library for plotting, you might need to transform your data so that you can plot with this library. We also explored stacking and unstacking data. Finally, we showed how to remove nested columns and indexes.

Exercises

With a dataset of your choice:

1. Melt two numeric column values into a single column. Add a new column to indicate what the values mean.
2. Un-melt the above.
3. Group by two columns, take the mean, and unstack the result.
4. Group by two columns, take the mean, unstack the result, and flatten the columns.

```
h
```

---

1. http://vita.had.co.nz/papers/tidy-data.html

2. https://github.com/pandas-dev/pandas/issues/43547

Working with Time Series

This chapter will explore how to manipulate and work with time-series data. One thing to note, when we say "time-series", we are not talking about the pandas `Series` object, but rather data that has a date component. Often we will have that date component in the index of a pandas series or dataframe because that allows us to do time aggregations easily.

Loading the Data

For this section, I'm going to explore a dataset from the US Geologic Survey that deals with river flow of a river in Utah called the Dirty Devil river[1].

This data is a tab-delimited ASCII file in detail described here[2].

The columns are:

- *agency_cd* - Agency collecting data
- *site_no* - USGS identification number of site
- *datetime* - Date
- *tz_cd* - Timezone
- *144166_00060* - Discharge (cubic feet per second)
- *144166_00060_cd* - Status of discharge. "A" (approved), "P" (provisional), "e" (estimate).
- *144167_00065* - Gage height (feet)
- *144167_00065_cd* - Status of gage_height. "A" (approved), "P" (provisional), "e" (estimate).

Here is my code to load the data. I have also included a tweak function that converts the date information to actual dates and renames some columns. Note that the file is not a CSV file, but we can specify a tab as a separator. Also, we need to skip a few of the rows:

This data provided by the US government is not really a CSV file. It has many lines at the top that we want to skip. Then, it has the columns. Then, we want to skip one more line and keep the rest. If you use the pandas engine for parsing the CSV, you use this parameter to accomplish that:

```
skiprows=lambda num: num <34 or num == 35
```

The pyarrow library in pandas 2.2 doesn't accept anything other than an integer for the `skiprows` parameter [3], so I'm going to write a function to remove the lines for me:

```
import urllib.request

def download_and_modify_url(url, local_filename):
    # Download the file from the URL
    urllib.request.urlretrieve(url, local_filename)
    with open(local_filename, 'r') as file:
        lines = file.readlines()

    with open(local_filename, 'w') as file:
        for i, line in enumerate(lines):
            if i <34 or i == 35:
                continue
            file.write(line)

url = 'https://github.com/mattharrison/datasets/raw/master'\
    '/data/dirtydevil.txt'
local_filename = 'data/devilclean.txt'
download_and_modify_url(url, local_filename)
```

Now, I'm going to load the cleaned up file:

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def tweak_river(df_):
...     return (df_
...  .assign(datetime=pd.to_datetime(df_.datetime))
...  .rename(columns={'144166_00060': 'cfs',
...                   '144167_00065': 'gage_height'})
...  .set_index('datetime')
...  .loc[:, ['agency_cd', 'site_no', 'tz_cd', 'cfs', 'gage_height'] ]
... )

>>> dd = tweak_river(df)
>>> print(dd)
```

```
                    agency_cd  site_no tz_cd   cfs  gage_height
datetime
2001-05-07 01:00:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:15:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:30:00      USGS  9333500   MDT  71.0         <NA>
2001-05-07 01:45:00      USGS  9333500   MDT  70.0         <NA>
2001-05-07 02:00:00      USGS  9333500   MDT  70.0         <NA>
...                       ...      ...   ...   ...          ...
2020-09-28 08:30:00      USGS  9333500   MDT  9.53         6.16
2020-09-28 08:45:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:00:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:15:00      USGS  9333500   MDT   9.2         6.15
2020-09-28 09:30:00      USGS  9333500   MDT   9.2         6.15

[539305 rows x 5 columns]
```

Adding Timezone Information

Many times the date column is missing timezone information. In the Dirty Devil dataset, the *tz_cd* column has offset abbreviations:

```
>>> dd.tz_cd
datetime
2001-05-07 01:00:00    MDT
2001-05-07 01:15:00    MDT
2001-05-07 01:30:00    MDT
2001-05-07 01:45:00    MDT
2001-05-07 02:00:00    MDT
                       ...
2020-09-28 08:30:00    MDT
2020-09-28 08:45:00    MDT
2020-09-28 09:00:00    MDT
2020-09-28 09:15:00    MDT
2020-09-28 09:30:00    MDT
Name: tz_cd, Length: 539305, dtype: string[pyarrow]
```

I ignored it above and have “naive” time data. Getting timezone information into a date column can be slow, buggy, or frustrating. I spent a few hours trying to add timezone information to this dataset.

My takeaway is that although the documentation and API make it appear that `pd.to_datetime` should handle timezone data, I would not go down that path. Generally, you should use `pd.to_datetime` to get a naive time and then convert the naive times to timezones with `.dt.tz_localize`.

I tried concatenating the *datetime* and *tz_cd* columns together and passing that into `pd.to_datetime`. That worked but took two minutes, whereas code to convert into a naive date column in a fraction of that time (54 ms). I tried using format strings, replacing the timezones with alternate spellings, and using offsets with `pd.to_datetime`[4] in an attempt to speed up the conversion. They silently failed or errored out.

With the help of the pandas core developers, I was able to get that 2 minutes down to 15 seconds with this code. The key points below are using numeric date offsets (not timezone abbreviations) and `utc=True`:

```
>>> def tweak_river(df_):
...     return (df_
...       .assign(datetime=lambda df_:
...           pd.to_datetime(df_.datetime + " " +
...               df_.tz_cd.str.replace('MST', '-0700')
...                  .str.replace('MDT', '-0600'),
...               format='%Y-%m-%d %H:%M %z', utc=True))
...       .rename(columns={'144166_00060': 'cfs',
...                        '144167_00065': 'gage_height'})
...       .set_index('datetime')
...     )
```

However, I was able to get the runtime down to 1 second. The code is more involved, but this is 15-120x faster than the other code.

For my dataset, I wrote the function, `to_america_denver_time`, to get my date parsing with timezone information down from 2 minutes to 2 seconds. I group by the offset column and then use the grouping name (the offset name) to call `.dt.tz_localize`. This creates a date with local times. However, they are using offsets and not timezones.

Note that this uses dictionary unpacking (`**`). Because the column name is passed in as a variable, `tz_col`, we can't use the variable as the named parameter for the column name in `.assign`, or it would make a new variable named *tz_col*. To get around that, we create a dictionary using the variable name as a key, and then unpack that dictionary in the call to `.assign`.

To add timezone, you need to use `.dt.tz_convert` after creating the local time:

```
>>> def to_america_denver_time(df_, time_col, tz_col):
...     return (df_
...             .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...             .groupby(tz_col)
```

```
...              .groupby(tz_col)
...              [time_col]
...              .transform(lambda s: pd.to_datetime(s)
...                  .dt.tz_localize(s.name, ambiguous=True)
...                  .dt.tz_convert('America/Denver'))
...             )

>>> def tweak_river(df_):
...     return (df_
...       .assign(datetime=to_america_denver_time(df_, 'datetime',
...               'tz_cd'))
...       .rename(columns={'144166_00060': 'cfs',
...                        '144167_00065': 'gage_height'})
...       .set_index('datetime')
...       .loc[:, ['agency_cd', 'site_no', 'tz_cd', 'cfs', 'gage_height'] ]
...      )

>>> dd = tweak_river(df)
```

Here is the resulting data:

```
>>> print(dd)
                          agency_cd  site_no tz_cd   cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                              ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15
```

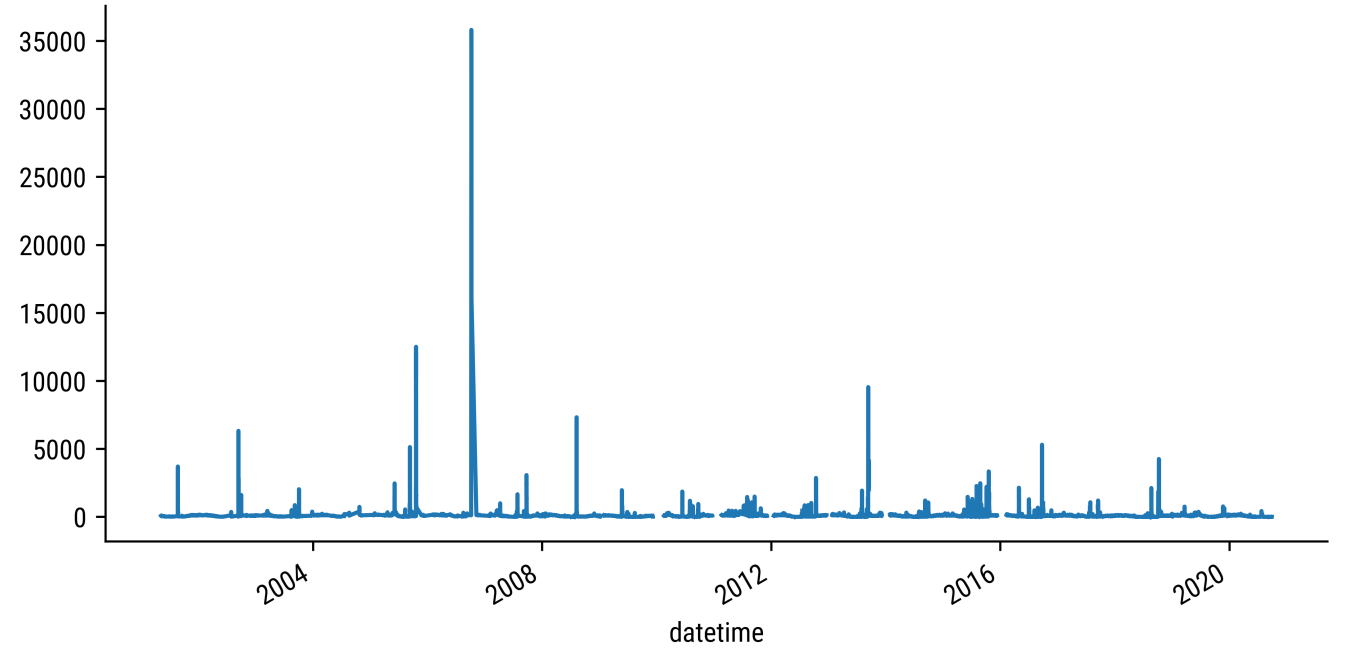
```
[539305 rows x 5 columns]
```

> Note
>
> One thing that bit me was I was trying to use `'MST'` and `'MDT'` as offset names. The underlying `pytz` library that handles timezone information didn't like them. (For a list of valid names, inspect `pytz.all_timezones`.) The timezone for this data is *America/Denver*.

Exploring the Data

I'm going to visualize the flow (cfs) of the river over time:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(dpi=600)
dd.cfs.plot(ax=ax)
```



Visualization of flow of Dirty Devil river.

From this visualization, it looks like there are some pretty big outliers. (Looking at a histogram or calling `.describe` would also confirm this.):

```
>>> dd.cfs.describe()
count      493124.0
mean     104.460537
std      477.341329
min             0.0
25%            34.7
50%            81.0
75%           115.0
max         35800.0
Name: cfs, dtype: double[pyarrow]
```

Slicing Time Series

We get some special slicing abilities because the dataframe has datetime data in the index. We can slice with strings representing dates (or parts of dates). Below, we will slice out the rows from 2018 onward:

```
>>> (dd
...  .cfs
...  .loc['2018':]
... )
datetime
2018-01-01 00:00:00-07:00    92.8
2018-01-01 00:15:00-07:00    88.3
2018-01-01 00:30:00-07:00    90.5
2018-01-01 00:45:00-07:00    90.5
2018-01-01 01:00:00-07:00    94.0
                             ...
2020-09-28 08:30:00-06:00    9.53
2020-09-28 08:45:00-06:00     9.2
2020-09-28 09:00:00-06:00     9.2
2020-09-28 09:15:00-06:00     9.2
2020-09-28 09:30:00-06:00     9.2
Name: cfs, Length: 95886, dtype: double[pyarrow]
```

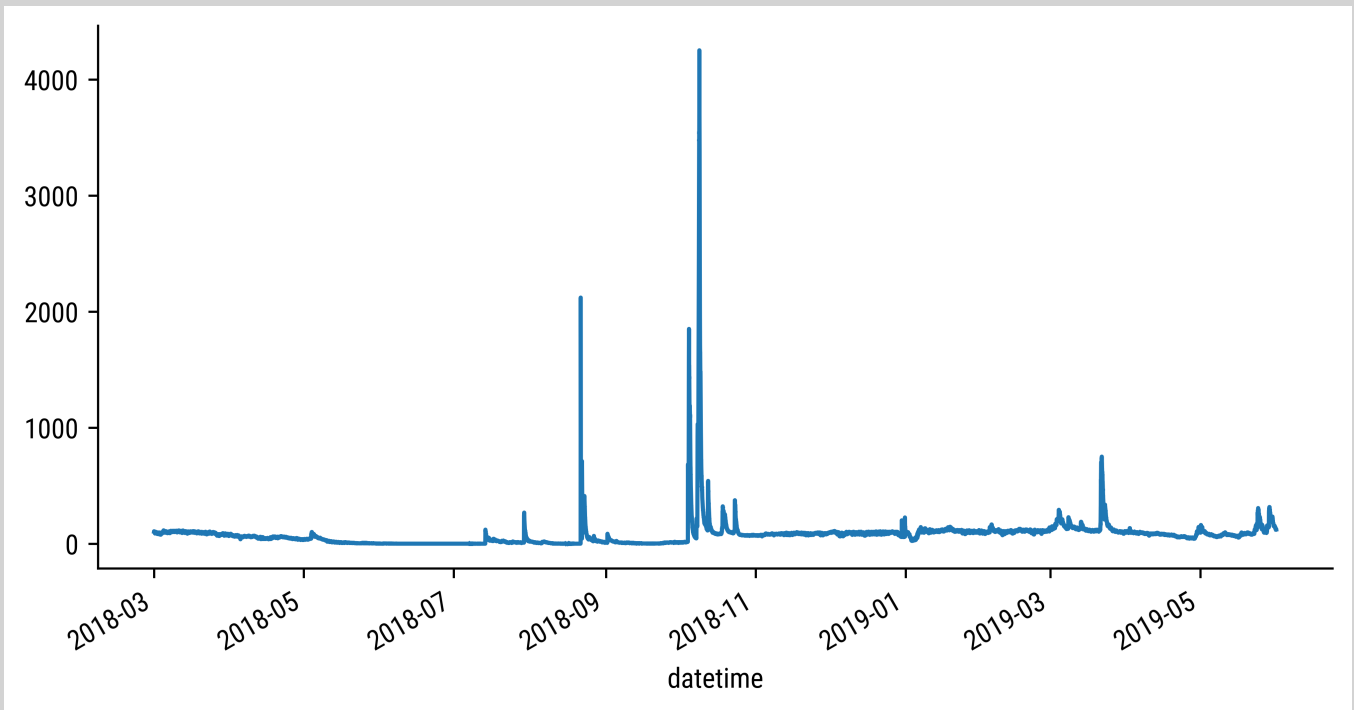
We can specify a slice including the month as well. Make sure the dates are sorted. When you specify just the month on an end slice, it includes all entries from that month on both the start and end slices (note that this is different behavior than both partial string slicing with `.loc` and position slicing with `.iloc`):

```
>>> (dd
...  .cfs
...  .sort_index()
...  .loc['2018-03-05':'2019/05']
```

```
... )
datetime
2018-03-05 00:00:00-07:00    104.0
2018-03-05 00:15:00-07:00    103.0
2018-03-05 00:30:00-07:00    103.0
2018-03-05 00:45:00-07:00    105.0
2018-03-05 01:00:00-07:00    106.0
                             ...
2019-05-31 22:45:00-06:00    121.0
2019-05-31 23:00:00-06:00    123.0
2019-05-31 23:15:00-06:00    123.0
2019-05-31 23:30:00-06:00    125.0
2019-05-31 23:45:00-06:00    123.0
Name: cfs, Length: 43478, dtype: double[pyarrow]
```

Let's visualize what that slice of data looks like:

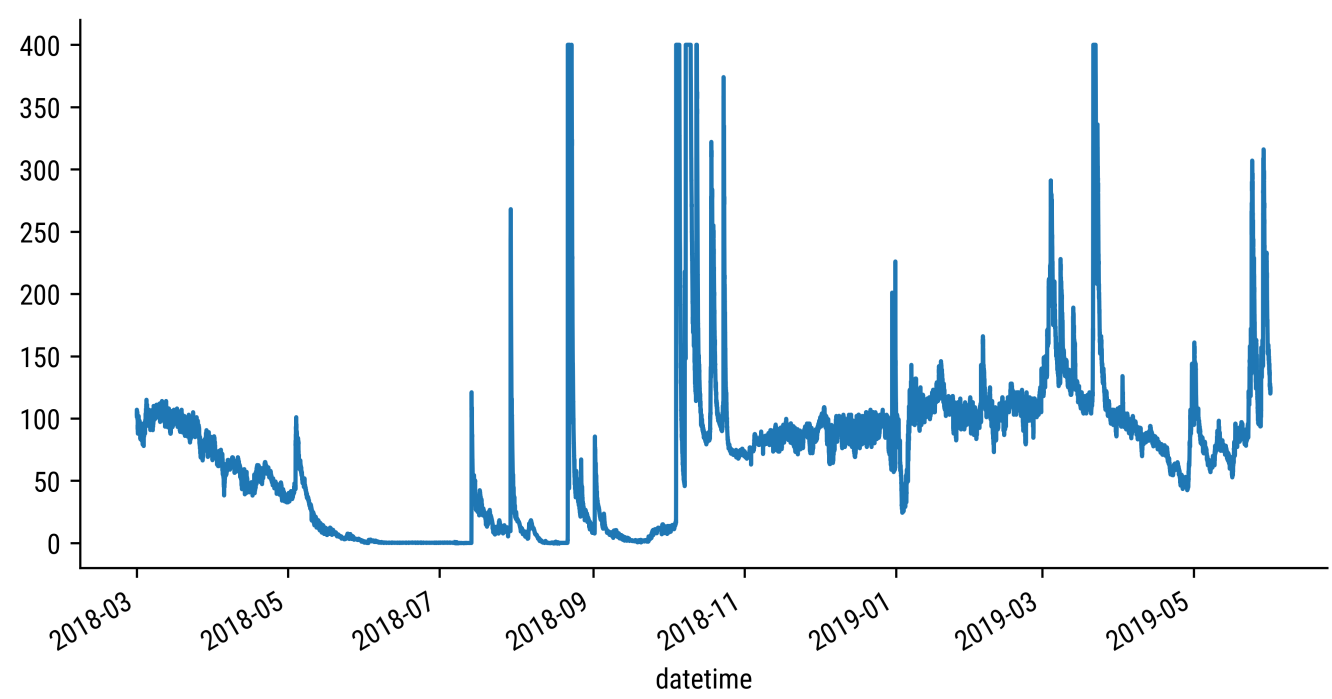
```
(dd
  .sort_index()
  .cfs
  .loc['2018/3':'2019/5']
  .plot()
)
```



Visualization of the flow of Dirty Devil River from March 2018 through May 2019.

I'm going to clip the visualization and limit the upper value to 400 and try the visualization again:

```
(dd
  .sort_index()
  .cfs
  .loc['2018/3':'2019/5']
  .clip(upper=400)
  .plot()
)
```

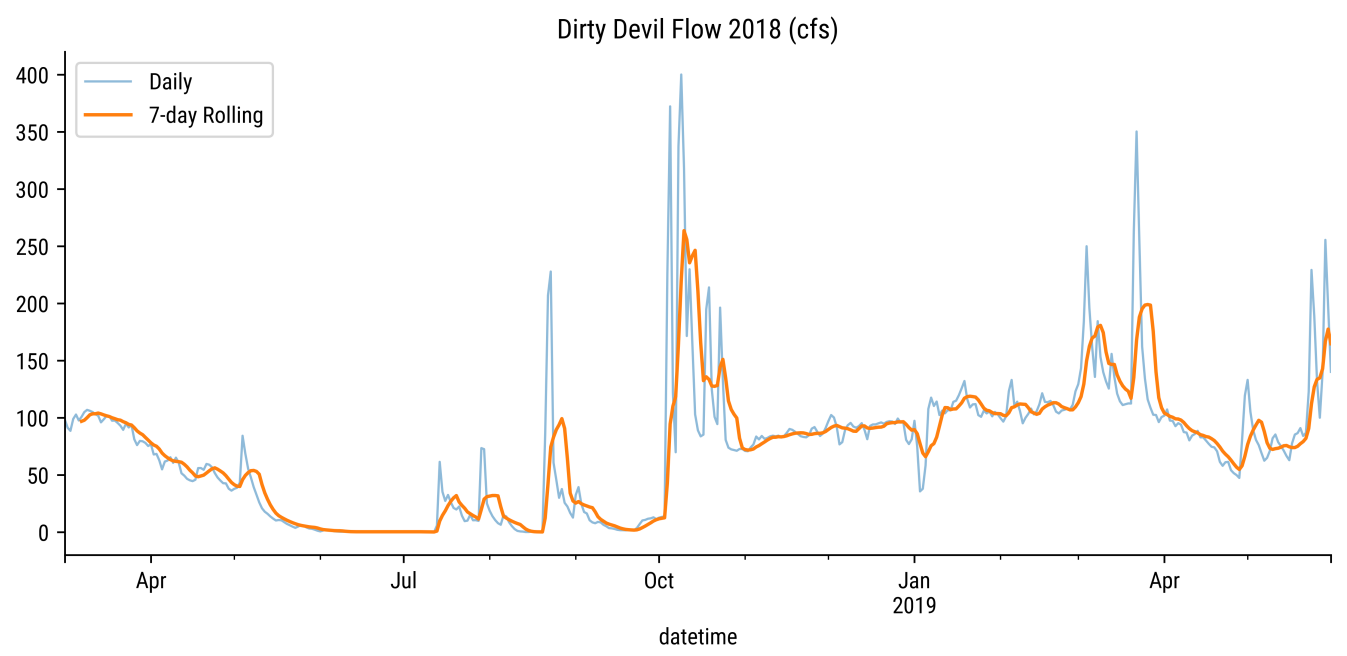


Visualization of the flow of Dirty Devil River from March 2018 through May 2019 with values clipped at 400.

Because the index is a time series, we can leverage the ability to resample. A typical operation these days is to plot rolling 7-day average data on top of daily data. The `.rolling` method accepts a moving window size, `window`, and like a grouping operation, you generally aggregate the result. Let's do it:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(8,4))
>>> dd2018 = (dd
...   .sort_index()
...   .cfs
...   .loc['2018/3':'2019/5']
...   .clip(upper=400))

>>> (dd2018
      .resample('D')
```

```
...   .resample('D')
...   .mean()
...   .plot(figsize=(10,4), alpha=.5, linewidth=1, label='Daily')
... )

>>> ax = (dd2018
...   .resample('D')
...   .mean()
...   .rolling(7)
...   .mean()
...   .plot(figsize=(10,4), ax=ax, label='7-day Rolling')
... )
>>> ax.legend()
>>> ax.set_title('Dirty Devil Flow 2018 (cfs)')
>>> sns.despine()
>>> fig.savefig('img/pandas2/dd4.png', dpi=600, bbox_inches='tight')
<Figure size 6000x2400 with 1 Axes>
```



Visualization of the flow of daily and weekly levels of Dirty Devil River from March 2018 through May 2019, with values clipped at 400

Missing Timeseries Data

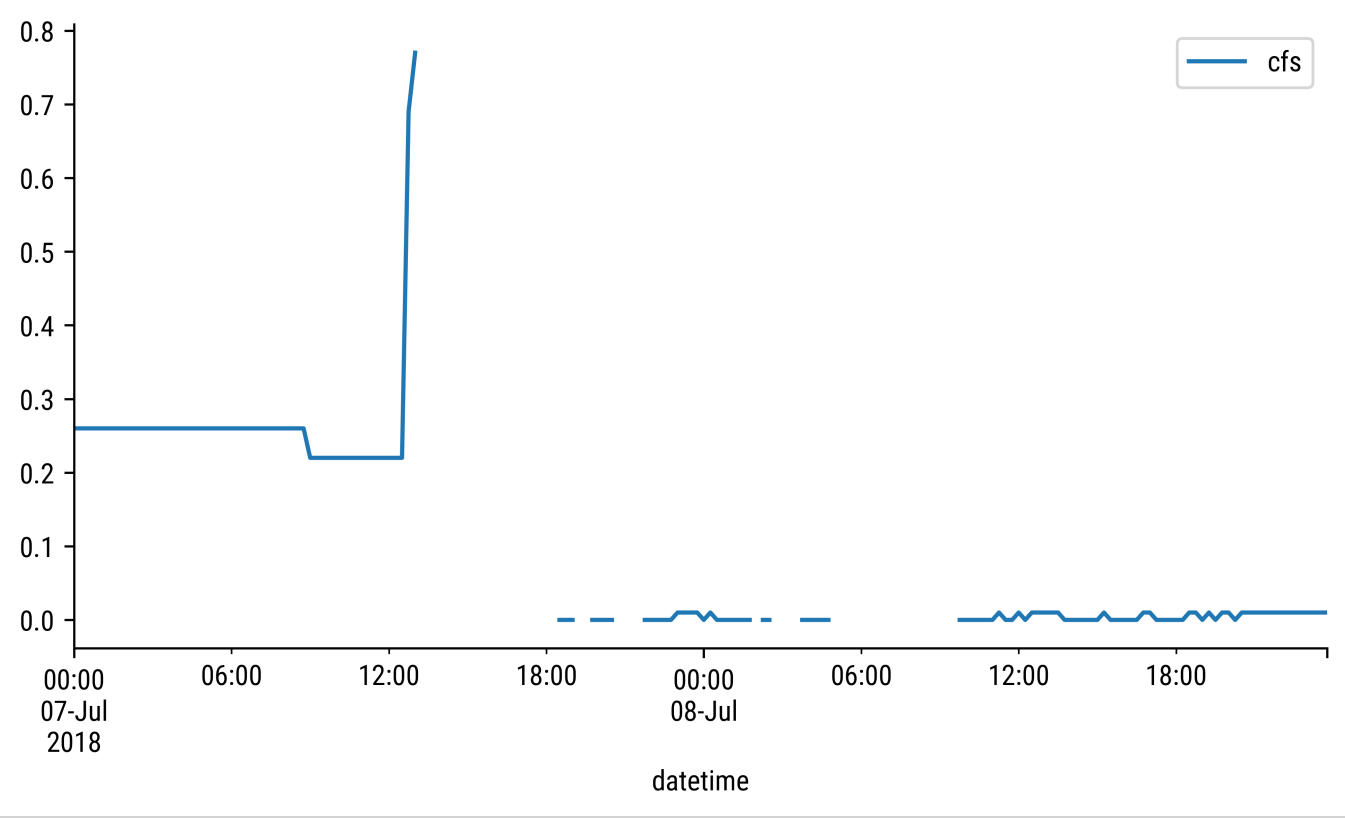
Let's look at dealing with missing data in timeseries data. First, we will search for it using `.isna`. One of the nice features of the `.query` method is that you

can call other methods in the string that you pass into it. Here we use `.query` and `.isna` to find missing values from the *cfs* column:

```
>>> print(dd
...   .sort_index()
...   [['cfs']]
...   .loc['2018/3':'2019/5']
...   .query('cfs.isna()')
... )
                           cfs
datetime
2018-07-07 13:15:00-06:00  <NA>
2018-07-07 13:30:00-06:00  <NA>
2018-07-07 13:45:00-06:00  <NA>
2018-07-07 14:00:00-06:00  <NA>
2018-07-07 14:15:00-06:00  <NA>
...                         ...
2018-08-18 08:15:00-06:00  <NA>
2018-08-18 08:30:00-06:00  <NA>
2018-08-18 08:45:00-06:00  <NA>
2018-08-18 09:15:00-06:00  <NA>
2018-08-18 10:30:00-06:00  <NA>

[337 rows x 1 columns]
```

Here is code to visualize the missing data from July 7-8. This will help us understand how the various methods work to deal with these missing values:

```
(dd
   .sort_index()
   [['cfs']]
   .loc['2018/7/7':'2018/7/8']
   .plot()
)
```



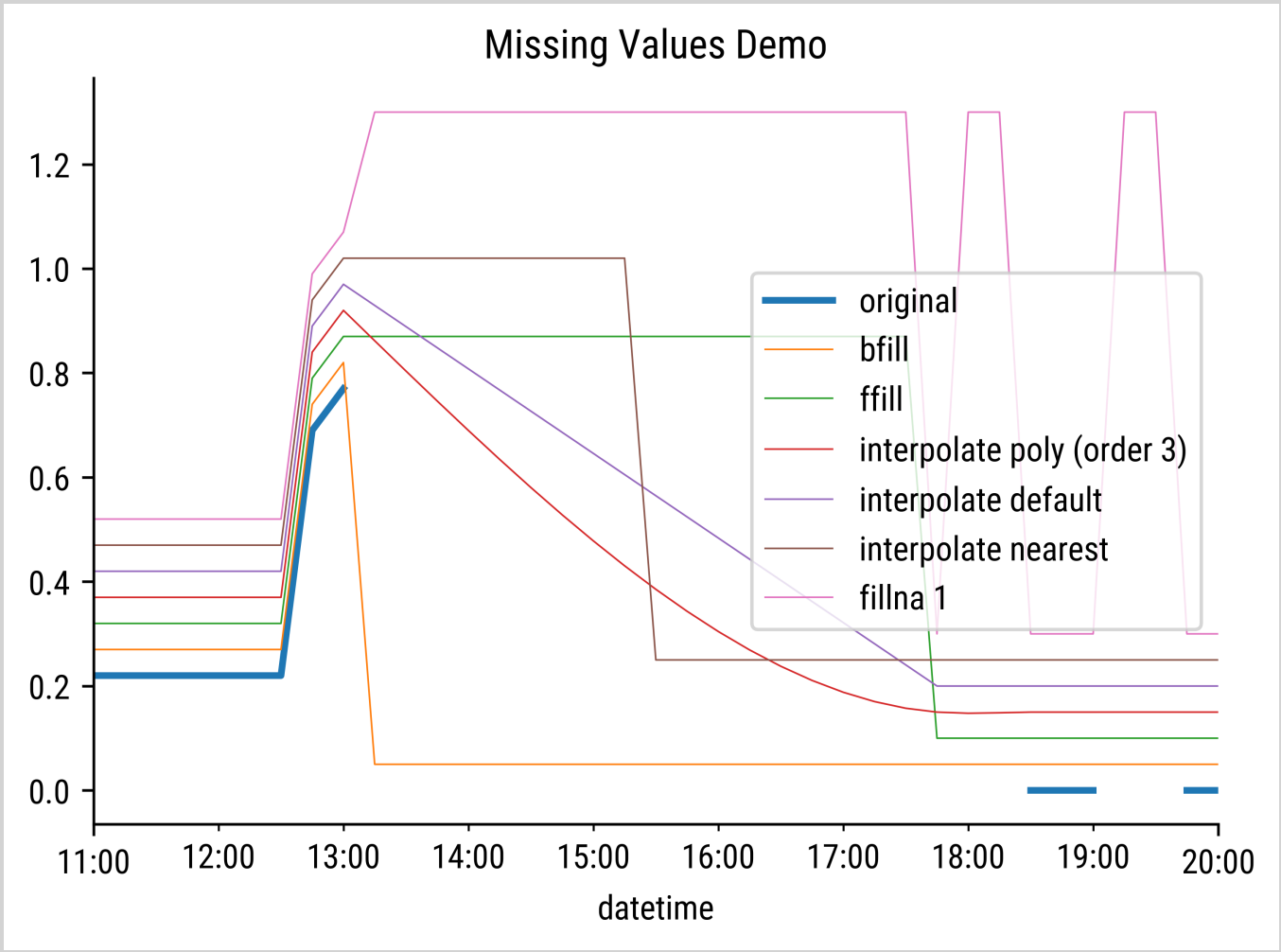
Visualization of missing data from flow of Dirty Devil river.

The series chapter discussed various methods for filling in missing data. Let's visualize those below. I'm adding an offset to each line so you can see the behavior:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(6,4))
>>> dd_july = (dd
...  .sort_index()
...  ['cfs']
...  .loc['2018/7/7 11:00':'2018/7/7 20:00']
... )

>>> dd_july.plot(ax=ax, label='original', linewidth=2)
>>> (dd_july
...  .bfill()
...  .add(.05)
...  .plot(label='bfill', ax=ax, linewidth=.5))

>>> (dd_july
...  .ffill()
...  .add(.1)
...  .plot(label='ffill', ax=ax, linewidth=.5))

>>> (dd_july
```

```
...  .astype(float)
...  .interpolate(method='polynomial', order=3)
...  .add(.15)
...  .plot(label='interpolate poly (order 3)', ax=ax, linewidth=.5))

>>> (dd_july
...  .astype(float)
...  .interpolate()
...  .add(.2)
...  .plot(label='interpolate default', ax=ax, linewidth=.5))

>>> (dd_july
...  .astype(float)
...  .interpolate(method='nearest')
...  .add(.25)
...  .plot(label='interpolate nearest', ax=ax, linewidth=.5))

>>> (dd_july
...  .fillna(1)
...  .add(.3)
...  .plot(label='fillna 1', ax=ax, linewidth=.5))

>>> ax.legend()
>>> ax.set_title('Missing Values Demo')
>>> sns.despine()
>>> fig.savefig('img/pandas2/dd6-na.png', dpi=600, bbox_inches='tight')
<Figure size 3600x2400 with 1 Axes>
```

Note that as of pandas 2.0.2, `.interpolate` doesn't like to run with pyarrow data.



Visualization of filling in missing data from flow of Dirty Devil river.

Exploring Seasonality

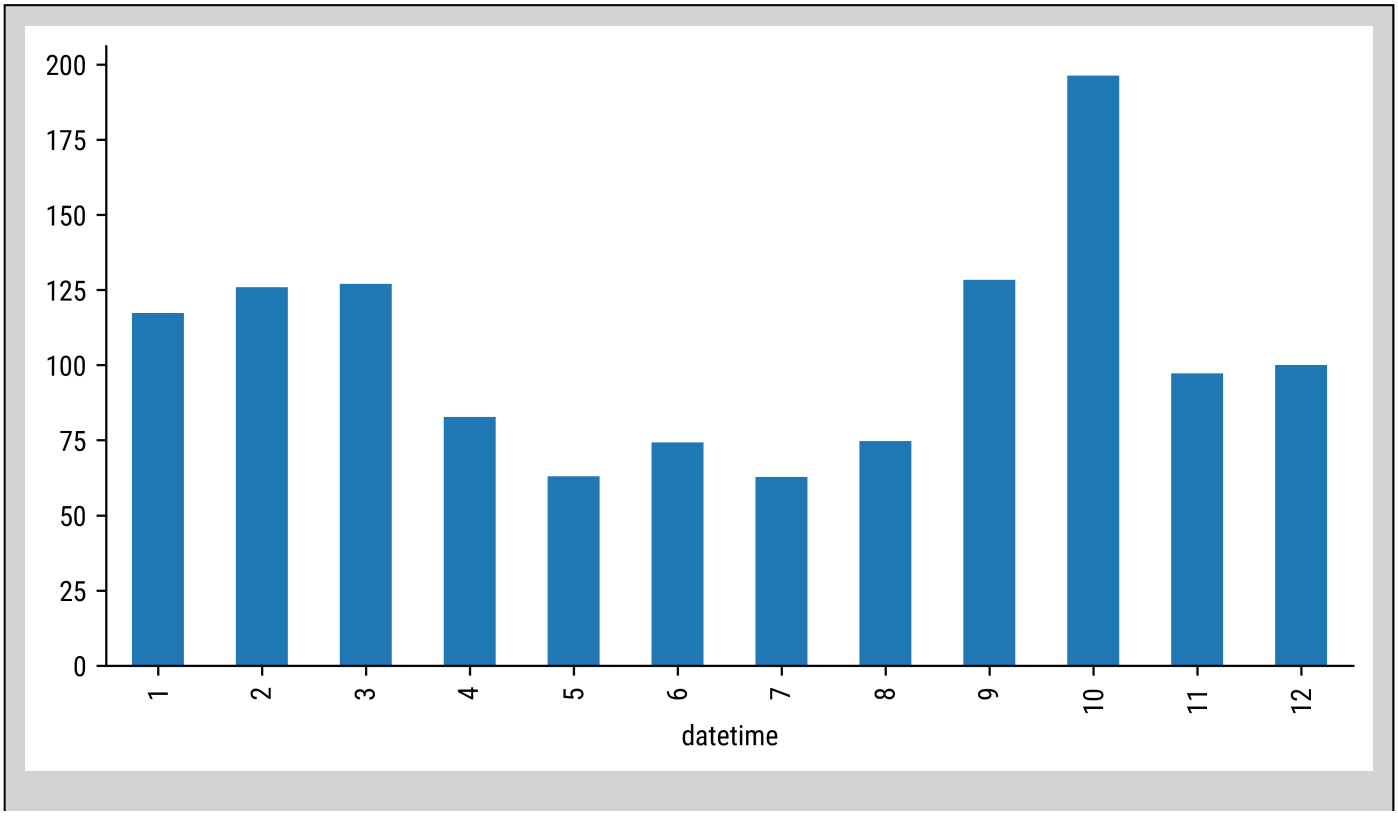
Time series data may have a seasonal component to it. Let's examine how to explore this with pandas (and related tools). We will explore the Dirty Devil dataset's cubic feet per second column (*cfs*). We can summarize monthly behavior in this column by combining `.groupby` and `.describe`. Note that we already have an index with date information, so one might suppose we could use `.resample` with `'M'` as an offset alias. However, a `.resample` operation will put the end date of each month in the index, while a `.grouby` on the month number will have only twelve entries in the index:

```
>>> print(dd
...   .groupby(dd.index.month)
...   .cfs
...   .describe()
```

```
... )
             count        mean  ...     75%      max
datetime                        ...
1          26011.0  117.268802  ...   132.0    265.0
2          41309.0  125.890293  ...   141.0    303.0
3          51807.0  127.037609  ...   136.0    750.0
4          50669.0   82.786214  ...    97.8   2140.0
5          49507.0   63.007851  ...    78.5   1960.0
...            ...         ...  ...     ...      ...
8          37584.0   74.676246  ...    59.1   7320.0
9          42272.0  128.309332  ...    55.9   9540.0
10         44647.0  196.285529  ...    80.9  35800.0
11         42165.0   97.194344  ...   105.0    766.0
12         28685.0  100.042608  ...   113.0    407.0

[12 rows x 8 columns]
```

We can also visualize these components by plotting. Here is a chain to plot the mean for each month as a bar plot:

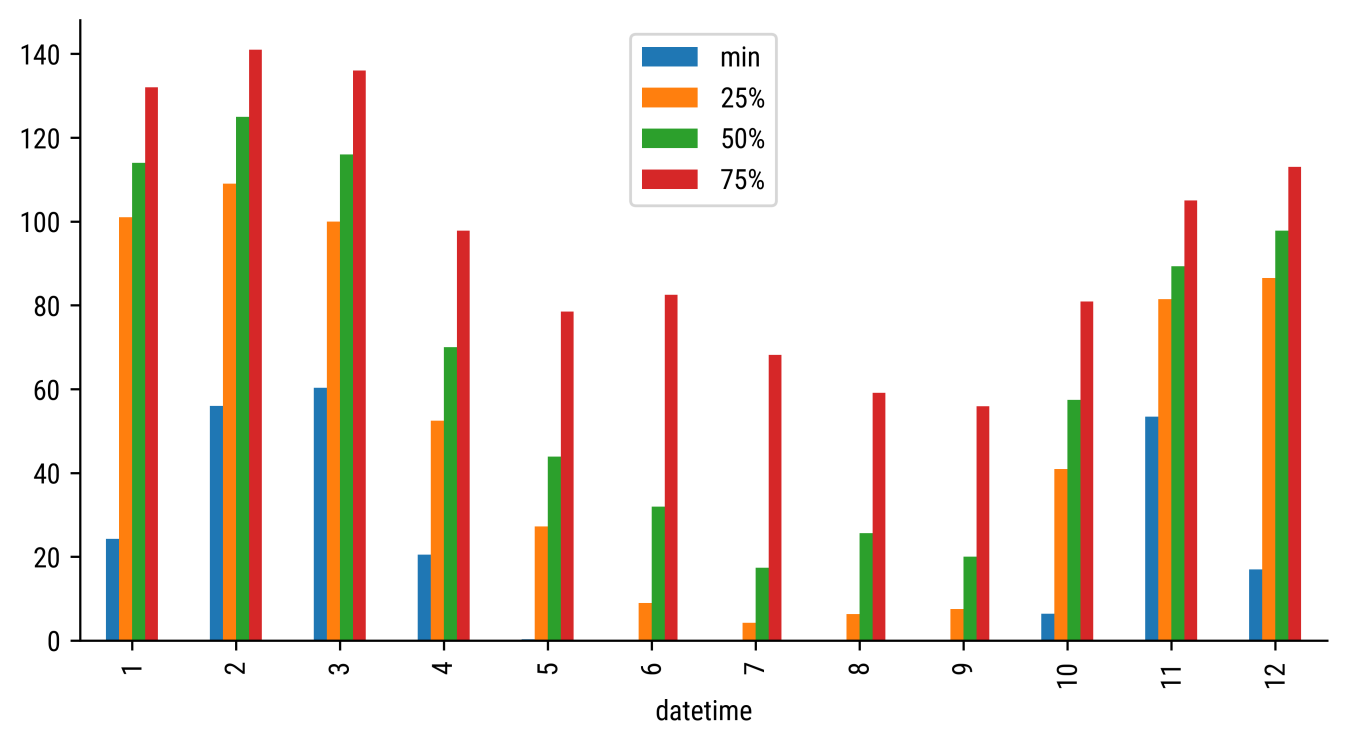
```
(dd
  .groupby(dd.index.month)
  ['cfs']
  .describe()
  ['mean']
  .plot.bar()
)
```



Visualization of monthly average of flow of Dirty Devil river.

We can also plot a line plot of each of the quantiles (I'm not showing the maximum value because it has so many outliers, it blows out the y-axis):

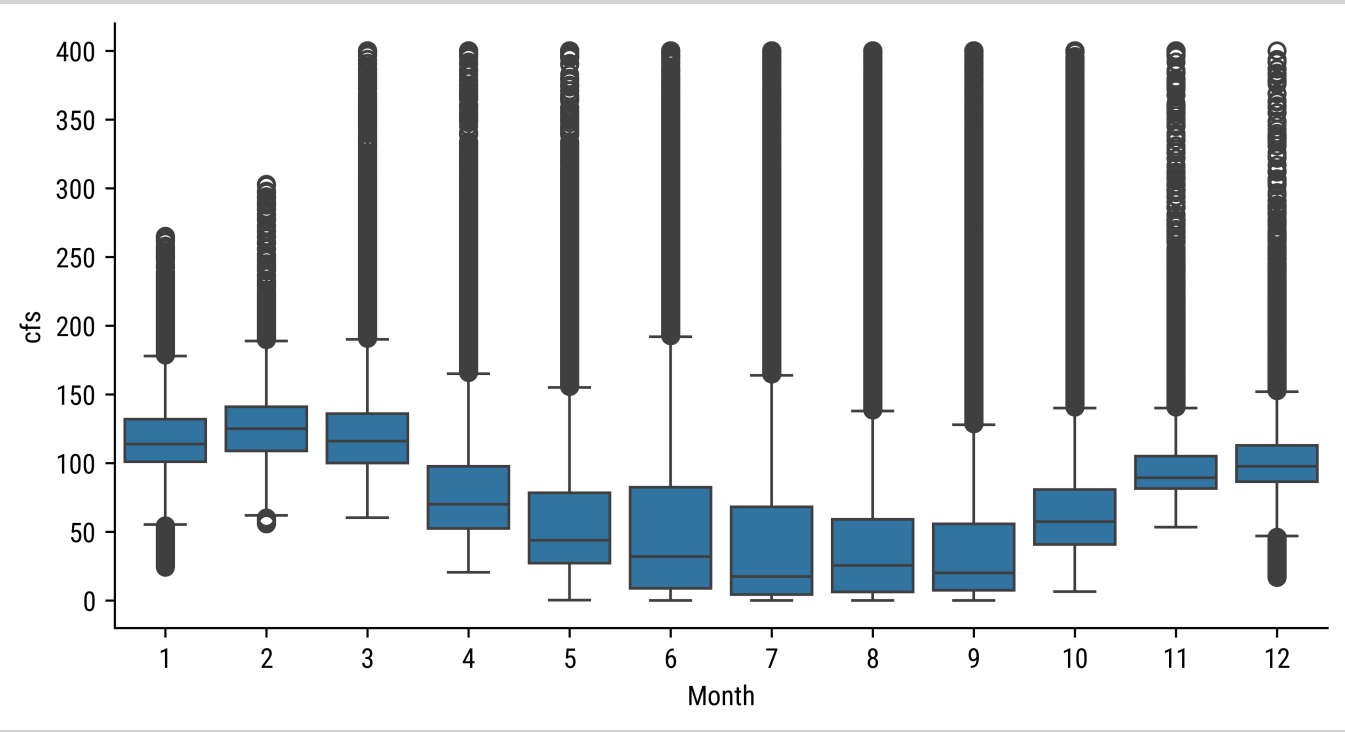
```
(dd
  .groupby(dd.index.month)
  ['cfs']
  .describe()
  .loc[:, 'min':'75%']
  .plot.bar()
)
```



Visualization of monthly quantiles of flow of Dirty Devil river.

To get much fancier, we could leverage the pandas `.boxplot` method, but at that point, I would prefer using Seaborn[5], which is built on top of Matplotlib and pandas provide a lot of power. I'm going to use the Seaborn `boxplot` function, and pass in clipped measurements to the `data` parameter. We must also specify what we plot in the x and y axes. I create a column from the index with the month data (and rename it from *datetime* to *Month*) and the *cfs* column for x and y, respectively:

```
import seaborn as sns
sns.boxplot(data=dd.assign(cfs=dd.cfs.clip(upper=400)),
    x=dd.index.month.rename('Month'), y='cfs')
```



Boxplot of monthly quantiles of the flow of Dirty Devil River.

Plots such as these can give us an understanding of the monthly patterns we see in the data.

Resampling Data

We explored resampling in the series section, but I want to show some of the power you get by using offset aliases. We will use the flow data from the Dirty Devil dataset to dive into resampling. This data has information sampled to 15-minute intervals:

```
>>> dd.cfs
datetime
2001-05-07 01:00:00-06:00    71.0
2001-05-07 01:15:00-06:00    71.0
2001-05-07 01:30:00-06:00    71.0
2001-05-07 01:45:00-06:00    70.0
2001-05-07 02:00:00-06:00    70.0
                             ...
2020-09-28 08:30:00-06:00    9.53
```

```
2020-09-28 08:45:00-06:00     9.2
2020-09-28 09:00:00-06:00     9.2
2020-09-28 09:15:00-06:00     9.2
2020-09-28 09:30:00-06:00     9.2
Name: cfs, Length: 539305, dtype: double[pyarrow]
```

Let's aggregate this information from a 15-minute interval to a daily interval. Because the index has date information in it, we can use `.resample` in combination with `'D'` (daily) as the offset alias. I am going to use `.median` as the aggregation method because the flow data is heavily skewed:

```
>>> print(dd
...   .resample('D')
...   .median(numeric_only=True)
... )
                           site_no   cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0  71.5         <NA>
2001-05-08 00:00:00-06:00  9333500.0  69.0         <NA>
2001-05-09 00:00:00-06:00  9333500.0  63.5         <NA>
2001-05-10 00:00:00-06:00  9333500.0  55.0         <NA>
2001-05-11 00:00:00-06:00  9333500.0  55.0         <NA>
...                              ...   ...          ...
2020-09-24 00:00:00-06:00  9333500.0  9.53         6.16
2020-09-25 00:00:00-06:00  9333500.0  10.2         6.18
2020-09-26 00:00:00-06:00  9333500.0  10.9          6.2
2020-09-27 00:00:00-06:00  9333500.0  10.2         6.18
2020-09-28 00:00:00-06:00  9333500.0  9.53         6.16

[7085 rows x 3 columns]
```

Rules with Offset Aliases

We could also provide a numeric *rule* before the alias if we wanted to combine multiple days. You can insert a number before the offset alias. In this example, we will aggregate every two days using the offset alias `'2D'`. Pay attention to the index of the result:

```
>>> print(dd
...   .resample('2D')
...   .median(numeric_only=True)
... )
                           site_no   cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0  69.0         <NA>
```

```
2001-05-09 00:00:00-06:00  9333500.0  56.0         <NA>
2001-05-11 00:00:00-06:00  9333500.0  54.0         <NA>
2001-05-13 00:00:00-06:00  9333500.0  47.0         <NA>
2001-05-15 00:00:00-06:00  9333500.0  54.0         <NA>
...                              ...   ...          ...
2020-09-20 00:00:00-06:00  9333500.0  6.83         6.07
2020-09-22 00:00:00-06:00  9333500.0  7.68          6.1
2020-09-24 00:00:00-06:00  9333500.0  9.86         6.17
2020-09-26 00:00:00-06:00  9333500.0  10.5         6.19
2020-09-28 00:00:00-06:00  9333500.0  9.53         6.16

[3543 rows x 3 columns]
```

Combining Offset Aliases

We can also combine offset aliases. If we want to aggregate at the three-day, 2-hour, and 10-minute intervals, we can combine all of these rules with the offset aliases into a single string:

```
>>> print(dd
...   .resample('3D2h10min')
...   .median(numeric_only=True)
... )
                           site_no    cfs  gage_height
datetime
2001-05-07 00:00:00-06:00  9333500.0   67.0         <NA>
2001-05-10 02:10:00-06:00  9333500.0   55.0         <NA>
2001-05-13 04:20:00-06:00  9333500.0   49.0         <NA>
2001-05-16 06:30:00-06:00  9333500.0   50.0         <NA>
2001-05-19 08:40:00-06:00  9333500.0   46.0         <NA>
...                              ...    ...          ...
2020-09-14 13:20:00-06:00  9333500.0   5.79         6.03
2020-09-17 15:30:00-06:00  9333500.0   6.04         6.04
2020-09-20 17:40:00-06:00  9333500.0   7.11         6.08
2020-09-23 19:50:00-06:00  9333500.0  10.03        6.175
2020-09-26 22:00:00-06:00  9333500.0   9.86         6.17

[2293 rows x 3 columns]
```

Anchored Offset Aliases

Some of the frequencies in offset aliases allow you to modify when the window for the frequency ends. You can use this operation on a weekly,

quarterly, or yearly frequency. Note that the default quarter ends in March, June, September, and December:

```
>>> print(dd
...   .resample('QE')
...   .median(numeric_only=True)
... )
                            site_no    cfs  gage_height
datetime
2001-06-30 00:00:00-06:00  9333500.0   44.0         <NA>
2001-09-30 00:00:00-06:00  9333500.0   27.0         <NA>
2001-12-31 00:00:00-07:00  9333500.0   85.0         <NA>
2002-03-31 00:00:00-07:00  9333500.0  122.0         <NA>
2002-06-30 00:00:00-06:00  9333500.0   46.0         <NA>
...                              ...    ...          ...
2019-09-30 00:00:00-06:00  9333500.0   13.3         6.21
2019-12-31 00:00:00-07:00  9333500.0   92.1         6.75
2020-03-31 00:00:00-06:00  9333500.0  126.0         6.99
2020-06-30 00:00:00-06:00  9333500.0   23.2         6.55
2020-09-30 00:00:00-06:00  9333500.0   5.79         5.96

[78 rows x 3 columns]
```

We can tack on `-JAN` to force the quarters to end in January, April, July, and October:

```
>>> print(dd
...   .resample('QE-JAN')
...   .median(numeric_only=True)
... )
                            site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0         <NA>
2001-10-31 00:00:00-07:00  9333500.0   39.0         <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0         <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0         <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0         <NA>
...                              ...    ...          ...
2019-10-31 00:00:00-06:00  9333500.0   12.8         6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0         6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0         6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9         6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5         5.49

[78 rows x 3 columns]
```

For annual and quarterly offset aliases, you can change the anchoring by using `-JAN`, `-FEB`, ... `-DEC`. For weekly offset aliases, you can change the

anchoring by using `-SUN`, `-MON`, ... `-SAT`.

Resampling to Finer-grain Frequency

Remember, this river flow data is at the 15-minute frequency. If we wanted to have it at a two-minute frequency, we could do the following:

```
>>> print(dd
...    .resample('2min')
...    .median(numeric_only=True)
... )
                             site_no   cfs  gage_height
datetime
2001-05-07 01:00:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:02:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:04:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:06:00-06:00       <NA>  <NA>         <NA>
2001-05-07 01:08:00-06:00       <NA>  <NA>         <NA>
...                              ...   ...          ...
2020-09-28 09:22:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:24:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:26:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:28:00-06:00       <NA>  <NA>         <NA>
2020-09-28 09:30:00-06:00  9333500.0   9.2         6.15

[5100736 rows x 3 columns]
```

You will notice that there is now a bunch of missing data. You will probably want to refer to the missing data section and adopt an appropriate option to deal with it. Below, we interpolate the missing values using a forward fill:

```
>>> print(dd
...    .resample('2min')
...    .median(numeric_only=True)
...    .ffill()
... )
                             site_no   cfs  gage_height
datetime
2001-05-07 01:00:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:02:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:04:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:06:00-06:00  9333500.0  71.0         <NA>
2001-05-07 01:08:00-06:00  9333500.0  71.0         <NA>
...                              ...   ...          ...
2020-09-28 09:22:00-06:00  9333500.0   9.2         6.15
```

```
2020-09-28 09:24:00-06:00  9333500.0   9.2          6.15
2020-09-28 09:26:00-06:00  9333500.0   9.2          6.15
2020-09-28 09:28:00-06:00  9333500.0   9.2          6.15
2020-09-28 09:30:00-06:00  9333500.0   9.2          6.15

[5100736 rows x 3 columns]
```

Grouping a Date Column with pd.Grouper

The `.resample` method is a powerful way to aggregate data with dates in the index. But what if you want to aggregate dataframes by a column with date information? Enter the `pd.Grouper` class.

Here is an anchored offset alias using `.resample` on the Dirty Devil data. It aggregates on quarters that end in January:

```
>>> print(dd
...   .resample('QE-JAN')
...   .median(numeric_only=True)
... )
                            site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0          <NA>
2001-10-31 00:00:00-07:00  9333500.0   39.0          <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0          <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0          <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0          <NA>
...                              ...    ...           ...
2019-10-31 00:00:00-06:00  9333500.0   12.8          6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0          6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0          6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9          6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5          5.49

[78 rows x 3 columns]
```

Assuming that we have a date column that we want to aggregate on (I'm going to move the index into a column, *datetime*), we could perform the same aggregation using `pd.Grouper`. The `key` parameter specifies the column to group on, and the `freq` parameter specifies the offset alias:

```
>>> print(dd
...   .reset_index()
...   .groupby(pd.Grouper(key='datetime', freq='QE-JAN'))
...   .median(numeric_only=True)
```

```
... )
                           site_no    cfs  gage_height
datetime
2001-07-31 00:00:00-06:00  9333500.0   42.0         <NA>
2001-10-31 00:00:00-07:00  9333500.0   39.0         <NA>
2002-01-31 00:00:00-07:00  9333500.0  116.0         <NA>
2002-04-30 00:00:00-06:00  9333500.0   96.0         <NA>
2002-07-31 00:00:00-06:00  9333500.0   13.0         <NA>
...                              ...    ...          ...
2019-10-31 00:00:00-06:00  9333500.0   12.8         6.25
2020-01-31 00:00:00-07:00  9333500.0  116.0         6.84
2020-04-30 00:00:00-06:00  9333500.0  116.0         6.98
2020-07-31 00:00:00-06:00  9333500.0   13.9         6.37
2020-10-31 00:00:00-06:00  9333500.0    0.5         5.49

[78 rows x 3 columns]
```

Chapter Methods

Method	Description
`pd.to_datetime(arg, errors='raise', dayfirst=False, yearfirst=False, utc=False, format=None, exact=True, unit=None, infer_datetime_format False, origin='unix', cache=True)`	Convert an `arg` to a datetime. Not guaranteed to return a `datetime64` type. Use `utc=True` to convert from naive to UTC (tz-aware) time. Specify strftime string with `format`. When parsing time since epoch, set `unit='s'` for seconds.
`s.dt.tz_localize(tz, ambiguous='raise', nonexistent='raise')`	Return a date converted to a timezone. Set `tz=None` to convert to naive time. For ambiguous times (when clocks move back for daylight savings) set to `'infer'` to base on order, array of `True`/`False` for DST, non-DST time, `'NaT'` to leave empty. For nonexistent times (when the clock moves forward) set `nonexistent` to `'shift_forward'`, `'shift_backward'`, `'NaT'`, or timedelta object.
`s.dt.tz_convert(tz)`	Convert from an existing timezone to another timezone. Set `tz=None` to convert to UTC time.

Method	Description
`df.loc`	If you have a dataframe/series with a datetime index, you can slice on partial date strings.
`df.resample(rule, axis=0, closed=None, label=None, convention='start', kind=None, on=None, level=None, origin='start_day')`	Return a resampled dataframe (with a date in the index, or specify the date column with `on`). Set closed to `'right'` to include the right side of the interval (default is `'right'` for `M/A/Q/BM/BQ/W`). Set the label to `'right'` to use the right label for the bucket. Can specify the timestamp to start `origin`.
`df.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0, closed=None, method='single')`	Return a window object to perform aggregations on.
`s.bfill(axis=0, limit=None, downcast=None)`	Backward fill the missing values. Alternate syntax for `s.fillna(method='bfill')`
`s.ffill(axis=0, limit=None, downcast=None)`	Forward fill the missing values. Alternate syntax for `s.fillna(method='ffill')`
`s.interpolate( method='linear', axis=0, limit=None, limit_direction='forward', limit_area=None, downcast=None, **kwargs,)`	Return a series with interpolated values.
`s.fillna(value=None, method=None, axis=0, limit=None, downcast=None)`	Use the `value` (scalar, dict, series) or `method` (`'ffill'`, `'bfill'`, or `'nearest'`) for filling in missing data.
`pd.Grouper(key=None, level=None, freq=None, axis=0, sort=False, closed=None, label=None, convention=None, origin='start_day', offset=None, dropna=True)`	Return a groupby object based on the column (`key`) or date index (`key=None`) and offset alias (`freq`).

Summary

There are many tools to manipulate time-series data in pandas. I recommend combining liberal amounts of visualizations when manipulating the data to validate the results.

Exercises

With a dataset of your choice:

1. Convert a date column from a string to a valid date.
2. Group the data by month names and look at the mean values.
3. Group the data by each month of every year and look at the mean values.
4. Insert the date column in the index and slice out a portion of the rows by date.

---

1. https://nwis.waterdata.usgs.gov/usa/nwis/uv/?cb_00060=on&cb_00065=on-&format=rdb&site_no=09333500&period=&begin_date-=2000-01-01&end_date=2020-09-28

2. https://help.waterdata.usgs.gov/faq/about-tab-delimited-output Also, see this link for a description of the spelling of “gage” https://www.usgs.gov/faqs/why-does-usgs-use-spelling-gage-instead-gauge

3. https://github.com/pandas-dev/pandas/issues/38872

4. https://github.com/pandas-dev/pandas/issues/43140

5. https://seaborn.pydata.org/

Combining and Joining Data

Dataframes hold tabular data. Databases hold tabular data. You can perform many of the same operations on dataframes that you do to database tables. In this section, we will look at the theory for joining dataframes. Then, we will look at a real-world example of joining.

Data for Joining

Here are the two tables we will be using for examples: hs

Table of favorite colors

Index	Color	Name
0	Blue	John
1	Blue	George
2	Purple	Ringo

Table of car colors

Index	Car Color	Name
3	Red	Paul
1	Blue	George
2		Ringo

Adding Rows to Dataframes

Let's assume we have two dataframes that we want to combine into a single dataframe, with rows from both. The simplest way to do this is with the

concat function. Below, we create the dataframes:

```
>>> import pandas as pd
>>> import numpy as np
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> df1 = (pd.DataFrame({'name': ['John', 'George', 'Ringo'],
...                      'color': ['Blue', 'Blue', 'Purple']})
...        .astype(string_pa))
>>> df2 = (pd.DataFrame({'name': ['Paul', 'George', 'Ringo'],
...                      'carcolor': ['Red', 'Blue', np.nan]},
...                     index=[3, 1, 2])
...        .astype(string_pa))

>>> print(df1)
     name   color
0    John    Blue
1  George    Blue
2   Ringo  Purple

>>> print(df2)
     name carcolor
3    Paul      Red
1  George     Blue
2   Ringo     <NA>
```

The concat function in the pandas library accepts a list of dataframes to combine. This function is useful when combining multiple files into one dataframe. By default (axis=0), it will stack in the vertical direction. It will find any columns that have the same name and stack them into a single column. In this case, *name* is common to both dataframes:

```
>>> print(pd.concat([df1, df2]))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     <NA>
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
1  George    <NA>     Blue
2   Ringo    <NA>     <NA>
```

Note that .concat preserves index values, so the resulting dataframe has duplicate index values. If you would prefer an error when duplicates appear, you can pass the verify_integrity=True parameter setting:

```
>>> pd.concat([df1, df2], verify_integrity=True)
Traceback (most recent call last):
```

```
  ...
ValueError: Indexes have overlapping values:
  Int64Index([1, 2], dtype='int64')
```

Alternatively, if you would prefer that pandas create new index values for you, pass in `ignore_index=True` as a parameter:

```
>>> print(pd.concat([df1, df2], ignore_index=True))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     <NA>
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
4  George    <NA>     Blue
5   Ringo    <NA>     <NA>
```

Adding Columns to Dataframes

The `concat` function also can align dataframes based on the index values, rather than using the columns. If you set `axis=1` (`axis='columns'`), we get this behavior. I do not use this operation often. Rather, I use `.assign` to create columns. However, here is an example of `concat` along the columns axis:

```
>>> print(pd.concat([df1, df2], axis=1))
     name   color    name carcolor
0    John    Blue    <NA>     <NA>
1  George    Blue  George     Blue
2   Ringo  Purple   Ringo     <NA>
3    <NA>    <NA>    Paul      Red
```

Note that this repeats the `name` column. Using SQL, we can *join* two database tables together based on common columns. If we want to perform a join similar to a database join on a dataframe, we need to use the `.merge` method. We will cover that in the next section.

Joins

Databases have different types of joins. The four common ones include inner, outer, left, and right. However, there is also a cross-join and an anti-join. We will cover those as well. The dataframe has two methods to support these operations, `.join` and `.merge`. I prefer the `.merge` method.

Inner Join

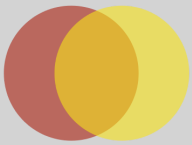
df1

	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

df2

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

```
(df1
  .merge(df2.assign(name=df2.Name))
)
```



	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red

Note every Suzy row matches with every Suzy in df2!

The `.merge` method performs an inner join by default. The resulting dataframe will only have rows where the merge column value exists in both dataframes.

Left Join

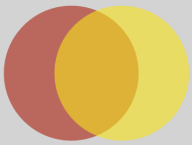
df1

	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

df2

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

```
(df1
  .merge(df2.assign(name=df2.Name), how='left')
)
```



	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red
7	Bob	Fish	nan	nan

Note every Suzy row matches with every Suzy in df2! Bob has missing values

A left join keeps all values from the left merge column (orange and red). The values that are unique to the right dataframe (yellow) are dropped. Note the combinatoric explosion for *Suzy* because each left value is matched with all the values in the right.

Right Join

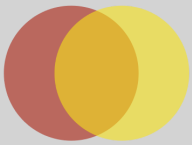
df1

	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

df2

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

```
(df1
  .merge(df2.assign(name=df2.Name), how='right')
)
```



	name	pet	Name	Color
0	Suzy	Dog	Suzy	Black
1	Suzy	Cat	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Cat	Suzy	Blue
4	Suzy	Dog	Suzy	Red
5	Suzy	Cat	Suzy	Red
6	Fred	Dog	Fred	Green
7	Joe	nan	Joe	Yellow
8	Joe	nan	Joe	Blue

Note every Suzy row matches with every Suzy in df2! Joe has missing values

A right join keeps all values from the right merge column (orange and yellow). The values that are unique to the left dataframe (red) are dropped.

Outer Join

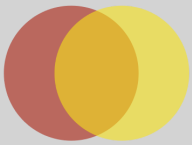
df1

	name	pet
0	Fred	Dog
1	Suzy	Dog
2	Suzy	Cat
3	Bob	Fish

df2

	Name	Color
0	Suzy	Black
1	Suzy	Blue
2	Suzy	Red
3	Fred	Green
4	Joe	Yellow
5	Joe	Blue

```
(df1
  .merge(df2.assign(name=df2.Name), how='outer')
)
```



	name	pet	Name	Color
0	Fred	Dog	Fred	Green
1	Suzy	Dog	Suzy	Black
2	Suzy	Dog	Suzy	Blue
3	Suzy	Dog	Suzy	Red
4	Suzy	Cat	Suzy	Black
5	Suzy	Cat	Suzy	Blue
6	Suzy	Cat	Suzy	Red
7	Bob	Fish	nan	nan
8	Joe	nan	Joe	Yellow
9	Joe	nan	Joe	Blue

Note every Suzy row matches with every Suzy in df2! Bob and Joe have missing values

An outer join keeps all values from the left and right merge columns.

Note

The `.join` method is meant for joining based on the index rather than columns. In practice, I join based on columns instead of index values.

If you want the `.join` method to join based on column values, you need to set that column as the index first:

```
>>> print(df1.set_index('name').join(df2.set_index('name')))
         color carcolor
name
John      Blue      <NA>
George    Blue      Blue
Ringo   Purple      <NA>
```

It is easier to just use the `.merge` method.

The default join type for the `.merge` method is an *inner join*. The `.merge` method looks for common column names in the dataframes it is going to join. The method aligns the values in those columns. If both columns have values that are the same, they are kept along with the remaining columns from both data frames. Rows with values in the aligned columns that only appear in one data frame are discarded:

```
>>> print(df1.merge(df2))  # inner join
     name   color carcolor
0  George    Blue     Blue
1   Ringo  Purple     <NA>
```

When the `how='outer'` parameter setting is passed in, an *outer join* is performed. Again, the method looks for common column names. It aligns the values for those columns and adds the values from the other columns of both data frames. If either dataframe had a value in the field that we joined on that was absent from the other, the new columns are filled with `<NA>`:

```
>>> print(df1.merge(df2, how='outer'))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     Blue
2   Ringo  Purple     <NA>
3    Paul    <NA>      Red
```

To perform a *left join*, pass the `how='left'` parameter setting. A left join keeps only the values from the columns in the dataframe that the `.merge` method is called on. If the other dataframe is missing aligned values, `<NA>` is used to fill in their values:

```
>>> print(df1.merge(df2, how='left'))
     name   color carcolor
0    John    Blue     <NA>
1  George    Blue     Blue
2   Ringo  Purple     <NA>
```

Finally, there is support for a *right join* as well. A right join keeps the values from the dataframe that are passed in as the first parameter of the `.merge` method. If the dataframe that `.merge` was called on has aligned values, they are kept, otherwise `<NA>` is used to fill in the missing values:

```
>>> print(df1.merge(df2, how='right'))
     name   color carcolor
0    Paul    <NA>      Red
```

```
1  George    Blue     Blue
2   Ringo  Purple     <NA>
```

If we want to join on columns that don't have the same name, we can use the `left_on` and `right_on` parameters. We can also specify a subset of columns if we don't want to merge on all of the common columns:

```
>>> print(df1.merge(df2, how='right', left_on='color',
...     right_on='carcolor'))
   name_x color  name_y carcolor
0    <NA>  <NA>    Paul      Red
1    John  Blue  George     Blue
2  George  Blue  George     Blue
3    <NA>  <NA>   Ringo     <NA>
```

Next, we have a *cross-join*. A cross join is a join where every row from the first dataframe is matched with every row from the second dataframe. This is also known as a *Cartesian product*. Pass in `how='cross'` to perform a cross-join:

```
>>> print(df1.merge(df2, how='cross'))
   name_x   color  name_y carcolor
0    John    Blue    Paul      Red
1    John    Blue  George     Blue
2    John    Blue   Ringo     <NA>
3  George    Blue    Paul      Red
4  George    Blue  George     Blue
5  George    Blue   Ringo     <NA>
6   Ringo  Purple    Paul      Red
7   Ringo  Purple  George     Blue
8   Ringo  Purple   Ringo     <NA>
```

Imagine you worked in a restaurant that had three types of curries and four types of protein. You could use a cross-join to generate all the possible combinations of curry and protein.

The `.merge` method has a few other parameters that are useful in practice. The table below lists them:

Table for `.merge` method parameters.

Parameter	Meaning
`on`	Column names to join on. String or list. (Default is the intersection of names).

Parameter	Meaning
`left_on`	Column names for left dataframe. String or list. Used when names don't overlap.
`right_on`	Column names for right dataframe. String or list. Used when names don't overlap.
`left_index`	Join based on left dataframe index. Boolean
`right_index`	Join based on right dataframe index. Boolean

After the next section, I'll discuss one more type of join, the anti-join.

Join Indicators

The `.merge` method can add a column that indicates where the data in the row can come from. If you include the `indicator=True` parameter, pandas will create a column called *_merge*. The `indicator` parameter can also be a string, in which the new column will be the name of the string rather than *_merge*.

The *_merge* column will have the values of `left_only`, `right_only`, or `both` to indicate the row came from the dataframe `.merge` was called on, the data frame passed in, or both of them, respectively:

```
>>> print(df1.merge(df2, how='outer',
...     indicator=True))
     name   color carcolor      _merge
0    John    Blue     <NA>   left_only
1  George    Blue     Blue        both
2   Ringo  Purple     <NA>        both
3    Paul    <NA>      Red  right_only
```

Anti Joins

An *anti-join* is a join where we keep only the rows that don't match between two dataframes. This is useful during data cleaning and validation. You can inspect the rows that don't match and decide what to do with them.

We can perform an anti-join by using the `.merge` method and passing in `how='outer'` and `indicator=True`. Then, we can filter out the rows that have a

value of `both` in the *_merge* column. The values that are *left_only* are the rows in the left dataframe but not the right dataframe. The values that are *right_only* are the rows that were in the right dataframe but not the left dataframe:

```
>>> print(df1.merge(df2, on='name', how='outer', indicator=True)
...   .query('_merge != "both"')
... )
   name color carcolor      _merge
0  John  Blue     <NA>   left_only
3  Paul  <NA>      Red  right_only
```

Merge Validation

The `.merge` method recently added a useful option, the `validate` parameter. It will raise a `MergeError` if the join validates a constraint. The constraint can be `'1:1'`, `'1:m'`, or `'m:1'` to ensure that the join keys are indeed one-to-one, one-to-many, or many-to-one. You can also specify `'m:m'` for many to many, but that constraint is always ignored.

In the following example, the left key is *color*, which has non-unique values (many), and the right key is *carcolor*, which is unique (one), so the constraint should be `'m:1'`. If we pass in a wrong constraint, like a one-to-many constraint, the `MergeError` is raised:

```
>>> df1.merge(df2, how='right', left_on='color',
...     right_on='carcolor', validate='1:m')
Traceback (most recent call last):
    ...
pandas.errors.MergeError: Merge keys are not
    unique in left dataset; not a one-to-many merge
```

This parameter is helpful to check that your data looks like you think it should. I recommend validating your data after merges.

Dirty Devil Flow and Weather Data

In the previous section, we discussed the theory behind joining data. In this section, we will look at a concrete example.

Most of the data we have looked at in the book has been delivered in a single CSV file. Sometimes, we have data from multiple sources, and we need to combine them. This section will explore joining a real-world dataset.

In this section, we will revisit the Dirty Devil data. Let's load the flow and gage height data. In this case, we will leave the *datetime* column as a column and not use it for the index:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...     )

>>> dd = tweak_river(df)
>>> print(dd)
                       datetime agency_cd  ...   cfs gage_height
0      2001-05-07 01:00:00-06:00      USGS  ...  71.0        <NA>
1      2001-05-07 01:15:00-06:00      USGS  ...  71.0        <NA>
2      2001-05-07 01:30:00-06:00      USGS  ...  71.0        <NA>
3      2001-05-07 01:45:00-06:00      USGS  ...  70.0        <NA>
4      2001-05-07 02:00:00-06:00      USGS  ...  70.0        <NA>
...                          ...       ...  ...   ...         ...
539300 2020-09-28 08:30:00-06:00      USGS  ...  9.53        6.16
539301 2020-09-28 08:45:00-06:00      USGS  ...   9.2        6.15
539302 2020-09-28 09:00:00-06:00      USGS  ...   9.2        6.15
539303 2020-09-28 09:15:00-06:00      USGS  ...   9.2        6.15
539304 2020-09-28 09:30:00-06:00      USGS  ...   9.2        6.15

[539305 rows x 6 columns]
```

I'm also going to load some meteorological data[1] from Hanksville, Utah, a city near the river. We will then combine both datasets to have flow data and temperature and precipitation information in the same dataset.

Some of the interesting columns are:

- *DATE* - Date
- *PRCP* - Precipitation in inches
- *TMIN* - Minimum temperature (F) for day
- *TMAX* - Maximum temperature (F) for day
- *TOBS* - Observed temperature (F) when measurement made

```
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       'hanksville.csv'

>>> temp_df = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> def tweak_temp(df_):
...     return (df_
...             .assign(DATE=pd.to_datetime(df_.DATE)
...                  .dt.tz_localize('America/Denver', ambiguous=False))
...             .loc[:,['DATE', 'PRCP', 'TMIN', 'TMAX', 'TOBS']]
...     )

>>> temp_df = tweak_temp(temp_df)
>>> print(temp_df)
                          DATE  PRCP  TMIN  TMAX  TOBS
0    2000-01-01 00:00:00-07:00  0.02    21    43    28
1    2000-01-02 00:00:00-07:00  0.03    24    39    24
2    2000-01-03 00:00:00-07:00   0.0     7    39    18
3    2000-01-04 00:00:00-07:00   0.0     5    39    25
4    2000-01-05 00:00:00-07:00   0.0    10    44    22
...                        ...   ...   ...   ...   ...
6843 2020-09-20 00:00:00-06:00   0.0    46    92    83
6844 2020-09-21 00:00:00-06:00   0.0    47    92    84
6845 2020-09-22 00:00:00-06:00   0.0    54    84    77
6846 2020-09-23 00:00:00-06:00   0.0    47    91    87
6847 2020-09-24 00:00:00-06:00   0.0    43    94    88

[6848 rows x 5 columns]
```

Joining Data

The pandas API provides a function for merging data, `pd.merge`. It also has two methods for joining data, `.join` and `.merge`, that wrap that function. I will use

the `.merge` method.

Let's try to use `.merge` and merge by date. This method will try to merge columns that have the same name. The `dd` dataframe has a *datetime* column, and `temp_df` has a *DATE* column. We can use the `left_on` and `right_on` parameters to help pandas align the data. The `.merge` method tries to do an *inner join* by default. That means that rows with values that are the same in the merge columns will be joined together:

```
>>> print(dd
...  .merge(temp_df, left_on='datetime', right_on='DATE')
... )
                     datetime agency_cd  ...  TMAX TOBS
0    2001-05-08 00:00:00-06:00      USGS  ...    85   58
1    2001-05-09 00:00:00-06:00      USGS  ...    92   64
2    2001-05-10 00:00:00-06:00      USGS  ...    92   67
3    2001-05-11 00:00:00-06:00      USGS  ...    87   60
4    2001-05-12 00:00:00-06:00      USGS  ...    93   72
...                        ...       ...  ...   ...  ...
4968 2020-09-20 00:00:00-06:00      USGS  ...    92   83
4969 2020-09-21 00:00:00-06:00      USGS  ...    92   84
4970 2020-09-22 00:00:00-06:00      USGS  ...    84   77
4971 2020-09-23 00:00:00-06:00      USGS  ...    91   87
4972 2020-09-24 00:00:00-06:00      USGS  ...    94   88

[4973 rows x 11 columns]
```

This appears to have worked but is somewhat problematic. Remember that the `dd` dataset has a 15-minute frequency, but `temp_df` only has daily data, so we only use the value from midnight. We should probably use our resampling skills to calculate the median flow value for each date and then merge. In that case, we will want to use the index of the grouped data to merge, so we specify `left_index=True`:

```
>>> print(dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE')
... )
        site_no   cfs  ...  TMAX TOBS
492   9333500.0  71.5  ...    82   55
493   9333500.0  69.0  ...    85   58
494   9333500.0  63.5  ...    92   64
495   9333500.0  55.0  ...    92   67
496   9333500.0  55.0  ...    87   60
...         ...   ...  ...   ...  ...
```

```
6843  9333500.0  6.83  ...    92   83
6844  9333500.0  6.83  ...    92   84
6845  9333500.0  7.39  ...    84   77
6846  9333500.0  7.97  ...    91   87
6847  9333500.0  9.53  ...    94   88

[6356 rows x 8 columns]
```

That looks better (and gives us a few more rows of data).

Validating Joined Data

Let's validate that we had a one-to-one join, i.e., each date from the flow data matched up with a single date from the temperature data. We can use the `validate` parameter to do this:

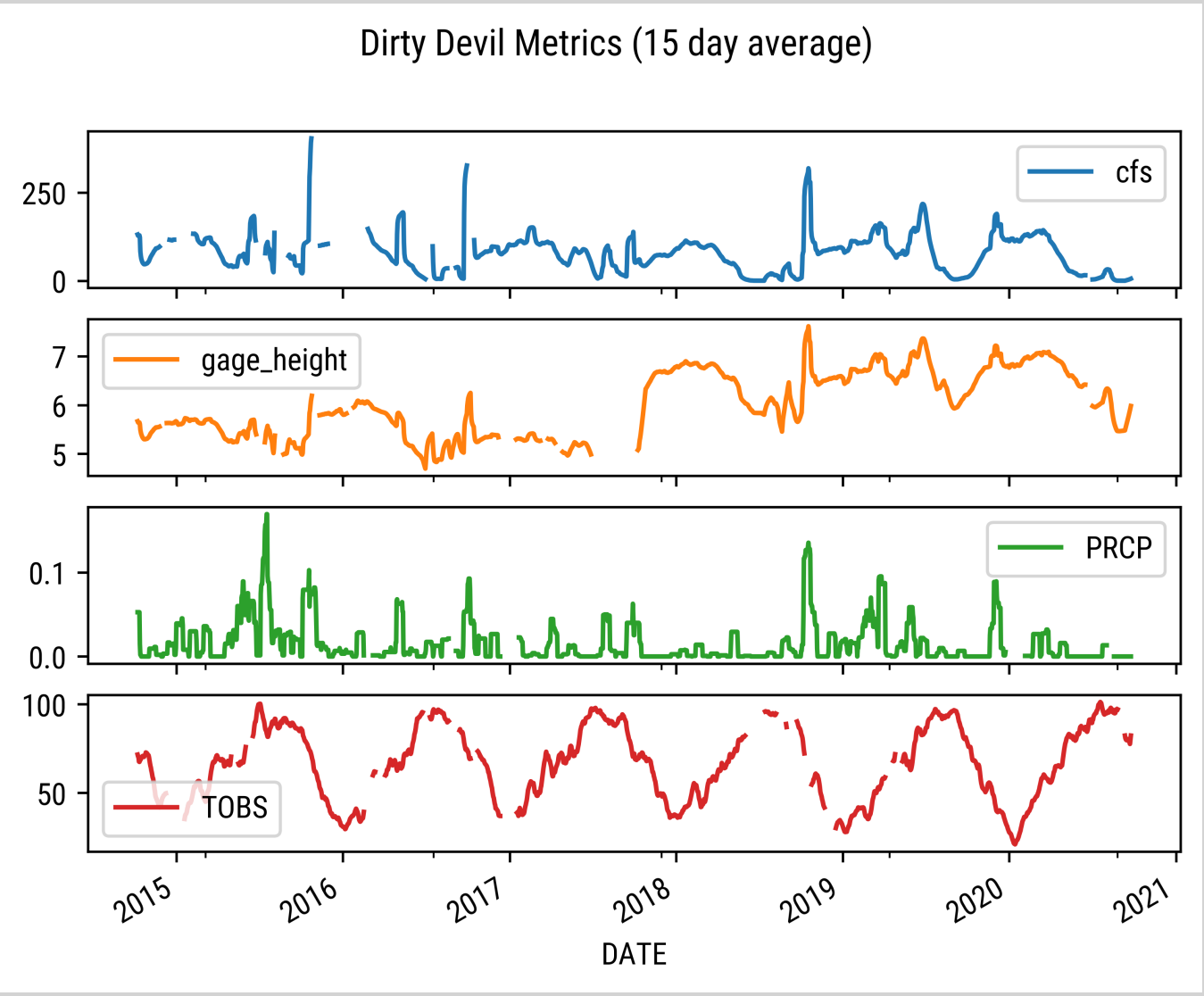
```
>>> print(dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...         validate='1:1')
... )
        site_no   cfs  ...  TMAX TOBS
492   9333500.0  71.5  ...    82   55
493   9333500.0  69.0  ...    85   58
494   9333500.0  63.5  ...    92   64
495   9333500.0  55.0  ...    92   67
496   9333500.0  55.0  ...    87   60
...         ...   ...  ...   ...  ...
6843  9333500.0  6.83  ...    92   83
6844  9333500.0  6.83  ...    92   84
6845  9333500.0  7.39  ...    84   77
6846  9333500.0  7.97  ...    91   87
6847  9333500.0  9.53  ...    94   88

[6356 rows x 8 columns]
```

Because this did not raise a `MergeError`, we know that our data had non-repeating date fields.

Visualization of Merged Data

You know that I'm a big fan of visualization. Let's visualize the merged time series. We will add it to our merge chain, stick the date in the index, pull out the years from 2014 forward, use the *cfs*, *gage_height*, *PRCP*, and *TOBS* columns, interpolate the missing values, do a rolling 15-day average and plot the result in their own subplot:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(dpi=600)
>>> (dd
...  .groupby(pd.Grouper(key='datetime', freq='D'))
...  .median(numeric_only=True)
...  .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...         validate='1:1')
...  .set_index('DATE')
...  .loc['2014':,['cfs', 'gage_height', 'PRCP', 'TOBS']]
...  .rolling(15)
...  .mean()
...  .plot(subplots=True, figsize=(10,8), ax=ax, sharex=True)
... )
>>> fig.suptitle('Dirty Devil Metrics (15 day average)')

<Figure size 3840x2880 with 4 Axes>
```

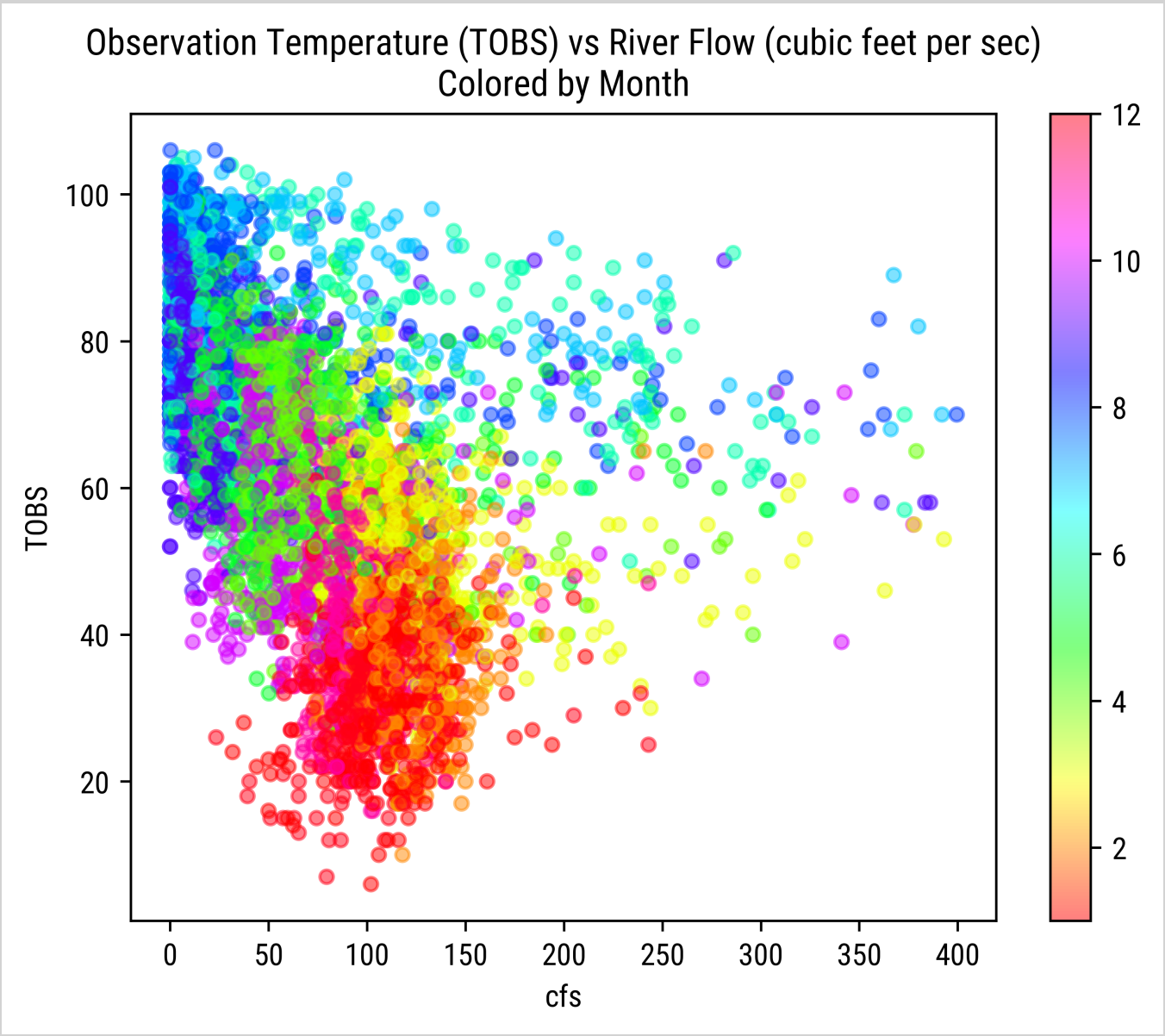


Visualization of 15-day average for the Dirty Devil River metrics.

hs Here's a scatterplot of temperature against river flow. I'm coloring this by month of the year:

```
>>> fig, ax = plt.subplots(dpi=600)
>>> dd2 = (dd
...   .groupby(pd.Grouper(key='datetime', freq='D'))
...   .median(numeric_only=True)
...   .merge(temp_df, left_index=True, right_on='DATE', how='inner',
...          validate='1:1')
...   .query('cfs < 400')
... )
>>> (dd2
...   .plot.scatter(x='cfs', y='TOBS', c=dd2.DATE.dt.month,
...                 ax=ax, cmap='hsv', alpha=.5)
... )
>>> ax.set_title('Observation Temperature (TOBS) '
```

```
...     'vs River Flow (cubic feet per sec)\nColored by Month')
>>> fig.savefig('img/pandas2/dd-scat.png', dpi=600, bbox_inches='tight')
<Figure size 3840x2880 with 2 Axes>
```



Scatterplot of temperature against river flow, colored by month.

Summary

Data can often have more utility if we combine it with other data. In the '70s, *relational algebra* was invented to describe various joins among tabular data. The `.merge` method of the `DataFrame` lets us apply these operations to tabular data in the pandas world. This chapter described concatenation and the four basic joins that are possible via `.merge`.

Exercises

1. Create a dataframe for employees. It should have:

Index	name	company
0	Fred	AMZN
1	John	GOOG
2	Sally	GOOG
3	Annie	NFLX

Create a dataframe for location. It should have:

Index	ticker	location
0	AMZN	Seattle
1	GOOG	SF

2. What type of join do we need to do to get the location of each employee?
3. How would you validate the join?

---

1. https://www.ncdc.noaa.gov/cdo-web/

Exporting Data

This book has dealt with exploring, tweaking, and visualizing data. In addition, you may need to share data with others. This chapter will explore some of the mechanisms for exporting data.

Dirty Devil Data

In this section, we will revisit the Dirty Devil data. Let's load the flow and gage height data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...      .set_index('datetime')
...     )

>>> dd = tweak_river(df)
```

```
>>> print(dd)
                          agency_cd  site_no tz_cd   cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                             ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

Reading and Writing

There are a bunch of functions in pandas that deal with ingesting data. They all begin with `read_`. Similarly, there are analogous exporting methods on the dataframe object. These exporting methods start with `.to_`. We will talk about the common methods for exporting in this chapter.

Creating CSV Files

The Comma Separated Value (CSV) file is ubiquitous. It has been around since the early 70s. This format has the benefit of being human-readable, and that is about where the benefits end. For a long time, there was no standard

for CSV files. In 2005, a standard was released [1], but the damage was already done. As such, escaping mechanisms, encoding, header handling, and data types all suffer. You can see the pandas developers' attempts to deal with these issues when you look at the interface for the `pd.read_csv` function. It has over 40 parameters!

We can use the `.to_csv` method to write our data to a file. One thing to be aware of is that by default, pandas will write the index values in a CSV, but when reading a CSV, it will create a new index unless we specify a column for the index:

```
>>> dd.to_csv('/tmp/dd.csv')
```

> Note
>
> If you don't provide a filename, `.to_csv` will return the string content that would go into the file rather than writing the file. We will take advantage of that in this book to examine what the export looks like.

Let's look at what the first five lines of the export look like:

```
>>> print(dd.head(5).to_csv())
datetime,agency_cd,site_no,tz_cd,cfs,gage_height
2001-05-07 01:00:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:15:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:30:00-06:00,USGS,9333500,MDT,71.0,
2001-05-07 01:45:00-06:00,USGS,9333500,MDT,70.0,
2001-05-07 02:00:00-06:00,USGS,9333500,MDT,70.0,
```

If we wanted to read this and stick *datetime* in the index, we could use this code:

```
>>> dd2 = pd.read_csv('/tmp/dd.csv', index_col='datetime',
...    dtype_backend='pyarrow', engine='pyarrow')
```

Note that CSV files don't do much type conversion other than trying to convert strings to numbers. You can use the `parse_dates` parameter to attempt to convert the index into proper dates, but I would recommend creating a tweak function and revisiting the section on dealing with timezones to

properly handle this (hint: it will look much like the `tweak_river` function from above).

There are a bunch of optional parameters for exporting CSV files, but I usually don't adjust them.

Reading ZIP Files with CSVs

If a single CSV is zipped up, pandas can read it directly with the `pd.read_csv` function. However, if multiple files are in the zip file, we need to use the `zipfile` module to extract the file and then read it. You could use a function like this:

```
import zipfile

def extract_csv_from_zip(zip_file, csv_file):
    with zipfile.ZipFile(zip_file) as z:
        z.extract(csv_file)

extract_csv_from_zip('/tmp/dd.zip', 'dd.csv')
pd.read_csv('dd.csv')
```

The `pd.read_csv` function can also read from a URL. Other `read_` functions will probably not support reading from a URL or a ZIP file.

Exporting to Excel

Another commonly used option is exporting the data frame to an Excel spreadsheet. The benefits of this method are that the world revolves around Excel. Everyone was taught how to use it in Kindergarten and business schools still teach it today. As such, Excel drives most of the business world.

> Note
>
> You must ensure the `openpyxl` library is installed to use Excel support. Simply installing the pandas library usually will not install full Excel support. Using pip is usually sufficient:

```
$ pip install openpyxl
```

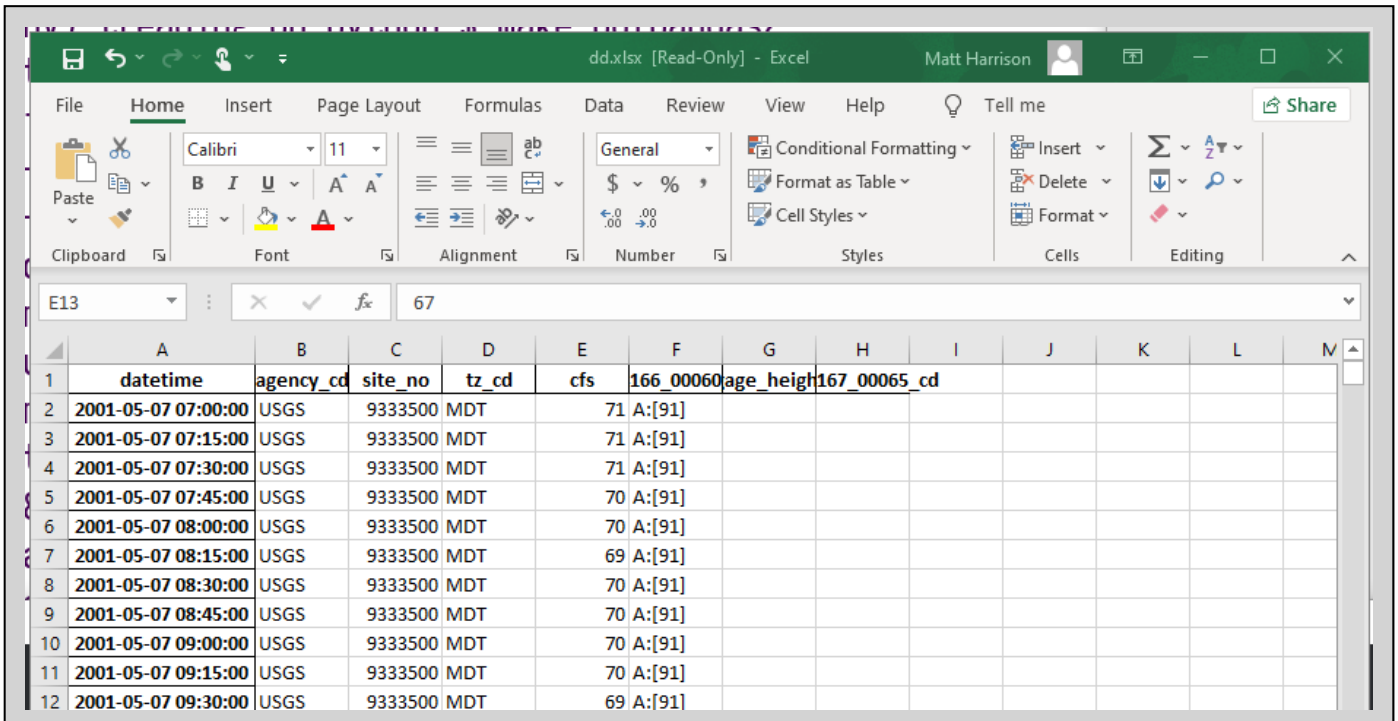
Let's export the data to Excel:

```
>>> dd.to_excel('/tmp/dd.xlsx')
Traceback (most recent call last):
  ...
ValueError: Excel does not support datetimes with timezones.
Please ensure that datetimes are timezone unaware before writing to Excel.
```

Whoops! That didn't quite work. We will need to strip the timezone information before exporting to Excel.

Note that exporting to Excel is a bit slower than writing CSV files. (Also note that Excel reads CSV files, so if you can deal without the limited formatting and type information that pandas inserts in an XLSX file, you might be ok with sending out CSV files to your Excel-junkie friends.):

```
>>> (dd
...  .reset_index()
...  .assign(datetime=lambda df_: df_.datetime.dt.tz_convert(tz=None))
...  .set_index('datetime')
...  .to_excel('/tmp/dd.xlsx')
... )
```



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	datetime	agency_cd	site_no	tz_cd	cfs	166_00060	age_heigh	167_00065_cd					
2	2001-05-07 07:00:00	USGS	9333500	MDT	71	A:[91]							
3	2001-05-07 07:15:00	USGS	9333500	MDT	71	A:[91]							
4	2001-05-07 07:30:00	USGS	9333500	MDT	71	A:[91]							
5	2001-05-07 07:45:00	USGS	9333500	MDT	70	A:[91]							
6	2001-05-07 08:00:00	USGS	9333500	MDT	70	A:[91]							
7	2001-05-07 08:15:00	USGS	9333500	MDT	69	A:[91]							
8	2001-05-07 08:30:00	USGS	9333500	MDT	70	A:[91]							
9	2001-05-07 08:45:00	USGS	9333500	MDT	70	A:[91]							
10	2001-05-07 09:00:00	USGS	9333500	MDT	70	A:[91]							
11	2001-05-07 09:15:00	USGS	9333500	MDT	70	A:[91]							
12	2001-05-07 09:30:00	USGS	9333500	MDT	69	A:[91]							

Excel export of pandas data frame.

Another benefit of Excel is writing a spreadsheet with multiple sheets. In this example, we write the 2010 data on one sheet, and 2011 data to another:

```
>>> writer = pd.ExcelWriter('/tmp/dd2.xlsx')
>>> dd2 = (dd
...  .reset_index()
...  .assign(datetime=lambda df_: df_.datetime.dt.tz_convert(tz=None))
...  .sort_values('datetime')
...  .set_index('datetime')
... )
>>> (dd2
...  .loc['2010':'2010-12-31']
...  .to_excel(writer, sheet_name='2010')
... )
>>> (dd2
...  .loc['2011':'2011-12-31']
...  .to_excel(writer, sheet_name='2011')
... )
```

Parquet

Parquet is a standard format for folks dealing with large amounts of data. It is a columnar format designed to be fast to read and write. It is also intended to be portable across different languages. It is a binary format, so it is not human-readable.

Let's export the data to Parquet:

```
>>> dd.to_parquet('/tmp/dd.parquet')
```

Let's roundtrip the data to make sure it is the same. As of pandas 2.2, I can't use the `pyarrow` backend when reading [2] because it has timezone information.:

```
>>> dd2 = pd.read_parquet('/tmp/dd.parquet')
>>> dd2.equals(dd)
Traceback (most recent call last)
  ...
ArrowInvalid: Timestamps already have a timezone: 'America/Denver'.
     Cannot localize to 'utc'.
```

Let's see if we can discern the differences between the two data frames.

```
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  string[pyarrow]
[right]: string[pyarrow]
```

This is odd because the types of the *agency_cd* column look the same.

```
>>> dd.agency_cd.dtype
string[pyarrow]

>>> dd2.agency_cd.dtype
string[pyarrow]
```

However, as of pandas 2.2, there are two different `string[pyarrow]` types.

```
>>> type(dd.agency_cd.dtype), type(dd2.agency_cd.dtype)
(pandas.core.dtypes.dtypes.ArrowDtype,
     pandas.core.arrays.string_.StringDtype)
```

If we ignore the types, the data frames are the same:

```
>>> pd.testing.assert_series_equal(dd2.agency_cd, dd.agency_cd,
...                                check_dtype=False)
```

Feather

Here is an option that is a relative newcomer. Feather is a binary file format for persisting columnar data in data frames. This is not a surprise because the creator of pandas works on it. Feather tends to be fast and keeps type information (for the most part). It is also supposed to be supported by other languages if you have to process data in R, Julia, or other languages.

> Note
>
> You will need to install the `feather-format` library to leverage this functionality.

Let's try exporting our data:

```
>>> (dd
...  .to_feather('/tmp/dd.fea')
... )
```

Let's see how this did in preserving our information:

```
>>> dd2 = pd.read_feather('/tmp/dd.fea')
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  string[pyarrow]
[right]: string[pyarrow]
```

Looks like this has the same issues as Parquet. Let's ignore the types:

```
>>> pd.testing.assert_frame_equal(dd2, dd, check_dtype=False)
```

Awesome! It looks like this works. Feather is relatively quick and supports most datatypes.

SQL

You can stick a data frame into a SQL table with the `.to_sql` method. In this example, we will create a SQLite database and insert our data into a table named *dd*.

> Note
>
> You will need to install `sqlalchemy` for SQL functionality.
>
> ```
> % pip install sqlachemy
> ```

```
>>> import sqlite3
>>> con = sqlite3.connect('dd.db')
```

```
>>> dd.to_sql('dd', con, if_exists='replace')
539305
```

Let's read from the database:

```
>>> import sqlalchemy as sa
>>> eng = sa.create_engine('sqlite:///dd.db')
>>> sa_con = eng.connect()
>>> dd2 = pd.read_sql('dd', sa_con, index_col='datetime',
...     dtype_backend='pyarrow')
>>> pd.testing.assert_frame_equal(dd2, dd, check_dtype=False)
Traceback (most recent call last)
  ...
AssertionError: DataFrame.index are different

DataFrame.index classes are different
[left]:  Index([2001-05-07 01:00:00-06:00, 2001-05-07 01:15:00-06:00,
       2001-05-07 01:30:00-06:00, 2001-05-07 01:45:00-06:00,
       2001-05-07 02:00:00-06:00, 2001-05-07 02:15:00-06:00,
       2001-05-07 02:30:00-06:00, 2001-05-07 02:45:00-06:00,
       2001-05-07 03:00:00-06:00, 2001-05-07 03:15:00-06:00,
       ...
       2020-09-28 07:15:00-06:00, 2020-09-28 07:30:00-06:00,
       2020-09-28 07:45:00-06:00, 2020-09-28 08:00:00-06:00,
       2020-09-28 08:15:00-06:00, 2020-09-28 08:30:00-06:00,
       2020-09-28 08:45:00-06:00, 2020-09-28 09:00:00-06:00,
       2020-09-28 09:15:00-06:00, 2020-09-28 09:30:00-06:00],
      dtype='object', name='datetime', length=539305)
[right]: DatetimeIndex(['2001-05-07 01:00:00-06:00',
                '2001-05-07 01:15:00-06:00',
                '2001-05-07 01:30:00-06:00',
                '2001-05-07 01:45:00-06:00',
                '2001-05-07 02:00:00-06:00',
                '2001-05-07 02:15:00-06:00',
                '2001-05-07 02:30:00-06:00',
                '2001-05-07 02:45:00-06:00',
                '2001-05-07 03:00:00-06:00',
                '2001-05-07 03:15:00-06:00',
                ...
                '2020-09-28 07:15:00-06:00',
                '2020-09-28 07:30:00-06:00',
                '2020-09-28 07:45:00-06:00',
                '2020-09-28 08:00:00-06:00',
                '2020-09-28 08:15:00-06:00',
                '2020-09-28 08:30:00-06:00',
                '2020-09-28 08:45:00-06:00',
                '2020-09-28 09:00:00-06:00',
                '2020-09-28 09:15:00-06:00',
                '2020-09-28 09:30:00-06:00'],
```

```
              dtype='datetime64[ns, America/Denver]',
     name='datetime', length=539305, freq=None)

>>> print(dd2)
                     agency_cd  site_no tz_cd    cfs  gage_height
datetime
2001-05-07 01:00:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:15:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:30:00      USGS  9333500   MDT  71.00         <NA>
2001-05-07 01:45:00      USGS  9333500   MDT  70.00         <NA>
2001-05-07 02:00:00      USGS  9333500   MDT  70.00         <NA>
...                       ...      ...   ...    ...          ...
2020-09-28 08:30:00      USGS  9333500   MDT   9.53         6.16
2020-09-28 08:45:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:00:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:15:00      USGS  9333500   MDT   9.20         6.15
2020-09-28 09:30:00      USGS  9333500   MDT   9.20         6.15

[539305 rows x 5 columns]
```

We could read the table from the database, but it was not equal to the original data. Closer inspection reveals that our index with timezone-aware dates was stored with timezone data, but this information was dropped when the data came out from the database.

Here is an example of using the `sqlite3` command-line tool to inspect the database:

```
$ sqlite3 dd.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE IF NOT EXISTS "dd" (
"datetime" TIMESTAMP,
  "agency_cd" TEXT,
  "site_no" INTEGER,
  "tz_cd" TEXT,
  "cfs" REAL,
  "144166_00060_cd" TEXT,
  "gage_height" REAL,
  "144167_00065_cd" TEXT
);
CREATE INDEX "ix_dd_datetime"ON "dd" ("datetime");
sqlite> SELECT * FROM dd LIMIT 1;
2001-05-07 01:00:00-06:00|USGS|9333500|MDT|71.0|A:[91]||
sqlite>
```

If we update the index with timezone information, our dataframe is equal to the original data:

```
>>> pd.testing.assert_frame_equal(dd2
...     .reset_index()
...     .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, utc=True)
...           .dt.tz_convert('America/Denver'))
...     .set_index('datetime'),
...  dd, check_dtype=False
... )
```

JSON

Those who implement backend services often need to serialize data with JavaScript Object Notation (JSON). The pandas library has a `.to_dict` method to format data as a dictionary. It also has a `.to_json` method which supports exporting data formatted as JSON in multiple layouts.

Let's try out `.to_dict` first. While not strictly JSON, they are both dictionary representations (with JSON being serialized as a string):

```
>>> obj = dd.to_dict()
```

The result of `.to_dict` is a dictionary of dictionaries. The outer dictionary is keyed by the column names. The inner dictionary is keyed by the index. The values are the data in the data frame. Here is an example of the export of the first two rows of data:

```
>>> dd.head(2).to_dict()
{'agency_cd': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 'USGS',
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 'USGS'},
 'site_no': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 9333500,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'):
     9333500},
 'tz_cd': {Timestamp('2001-05-07 01:00:00-0600',
     tz='America/Denver'): 'MDT',
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 'MDT'},
 'cfs': {Timestamp('2001-05-07 01:00:00-0600', tz='America/Denver'):
     71.0,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): 71.0},
 'gage_height': {Timestamp('2001-05-07 01:00:00-0600',
```

```
     tz='America/Denver'): None,
  Timestamp('2001-05-07 01:15:00-0600', tz='America/Denver'): None}}
```

There is no corresponding `pd.read_dict` function. Rather, there is a class method on the data frame called `.from_dict`. Let's see how round tripping works with this method:

```
>>> dd2 = pd.DataFrame.from_dict(obj)
>>> pd.testing.assert_frame_equal(dd2, dd)
Traceback (most recent call last)
  ...
AssertionError: DataFrame.index are different

Attribute "names" are different
[left]:  [None]
[right]: ['datetime']
```

It looks like the index name was dropped. Not a big deal, let's fix that and try again:

```
>>> pd.testing.assert_frame_equal((dd2
...   .rename_axis(index='datetime')), dd)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="agency_cd") are different

Attribute "dtype" are different
[left]:  object
[right]: string[pyarrow]
```

Now it is complaining about type mismatches. Remember, I've been focusing on the optimized PyArrow types in this book, but the dataframe from `.from_dict` uses legacy types.

```
>>> dd2.dtypes
agency_cd       object
site_no          int64
tz_cd           object
cfs            float64
gage_height    float64
dtype: object
```

Let's address the types and test for equality again:

```
>>> pd.testing.assert_frame_equal((dd2
...   .rename_axis(index='datetime')
```

```
...   .astype(dict(dd.dtypes))), dd)
```

> Note
>
> Dictionary exports do not support duplicated index names. Unlike `.to_json` (when called with `orient='columns'` which raises a `ValueError`), it will silently drop data.

Ok, now on to `.to_json`. For pyarrow types (as of pandas 2.2), we need to specify the `orient` and `lines` parameter as below. We also need to push the index into a column since the *records* orientation that is required for pyarrow loading doesn't support index entries:

```
>>> dd.reset_index().to_json('/tmp/dd.json.gz', orient='records',
...       index=False, lines=True)
>>> dd2 = (pd.read_json('/tmp/dd.json.gz', orient='records', lines=True,
...                     dtype_backend='pyarrow', engine='pyarrow')
...   .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, utc=True)
...               .dt.tz_convert('America/Denver')))
>>> pd.testing.assert_frame_equal(dd2, dd.reset_index())
Traceback (most recent call last)
  ...
AssertionError: DataFrame.iloc[:, 0] (column name="datetime") are
     different

DataFrame.iloc[:, 0] (column name="datetime") values are different
     (100.0 %)
[index]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
     17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
     34, ...
[left]:  [989218800000, 989219700000, 989220600000, 989221500000,
     989222400000, 989223300000, 989224200000, 989225100000,
     989226000000, 9892...
[right]: [989218800000000000, 989219700000000000, 989220600000000000,
     989221500000000000, 989222400000000000, 989223300000000000,
     9892242000...
```

These are not equal because the dates were stored as integers (milliseconds past the UNIX epoch of 1970). Let's put them back into *America/Denver* dates:

```
>>> dd3 = (dd2
...  .assign(datetime=lambda df_: pd.to_datetime(df_.datetime, unit='ms')
```

```
...           .dt.tz_localize(tz='UTC')
...           .dt.tz_convert('America/Denver'))
...  .set_index('datetime')
... )

>>> print(dd3)
                          agency_cd  site_no tz_cd    cfs  \
datetime
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.00
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.00
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.00
...                             ...      ...   ...    ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT   9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.20
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.20

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

Let's check if they are equal now:

```
>>> pd.testing.assert_frame_equal(dd3, dd)
```

> Note
>
> The `.to_json` method exports dates as epoch integers in strings:
>
> ```
> >>> dd.head()
>                           agency_cd   ... gage_height  144167_00065_cd
> datetime                              ...
> ```

```
2001-05-07 01:00:00-06:00      USGS   ...         NaN                NaN
2001-05-07 01:15:00-06:00      USGS   ...         NaN                NaN
2001-05-07 01:30:00-06:00      USGS   ...         NaN                NaN
2001-05-07 01:45:00-06:00      USGS   ...         NaN                NaN
2001-05-07 02:00:00-06:00      USGS   ...         NaN                NaN

[5 rows x 7 columns]

>>> dd.head().to_json()[:60]
'{"agency_cd":{"989218800000":"USGS","989219700000":"USGS",'
```

When we read the JSON, pandas converts the epoch integers into naive dates (they have no timezone information).

Chapter Methods

Method	Description
`df.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding='utf8', compression='infer', quoting=csv.QUOTE_MINIMAL, quotechar='"', line_terminator=os.linesep, chunksize=None, date_format=None, doublequote=True, escapechar=None, decimal='.', errors='strict', storage_options=None)`	Write to a CSV file (or stdout if not specified). Can specify `float_format` with `'%.3f'` (.1234 to .123).

Method	Description
`pd.read_csv(filepath_or_buffer, sep=',', header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix='', mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, error_bad_lines=None, ...)`	Create a dataframe from a CSV file. Use `sep` to parse files with other delimiters. Use `names` to specify column names. Specify missing numeric values with `na_values`. Set `dayfirst=True` to use dates in a non US-centric environment.

Method	Description
`df.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options=None)`	Write an Excel formatted file or instance `ExcelWriter`.
`pd.ExcelWriter(path, engine=None, date_format=None, datetime_format=None, mode='w', storage_options=None, if_sheet_exists=None, engine_kwargs=None, **kwargs)`	Create a class for writing dataframes into sheets.
`pd.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=False, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, mangle_dupe_cols=True, storage_options=None)`	Create a dataframe from Excel file or dictionary (mapping sheet name to dataframe) if sheet_name is a list.
`df.to_feather(path)`	Write a Feather formatted file.

Method	Description
`pd.read_feather(path, columns=None, use_threads=True, storage_options=None)`	Create a dataframe from a Feather file.
`sqlite3.connect(database, timeout=None, detect_types=None, isolation_level=None, check_same_thread=None, factory=None, cached_statements=None, uri=None)`	Open a connection to a SQLite database. Use `database=':memory:'` to create RAM database.
`sa.create_engine(url, **kwargs)`	Create a SQLAlchemy engine from a database connection string.
`eng.connect()`	Get the database connection from a SQLAlchemy engine.
`df.to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None)`	Create a SQL table with `name` from the dataframe. Store the results in database specified by connection `con`. Can specify `'replace'` or `'append'` for `if_exists`.
`pd.read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None,)`	Create a dataframe from a SQL query.
`df.to_dict(orient='dict', into=dict)`	Serialize a dataframe into a dictionary. Orientation can be `'dict'` (column to dict of index to value), `'list'` (column to list of values), `'series'` (column to series), `'split'` (dictionary with index, columns, and data keys), `'records'` (list of dictionary (column to value)), `'index'` (dictionary of index to dictionary of column to value).

Method	Description
`pd.Dataframe.from_dict(data, orient='columns', dtype=None, columns=None)`	Create a dataframe from a dictionary. Orientation can be `'columns'` (like `'dict'` in `.to_dict`) or `'index'`.
`df.to_json(path_or_buf=None, orient=None, date_format='epoch', double_precision=10, force_ascii=True, date_unit='ms', default_handler=None, lines=False, compression='infer', index=True, indent=None, storage_options=None)`	Serialize a dataframe to JSON. Orientation can be `'columns'` (column to dict of index to value), `'list'` (column to list of values), `'series'` (column to series), `'split'` (dictionary with index, columns, and data keys), `'records'` (list of dictionary (column to value)), `'index'` (dictionary of index to dictionary of column to value), `'data'` (list of values), `'values'` (values array), `'table'` (dictionary of schema and data). Can change date format with `'iso'` (ISO8601).
`pd.read_json(path_or_buf=None, orient=None, typ='frame', dtype=None, convert_axes=None, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding='utf-8', encoding_errors='strict', lines=False, chunksize=None, compression='infer', nrows=None, storage_options=None)`	Create a dataframe from JSON.
`df.round(decimals=0)`	Create a dataframe with decimals rounded to given places.
`df.equals(other)`	Compares two dataframes if they have the same shape and values. Columns should have the same type.

Method	Description
`.to_parquet(path, engine='auto', compression='snappy', index=None, partition_cols=None, storage_options=None, **kwargs)`	Export a dataframe to a Parquet file. Use the pyarrow engine by default if avaliable. Partiction columns can be specified as a list of column names. Storage options are passed to the backend file-system. For example if accessing a file on S3, you can pass `storage_options={'key': 'value'}` for host, port, and other options.
`pd.read_parquet(path, engine='auto', columns=None, storage_options=None, use_nullable_dtypes=None, filesystem=None, filters=None, **kwargs)`	Create a dataframe from a Parquet file. Use the pyarrow engine by default if avaliable. You can specify specific columns to load. Storage options are passed to the backend file-system. For example, if accessing a file on S3, you can pass `storage_options={'key': 'value'}` for host, port, and other options.

Summary

There are many formats for exporting data with pandas. As I keep mentioning in this book, you will want to double-check your data after exporting it to know what is in there. Some formats, like CSV, lose most type information. Others try to preserve it but may get hung up on timezones or rounding issues.

Exercises

With a dataset of your choice:

1. Export the data from a dataframe into a CSV file.
2. Export the data from a dataframe into a SQLite database.
3. Export the data from a dataframe into a Feather file.
4. Export the data from a dataframe into JSON.

1. https://www.ietf.org/rfc/rfc4180.txt

2. https://github.com/pandas-dev/pandas/issues/56282

Styling Dataframes

In this chapter, I will demonstrate how to style a dataframe inside of Jupyter.

Loading the Data

We are going to use the Dirty Devil dataset for this section.

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...      .set_index('datetime')
...     )

>>> dd = tweak_river(df)
>>> print(dd)

                         agency_cd  site_no tz_cd   cfs  \
datetime
```

```
2001-05-07 01:00:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:15:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:30:00-06:00      USGS  9333500   MDT  71.0
2001-05-07 01:45:00-06:00      USGS  9333500   MDT  70.0
2001-05-07 02:00:00-06:00      USGS  9333500   MDT  70.0
...                             ...      ...   ...   ...
2020-09-28 08:30:00-06:00      USGS  9333500   MDT  9.53
2020-09-28 08:45:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:00:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:15:00-06:00      USGS  9333500   MDT   9.2
2020-09-28 09:30:00-06:00      USGS  9333500   MDT   9.2

                           gage_height
datetime
2001-05-07 01:00:00-06:00         <NA>
2001-05-07 01:15:00-06:00         <NA>
2001-05-07 01:30:00-06:00         <NA>
2001-05-07 01:45:00-06:00         <NA>
2001-05-07 02:00:00-06:00         <NA>
...                                ...
2020-09-28 08:30:00-06:00         6.16
2020-09-28 08:45:00-06:00         6.15
2020-09-28 09:00:00-06:00         6.15
2020-09-28 09:15:00-06:00         6.15
2020-09-28 09:30:00-06:00         6.15

[539305 rows x 5 columns]
```

Now that we have the basic data, I will do some aggregations and column creation. See if you can review the following code and determine what it is doing. I'll explain right after showing it, but after going through this book, you should start practicing reading code and making sure that you can understand what it is doing. You will need the sparklines library if you want to follow along.

```
>>> import sparklines
>>> agg_flow = (dd
...  #.resample('M')  # resample .agg doesn't support named aggregations
...  .groupby(pd.Grouper(freq='M'))
...  .agg(cfs=('cfs', 'median'),
...       total_flow=('cfs', lambda ser:(ser*15*60).sum()),
...       gage_height=('gage_height', 'median'),
...       flow_trend=('cfs', lambda ser: sparklines.sparklines(
...            ser
...            .fillna(0)
...            .resample('2D')
...            .median()
...            .fillna(0))
```

```
...             [0])
...       )
...   .assign(quarterly_flow=lambda df_: df_
...               .total_flow
...               .resample('Q')
...               .transform('sum'),
...           percent_quarterly_flow=lambda df2_: df2_
...               .total_flow / df2_.quarterly_flow,
...           off_goal=lambda df3_: df3_.percent_quarterly_flow-.33,
...           cost=lambda df4_: df4_.total_flow * .0002)
... )
>>> print(agg_flow.iloc[:,:4])
                             cfs   total_flow  gage_height  \
datetime
2001-05-31 00:00:00-06:00   47.0  105383700.0         <NA>
2001-06-30 00:00:00-06:00   23.0   17843400.0         <NA>
2001-07-31 00:00:00-06:00   17.0    7781400.0         <NA>
2001-08-31 00:00:00-06:00   52.5  192848220.0         <NA>
2001-09-30 00:00:00-06:00   26.0   42819300.0         <NA>
...                          ...          ...          ...
2020-05-31 00:00:00-06:00  21.25   60721029.0         6.51
2020-06-30 00:00:00-06:00   10.2   24475410.0         6.28
2020-07-31 00:00:00-06:00   10.8   67073337.0         6.05
2020-08-31 00:00:00-06:00   0.32   11042316.0         5.48
2020-09-30 00:00:00-06:00   5.79   10369692.0         6.01

                                flow_trend
datetime
2001-05-31 00:00:00-06:00
2001-06-30 00:00:00-06:00
2001-07-31 00:00:00-06:00
2001-08-31 00:00:00-06:00
2001-09-30 00:00:00-06:00
...                                   ...
2020-05-31 00:00:00-06:00
2020-06-30 00:00:00-06:00
2020-07-31 00:00:00-06:00
2020-08-31 00:00:00-06:00
2020-09-30 00:00:00-06:00

[233 rows x 4 columns]
```

There might have been a curveball in here... the sparklines library. Let's skip that for now and describe the rest of the chain.

Group by the months in the index (note that I'm using named aggregations and that, as the comment states, the result of the `.resample` method does not support named aggregations). For each group, calculate the median of the *cfs*

column, calculate *total_flow* from the the *cfs* column (it is the 15-minute value, so we multiply it by 15 to get the minutes and 60 to get the seconds), and create a *flow_trend* column that uses sparklines.

After grouping, we are going to make some more columns. *quarterly_flow* resamples our monthly data to the quarterly level and sums it up. *percent_quarterly_flow* divides *total_flow* by *quarterly_flow*. The *off_goal* column assumes that each month should contribute 33% of the quarterly water flow and measures how far off we are from that goal. The *cost* column calculates expense, assuming it costs two-hundredths of a cent per cubic foot of water.

Sparklines

A sparkline[1] is a small plot drawn without axes or coordinates created by Edward Tufte. The intent is to show a general trend. The sparklines[2] library in Python is a Unicode bar chart implementation of this idea.

If you have a series of numbers, you can create a Unicode string that represents them:

```
>>> import sparklines
>>> sparklines.sparklines(range(10))
['▁▂▃▄▅▆▇█']
```

So let's revisit this chunk of code:

```
...       flow_trend=('cfs', lambda ser: sparklines.sparklines(
...            ser
...            .resample('2D')
...            .median()
...            .fillna(0))
...            [0])
```

We use the *cfs* column, resample to every two days (remember this series, `ser`, is data for every 15 minutes for a single month), calculate the median value of river flow, and fill in missing values with zero. This gives us a series with the median two-day value. We pass this data into the sparklines library to generate a Unicode bar plot. The sparklines library returns a list with the string inside of it, so we pull the chart out of the list.

The resulting column looks like this:

```
>>> agg_flow.flow_trend
datetime
2001-05-31 00:00:00-06:00
2001-06-30 00:00:00-06:00
2001-07-31 00:00:00-06:00
2001-08-31 00:00:00-06:00
2001-09-30 00:00:00-06:00
                          ...
2020-05-31 00:00:00-06:00
2020-06-30 00:00:00-06:00
2020-07-31 00:00:00-06:00
2020-08-31 00:00:00-06:00
2020-09-30 00:00:00-06:00
Freq: M, Name: flow_trend, Length: 233, dtype: object
```

The .style Attribute

Up to this point, most of the results of our chains have been a series or dataframe. The `.style` attribute of a dataframe allows you to chain, but you can only chain more styling methods, you cannot update the dataframe. If you want to style the output, you should do that as your chain's last step(s).

Formatting

One thing you can do with styling is control the formatting. Let's make the *cost* column show dollar signs, change the format of the *datetime* column, convert *percent_quarterly_flow* to a percentage and add a plus or minus to the *off_goal* column. This is done with the `.format` method.

For each column, we pass a dictionary with the column name as the key and the format string as the value. The format string syntax is the same as the format string syntax for the `.str.format` method. Inside the curly braces, you can specify the format string following a colon. For example, the format string for the *cost* column is `'${:,.0f}'`. The dollar sign will be added to the front of the number, the comma will be added to separate thousands, and the `.0f` will round the number to zero decimal places.

The *datetime* column is formatted with the format string `'{:%Y-%m}/01'`. The curly braces are replaced with the value of the column and the `:%Y-%m` is a special format string that formats the date as `YYYY-MM` and the `/01` adds the day and formats it as `YYYY-MM/01`.

I use a dictionary comprehension because the format strings are the same for the *cfs*, *total_flow*, and *quarterly_flow* columns. The dictionary comprehension creates a dictionary with the column name as the key and the format string as the value. The `**` operator unpacks the dictionary comprehension into the other dictionary.

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
         na_rep='Missing')
)
```

✓ 0.1s

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Changing the style of the columns.

Embedding Bar Plots

Next, we will embed a bar plot in the cell background. We use the `.bar` method for that.

The *cfs* bars are clipped (via `vmax`) to the 95% quantile, otherwise, they don't show up due to outliers. The *off_goal* bars specify two colors to distinguish positive from negative.

```
(agg_flow # HIDECELL
.ffill() # .bar doen't work with NaNs
.reset_index()
.style
# after this we are not working with a dataframe but a "Styler" object
.format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
         'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
         **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']}},
        na_rep='Missing')
.bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
.bar(subset='off_goal', color=['red', 'green'], align='mid')
.highlight_null(color='#fef70c')
.highlight_max(color='lightgreen', axis='index')
)
```

✓ 0.2s Python

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480
8	2002/01/01	136.0	197745300.0	Missing		638525700.0	31.0%	-2.0%	$39,549

Adding bar plots to *cfs* and *off_goal* columns. Highlighting missing and maximum values.

Highlighting

There are a few styling methods to highlight values. You can highlight missing values, the minimum, the maximum, a range, or a quantile range. Our example highlights missing and maximum values.

Heatmaps and Gradients

You can shade the background based on the value of the cell. We demoed this in the cross-tabulation section. Here, we will use a red colormap to color the *cost* column. We will set `vmax` to indicate that anything over $25,000 is over budget.

Depending on the data, you may want to choose a different colormap. For correlations, you want to use a diverging colormap. For positive numeric data, you may consider an increasing or continuous colormap. You can use a Matplotlib colormap. They are shown at the end of the chapter.

Captions

The `.set_caption` allows you to specify text for a caption. This will appear before the dataframe.

CSS Properties

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']},
     na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds', vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color: {"#4589ae" if val > 50 else "#a05cbc"}',
          subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props': 'background-color: pink; font-size:14pt;'}])
)
```

✓ 0.1s

Dirty Devil River Flow

	datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
0	2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
1	2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2	2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
3	2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
4	2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
5	2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
6	2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
7	2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Using `.set_properties` to set CSS properties on the *datetime* column. Notice that the *datetime* column is gray. Using `.map` to set CSS properties on the *cfs* column. Notice that the font and background of *cfs* has changed. Using `.set_table_styles` to set CSS properties on hovering. Notice that when you hover over a cell, the style gets set.

The `.set_properties` method lets you set CSS properties to each cell.

The `.map` method will also let you place CSS properties. You pass in a function that takes the value of the cell and returns a string with the CSS properties for that cell.

Another way to set CSS styling is with the `.set_table_styles` method. This method allows you to specify the selector and its properties.

Stickiness

If you find it annoying to lose the column headers when scrolling down a dataframe or losing the index when scrolling to the side, you are in luck. The `.set_sticky` method will keep the headers in place when scrolling. Note, however, that you should call this method at the end of your chain because you might lose the stickiness if you set some CSS styles after it.

Hiding the Index

```
(agg_flow # HIDECELL
 .ffill() # .bar doen't work with NaNs
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow', 'quarterly_flow']},
     na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds', vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color: {"#4589ae" if val > 50 else "#a05cbc"}',
          subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props': 'background-color: pink; font-size:14pt;'}])
 .set_sticky(axis='columns')
 .hide(axis='index')
)
```

✓ 0.1s

Dirty Devil River Flow

datetime	cfs	total_flow	gage_height	flow_trend	quarterly_flow	percent_quarterly_flow	off_goal	cost
2001/05/01	47.0	105383700.0	Missing		123227100.0	85.5%	52.5%	$21,077
2001/06/01	23.0	17843400.0	Missing		123227100.0	14.5%	-18.5%	$3,569
2001/07/01	17.0	7781400.0	Missing		243448920.0	3.2%	-29.8%	$1,556
2001/08/01	52.5	192848220.0	Missing		243448920.0	79.2%	46.2%	$38,570
2001/09/01	26.0	42819300.0	Missing		243448920.0	17.6%	-15.4%	$8,564
2001/10/01	54.0	134975700.0	Missing		480483900.0	28.1%	-4.9%	$26,995
2001/11/01	104.0	203110200.0	Missing		480483900.0	42.3%	9.3%	$40,622
2001/12/01	115.0	142398000.0	Missing		480483900.0	29.6%	-3.4%	$28,480

Using `.hide` to hide the index.

Finally, you can hide the index. In the image, you can see that we have made the index disappear. Through this book, we have seen that unless you are doing grouping, I recommend keeping essential data out of the index (and then moving it out when you are done with grouping).

If you display a dataframe, removing the index can remove a potential distraction. Use the `.hide` method accessed from the `.style` attribute to hide it when displaying data in Jupyter.

Final Styling Code

The final styling code is shown below.

```
(agg_flow
 .reset_index()
 .style
 # after this we are not working with a dataframe but a "Styler" object
 .format({'cost': '${:,.0f}', 'datetime': '{:%Y/%m}/01',
          'percent_quarterly_flow': '{:.1%}', 'off_goal': '{:.1%}',
          **{col: '{:.1f}' for col in ['cfs', 'total_flow',
                                       'quarterly_flow']}},
    na_rep='Missing')
 .bar(subset='cfs', color='#c07fef', vmax=agg_flow.cfs.quantile(.95))
 .bar(subset='off_goal', color=['red', 'green'], align='mid')
 .highlight_null(color='#fef70c')
 .highlight_max(color='lightgreen', axis='index')
 .background_gradient(subset='cost', axis='index', cmap='Reds',
                      vmin=1_000, vmax=25_000)
 .set_caption('Dirty Devil River Flow')
 .set_properties(**{'background-color': '#999'}, subset='datetime')
 .map(lambda val: f'color: "grey"; opacity: 80%; background-color:
          {"#4589ae" if val > 50 else "#a05cbc"}',
          subset='cfs')
 .set_table_styles([{'selector': 'th:hover', 'props':

                     'background-color: pink; font-size:14pt;'}])
 .set_sticky(axis='columns')
 .hide(axis='index')
)
```

Display Options

Pandas has a few options for displaying dataframes.

You can inspect the default values by printing them off of the `pd.options.display` attribute.

I generally leave these alone, but often, my students ask how to view more rows or columns. The `display.max_rows` option finds the default value for the number of displayed rows.

```
>>> pd.options.display.max_rows
10
```

You can override the default by setting the value to a different number. But I recommend using the context manager `pd.option_context` to change the value temporarily. You do that like this:

```
>>> with pd.option_context('display.max_rows', 4,
...                        'display.max_columns', 2):
...     print(agg_flow)
                             cfs  ...        cost
datetime                          ...
2001-05-31 00:00:00-06:00  47.00  ...  21076.7400
2001-06-30 00:00:00-06:00  23.00  ...   3568.6800
...                          ...  ...         ...
2020-08-31 00:00:00-06:00   0.32  ...   2208.4632
2020-09-30 00:00:00-06:00   5.79  ...   2073.9384

[233 rows x 8 columns]
```

You can use the `pd.describe_option` function to get more information about the option.

```
>>> pd.describe_option('display.max_rows')
display.max_rows : int
    If max_rows is exceeded, switch to truncate view. Depending on
    `large_repr`, objects are either centrally truncated or printed as
    a summary view. 'None' value means unlimited.

    In case python/IPython is running in a terminal and `large_repr`
    equals 'truncate' this can be set to 0 and pandas will auto-detect
    the height of the terminal and print a truncated object which fits
    the screen height. The IPython notebook, IPython qtconsole, or
    IDLE do not run in a terminal and hence it is not possible to do
    correct auto-detection.
    [default: 60] [currently: 10]
```

Display Options in Pandas

Option Name	Description
`chop_threshold`	Controls at what level floating point numbers are truncated (chopped).
`colheader_justify`	Controls the justification of the headers. Values can be ‘right’, ‘left’, etc.
`date_dayfirst`	Displays the date with the day first, such as ‘13/01/2020’ for January 13, 2020.

Option Name	Description
`date_yearfirst`	Displays the date with the year first, such as '2020/01/13' for January 13, 2020.
`encoding`	Sets the default encoding for outputting objects.
`expand_frame_repr`	Whether to stretch the DataFrame representation across the console or not.
`float_format`	Formatter for floating point numbers.
`html`	Settings related to rendering DataFrames as HTML.
`large_repr`	Sets the method of displaying large DataFrames.
`max_categories`	Maximum number of categories displayed when printing a categorical column.
`max_columns`	Maximum number of columns displayed when printing a DataFrame.
`max_colwidth`	Maximum width of each column.
`max_dir_items`	Maximum number of items displayed in the `dir()` of a Pandas object.
`max_info_columns`	Maximum number of columns for which a frame will be considered narrow (for printing info).
`max_info_rows`	Maximum number of rows for which to display a summary (for printing info).
`max_rows`	Maximum number of rows to display.
`max_seq_items`	Maximum number of elements in each sequence (rows, columns) to print.
`memory_usage`	Specifies whether total memory usage of the DataFrame elements (including index) should be shown when printing.
`min_rows`	The minimum number of rows to show in the DataFrame output.
`multi_sparse`	Controls sparsifying of hierarchical indices.

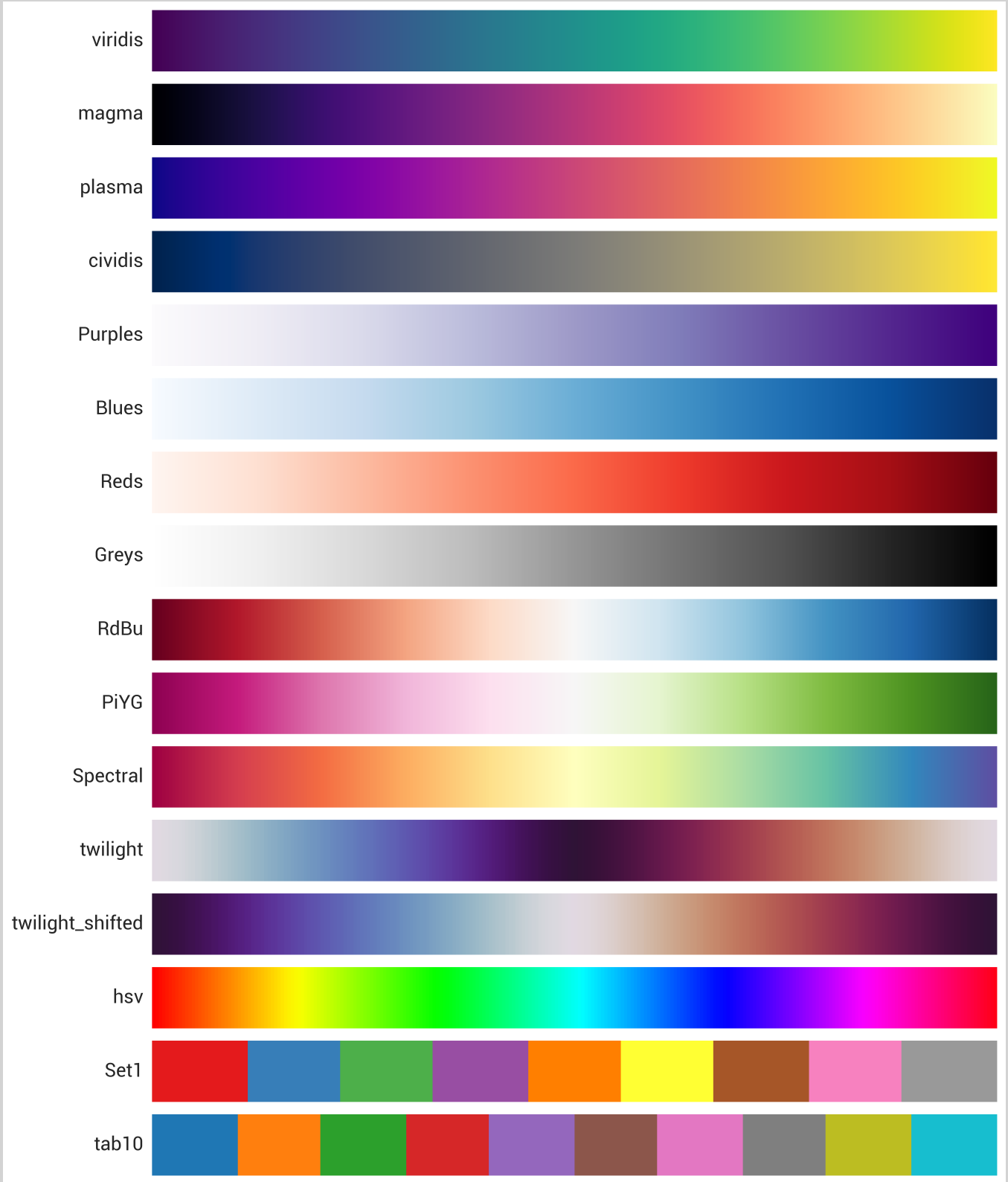
Option Name	Description
notebook_repr_html	Whether to use HTML representation in an IPython notebook.
pprint_nest_depth	Controls the depth to which nested structures are printed.
precision	Sets the precision of floating point numbers.
show_dimensions	Whether to print the dimensions of the DataFrame.
unicode	Controls whether to use Unicode characters to pretty-print DataFrame objects.
width	Width of the display in characters.

Styling Methods

Method	Description
.format(formatter=None, subset=None, na_rep=None, precision=None, decimal='.', thousands=None, escape=None)	Return a Styler. formatter can be a string, a callable that takes a value and returns the string representation, or a dictionary mapping column names to Python format specifiers or callables. subset is a column or list of columns to apply (if not using a dictionary formatter). Use na_rep to specify alternate representation for missing numbers. Use precision to specify floating point decimal places. Use decimal to change decimal separator. Use thousands to specify character to insert for thousands separator. The escape parameter can specify html or latex to provide properly escaped cells.

Method	Description
`.bar( subset=None, axis=0, color='#d65f5f', width=100, align='left', vmin=None, vmax=None)`	Return a `Styler`. Draw a bar chart in cell background. `subset` is a column or list of columns to apply to. If you specify a two-tuple for `color`, the first is for negative values. `width` is the percentage of the cell to use. `align` defaults to `left` side, you can specify `zero` for the center of the cell, or `mid` for center to right aligned if all values are negative or (max-min)/2. Use `vmin` and `vmax` to clip values.
`.highlight_max( null_color='red', subset=None, axis=0, props=None)`	Return a `Styler` that highlights maximum values. You can specify CSS properties with `props`.
`.highlight_null( null_color='red', subset=None, props=None)`	Return a `Styler` that highlights missing values. You can specify CSS properties with `props`.
`.background_gradient( cmap='PuBu', low=0, high=0, axis=0, subset=None, text_color_threshold=0.408, vmin=None, vmax=None, gmap=None)`	Return a `Styler` that highlights background colors based on values. Use `cmap` to specify a Matplotlib colormap.
`.set_caption(caption)`	Return a `Styler`. Create HTML caption. If using LaTex, can specify a tuple with full and short captions.
`.set_properties( subset=None, **kwargs)`	Return a `Styler`. Set CSS properties on each cell. You can specify them as keyword arguments, but will probably need to use an unpacked dictionary since many CSS properties have dashes in them (ie: `** {'background-color': 'red'}`).

Method	Description
`.map( func, subset=None, **kwargs)`	Return a `Styler`. Set CSS properties on each cell. The `func` takes the current value of the cell and returns a string with the CSS properties. You can pass additional arguments to `func` with `kwargs`.
`.set_table_styles( table_styles, axis=0, overwrite=True)`	Return a `Styler`. Set CSS properties on table, columns, rows, or HTML selectors. `table_styles` can be a list of dictionaries (mapping `'selector'` to CSS selector, and `'props'` to CSS properties) or a dictionary (mapping column names (or index names if `axis=1`) to row CSS selectors (a list of the selector and the property).
`.set_sticky( axis=0, pixel_size=None, levels=None)`	Return a `Styler`. Sets columns to sticky if `axis=1`. Set index to stick if `axis=0`. Make sure you call this as one of the last styling operations, otherwise it might not work.
`.hide(subset=None, axis=0, level=None, names=False)`	Return a `Styler`. Hide the index or columns or specified values.



Select Matplotlib colormaps. Continuous (viridis through cividis). Increasing (Purples through Greys). Diverging (RdBu through Spectral). Cyclic (twilight through hsv). Categorical (Set1 and tab10).

Summary

In this chapter, we demonstrated many of the styling features of pandas. There are other features that we didn't explain. Feel free to explore those and see if they will be useful. We also demonstrated how to create a sparkplot as Unicode.

Exercises

With a dataset of your choice:

1. Color the background of the first two columns blue.
2. Format the numeric values by specifying precision and thousands separator.
3. Include a bar plot in a column.
4. Set a background gradient for a column.
5. Make the column headers sticky.

---

1. This creative use of embedding sparklines was inspired by this tweet https://twitter.com/pmbaumgartner/status/1084645440224559104

2. https://github.com/deeplook/sparklines

Debugging Pandas

In this chapter, we will explore various techniques for debugging Pandas.

Checking if Dataframes are Equal

The first technique we will explore is checking whether two dataframes are equal. This is especially useful after serializing and deserializing data and, unfortunately, is a little more complicated than it should be. We can use the `.equals` method to check if two dataframes are equal, but if they are not, diagnosing the problem is hard.

Let's step through an example with our Dirty Devil data:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master'\
...     '/data/dirtydevil.txt'
>>> df = pd.read_csv('data/devilclean.txt',
...                  sep='\t', dtype_backend='pyarrow', engine='pyarrow')
>>> def to_denver_time(df_, time_col, tz_col):
...     return (df_
...      .assign(**{tz_col: df_[tz_col].replace('MDT', 'MST7MDT')})
...      .groupby(tz_col)
...      [time_col]
...      .transform(lambda s: pd.to_datetime(s)
...          .dt.tz_localize(s.name, ambiguous=True)
...          .dt.tz_convert('America/Denver'))
...     )

>>> def tweak_river(df_):
...     return (df_
...      .assign(datetime=to_denver_time(df_, 'datetime', 'tz_cd'))
...      .rename(columns={'144166_00060': 'cfs',
...                       '144167_00065': 'gage_height'})
...      .loc[:, ['datetime', 'agency_cd', 'site_no', 'tz_cd', 'cfs',
...               'gage_height'] ]
...     )
```

```
>>> dd = tweak_river(df)
>>> print(dd)

                       datetime agency_cd  ...   cfs gage_height
0      2001-05-07 01:00:00-06:00      USGS  ...  71.0        <NA>
1      2001-05-07 01:15:00-06:00      USGS  ...  71.0        <NA>
2      2001-05-07 01:30:00-06:00      USGS  ...  71.0        <NA>
3      2001-05-07 01:45:00-06:00      USGS  ...  70.0        <NA>
4      2001-05-07 02:00:00-06:00      USGS  ...  70.0        <NA>
...                          ...       ...  ...   ...         ...
539300 2020-09-28 08:30:00-06:00      USGS  ...  9.53        6.16
539301 2020-09-28 08:45:00-06:00      USGS  ...   9.2        6.15
539302 2020-09-28 09:00:00-06:00      USGS  ...   9.2        6.15
539303 2020-09-28 09:15:00-06:00      USGS  ...   9.2        6.15
539304 2020-09-28 09:30:00-06:00      USGS  ...   9.2        6.15

[539305 rows x 6 columns]
```

Now, let's roundtrip this through JSON and evaluate whether we get the same data back:

```
>>> dd2 = pd.read_json(dd.to_json(), dtype_backend='pyarrow')
>>> dd.equals(dd2)
False
```

Nope, the data is different! Our task is to find out why `dd` and `dd2` are different. The `.equals` method is not particularly helpful in helping us figure out why they aren't equal. Let's dive in a little more.

Let's see if the `.eq` method will help us out. It returns a dataframe of booleans indicating where values are equal:

```
>>> print(dd.eq(dd2))
        datetime agency_cd  ...   cfs gage_height
0          False      True  ...  True        <NA>
1          False      True  ...  True        <NA>
2          False      True  ...  True        <NA>
3          False      True  ...  True        <NA>
4          False      True  ...  True        <NA>
...          ...       ...  ...   ...         ...
539300     False      True  ...  True        True
539301     False      True  ...  True        True
539302     False      True  ...  True        True
539303     False      True  ...  True        True
539304     False      True  ...  True        True
```

```
[539305 rows x 6 columns]
```

We can use the `.sum` or `.mean` trick to quantify the counts or percentages of values that are the same.

```
>>> (dd
...   .eq(dd2)
...   .sum()
... )
datetime            0.0
agency_cd      539305.0
site_no        539305.0
tz_cd          539305.0
cfs            491257.0
gage_height    413649.0
dtype: double[pyarrow]
```

The pandas library has a function hidden away in the `testing` namespace that helps a little, `pd.testing.assert_frame_equal`. This function is meant to be used for the core developers of pandas when developing and testing the library, but let's try it here:

```
>>> pd.testing.assert_frame_equal(dd, dd2)
Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
     name="datetime") are different

Attribute "dtype" are different
[left]:  datetime64[ns, America/Denver]
[right]: timestamp[ns][pyarrow]
```

Ok, it hints that the *datetime* column has different types. As we saw in the JSON serialization section, we lose timezone information when we serialize. Let's address that and try again:

```
>>> from_json = (dd2
...     .assign(datetime=dd2.datetime
...         .dt.tz_localize('UTC')
...         .dt.tz_convert('America/Denver'))
... )
>>> pd.testing.assert_frame_equal(dd, from_json)

Traceback (most recent call last)
  ...
AssertionError: Attributes of DataFrame.iloc[:, 0] (column
```

```
     name="datetime") are different

Attribute "dtype" are different
[left]:  datetime64[ns, America/Denver]
[right]: timestamp[ns, tz=America/Denver][pyarrow]
```

Now it complains that the types are different. We can ask the method to ignore type information:

```
>>> pd.testing.assert_frame_equal(dd, from_json, check_dtype=False)
```

In this case, no assertion is raised, it is quiet! However `.equals` still fails:

```
>>> dd.equals(from_json)
False
```

Let's change the types using `.astypes` and try again.

```
>>> pd.testing.assert_frame_equal(dd, from_json.astype(dict(dd.dtypes)))
```

It doesn't complain, so we should feel confident they are now equal.

For good measure, let's check `.equals`:

```
>>> (dd.equals(from_json.astype(dict(dd.dtypes))))
False
```

That seems weird. The `.equals` method is returning `False`.

Let's try the `check_exact` parameter for `assert_frame_equals` and see if we can see what is different:

```
>>> pd.testing.assert_frame_equal(dd, from_json.astype(dict(dd.dtypes)),
...  check_exact=True
... )
```

This works as well. We will do a little more exploring. I'm going to store my changes to `dd2` in `dd3`.

```
dd3 = from_json.astype(dict(dd.dtypes))
```

Let's use the `.eq` method combined with `.all` to see the differences. It looks like some of the values in the *cfs* column differ.

```
>>> dd.eq(dd3).all()
datetime        True
agency_cd       True
site_no         True
tz_cd           True
cfs            False
gage_height    False
dtype: bool
```

Let's examine those with the `.ne` method. This method will return a boolean array where the values are not equal in a series:

```
>>> print(dd[dd.cfs.ne(dd3.cfs)])
                       datetime agency_cd  ...   cfs gage_height
96246  2007-07-03 19:45:00-06:00      USGS  ...   1.7        <NA>
96247  2007-07-03 20:00:00-06:00      USGS  ...   1.7        <NA>
96248  2007-07-03 20:15:00-06:00      USGS  ...   1.7        <NA>
96249  2007-07-03 20:30:00-06:00      USGS  ...   1.7        <NA>
96250  2007-07-03 20:45:00-06:00      USGS  ...   1.7        <NA>
...                          ...       ...  ...   ...         ...
538678 2020-09-21 21:00:00-06:00      USGS  ...  6.56        6.06
538728 2020-09-22 09:30:00-06:00      USGS  ...  6.56        6.06
538735 2020-09-22 11:15:00-06:00      USGS  ...  6.56        6.06
538739 2020-09-22 12:15:00-06:00      USGS  ...  6.56        6.06
538753 2020-09-22 15:45:00-06:00      USGS  ...  6.56        6.06

[1867 rows x 6 columns]
```

The index here are rows where values are different. Ok, let's look at the values for *cfs* from row label *96246* from both of the datasets:

```
>>> dd.loc[96246].cfs, dd3.loc[96246].cfs
(1.7, 1.7000000000000002)
```

We found a culprit! It looks like we have rounding issues.

Here is a little function I wrote to help diagnose where dataframes are not the same:

```
>>> def cmp_dfs(df1, df2, round_amt=3):
...     diff_cols = set(df1.columns) ^ set(df2.columns)
...     if diff_cols:
...         print(f'Different columns {diff_cols}')
...     if df1.shape != df2.shape:
...         print(f'Different shapes {df1.shape} {df2.shape}')
...     bad = False
...     for col in df1.columns:
```

```
...         s1 = df1[col]
...         s2 = df2[col]
...         if s1.equals(s2):
...             continue
...         bad = True
...         if s1.dtype != s2.dtype:
...             print(f'{col} types differ {s1.dtype} vs {s2.dtype}')
...         if s1.dtype in [float, 'double[pyarrow]']:
...             if s1.round(round_amt).equals(s2.round(round_amt)):
...                 print(f'{col} has rounding differences'
...                       f'{df1[s1.ne(s2)][col].dropna().iloc[0]} '
...                       f'vs {df2[s1.ne(s2)][col].dropna().iloc[0]}')
...         else:
...             diff = (df1
...                     .loc[s1.ne(s2)]
...                     .assign(other=s2)
...                     .loc[[col, "other"]]
...                     .dropna())
...             print(f'{col} differs {diff}')
...     if not bad:
...         print('Same')

>>> cmp_dfs(dd, dd3)
cfs has rounding differences1.7 vs 1.7000000000000002
gage_height has rounding differences3.28 vs 3.2800000000000002
```

Feel free to leverage this function and the others described in this section to discover why your dataframes are not equal.

Hopefully this section gave you some insight into determining what equality really means for your dataframes.

Debugging Chains

In this section, we will explore debugging chains of operations on dataframes or series. I have taught thousands of people pandas during my career. I've also seen a lot of pandas code from clients and students. Almost universally, it is messy code. I get it. I used to write pandas code that way too. Making liberal use of chaining and creating functions to tweak my data has gone a long way toward remedying my ails.

I have been a vocal proponent of chaining on social media. Occasionally, I will hear someone protest that they don't like chaining. When asked why, they usually flounder. Excuses like excess code, copying data (yes, there are copies,

but no more than non-chained pandas), and “hard to debug” are common complaints. I don’t buy excess code. In fact, I think chaining produces less code. The pandas library is an in-memory library that works by copying data. This argument is a moot point. Let’s address the debugging complaint.

I’m going to show a “tweak” function that I created to analyze fuel economy data.

Let’s load the raw data:

```
>>> import pandas as pd
>>> autos = pd.read_csv('https://github.com/mattharrison/datasets/raw/'
...     'master/data/vehicles.csv.zip', dtype_backend='pyarrow',
...     engine='pyarrow')

>>> print(autos)
       barrels08  barrelsA08  ...  phevHwy  phevComb
0      15.695714         0.0  ...        0         0
1      29.964545         0.0  ...        0         0
2      12.207778         0.0  ...        0         0
3      29.964545         0.0  ...        0         0
4      17.347895         0.0  ...        0         0
...          ...         ...  ...      ...       ...
41139  14.982273         0.0  ...        0         0
41140   14.33087         0.0  ...        0         0
41141  15.695714         0.0  ...        0         0
41142  15.695714         0.0  ...        0         0
41143  18.311667         0.0  ...        0         0

[41144 rows x 83 columns]
```

Here is my tweak function:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',

...             'year']
```

```
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                   'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      .loc[:, final_cols]
...     )
```

Say you came across this `tweak_autos` function and wanted to understand what it does. First of all, realize that it is written like a recipe, step by step:

- Limit the columns to `col_cols` to make it easier to deal with.
- Create various columns (`.assign`).
- Convert column types (`.astype`).
- Keep only the columns in `final_cols`.

Haters of chaining say there is no way to debug this. I have a few ways to debug the chain. The first is using comments. I comment out all of the operations and then go through them one at a time. This comes in really handy to visually see what is happening as the chain progresses. Let's look at all four steps with debugging.

First, pulling out the columns. The raw data has 83 columns. I don't want all of those, so I limit the columns to have less distraction during my data

cleanup. I would make a simple function that did just that. Then, I would validate that the function works as intended:

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      .loc[:, orig_cols]
...     )

>>> print(tweak_autos(autos))
       city08  comb08  ...                  createdOn  year
0          19      21  ...  Tue Jan 01 00:00:00 E...  1985
1           9      11  ...  Tue Jan 01 00:00:00 E...  1985
2          23      27  ...  Tue Jan 01 00:00:00 E...  1985
3          10      11  ...  Tue Jan 01 00:00:00 E...  1985
4          17      19  ...  Tue Jan 01 00:00:00 E...  1993
...       ...     ...  ...                        ...   ...
41139      19      22  ...  Tue Jan 01 00:00:00 E...  1993
41140      20      23  ...  Tue Jan 01 00:00:00 E...  1993
41141      18      21  ...  Tue Jan 01 00:00:00 E...  1993
41142      18      21  ...  Tue Jan 01 00:00:00 E...  1993
41143      16      18  ...  Tue Jan 01 00:00:00 E...  1993

[41144 rows x 14 columns]
```

The next step is creating the new columns that I care about. I would add them one at a time to an `.assign` method. Checking with each of them that they work. Here's the first column. Updating the *drive* column, by filling in missing values and then converting it to a category:

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              )
...     )
```

Let's check that it works. I would do this by calling `.value_counts` on the new column:

```
>>> (tweak_autos(autos)
...   .drive
...   .value_counts()
... )
drive
Front-Wheel Drive             14236
Rear-Wheel Drive              13831
4-Wheel or All-Wheel Drive     6648
All-Wheel Drive                3015
4-Wheel Drive                  1460
Other                          1189
2-Wheel Drive                   507
Part-time 4-Wheel Drive         258
Name: count, dtype: int64
```

That's better. Here's the rest of the columns. I'm not going to go over checking each column here. But I hope you get the idea.

The *trany* (transmission) column appears to encode two pieces of data, the number of speeds and whether the automobile is automatic or manual. I'm going to pull out those features into their own columns and then we delete *trany*. The *eng_desc* (engine description) column appears almost freeform. I will create an *indicator column* indicating whether the string *FFS* (fuel feedback system) was in the description. Then, I will discard the *eng_desc* column later.

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
```

```
...                     .astype('int8[pyarrow]'),
...                offset=autos.createdOn
...                       .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                       .replace('EDT', 'EST5EDT'),
...                str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                          autos.createdOn.str.slice(-4)),
...                createdOn=lambda df_: to_tz(df_, 'str_date',
...                          'offset', 'America/New_York'),
...                ffs=autos.eng_dscr.str.contains('FFS')
...               )
...     )
```

The next step is to check the types. I do that with the `.dtypes` attribute.

```
>>> tweak_autos(autos).dtypes
city08          int64[pyarrow]
comb08          int64[pyarrow]
highway08       int64[pyarrow]
cylinders       int64[pyarrow]
displ          double[pyarrow]
                    ...
automatic        bool[pyarrow]
speeds           int8[pyarrow]
offset         string[pyarrow]
str_date       string[pyarrow]
ffs              bool[pyarrow]
Length: 19, dtype: object
```

These types are looking pretty good. In this case, pandas limits the rows shown. So, I don't know if other columns have problematic types. Rather than try and bump that limit, I would look at the value counts to see the unique types.

```
>>> (tweak_autos(autos)
...   .dtypes
...   .value_counts())
int64[pyarrow]                     7
string[pyarrow]                    6
bool[pyarrow]                      2
double[pyarrow]                    1
category                           1
datetime64[ns, America/New_York]   1
int8[pyarrow]                      1
Name: count, dtype: int64
```

From this output, it looks like we have many `int64[pyarrow]` types that I can probably shrink. Let's check by using the `.describe` method and look at the

max row. (In this case, I will transpose it to see a little more data):

```
>>> print(tweak_autos(autos)
...   .describe()
...   .T)
              count          mean  ...      75%     max
city08      41144.0    18.369045  ...     20.0   150.0
comb08      41144.0    20.616396  ...     23.0   136.0
highway08   41144.0    24.504667  ...     28.0   124.0
cylinders   40938.0     5.717084  ...      6.0    16.0
displ       40940.0     3.294238  ...      4.3     8.4
fuelCost08  41144.0  2362.335942  ...   2700.0  7400.0
range       41144.0     0.793506  ...      0.0   370.0
year        41144.0  2001.535266  ...   2011.0  2020.0
speeds      41144.0     5.325029  ...      6.0    20.0

[9 rows x 8 columns]
```

Let's inspect the string columns and see if it makes sense to convert those to categoricals:

```
>>> print(tweak_autos(autos)
...  .select_dtypes('string')
...  .nunique())
eng_dscr     557
make         136
model       4058
trany         37
offset         2
str_date     269
dtype: int64
```

We are going to drop *eng_dscr*, *trany*, and *str_date*. The other columns could be categories.

I'm going to make a dictionary with the types for the columns and pass that into `.astype`:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )
```

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...            'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...            'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...            'make': 'category', 'model': 'category',
...            'cylinders': 'int8[pyarrow]'}
...     return (autos
...      .loc[:, orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...     )
```

Let's check the types now:

```
>>> tweak_autos(autos).dtypes
city08        int16[pyarrow]
comb08        int16[pyarrow]
highway08      int8[pyarrow]
cylinders      int8[pyarrow]
displ        double[pyarrow]
                 ...
automatic      bool[pyarrow]
speeds         int8[pyarrow]
offset       string[pyarrow]
str_date     string[pyarrow]
ffs            bool[pyarrow]
Length: 19, dtype: object
```

That is looking good. Our next step is to drop columns that we don't need. I generally do this in two steps. First, I use the `.drop` method to remove the

columns I don't want. Then, I inspect the columns of the result and change the `.drop` into a `.loc`.

Why go through the hassle? This is a lesson the creator of Polars, Ritchie Vink, taught me. I had never considered it before, but once he said it, it made perfect sense. Ritche said rather than dropping columns, you should select the columns that you want to keep. I guess this is a variation of the *Robustness Principle* or *Postel's Law*: be strict in what you do and tolerant of what others do.

The variation is that you should focus on the columns you want, not those you don't want. This can also protect you in the future. If a dataset adds features and you only worry about dropping specific columns, the dimensions of your data (the number of columns) could change, which is problematic for applications like machine learning.

Because I'm a lazy programmer and don't want to type out all of the columns manually, I use `.drop` and inspect the `.columns` attribute to "type" the columns for me. Then, I copy those columns and replace the `.drop` with `.loc`:

Here's my `.drop`:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...            'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...            'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...            'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
```

```
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...                offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...                str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                          autos.createdOn.str.slice(-4)),
...                createdOn=lambda df_: to_tz(df_, 'str_date',
...                          'offset', 'America/New_York'),
...                ffs=autos.eng_dscr.str.contains('FFS')
...               )
...       .astype(types)
...       .drop(columns=['trany', 'eng_dscr', 'offset', 'str_date'])
...     )
```

Now let's get the columns that I want from this:

```
>>> tweak_autos(autos).columns
Index(['city08', 'comb08', 'highway08', 'cylinders', 'displ',
       'drive', 'fuelCost08', 'make', 'model', 'range', 'createdOn',
       'year', 'automatic', 'speeds', 'ffs'],
      dtype='object')
```

And change the `.drop` to `.loc`

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...               .groupby(tz_offset)
...               [time_col]
...               .transform(lambda s: pd.to_datetime(s)
...                   .dt.tz_localize(s.name, ambiguous=True)
...                   .dt.tz_convert(tz_name))
...             )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...           'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...           'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...           'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...           'displ', 'drive', 'fuelCost08', 'make', 'model', 'range',
...           'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
```

```
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      #.drop(columns=['trany', 'eng_dscr', 'offset', 'str_date'])
...      .loc[:, final_cols]
...     )
```

Let's make sure that it works:

```
>>> print(tweak_autos(autos))
       city08  comb08  ...  speeds    ffs
0          19      21  ...       5   True
1           9      11  ...       5  False
2          23      27  ...       5   True
3          10      11  ...       3   <NA>
4          17      19  ...       5   True
...       ...     ...  ...     ...    ...
41139      19      22  ...       4   True
41140      20      23  ...       5   True
41141      18      21  ...       4   True
41142      18      21  ...       5   True
41143      16      18  ...       4   True

[41144 rows x 15 columns]
```

If you were to come across this function and wanted to debug it, I would comment out the operations in the chain. Then, I would run the function, uncommenting and inspecting the output as I went through the code.

This is effectively what I did to create the chain in the first place.

Don't let a long chain scare you. Look at it as steps of a recipe. If you know what each step is doing, you will be in a good place.

Commenting out chain operations is an effective debugging technique.

Debugging Chains Part II

I won't stop with the debugging techniques. Here's another one that allows you to look at the intermediate state after any method call in a chain. Remember that the `.pipe` method will pass the current state of a dataframe or series into a function. This function can return anything but normally returns a dataframe or a series.

Imagine a function that returns the dataframe (or series) that was passed into it, but it also prints out the representation on the screen. That is what the `show` function below does. This function leverages the `display` function in Jupyter to create an optional HTML header and display the dataframe as HTML rather than a string version:

```
>>> from IPython.display import display, HTML
>>> def show(df_, rows=20, cols=30, title=None):
...     if title:
...         display(HTML(f'<h2>{title}</h2>'))
...     with pd.option_context('display.min_rows', rows,
...                            'display.max_columns', cols):
...         display(df_)
...     return df_
```

Let's stick `show` into the `tweak_autos` function right after the new columns are created but before we convert the types. The image shows the new output.

```
from IPython.display import display, HTML
def show(df_, rows=20, cols=30, title=None):
    if title:
        display(HTML(f'<h2>{title}</h2>'))
    with pd.option_context('display.min_rows', rows, 'display.max_columns', cols):
        display(df_)
    return df_

def tweak_autos(autos):
    cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ', 'drive', 'eng_dscr',
        'fuelCost08', 'make', 'model', 'trany', 'range', 'createdOn', 'year']
    return (autos
     [cols]
     .assign(cylinders=autos.cylinders.fillna(0).astype('int8'),
             displ=autos.displ.fillna(0).astype('float16'),
             drive=autos.drive.fillna('Other').astype('category'),
             automatic=autos.trany.str.contains('Auto'),
             speeds=autos.trany.str.extract(r'(\d)+').fillna('20').astype('int8'),
             tz=autos.createdOn.str.extract(r'\d\d:\d\d ([A-Z]{3}?)').replace('EDT', 'EST5EDT'),
             str_date=(autos.createdOn.str.slice(4,19) + ' ' + autos.createdOn.str.slice(-4)),
             createdOn=lambda df_: to_tz(df_, 'str_date', 'tz', 'US/Eastern'),
             ffs=autos.eng_dscr.str.contains('FFS')
            )
     .pipe(show, rows=2, title='New Cols')
     .astype({'highway08': 'int8', 'city08': 'int16', 'comb08': 'int16', 'fuelCost08': 'int16',
              'range': 'int16',  'year': 'int16', 'make': 'category'})
     .drop(columns=['trany', 'eng_dscr'])
    )

tweak_autos(autos)
```

New Cols

	city08	comb08	highway08	cylinders	displ	drive	eng_dscr	fuelCost08	make	model	trany	range	createdOn	year	automatic	speeds	tz	str_date	ffs
0	19	21	25	4	2.000000	Rear-Wheel Drive	(FFS)	2000	Alfa Romeo	Spider Veloce 2000	Manual 5-spd	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
41143	16	18	21	4	2.199219	4-Wheel or All-Wheel Drive	(FFS,TRBO)	2900	Subaru	Legacy AWD Turbo	Automatic 4-spd	0	2013-01-01 00:00:00-05:00	1993	True	4	EST	Jan 01 00:00:00 2013	True

41144 rows × 19 columns

Out[157]:

	city08	comb08	highway08	cylinders	displ	drive	fuelCost08	make	model	range	createdOn	year	automatic	speeds	tz	str_date	ffs
0	19	21	25	4	2.000000	Rear-Wheel Drive	2000	Alfa Romeo	Spider Veloce 2000	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
1	9	11	14	12	4.898438	Rear-Wheel Drive	3850	Ferrari	Testarossa	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	False
2	23	27	33	4	2.199219	Front-Wheel Drive	1550	Dodge	Charger	0	2013-01-01 00:00:00-05:00	1985	False	5	EST	Jan 01 00:00:00 2013	True
3	10	11	12	8	5.199219	Rear-Wheel Drive	3850	Dodge	B150/B250 Wagon 2WD	0	2013-01-01 00:00:00-05:00	1985	True	3	EST	Jan 01 00:00:00 2013	NaN

Inserting `show` function inside of chain to debug intermediate state.

Another helpful tool during chaining is to inspect the shape of the intermediate dataframes to ensure that you are not accidentally removing all the rows or that you don't have a combinatoric explosion of data following a merge. You could leverage `.pipe` with a function that prints out the shape of the data:

```
>>> def shape(df_):
...     print(df_.shape)
...     return df_
```

Debugging Chains Part III

We are on a roll with debugging. Let's keep going!

Another complaint that people who justify not using chains is that they really want to have the intermediate states of each operation. For example, they might write the `tweak_autos` chain like this:

```
cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
        'drive', 'eng_dscr', 'fuelCost08', 'make', 'model',
        'trany', 'range', 'createdOn', 'year']
autos2 = autos[cols]
cyl_nona = autos.cylinders.fillna(0)
cyl_int8 = cyl_nona.astype('int8')
autos2['cylinders'] = cyl_int8
displ_nona = autos.displ.fillna(0)
displ_float16 = displ_nona.astype('float16')
autos2['displ'] = displ_float16
...
autos2.drop(columns=['trany', 'eng_dscr'], inplace=True)
```

I left out much of the column updating and type changing, but I think you get the point: most users pull out a column, mess with it, and finally stick it back in. Anti-chainers claim that this ability to inspect the state using any of these variables is useful. (Nevermind that the variables sit around in global memory, wasting space.)

Admittedly, the intermediate state might be useful during the development (in fact, you saw that I was inspecting the internal state as I built up the chain), but that utility quickly fades away during analysis and also creates a mess. I don't care about the intermediate state when my chain is done. I care about the output. The intermediate state is just noise.

If you really want the intermediate state of the dataframe, guess what? You can get that by leveraging `.pipe`. Below is a function, `get_var`, that will create a global variable with the contents of the intermediate value of a dataframe. Just shim this function into the chain with `.pipe`:

```
>>> def get_var(df, var_name):
...     globals()[var_name] = df
...     return df
```

Let's use `get_var` to create a variable, `new_cols`, with the state of `tweak_autos` immediately after creating the new columns:

```
>>> def to_tz(df_, time_col, tz_offset, tz_name):
...     return (df_
...             .groupby(tz_offset)
...             [time_col]
...             .transform(lambda s: pd.to_datetime(s)
...                 .dt.tz_localize(s.name, ambiguous=True)
...                 .dt.tz_convert(tz_name))
...            )

>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',
...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                    'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.replace('', 'Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .pipe(get_var, 'new_cols')
...      .astype(types)
...      .loc[:, final_cols]
...     )
```

Let's inspect the intermediate state stored in `new_cols`:

```
>>> tweak_autos(autos)
>>> print(new_cols)
       city08  comb08  ...              str_date    ffs
0          19      21  ...  Jan 01 00:00:00 2013   True
1           9      11  ...  Jan 01 00:00:00 2013  False
2          23      27  ...  Jan 01 00:00:00 2013   True
3          10      11  ...  Jan 01 00:00:00 2013   <NA>
4          17      19  ...  Jan 01 00:00:00 2013   True
...       ...     ...  ...                   ...    ...
41139      19      22  ...  Jan 01 00:00:00 2013   True
41140      20      23  ...  Jan 01 00:00:00 2013   True
41141      18      21  ...  Jan 01 00:00:00 2013   True
41142      18      21  ...  Jan 01 00:00:00 2013   True
41143      16      18  ...  Jan 01 00:00:00 2013   True

[41144 rows x 19 columns]
```

You can use the `.pipe` method to debug intermediate states of chained operations.

Debugging Chains Part IV

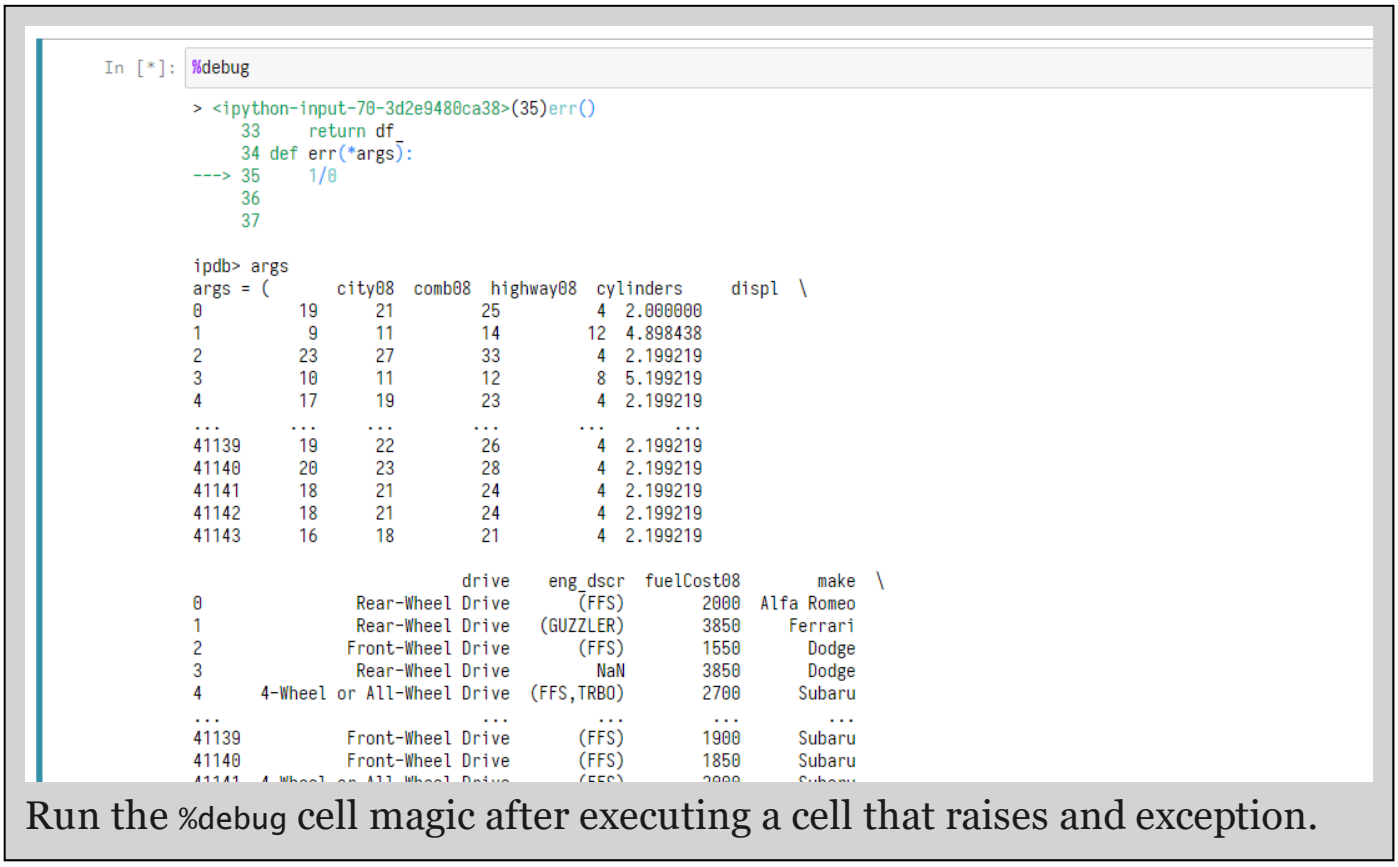
Another option for debugging code in Jupyter is to leverage the pdb debugger. In Jupyter notebook, there are two main options to do this. One is to run the command `%debug` command immediately after encountering an exception. The other way to invoke the debugger is to explicitly invoke the `set_trace` function.

Let's look at the first option. I will insert a link into the chain to call an `err` function that raises an exception. When we run this, it will raise an exception:

```
>>> def err(*args):
...     1/0

>>> def tweak_autos(autos):
...     cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     return (autos
...      [cols]
...      .assign(cylinders=autos.cylinders.fillna(0).astype('int8'),
...              displ=autos.displ.fillna(0).astype('float16'),
...              drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .pipe(err)
...      .astype({'highway08': 'int8', 'city08': 'int16',
...               'comb08': 'int16', 'fuelCost08': 'int16',
...               'range': 'int16',  'year': 'int16',
...               'make': 'category'})
...      .drop(columns=['trany', 'eng_dscr'])
...     )

>>> res = tweak_autos(autos)
```

```
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

This raises an exception. But if you run this in Jupyter, you can drop into a debugger after raising the exception. In a new cell, run the command `%debug`.



Run the `%debug` cell magic after executing a cell that raises and exception.

You are now in the debugger. Here is a brief overview of the pdb commands that I find useful:

- `h` - (help) Show the commands.
- `l` - (list) List code around break.
- `s` - (step) Step into function/method.
- `w` - (where) Show where you are in stack.
- `u` - (up) Move up in the stack.
- `d` - (down) Move down in the stack.
- `c` - (continue) Continue running code.
- `q` - (quit) Quit running code.

Another mechanism to drop into the debugger is to call the `set_trace` function. Replace `err` with this function:

```
>>> from IPython.core.debugger import set_trace
>>> def err(*args):
...     set_trace()
```

> Note
>
> While the debugger is running in Jupyter, no other cells can run. Make sure you type `c` or `q` to finish your debugging session before executing other cells.

Debugging Apply (and Friends)

It can be confusing to keep track of what pandas passes around when you call `.apply`, `.assign`, `.groupby(...) .apply`, `.groupby(...) .agg`, `.groupby(...) .transform`, `.pipe`, and others. What is getting passed in? A series, dataframe, group? One answer is to look at the documentation, which is generally good (although there are some holes). Also, it can be useful to have access to the object being passed around so you can play with it in Jupyter and figure out what you want your `.apply` (or `.groupby(...) .apply` or `.groupby(...) .agg` ...) to do.

We can take a similar approach to debugging with `.pipe` and create a function to help us. The `debug_var` function accepts an item (this is what we want to check). This function will store the item in the `debug_item` variable (we can overwrite this if we desire) for future inspection. Then, the function raises a `DebugException` to prevent further processing. We will pass this function into `.apply`.

Here is the function:

```
>>> class DebugException(Exception):
...     pass

>>> def debug_var(thing, *, name='debug_item', raise_ex=True):
...     globals()[name] = thing
...     if raise_ex:
...         raise DebugException
...     return thing
```

Make sure you have our normal tweak function.

```
>>> def tweak_autos(autos):
...     orig_cols = ['city08', 'comb08', 'highway08', 'cylinders',
...             'displ', 'drive', 'eng_dscr', 'fuelCost08',
...             'make', 'model', 'trany', 'range', 'createdOn',
...             'year']
...     types = {'highway08': 'int8[pyarrow]', 'city08': 'int16[pyarrow]',
...              'comb08': 'int16[pyarrow]', 'fuelCost08': 'int16[pyarrow]',
...              'range': 'int16[pyarrow]',  'year': 'int16[pyarrow]',
...              'make': 'category', 'cylinders': 'int8[pyarrow]'}
...     final_cols = ['city08', 'comb08', 'highway08', 'cylinders', 'displ',

...                   'drive', 'fuelCost08', 'make', 'model', 'range',
...                   'createdOn', 'year', 'automatic', 'speeds', 'ffs']
...     return (autos
...      [orig_cols]
...      .assign(drive=autos.drive.fillna('Other').astype('category'),
...              automatic=autos.trany.str.contains('Auto'),
...              speeds=autos.trany
...                     .str.extract(r'(?P<speeds>\d+)')
...                     .fillna('20')
...                     .astype('int8[pyarrow]'),
...              offset=autos.createdOn
...                     .str.extract(r'\d\d:\d\d (?P<offset>[A-Z]{3}?)')
...                     .replace('EDT', 'EST5EDT'),
...              str_date=(autos.createdOn.str.slice(4,19) + ' ' +
...                        autos.createdOn.str.slice(-4)),
...              createdOn=lambda df_: to_tz(df_, 'str_date',
...                        'offset', 'America/New_York'),
...              ffs=autos.eng_dscr.str.contains('FFS')
...             )
...      .astype(types)
...      .loc[:, final_cols]
...     )
```

Let's use the function to explore how `.apply` works. What gets passed into the `.apply` method? Plug in the `debug_var` function and find out. Let's use it on the Fuel Economy data:

```
>>> autos2 = tweak_autos(autos)
>>> autos2.apply(debug_var, name='this')
Traceback (most recent call last)
  ...
DebugException:
```

What is `this`?

```
>>> this
0        19
1         9
2        23
3        10
4        17
         ..
41139    19
41140    20
41141    18
41142    18
41143    16
Name: city08, Length: 41144, dtype: int16[pyarrow]
```

It looks like `this` is a single column (or series). The `.apply` method will call our function on every single column.

I've removed the stack trace from the exception above, but I try to convince my students that they should try to understand it. In a previous section, we discussed the debugger and how to step through the stack to explore what is happening.

Let's re-run this, but with the `axis=1` parameter to see what gets passed into our function:

```
>>> autos2.apply(debug_var, axis=1)
Traceback (most recent call last):
  ...
DebugException

>>> debug_item
city08                                19
comb08                                21
highway08                             25
cylinders                              4
displ                                2.0
drive                  Rear-Wheel Drive
fuelCost08                          2000
make                          Alfa Romeo
model                 Spider Veloce 2000
range                                  0
createdOn     2013-01-01 00:00:00-05:00
year                                1985
automatic                          False
speeds                                 5
tz                                   EST
str_date           Jan 01 00:00:00 2013
```

```
ffs                                  True
Name: 0, dtype: object
```

It looks like it is passing in a row represented as a series.

Let's try it with `.assign`:

```
>>> (autos2
...  .assign(new_col=debug_var)
... )
Traceback (most recent call last):
  ...
DebugException

>>> debug_item
       city08  comb08  highway08  ...   tz              str_date    ffs
0          19      21         25  ...  EST  Jan 01 00:00:00 2013   True
1           9      11         14  ...  EST  Jan 01 00:00:00 2013  False
2          23      27         33  ...  EST  Jan 01 00:00:00 2013   True
3          10      11         12  ...  EST  Jan 01 00:00:00 2013    NaN
4          17      19         23  ...  EST  Jan 01 00:00:00 2013   True
...       ...     ...        ...  ...  ...                   ...    ...
41139      19      22         26  ...  EST  Jan 01 00:00:00 2013   True
41140      20      23         28  ...  EST  Jan 01 00:00:00 2013   True
41141      18      21         24  ...  EST  Jan 01 00:00:00 2013   True
41142      18      21         24  ...  EST  Jan 01 00:00:00 2013   True
41143      16      18         21  ...  EST  Jan 01 00:00:00 2013   True

[41144 rows x 17 columns]

>>> (autos2
...  .assign(new_col=debug_var)
... )
Traceback (most recent call last)
  ...
DebugException:

>>> print(debug_item)
       city08  comb08  ...  speeds    ffs
0          19      21  ...       5   True
1           9      11  ...       5  False
2          23      27  ...       5   True
3          10      11  ...       3  False
4          17      19  ...       5   True
...       ...     ...  ...     ...    ...
41139      19      22  ...       4   True
41140      20      23  ...       5   True
41141      18      21  ...       4   True
41142      18      21  ...       5   True
```

```
41143      16      18  ...       4   True

[41144 rows x 15 columns]
```

It looks like `debug_item` is the whole dataframe.

Let's try it when we call `.groupby(...).agg` with a dictionary:

```
>>> (autos2.groupby('make').agg({'city08': debug_var}))
Traceback (most recent call last):
  ...
DebugException

>>> debug_item
Series([], Name: city08, dtype: int16)
```

Looks like `debug_item` is the *city08* column.

You get the idea. With the intermediate variable in hand, you should be able to make progress on your analysis.

> Note
>
> In addition to creating a variable, you can also combine this technique with the `%debug` cell magic. This will drop you into a debugger at the point that the exception was raised.

Memory Usage

Because pandas requires that you load your data into RAM, you need to be aware of the size of your data. Because pandas doesn't mutate data (in general), you will need some overhead to work with data. I typically recommend that my clients have 3-10x more memory than the size of the data they are analyzing.

One way to explore the data is to look at the `.info` method. Just remember to use the `memory_usage='deep'` option so you take into account any Python objects the dataframe might use (strings for example):

```
>>> dd.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 539305 entries, 0 to 539304
Data columns (total 6 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   datetime     539305 non-null  datetime64[ns, America/Denver]
 1   agency_cd    539305 non-null  string[pyarrow]
 2   site_no      539305 non-null  int64[pyarrow]
 3   tz_cd        539305 non-null  string[pyarrow]
 4   cfs          493124 non-null  double[pyarrow]
 5   gage_height  433377 non-null  double[pyarrow]
dtypes: datetime64[ns, America/Denver](1), double[pyarrow](2),
     int64[pyarrow](1), string[pyarrow](2)
memory usage: 24.2 MB
```

Another option is to use the 3rd party library *memory-profiler*. You can install this with pip:

```
pip install memory-profiler
```

If you are using Jupyter, you will want to load the extension to access the `%%memit` cell magic. Run this command in a cell in Jupyter:

```
%load_ext memory_profiler
```

Now, you can leverage the `%%memit` cell magic. This will run a cell and track from the operating system's point of view how much memory the process has allocated. It also reports how much the memory usage has grown:

```
>>> %%memit
>>> dd = tweak_river(df)
peak memory: 304.42 MiB, increment: 254.99 MiB
```

If you find that you are using too much memory, consider:

- Sampling rows to limit the data
- Only loading columns you need
- Changing types to more efficient types (i.e., using `'int8[pyarrow]'` instead of `'int64[pyarrow]'` when representing human ages, or using `'category'` for categorical data)
- Acquiring more memory (or using a machine with more memory)

Copy On Write

One of the best features of pandas 2 is called copy on write. In legacy pandas, when you operated on a dataframe, pandas would often make a copy of the data. This was a problem when you had a large dataframe and limited memory. In pandas 2, pandas will only make a copy of the data if you modify the data. This is a vast improvement.

But you need to enable this feature. You can do this by setting `pd.options.mode.copy_on_write` to `True`. Before I do that, I'm going to use the `psutil` library to see how much memory the `tweak_jb` function is using.

Make sure that you install `psutil` (using `pip` or your favorite tool):

```
pip install psutil
```

Let's write a helper function to see how much memory is being used by a process. Because I need to track the previous memory usage, I'm going to create a class:

```
import psutil

import os

class MemoryTracker:
    def __init__(self):
        self.previous_memory = self._get_process_memory()

    def _get_process_memory(self):
        process = psutil.Process(os.getpid())
        memory_info = process.memory_info()
        return memory_info.rss / (1024 ** 2)  # Convert bytes to megabytes

    def __call__(self, df, txt=''):
        current_memory = self._get_process_memory()
        memory_growth = current_memory - self.previous_memory
        print(f"{txt} Process memory usage: {current_memory:.2f} MB"\
              f" (growth: {memory_growth:.2f} MB)")
        self.previous_memory = current_memory
        return df
```

Now, I'm going to instrument the `tweak_jb` function to see how much memory it uses as the chain of operations is applied.

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

>>> import collections

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

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

>>> def predict_col(df, col):
...     df = prep_for_ml(df)
...     missing = df.query(f'~{col}.isna()')
...     cat_idx = [i for i,typ in enumerate(df.drop(columns=[col]).dtypes)
...                if str(typ) == 'object']
...     X = (missing
...          .drop(columns=[col])
...          .values
...         )
...     y = missing[col]
...     model = cb.CatBoostRegressor(iterations=20, cat_features=cat_idx)
...     model.fit(X,y, cat_features=cat_idx)
...     pred = model.predict(df.drop(columns=[col]))
...     return df[col].where(~df[col].isna(), pred)

>>> def tweak_jb_mt(jb, mem_tracker):
...     uniq_cols = get_uniq_cols(jb)
...     return (jb
```

```
...      .pipe(mem_tracker, txt='Start')
...      [uniq_cols]
...      .pipe(mem_tracker, txt='After uniq_cols')
...      .rename(columns=lambda c: c.replace('.', '_'))
...      .pipe(mem_tracker, txt='After rename')
...      .assign(age=lambda df_:df_.age.str.slice(0,2)
...                                .astype('int8[pyarrow]'),
...          are_you_datascientist=lambda df_: df_.are_you_datascientist
...             .replace({'Yes': '1', 'No': '0', '': '0', 'Other': '0'})
...             .astype('bool[pyarrow]')  ,
...          company_size=lambda df_:df_.company_size.replace(
...            {'Just me': '1', '': pd.NA,
...            'Not sure': pd.NA, 'More than 5,000': '5000', '2-10': '2',
...            '11-50':'11','51-500': '51', '501-1,000':'501',
...            '1,001-5,000':'1001'}).astype('int64[pyarrow]'),
...          country_live=lambda df_:df_.country_live.astype('category'),
...          employment_status=lambda df_:df_.employment_status
...             .fillna('Other').astype('category'),
...          is_python_main=lambda df_:df_.is_python_main
...             .astype('category'),
...          team_size=lambda df_:df_.team_size
...             .str.split(r'-', n=1, expand=True)
...             .iloc[:,0].replace('More than 40 people', '41')
...             .where(df_.company_size!=1, '1')
...             .replace('', pd.NA)
...             .astype('int8[pyarrow]'),
...          years_of_coding=lambda df_:df_.years_of_coding.astype(str)
...             .replace('Less than 1 year', '.5')
...             .str.extract(r'(\.?\d+)').astype('float64[pyarrow]'),
...          python_years=lambda df_:df_.python_years
...              .replace('Less than 1 year', '.5')
...              .str.extract(r'(?P<python_years>\.?\d+)')
...              .astype('float64[pyarrow]'),
...          python3_ver=lambda df_:df_.python3_version_most
...              .str.replace('_', '.')
...              .str.extract(r'(?P<python3_ver>\d\.\d)'),
...          use_python_most=lambda df_:df_.use_python_most
...              .fillna('Unknown'))
...      .pipe(mem_tracker, txt='After assign')
...       .assign(
...             team_size=lambda df_:predict_col(df_, 'team_size')
...              .astype(int))
...      .pipe(mem_tracker, txt='After predict_col')
...  .loc[:, ['age', 'are_you_datascientist', 'company_size',
...       'country_live', 'employment_status',
...       'first_learn_about_main_ide', 'how_often_use_main_ide',
...       'ide_main', 'is_python_main', 'job_team', 'main_purposes',
...       'missing_features_main_ide', 'nps_main_ide', 'python_years',
...       'python3_version_most', 'several_projects', 'team_size',
```

```
...       'use_python_most', 'years_of_coding', 'python3_ver']]
...      .pipe(mem_tracker, txt='After loc')
... )

>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...    '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> mt = MemoryTracker()
>>> jb2 = tweak_jb_mt(jb, mt)
Start Process memory usage: 585.14 MB (growth: 0.00 MB)
After uniq_cols Process memory usage: 585.17 MB (growth: 0.03 MB)
After rename Process memory usage: 585.17 MB (growth: 0.00 MB)
After assign Process memory usage: 607.91 MB (growth: 22.73 MB)
Learning rate set to 0.5
0:  learn: 2.9758568    total: 84.5ms   remaining: 1.61s
1:  learn: 2.8841040    total: 95.8ms   remaining: 862ms
2:  learn: 2.8443484    total: 113ms    remaining: 643ms
3:  learn: 2.8105584    total: 128ms    remaining: 511ms
4:  learn: 2.7922983    total: 139ms    remaining: 417ms
5:  learn: 2.7803329    total: 153ms    remaining: 358ms
6:  learn: 2.7756137    total: 178ms    remaining: 330ms
7:  learn: 2.7706510    total: 189ms    remaining: 284ms
8:  learn: 2.7571563    total: 212ms    remaining: 259ms
9:  learn: 2.7564631    total: 231ms    remaining: 231ms
10: learn: 2.7503591    total: 253ms    remaining: 207ms
11: learn: 2.7494745    total: 275ms    remaining: 183ms
12: learn: 2.7481258    total: 285ms    remaining: 154ms
13: learn: 2.7477180    total: 310ms    remaining: 133ms
14: learn: 2.7449738    total: 342ms    remaining: 114ms
15: learn: 2.7409940    total: 359ms    remaining: 89.7ms
16: learn: 2.7408640    total: 382ms    remaining: 67.4ms
17: learn: 2.7365108    total: 403ms    remaining: 44.8ms
18: learn: 2.7346780    total: 416ms    remaining: 21.9ms
19: learn: 2.7287662    total: 443ms    remaining: 0us
After predict_col Process memory usage: 643.61 MB (growth: 35.70 MB)
After loc Process memory usage: 643.62 MB (growth: 0.02 MB)
```

Now, I'm going to enable copy on write and see how much memory is used:

```
>>> pd.options.mode.copy_on_write = True

>>> mt2 = MemoryTracker()
>>> jb3 = tweak_jb_mt(jb, mt2)
Start Process memory usage: 603.28 MB (growth: 0.00 MB)
After uniq_cols Process memory usage: 603.28 MB (growth: 0.00 MB)
After rename Process memory usage: 603.28 MB (growth: 0.00 MB)
After assign Process memory usage: 604.98 MB (growth: 1.70 MB)
Learning rate set to 0.5
0:  learn: 2.9758568    total: 12.4ms   remaining: 236ms
```

```
1:  learn: 2.8841040    total: 35.4ms   remaining: 319ms
2:  learn: 2.8443484    total: 50.7ms   remaining: 287ms
3:  learn: 2.8105584    total: 60.8ms   remaining: 243ms
4:  learn: 2.7922983    total: 76.4ms   remaining: 229ms
5:  learn: 2.7803329    total: 88.6ms   remaining: 207ms
6:  learn: 2.7756137    total: 100ms    remaining: 186ms
7:  learn: 2.7706510    total: 122ms    remaining: 183ms
8:  learn: 2.7571563    total: 135ms    remaining: 164ms
9:  learn: 2.7564631    total: 155ms    remaining: 155ms
10: learn: 2.7503591    total: 170ms    remaining: 139ms
11: learn: 2.7494745    total: 186ms    remaining: 124ms
12: learn: 2.7481258    total: 211ms    remaining: 114ms
13: learn: 2.7477180    total: 227ms    remaining: 97.3ms
14: learn: 2.7449738    total: 254ms    remaining: 84.6ms
15: learn: 2.7409940    total: 271ms    remaining: 67.7ms
16: learn: 2.7408640    total: 295ms    remaining: 52ms
17: learn: 2.7365108    total: 314ms    remaining: 34.9ms
18: learn: 2.7346780    total: 328ms    remaining: 17.3ms
19: learn: 2.7287662    total: 347ms    remaining: 0us
After predict_col Process memory usage: 643.20 MB (growth: 38.22 MB)
After loc Process memory usage: 643.20 MB (growth: 0.00 MB)
```

It looks like copy on write is using less memory. I encourage you to turn on copy on write to save memory.

Timing Information

In addition to how much memory your data is using, you probably want your code to run as fast as possible. Throughout this book, we have emphasized best practices, but we have also seen that pandas often has two (or three or four) ways of doing something.

When clients ask what is faster, my general response is, “it depends”. And that is true. If you compare two pieces of code and benchmark them on a small amount of data, there is no guarantee that the fast code will still be faster when bombarded with more data. (Pay special attention to `.apply`, `.query`, and date conversion.)

After saying, “It depends,” I follow that up with, “Benchmark it and see.” You can use the `%%time` cell magic to measure the clock time of a cell in Jupyter:

```
>>> %%time
>>> dd = tweak_river(df)
```

```
CPU times: user 228 ms, sys: 8.8 ms, total: 237 ms
Wall time: 235 ms
```

Another cell magic that provides timing information is `%%timeit`. This will run the cell a few times and give you the mean and standard deviation of the runtime:

```
>>> %%timeit
>>> dd = tweak_river(df)
233 ms ± 9.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Chapter Methods

Method	Description
`df.equals(other)`	Compares two dataframes if they have the same shape and values. Columns should have the same type.
`df.eq(other, axis='columns', level=None)`	Return dataframe with same index and columns but boolean values indicating whether values are the same elementwise.
`df.ne(other, axis='columns', level=None)`	Return dataframe with same index and columns but boolean values indicating whether values are different elementwise.
`pd.testing.assert_frame_equal(left, right, check_dtype=True, check_index_type='equiv', check_column_type='equiv', check_frame_type=True, check_names=True, by_blocks=False, check_exact=False, check_datetimelike_compat=False, check_categorical=True, check_like=False, check_freq=True, check_flags=True, rtol=1e-05, atol=1e-08, obj='DataFrame')`	Utility function to determine if two dataframes are the same. Can change numeric tolerance with `rtol` (relative tolerance) and `atol` (absolute tolerance).

Method	Description
`df.round(decimals=0)`	Create a dataframe with decimals rounded to given places.
`.pipe(func, *args, **kwargs)`	Apply a function to a dataframe. Return the result of function.
`IPython.display.display( *objs, include=None, exclude=None, metadata=None, transient=None, display_id=None, **kwargs)`	Displays `objs` in Jupyter.
`df.info(verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None)`	Print summary of dataframe to stdout. Use `memory_usage='deep'` to show object column memory usage.

Summary

In this chapter, we have shown various techniques for understanding what happens when you use pandas. One of the keys to success with pandas is understanding what operations do to your data and validating that the operation worked as you expected. We also showed how to profile memory usage and timing.

Exercises

With a dataset of your choice, create a tweak function to perform a chain of operations.

1. Use the debugger to step into the chain of your tweak function.
2. Capture an intermediate state of your chain into a variable.
3. Time how long the tweak function takes to run.
4. Determine how much memory the tweak function needs to run.

Refactoring Pandas Code

This chapter will explore a project from the book, *Algorithmic Short Selling with Python* [1]. The primary focus here is not to discuss the book's content but to take some Python code from the book's project and refactor it to improve readability and testability. I believe this coding style represents much of the pandas code in the wild.

I strongly recommend this book if you're interested in algorithmic short-selling.

Starting Code

```
# Chapter 13: Portfolio Management System

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf


K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])

raw_data = yf.download(tickers= ticker_list, period='6mo',
```

```
                        interval = "1d",group_by = 'column',
                        auto_adjust = True,
                         # had a typo w/ treads=True
                         prepost = True, threads = True, proxy = None)

price_df = round( raw_data['Close'],2)

print(price_df.shape)

bm_cost = price_df[bm_ticker][0]
bm_price = price_df[bm_ticker][-1]

port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
port['Cost'] = price_df.iloc[0,:]
port['Price'] = price_df.iloc[-1,:]

print(port)
```

The existing code structure starts with library imports. This is followed by a number of global variables.

The code calls the finance `download` function to fetch six months of stock ticker information at the one-day interval. This is stored in the `price_df` variable. Another dataframe, `port`, is created and printed.

I have changed this code a little from the original. I removed the start and end dates. I also read the data from yfinance into its own data frame. To make this code easy to reproduce I want to limit the amount of times I'm hitting a web service. I will just read the data from a CSV instead.

```
raw_data.to_csv('data/raw-yfinance.csv')
```

With that data in place, few more columns are derived:

```
# Chapter 13: Portfolio Management System

price_df['bm returns'] = round(
    np.exp(np.log(price_df[bm_ticker]/
    price_df[bm_ticker].shift()).cumsum()) - 1, 3)
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
```

```
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round(
    (rLong_MV + rShort_MV).div(abs(rMV).sum(axis=1)),3)

price_df['rReturns_Long'] = round(
    np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
price_df['rReturns_Short'] = - round(
    np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
price_df['rReturns'] = price_df['rReturns_Long'] + price_df['rReturns_Short']

MV = price_df.mul(port['Shares'])
Long_MV = MV[MV >0].sum(axis=1)
Short_MV = MV[MV <0].sum(axis=1)
price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
price_df['Net'] = round(
    (Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)

price_df['Returns_Long'] = round(
    np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
price_df['Returns_Short'] = - round(
    np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
price_df['Returns'] = price_df['Returns_Long'] + price_df['Returns_Short']

MV_Beta = MV.mul(port['Beta'])
Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
```

Finally, some plots are created:

```
# Chapter 13: Portfolio Management System
price_df[['bm returns','Returns','Gross','rNet_Beta','rNet' ]].plot(
    figsize=(20,8),grid=True, secondary_y=['Gross'],
    style= ['r.-','k','g--','g:','g-o','b:','c','c:'],
    title = 'bm returns, Returns, Gross, rNet_Beta, rNet')

price_df[['bm returns','Returns','rReturns','rReturns_Long',
          'rReturns_Short']].plot(
    figsize=(20,8),grid=True,
    style= ['r.-','k','b--o','b--^','b--v','g-.','g:','b:'],
    title='bm returns, Returns, rReturns, rReturns_Long, rReturns_Short')

price_df[['bm returns','Returns','rReturns',
          'rReturns_Long','rReturns_Short','Returns_Long',
          'Returns_Short']].plot(
    figsize=(20,8),grid=True,secondary_y=['Gross'],
```

```
    style= ['r.-','k','b--o','b--^','b--v','k:^','k:v',],
    title= 'Returns: benchmark, Long / Short absolute & relative')

price_df[['bm returns',
          'rReturns_Long','rReturns_Short','Returns_Long',
          'Returns_Short']].plot(
    figsize=(20,8),grid=True,secondary_y=['Gross'],
    style= ['r.-','b--^','b--v','m:.','m:.',],
    title= 'Returns: benchmark, Long / Short absolute & relative')
```

Our plan is to refactor the existing code and wrap tests around it.

In the original code, various objects are derived from `rel_price`, such as `rLong_MV`, `rShort_MV`, `rMV_beta`, and `rLong_MV_Beta`. New columns in the `price_df` data frame are created from these derived objects. Some of these columns have calculations that are derived from other columns.

Code Review

This section focuses on the code we just listed. It's important to clarify that this code isn't inherently bad. In fact, it reflects how many people, including students and industry professionals, write pandas code. Nonetheless, I wish to discuss certain aspects I find less than ideal.

The first issue lies in the prevalent use of global variables. While in the context of Jupyter notebooks global variables are commonplace, they can cause serious problems such as shadowing variables, unexpected state, the time travel paradox (change a global variable's value halfway through your notebook, then rerun an earlier cell? Welcome to a world where Marty McFly might not be born), notebook amnesia (revisit a notebook only to discover that global values have all disappeared), variable name lazy reuse, reproducibility nightmare, order dependency, naming creativity, Murphy's Law (when things can go wrong with globals they will) and more. Ok, maybe I slightly exaggerated, this list should be a little longer.

This dichotomy between Jupyter's easy-going approach to global variables and the more disciplined software engineering perspective can often lead to confusion. While exploratory data analysis has a certain degree of leniency, relying on global variables becomes problematic quickly. Especially when you

intend to move your code into production. We'll examine how to mitigate this by refactoring the code into functions.

The code's rigidity is another concern, stemming from heavy reliance on hard coding. A lack of precise inputs and outputs makes modifications a challenging task. We will create a more streamlined user interface, offering functions with clearly defined inputs that return predictable outputs. This approach not only makes it easier to revisit the code in the future but also facilitates code sharing.

You should also note the absence of documentation within this code. Though this chapter won't delve extensively into documentation, awareness of its importance is crucial. Comments can serve as simple starting points for documentation, but once you've refactored the code into functions, they become excellent candidates for more detailed documentation. Jupyter's shift-tab inspection feature is invaluable, allowing you to review your function's documentation and usage instructions quickly.

Lastly, this code lacks tests, a common occurrence given many individuals either lack the knowledge or the motivation to write tests. However, I firmly believe that tests are integral, especially for Python code, to establish confidence in your code's functionality. Consequently, I will demonstrate how to add tests to ensure the code maintains behavior as expected post-refactoring.

When encountering code in the future, I urge you to ask the following questions:

- Are global variables being used?
- Is the code organized into functions?
- Is the code usage clear?
- Is there adequate documentation?
- Are there tests in place?

If the answer to any of these questions is 'no', then there's room for improvement. Applying the principles and practices discussed in this chapter can enhance our ability to write robust, efficient, and maintainable pandas code.

Enhancing Your Coding Process

As we delve deeper into the intricacies of coding, I want to emphasize a habit I've cultivated over the years. Before typing away, I like to step back and contemplate my action plan. Taking the time to chart your path often leads to better code.

Before diving in, think about the code you're about to work on before diving in. Identify the main objects and what you want the output to be. Certain data structures might be candidates for global variable storage but should result from function calls. It's not advisable to pepper your code with global variables haphazardly.

Next, ponder on how you'll create the main objects. What parameters are needed for their creation? For instance, we have two main objects: a portfolio object (`port`) and a pricing object `priceDF`. However, there are also several intermediate variables created along the way. These intermediate objects are sources from which we derive columns for our `priceDF`.

Ask yourself, what role do these objects play? Are they crucial or merely side effects?

Also, consider how this code will be reused. We often focus on achieving functionality for the present moment, neglecting future use cases. What different parameters might be required? Perhaps, in the future, we'll need to look at data hourly rather than daily, or maybe we'll have different tickers in our portfolio.

As we go along, I'll be using the term "refactoring" quite often. *Refactoring* is a software engineering term indicating a change in the code's internal structure without altering its external behavior. Throughout this chapter, I'm not adding new features or making changes that would affect the output. We're simply altering how we achieve that output.

When refactoring, we want to have a test for the original code, and it should yield the same result after refactoring.

Identify your main objects, understand the steps to create them, plan for future needs, and ensure you have tests to validate the code's behavior. This

strategy can significantly enhance your coding process, allowing you to produce high-quality, reusable, and reliable code.

Analyzing Raw Data

This section will look at financial data fetched from Yahoo using the finance `download` function.

The code creates a `price_df` DataFrame object, which fetches and stores our financial data. We also instantiate a `port` object, a portfolio object that contains the number of shares for our portfolio.

```
K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])

raw_data = pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])

price_df = round(raw_data['Close'],2)
```

Creating a get_tickers Function

To improve the structure and readability of our code, we will refactor the process of pulling data into a new function called `get_tickers`.

```
def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))
```

This function takes as inputs a list of ticker symbols, a period, and an interval. Running this function confirms that it works as expected. I'll tack on a `_rf` on

my new object that stands for “refactored”.

```
>>> port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
...                  'UPS','F']
>>> bm_ticker = '^GSPC'
>>> ticker_list = [bm_ticker] + port_tickers
>>> period = '6mo'
>>> price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
```

I will also validate that it returns the same result as `price_df`.

It’s important to note that the `.equals` method differs from the `==` operator or the `.eq` method. Using `.eq` provides a DataFrame output with `True` or `False` for each cell, indicating whether the corresponding cells are equal.

```
>>> price_df.equals(price_df_rf)
True
```

In our case, `.equals` returns `True` for all values, confirming that our refactoring worked as intended.

Creating Portfolio Data

Now let’s explore the the process of creating our portfolio data.

```
K = 1000000
lot = 100
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])
```

First, let’s take a look at the existing code structure we have. The existing code comprises several variables:

- `K` - This is used in calculating the *Gross* column
- `lot` - This is not used

- `port_tickers` - The portfolio tickers
- `benchmark_ticker` - The benchmark tickers
- `ticker_list` - The combination of the tickers
- `df_data` - The seed data for the port data frame

We start by creating a data frame using the `df_data` dictionary. Then, we make a new column called *side*, reflecting whether the *Shares* column is positive or negative. (`np.sign` returns `-1` for negative numbers, `1` for positive numbers, and `0` for zero.)

```
bm_cost = price_df[bm_ticker][0]
bm_price = price_df[bm_ticker][-1]

port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
port['Cost'] = price_df.iloc[0,:]
port['Price'] = price_df.iloc[-1,:]
```

Next, we proceed to set up our benchmark information. We create a benchmark cost, `bm_cost`, which represents the initial cost of the benchmark item in our portfolio. We also compute the benchmark price, `bm_price`, which corresponds to the last item from the benchmark.

These values allow us to calculate the relative price, *rPrice*, and cost, *rCost*, to our benchmark. We place those in the portfolio data frame. We also add the non-relative versions of these to the portfolio. It's important to note a crucial point. The calculation of relative price depends on the `price_df` data frame. Hence, a dependency exists between these two objects. This is something to keep in mind during the refactoring process.

```
>>> print(port.loc[:, 'rCost':'Price'])
       rCost  rPrice    Cost   Price
QCOM   28.58   26.63  109.25  116.36
TSLA   32.79   59.85  125.35  261.47
NFLX   77.90   96.28  297.75  420.61
DIS    22.67   20.23   86.67   88.39
PG     39.32   34.29  150.30  149.79
MMM    30.92   22.95  118.20  100.27
IBM    35.92   29.97  137.30  130.91
BRK-B  79.19   77.12  302.69  336.91
UPS    45.11   39.46  172.42  172.37
F       2.75    3.27   10.51   14.28
```

Refactoring the Code

Let's refactor this code to create the `port` variable.

```
def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
            )
```

Upon refactoring, the code is written inside a function named `get_portfolio`. This function accepts the tickers, the dependent price data frame, and the benchmark data as arguments.

Let's try it out:

```
>>> print(get_portfolio(port_tickers, price_df_rf,
...                     price_df_rf.loc[:,bm_ticker])
...     .loc[:, 'rCost':'Price']
... )

       rCost  rPrice    Cost   Price
QCOM   28.58   26.63  109.25  116.36
TSLA   32.79   59.85  125.35  261.47
NFLX   77.90   96.28  297.75  420.61
DIS    22.67   20.23   86.67   88.39
PG     39.32   34.29  150.30  149.79
MMM    30.92   22.95  118.20  100.27
IBM    35.92   29.97  137.30  130.91
BRK-B  79.19   77.12  302.69  336.91
UPS    45.11   39.46  172.42  172.37
F       2.75    3.27   10.51   14.28
```

Chaining allows us to conduct operations one after another, where each operation returns a new object (a new data frame or series). The `.assign` method is especially useful here. Chaining forces us to think about the operations step by step, making it read like a recipe.

The use of a `lambda` function within the `.assign` method is particularly essential here. This is because, during chaining, we work with intermediate objects. The assign method accepts a function as an argument and passes in the current state of the data frame to that function. This allows us to work with the current state of the new data fram which now has a *Shares* column. There is no way to access the dataframe without the lambda inside of the `.assign`.

Next, we create the *rCost* column. This column is calculated based on the `price_df` data frame, and involves division of the first column by the first column of the benchmark data. Finally, it is multiplied by 1000 and rounded to two decimal places.

You might be wondering, how do we know if our *rCost* is correct? That's the crux of testing and verification. We should test the result of our calculation against known examples to ensure the accuracy of our code.

In fact, I'll just test the whole data frame:

```
>>> (get_portfolio(port_tickers, price_df_rf,
...   price_df_rf.loc[:,bm_ticker]).equals(port))
True
```

Yeah! It looks like we are doing well.

Notebook Reformatting

Let's delve into an often-overlooked aspect of data science work: **reformatting notebooks**. Unorganized notebooks can cause confusion and significantly slow down the workflow.

After you've executed some code refactoring, the best practice is to place this revised code at the top of your notebook. This way, it's straightforward to locate and execute.

You may want to restart your notebook to ensure your refactoring works as intended. This action can be performed simply by hitting 'zero' twice. A dialog box will pop up, prompting you to confirm if you want to restart the current kernel.

My top cell would not look like this:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
```

```
                    )
            )

port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
period = '6mo'
price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
port_rf = get_portfolio(port_tickers, price_df_rf,
                        price_df_rf.loc[:,bm_ticker])
```

When performing this kind of refactoring, keep in mind the treatment of global variables. Global variables can affect your functions and, in some cases, may be inadvertently left in your refactored code. Restarting your notebook and running the code from scratch ensures that your refactored functions aren't reliant on any lingering global variables. If they were, you would encounter an error, indicating that your refactoring needs more attention.

Keeping your Jupyter notebooks tidy is highly recommended for both solo and collaborative projects. Whenever you complete a portion of your work, move it to the top, restart your notebook, and rerun the code. This practice not only streamlines your work but also eases collaboration. It also simplifies your workflow the following day, as you won't have to wade through a bunch of cells determining which ones need to run and in what order.

More Refactoring

In this section, we're going to see the rest of the codebase. Previously, we looked at the initial code and refactored it into some functions. There's a lot of code we haven't explored yet, so let's see that.

```
price_df['bm returns'] = round(np.exp(np.log(price_df[bm_ticker]/
                        price_df[bm_ticker].shift()).cumsum()) - 1, 3)
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV
```

```
price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round((rLong_MV + rShort_MV)
                         .div(abs(rMV).sum(axis=1)),3)

price_df['rReturns_Long'] = round(
    np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
price_df['rReturns_Short'] = - round(
    np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
price_df['rReturns'] = price_df['rReturns_Long'] + \
    price_df['rReturns_Short']

MV = price_df.mul(port['Shares'])
Long_MV = MV[MV >0].sum(axis=1)
Short_MV = MV[MV <0].sum(axis=1)
price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
price_df['Net'] = round((Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)

price_df['Returns_Long'] = round(
    np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
price_df['Returns_Short'] = -round(
    np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
price_df['Returns'] = price_df['Returns_Long'] + price_df['Returns_Short']

MV_Beta = MV.mul(port['Beta'])
Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
```

I've extracted the code directly from the GitHub repository. You'll notice we have a dataframe called `price_df` and several intermediate objects. These intermediate objects are then used to create various columns in `price_df`. We aim to refactor the code so our new version can replicate these columns with the same values.

Let's go through this code. First, we need to create the *bm returns* column. This column is derived from the *bm_ticker* column in the same dataframe. We use several operations such as `.shift`, `np.log`, `.cumsum`, and `np.exp` to calculate the returns. I'll use the `.assign` method to create that column:

```
>>> # round(np.exp(np.log(price_df[bm_ticker]/
>>> #      price_df[bm_ticker].shift()).cumsum()) - 1, 3)
>>> print(price_df_rf.assign(**{
...  'bm returns': price_df_rf[bm_ticker]/price_df_rf[bm_ticker]
...  .shift().apply(np.log).cumsum().apply(np.exp).sub(1).round(3)})
... )
                           BRK-B    DIS  ...    ^GSPC    bm returns
```

```
Date                                      ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39           NaN
2022-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82  1.006131e+00
2022-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25  2.605570e-04
2022-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22  6.722594e-08
2022-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28  1.807978e-11
...                           ...    ...  ...      ...           ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84  0.000000e+00
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59  0.000000e+00
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71  0.000000e+00
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69  0.000000e+00
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72  0.000000e+00

[124 rows x 12 columns]
```

Let's run the original calculation and compare the results:

```
>>> price_df['bm returns']
Date
2022-12-22 00:00:00-05:00      NaN
2023-12-23 00:00:00-05:00    0.006
2023-12-27 00:00:00-05:00    0.002
2023-12-28 00:00:00-05:00   -0.010
2023-12-29 00:00:00-05:00    0.007
                             ...
2023-06-15 00:00:00-04:00    0.158
2023-06-16 00:00:00-04:00    0.154
2023-06-20 00:00:00-04:00    0.148
2023-06-21 00:00:00-04:00    0.142
2023-06-22 00:00:00-04:00    0.143
Name: bm returns, Length: 124, dtype: float64
```

It turns out that my re-writing of the logic in a chain had a operator precedence error. Good thing I checked that the values were the same. Here is a fixed version:

```
>>> # round(np.exp(np.log(price_df[bm_ticker]/
>>> # price_df[bm_ticker].shift()).cumsum()) - 1, 3)
>>> print(price_df_rf.assign(**{
...     'bm returns': (price_df_rf[bm_ticker]/price_df_rf[bm_ticker]
...      .shift()).apply(np.log).cumsum().apply(np.exp).sub(1).round(3)})
... )
                           BRK-B    DIS  ...    ^GSPC  bm returns
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39         NaN
2023-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82       0.006
2023-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25       0.002
2023-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22      -0.010
```

```
2022-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28       0.007
...                           ...    ...  ...      ...         ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84       0.158
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59       0.154
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71       0.148
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69       0.142
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72       0.143

[124 rows x 12 columns]
```

The code calculates the returns many times. So I'm going to refactor this calculation into its own function:

```
def returns(df, col):
    return (df[col]
            .div(df[col].shift())
            .apply(np.log)
            .cumsum()
            .apply(np.exp)
            .sub(1)
            .round(3))
```

Now, let's run the function:

```
>>> print(price_df_rf
...  .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
... )
                           BRK-B    DIS  ...    ^GSPC  bm returns
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...  3822.39         NaN
2023-12-23 00:00:00-05:00  306.49  88.01  ...  3844.82       0.006
2023-12-27 00:00:00-05:00  305.55  86.37  ...  3829.25       0.002
2023-12-28 00:00:00-05:00  303.43  84.17  ...  3783.22      -0.010
2023-12-29 00:00:00-05:00  309.06  87.18  ...  3849.28       0.007
...                           ...    ...  ...      ...         ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...  4425.84       0.158
2023-06-16 00:00:00-04:00  338.31  91.32  ...  4409.59       0.154
2023-06-20 00:00:00-04:00  338.67  89.75  ...  4388.71       0.148
2023-06-21 00:00:00-04:00  338.61  88.64  ...  4365.69       0.142
2023-06-22 00:00:00-04:00  336.91  88.39  ...  4368.72       0.143

[124 rows x 12 columns]
```

In the `returns` function, we use `.apply` with `np.log` and `np.exp`. You might think that `.apply` is slow. And it is when you use it with Python functions. However, when you're applying a NumPy function to a pandas series, it is fast. This is because it doesn't perform the operation element by element, but instead

applies the operation to the entire series at once due to the vectorized nature of NumPy functions.

If the `returns` function was slow during benchmarking, we might consider a NumPy, Cython, or Numba version. Here is a NumPy version:

```
from scipy.ndimage import shift

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(
        np.exp(np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)
```

Let's benchmarkk this new function:

```
>>> %%timeit
>>> np_returns(price_df_rf, bm_ticker)
46.4 µs ± 14.1 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
     each)

>>> %%timeit
>>> returns(price_df_rf, bm_ticker)
190 µs ± 8.69 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
     each)
```

On my machine the NumPy version is five times faster for this size of data. I'll leave it to the reader to try and implement a Cython and Numba version.

The rel_price Variable

This section looks at dependent objects—objects that our chain relies upon. The original code started with two original objects and created ancillary objects to develop columns that would feed back into the original objects.

The original code operated on a `priceDF` object. In my chain, however, I intend to make calls that will generate some global objects. Consequently, later operations dependent on these will be able to access those global objects and generate derived columns from them.

Consider this original code. The *RNet_Beta* column depends on *rLong_MV_Beta*, which depends on *rMV_Beta* and *rLong_MV* both of which depend on *rMV*, which itself depends on *rel_price*. With chaining, we might be able to just stick these variables into columns, however, *rep_price*, *rMV*, and *rMV_Beta* aren' series. They are dataframes. In my opinion, sticking these dataframes into an existing dataframe is not a win. It will make the code complicated.

```
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round((rLong_MV + rShort_MV)
    .div(abs(rMV).sum(axis=1)),3)
```

It seems counterintuitive to assign a dataframe into a column. So, how do we deal with this issue?

My solution is to create a new dataframe.

I'm going to create a new function called `make_var`. This function takes in a dataframe, a variable name, a function, and several arguments. It then inserts the variable name into the global namespace, calling the function I passed in with the dataframe and the arguments I passed into it.

This function allows me to pass any arguments into the underlying function, `fn`, that I choose.

I will create these variables inside a chain using this function:

```
def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df
```

Instead of using `.assign` to create a column called `relPrice2`, I'm going to use the `.pipe` method of pandas. The `.pipe` method allows you to pass in any function and return whatever you want.

I want to pass in a function that creates a global variable, and I'll return the current state of the dataframe.

We introduced a similar function during the debugging chapter. Now, we are using it in real-world code. I will use it to create these dataframes that other columns depend on:

```
>>> print(price_df_rf
...  .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
...  .pipe(make_var, var_name='rel_price2',
...        fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
...             .mul(1_000).round(2))
...  .pipe(make_var, var_name='rMV2',
...        fn=lambda df_: rel_price2.mul(port_rf['Shares']))
...  .pipe(make_var, var_name='rMV_Beta2',
...        fn=lambda df_: rMV2.mul(port_rf['Beta']))
...  .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
...          rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
...          rLong_MV_Beta=lambda df_: rMV_Beta2[rMV_Beta2 > 0]
...            .sum(axis='columns') / df_.rLong_MV,
...          rShort_MV_Beta=lambda df_: rMV_Beta2[rMV_Beta2 < 0]
...            .sum(axis='columns') / df_.rShort_MV,
...          rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta
...         )
... )
                           BRK-B    DIS  ...  rShort_MV_Beta  \
Date                                     ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...        0.659542
2022-12-23 00:00:00-05:00  306.49  88.01  ...        0.659418
2022-12-27 00:00:00-05:00  305.55  86.37  ...        0.654134
2022-12-28 00:00:00-05:00  303.43  84.17  ...        0.653003
2022-12-29 00:00:00-05:00  309.06  87.18  ...        0.657856
...                           ...    ...  ...             ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...        0.694658
2023-06-16 00:00:00-04:00  338.31  91.32  ...        0.692247
2023-06-20 00:00:00-04:00  338.67  89.75  ...        0.692130
2023-06-21 00:00:00-04:00  338.61  88.64  ...        0.684792
2023-06-22 00:00:00-04:00  336.91  88.39  ...        0.684952

                           rNet_Beta
Date
2022-12-22 00:00:00-05:00   0.343361
2023-06-23 00:00:00-05:00   0.343176
2022-12-27 00:00:00-05:00   0.348584
2022-12-28 00:00:00-05:00   0.349062
2022-12-29 00:00:00-05:00   0.344405
...                              ...
2023-06-15 00:00:00-04:00   0.310011
```

```
2023-06-16 00:00:00-04:00   0.312403
2023-06-20 00:00:00-04:00   0.311869
2023-06-21 00:00:00-04:00   0.318311
2023-06-22 00:00:00-04:00   0.318279

[124 rows x 17 columns]
```

While this technique can be powerful, it requires some caution. We don't want to create too many global variables, and we may need to consider whether to delete these global objects if they consume a significant amount of space.

For more complex operations, you could use something like a context manager to create a global variable during your data creation. When the context manager exits, it cleans up those global variables for you. This is not covered in this book, but it might be an interesting exercise if you're unfamiliar with how context managers work.

Here's my new chain with the new columns:

```
>>> print(price_df_rf
...  .assign(**{'bm returns':lambda df_:returns(df_, col=bm_ticker)})
...  .pipe(make_var, var_name='rel_price2',
...        fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
...                         .mul(1_000).round(2))
...  .pipe(make_var, var_name='rMV2',
...        fn=lambda df_: rel_price2.mul(port_rf['Shares']))
...  .pipe(make_var, var_name='rMV_Beta2',
...        fn=lambda df_: rMV2.mul(port_rf['Beta']))
...  .pipe(make_var, var_name='MV2',
...        fn=lambda df_: df_.mul(port_rf['Shares']))
...  .pipe(make_var, var_name='MV_Beta2',
...        fn=lambda df_: MV2.mul(port_rf['Beta']))
...  .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
...      rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
...      rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
...                                 .sum(axis='columns') / df_.rLong_MV),
...      rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
...                                 .sum(axis='columns') / df_.rShort_MV),
...      rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
...      rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
...                         .div(rMV.abs().sum(axis='columns')).round(3)),
...      rReturns_Long=lambda df_: returns(df_, 'rLong_MV'),
...      Long_MV=MV[MV > 0].sum(axis='columns'),
...      Short_MV=MV[MV < 0].sum(axis='columns'),
...      Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
...      Net=lambda df_:((df_.Long_MV + df_.Short_MV)
```

```
...                       .div(MV2.abs().sum(axis='columns')).round(3)),
...      Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
...      Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
...      Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,
...      Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
...                               .sum(axis='columns') / df_.Long_MV),
...      Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
...                                .sum(axis='columns') / df_.Short_MV),
...      Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
...      )
... )

                            BRK-B    DIS  ...  Short_MV_Beta  \
Date                                      ...
2022-12-22 00:00:00-05:00  302.69  86.67  ...       0.659551
2022-12-23 00:00:00-05:00  306.49  88.01  ...       0.659401
2022-12-27 00:00:00-05:00  305.55  86.37  ...       0.654096
2022-12-28 00:00:00-05:00  303.43  84.17  ...       0.653013
2022-12-29 00:00:00-05:00  309.06  87.18  ...       0.657848
...                           ...    ...  ...            ...
2023-06-15 00:00:00-04:00  339.82  92.94  ...       0.694644
2023-06-16 00:00:00-04:00  338.31  91.32  ...       0.692241
2023-06-20 00:00:00-04:00  338.67  89.75  ...       0.692140
2023-06-21 00:00:00-04:00  338.61  88.64  ...       0.684778
2023-06-22 00:00:00-04:00  336.91  88.39  ...       0.684991

                           Net_Beta
Date
2022-12-22 00:00:00-05:00  0.343349
2023-12-23 00:00:00-05:00  0.343220
2022-12-27 00:00:00-05:00  0.348606
2022-12-28 00:00:00-05:00  0.349037
2022-12-29 00:00:00-05:00  0.344450
...                             ...
2023-06-15 00:00:00-04:00  0.310061
2023-06-16 00:00:00-04:00  0.312408
2023-06-20 00:00:00-04:00  0.311862
2023-06-21 00:00:00-04:00  0.318336
2023-06-22 00:00:00-04:00  0.318226

[124 rows x 29 columns]
```

Again, to ensure that this works, you could restart your notebook and run it with only the new code.

Code Porting and Validation

In this section, we will finalize refactoring our code, ensuring all elements from the original code are included in our refactored version.

A crucial feature in my columns is the reusability of the `np_returns` function I created earlier. This refactoring allows me to leverage all the returns logic into one location.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.ndimage import shift
import yfinance as yf

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(np.exp(
        np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]

    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
            )

def make_var(df, var_name, fn, *args, **kwargs):
```

```
def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df

def get_price(df, bm_ticker, port, K):
    return (df
     .assign(**{'bm returns': lambda df_: np_returns(df_, bm_ticker)})
     .pipe(make_var, var_name='rel_price2',
           fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
               .mul(1_000).round(2))
     .pipe(make_var, var_name='rMV2',
           fn=lambda df_: rel_price2.mul(port['Shares']))
     .pipe(make_var, var_name='rMV_Beta2',
           fn=lambda df_: rMV2.mul(port['Beta']))
     .pipe(make_var, var_name='MV2',
           fn=lambda df_: df_.mul(port['Shares']))
     .pipe(make_var, var_name='MV_Beta2',
           fn=lambda df_: MV2.mul(port['Beta']))
     .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
        rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
        rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
                                   .sum(axis='columns') / df_.rLong_MV),
        rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
                                    .sum(axis='columns') / df_.rShort_MV),
        rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
        rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
                          .div(rMV2.abs().sum(axis='columns')).round(3)),
        rReturns_Long=lambda df_: np_returns(df_, 'rLong_MV'),
        rReturns_Short=lambda df_: -np_returns(df_, 'rShort_MV'),
        rReturns=lambda df_:df_.rReturns_Long + df_.rReturns_Short,
        Long_MV=MV2[MV2 > 0].sum(axis='columns'),
        Short_MV=MV2[MV2 < 0].sum(axis='columns'),
        Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
        Net=lambda df_: ((df_.Long_MV + df_.Short_MV)
                         .div(MV2.abs().sum(axis='columns')).round(3)),
        Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
        Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
        Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,

        Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
                                 .sum(axis='columns') / df_.Long_MV),
        Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
                                  .sum(axis='columns') / df_.Short_MV),
        Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
        )
     #.rename(columns={'benchmark_returns': 'bm returns'})
     .loc[:, ['BRK-B', 'DIS', 'F', 'IBM', 'MMM', 'NFLX', 'PG', 'QCOM',
       'TSLA', 'UPS', '^GSPC', 'bm returns', 'rNet_Beta', 'rNet',
       'rReturns_Long', 'rReturns_Short', 'rReturns', 'Gross', 'Net',
       'Returns_Long', 'Returns_Short', 'Returns', 'Net_Beta']]
    )
```

```
    )

def process_data():
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
    port_rf = get_portfolio(port_tickers, price_df_rf,
                            price_df_rf[bm_ticker])
    returns = get_price(price_df_rf, bm_ticker, port_rf, K)
    return returns
```

```
price_df_rf = process_data()
```

Let's check whether these are the same.

```
>>> pd.testing.assert_frame_equal(price_df, price_df_rf)
```

Success!

Summary

This chapter presented the various facets of refactoring using pandas. We began by discussing the importance of restructuring or refactoring code to make it cleaner, more efficient, and easier to understand. We started making functions and chains. We underscored the importance of comprehensive checks and validations after refactoring, introduced refined methods for validating the equivalence of DataFrames, and emphasized the need to stop and resolve issues as soon as they emerge.

Exercises

1. Reflect on your current practices in writing pandas code. Identify areas where you could apply the principles of refactoring to write more efficient and readable code.
2. Find a block of pandas code you have written and refactor it.
3. Rewrite `np_returns` in Cython.

4. Rewrite `np_returns` in Numba.
5. Convert the code in this chapter to use PyArrow types and ensure that the conversion didn't introduce errors.

---

1. https://github.com/PacktPublishing/Algorithmic-Short-Selling-with-Python-Published-by-Packt/blob/main/Chapter%2013/Chapter%2013.ipynb

Refactoring Code and Unit Testing

This chapter will discuss how to refactor code and run sanity checks. We then introduced a more formal approach to testing, using a library called PyTest. Testing is something that software engineers do all the time, but many folks don't have experience with it. I want to show you how to get started testing your pandas code.

Using PyTest

Before proceeding, ensure PyTest is installed in your environment. You can install it by running `pip install pytest` in your terminal or command prompt.

When running tests, it's essential to have consistent tests. This means you get the same output every time you run the tests. In our code, we are using Yahoo Finance to download data. This dependency could lead to inconsistent test results since there's no guarantee that the data from Yahoo Finance will be the same if you run the tests in the future.

We already did this in the previous chapter, but I will repeat it here. (Normally I would only do this for testing purposes.) I will force data consistency by saving the downloaded data to a CSV file. In my test, I will read data from the file instead of calling the resource that might change. (I might want to have another test to check that the format of the data coming from Yahoo Finance doesn't change. But I would want this test to be distinct from one that ensures my logic works.)

```
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
```

```
ticker_list = [bm_ticker] + port_tickers
period = '6mo'
price_df_rf = get_tickers(ticker_list=ticker_list, period=period)

price_df_rf.to_csv('data/tickers-raw.csv')
```

Writing Test Code

One of the significant challenges for Jupyter users is that they might not work
in a terminal or a shell like software developers do. Software developers will write code and test in a text editor and then use a command line tool like pytest to run the tests and report the results. (They might have an IDE that does this for them, but you should understand what the underlying tool does before blindly trusting the IDE.)

Let's write some test code. If I were in Jupyter, I would use a Jupyter cell magic called `%writefile`. This cell magic allows us to write a file directly from Jupyter.

```
%%writefile test_pd_refactor.py

import pandas as pd
import pytest


@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                       dtype_backend='pyarrow', engine='pyarrow')

def test_basic(raw_price_df):
    assert len(raw_price_df) > 1
```

This test code imports the `pandas` and `pytest` libraries. I created a function, `test_basic`, that takes in a `DataFrame` and checks if its length is greater than one. This function uses a PyTest feature known as *fixtures*. Fixtures are dependencies necessary for tests to run. The `test_basic` test needs a dataframe to run. The function `raw_price_df` will provide that dataframe. Using a decorator, `pytest.fixture`, we register the fixture. Any test that wants

to use that fixture just needs to put the name of the fixture function in the parameter list.

Here is where the challenge for Jupyter users is. Running the test. Software folks would do this from the command line. We can use the Jupyter feature to access the command line. We need to preface our code with an exclamation point (`!`).

We ran the test code using the command `pytest test_pd_refactored.py`.

Let's look at the output:

```
!pytest test_pd_refactored.py
test_pd_refactor.py .                                                [100%]

============================ 1 passed in 0.30s ============================
```

Following the file name is a period. That period is a convention that pytest uses to indicate that the test passed. On the right, it says 100 percent of the test from the file passed. At the bottom is a summary of the number of tests and the time it took to run them.

Writing More Compelling Test Cases

Our initial test case was relatively simple – checking if the length of the `DataFrame` was greater than one.

I'm going to test that the original code and the refactored code have the same behavior. I will create a file, `returns.py`, with a function to generate the original `price_df` called `orig_price_df`. It will also have the refactored logic, `process_data`.

```
%%writefile returns.py
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.ndimage import shift
import yfinance as yf

def np_returns(df, col):
    values = df[col].to_numpy()
    shifted = shift(values, 1, cval=np.NaN)
    res = np.round(np.subtract(
```

```
    res = np.round(np.subtract(
        np.exp(np.nancumsum(np.log(np.divide(values, shifted)))), 1), 3)
    res[0] = np.NaN
    return pd.Series(res,index=df.index)

def get_tickers(ticker_list, period, interval='1d'):
    return (pd.read_csv('data/raw-yfinance.csv', index_col=0, header=[0,1])
            ['Close']
            .round(2))

def get_portfolio(port_tickers, price_df, benchmark_data):
    df_data= {
    'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
    'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
    'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
    }
    benchmark_cost = benchmark_data.iloc[0]
    benchmark_price = benchmark_data.iloc[-1]
    start_price = price_df.iloc[0]
    end_price = price_df.iloc[-1]
    return (pd.DataFrame(df_data,index=port_tickers)
            .assign(Side=lambda df_:np.sign(df_.Shares),
                    rCost=((start_price / benchmark_cost)
                           .mul(1_000).round(2)),
                    rPrice=((end_price / benchmark_price)
                            .mul(1_000).round(2)),
                    Cost=start_price,
                    Price=end_price,
                    )
           )

def make_var(df, var_name, fn, *args, **kwargs):
    globals()[var_name] = fn(df, *args, **kwargs)
    return df

def get_price(df, bm_ticker, port, K):
    return (df

     .assign(**{'bm returns': lambda df_: np_returns(df_, bm_ticker)})
     .pipe(make_var, var_name='rel_price2',
           fn=lambda df_:df_.div(df_[bm_ticker], axis='index')
               .mul(1_000).round(2))
     .pipe(make_var, var_name='rMV2',
           fn=lambda df_: rel_price2.mul(port['Shares']))
     .pipe(make_var, var_name='rMV_Beta2',
           fn=lambda df_: rMV2.mul(port['Beta']))
     .pipe(make_var, var_name='MV2',
           fn=lambda df_: df_.mul(port['Shares']))
     .pipe(make_var, var_name='MV_Beta2',
           fn=lambda df_: MV2.mul(port['Beta']))
     .assign(rLong_MV=rMV2[rMV2 > 0].sum(axis='columns'),
```

```
        rShort_MV=rMV2[rMV2 < 0].sum(axis='columns'),
        rLong_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 > 0]
                                   .sum(axis='columns') / df_.rLong_MV),
        rShort_MV_Beta=lambda df_: (rMV_Beta2[rMV_Beta2 < 0]
                                    .sum(axis='columns') / df_.rShort_MV),
        rNet_Beta=lambda df_: df_.rLong_MV_Beta - df_.rShort_MV_Beta,
        rNet=lambda df_: ((df_.rLong_MV + df_.rShort_MV)
                          .div(rMV2.abs().sum(axis='columns')).round(3)),
        rReturns_Long=lambda df_: np_returns(df_, 'rLong_MV'),
        rReturns_Short=lambda df_: -np_returns(df_, 'rShort_MV'),
        rReturns=lambda df_:df_.rReturns_Long + df_.rReturns_Short,
        Long_MV=MV2[MV2 > 0].sum(axis='columns'),
        Short_MV=MV2[MV2 < 0].sum(axis='columns'),
        Gross=lambda df_:(df_.Long_MV - df_.Short_MV).div(K).round(3),
        Net=lambda df_: ((df_.Long_MV + df_.Short_MV)
                         .div(MV2.abs().sum(axis='columns')).round(3)),
        Returns_Long=lambda df_:np_returns(df_, 'Long_MV'),
        Returns_Short=lambda df_:-np_returns(df_, 'Short_MV'),
        Returns=lambda df_:df_.Returns_Long + df_.Returns_Short,
        Long_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 > 0]
                                 .sum(axis='columns') / df_.Long_MV),
        Short_MV_Beta=lambda df_:(MV_Beta2[MV_Beta2 < 0]
                                  .sum(axis='columns') / df_.Short_MV),
        Net_Beta=lambda df_:df_.Long_MV_Beta - df_.Short_MV_Beta,
            )
     #.rename(columns={'benchmark_returns': 'bm returns'})
     .loc[:, ['BRK-B', 'DIS', 'F', 'IBM', 'MMM', 'NFLX', 'PG', 'QCOM',
       'TSLA', 'UPS', '^GSPC', 'bm returns', 'rNet_Beta', 'rNet',
       'rReturns_Long', 'rReturns_Short', 'rReturns', 'Gross', 'Net',
       'Returns_Long', 'Returns_Short', 'Returns', 'Net_Beta']]
    )

def process_data():
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']

    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    price_df_rf = get_tickers(ticker_list=ticker_list, period=period)
    port_rf = get_portfolio(port_tickers, price_df_rf,
                            price_df_rf[bm_ticker])
    returns = get_price(price_df_rf, bm_ticker, port_rf, K)
    return returns

def orig_price_df(price_df):
    price_df = price_df.copy()
    K = 1000000
    lot = 100
```

```
port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                'UPS','F']
bm_ticker= '^GSPC'
ticker_list = [bm_ticker] + port_tickers
df_data= {
'Beta':[1.34,2,0.75,1.2,0.41,0.95,1.23,0.9,1.05,1.15],
'Shares':[-1900,-100,-400,-800,-5500,1600,1800,2800,1100,20800],
'rSL':[42.75,231,156,54.2,37.5,42.75,29.97,59.97,39.97,2.10]
}
port = pd.DataFrame(df_data,index=port_tickers)
port['Side'] = np.sign(port['Shares'])

bm_cost = price_df[bm_ticker][0]
bm_price = price_df[bm_ticker][-1]

port['rCost'] = round(price_df.iloc[0,:].div(bm_cost) *1000,2)
port['rPrice'] = round(price_df.iloc[-1,:].div(bm_price) *1000,2)
port['Cost'] = price_df.iloc[0,:]
port['Price'] = price_df.iloc[-1,:]
price_df['bm returns'] = round(
    np.exp(np.log(price_df[bm_ticker]/
            price_df[bm_ticker].shift()).cumsum()) - 1, 3)
rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)

rMV = rel_price.mul(port['Shares'])
rLong_MV = rMV[rMV >0].sum(axis=1)
rShort_MV = rMV[rMV <0].sum(axis=1)
rMV_Beta = rMV.mul(port['Beta'])
rLong_MV_Beta = rMV_Beta[rMV_Beta >0].sum(axis=1) / rLong_MV
rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV

price_df['rNet_Beta'] = rLong_MV_Beta - rShort_MV_Beta
price_df['rNet'] = round(
    (rLong_MV + rShort_MV).div(abs(rMV).sum(axis=1)),3)

price_df['rReturns_Long'] = round(

    np.exp(np.log(rLong_MV/rLong_MV.shift()).cumsum())-1,3)
price_df['rReturns_Short'] = -round(
    np.exp(np.log(rShort_MV/rShort_MV.shift()).cumsum())-1,3)
price_df['rReturns'] = price_df['rReturns_Long'] +\
         price_df['rReturns_Short']

MV = price_df.mul(port['Shares'])
Long_MV = MV[MV >0].sum(axis=1)
Short_MV = MV[MV <0].sum(axis=1)
price_df['Gross'] = round((Long_MV - Short_MV).div(K),3)
price_df['Net'] = round(
    (Long_MV + Short_MV).div(abs(MV).sum(axis=1)),3)
```

```
    price_df['Returns_Long'] = round(
        np.exp(np.log(Long_MV/Long_MV.shift()).cumsum())-1,3)
    price_df['Returns_Short'] = -round(
        np.exp(np.log(Short_MV/Short_MV.shift()).cumsum())-1,3)
    price_df['Returns'] = price_df['Returns_Long'] + \
        price_df['Returns_Short']

    MV_Beta = MV.mul(port['Beta'])
    Long_MV_Beta = MV_Beta[MV_Beta >0].sum(axis=1) / Long_MV
    Short_MV_Beta = MV_Beta[MV_Beta <0].sum(axis=1)/ Short_MV
    price_df['Net_Beta'] = Long_MV_Beta - Short_MV_Beta
    return price_df
```

Let's create a new fixture that depends on the first fixture to do something more interesting. This new fixture processes the raw data with the original logic. We then wrote another test to ensure that this processed `DataFrame` also has a length greater than one.

```
%%writefile test_pd_refactor.py
import numpy as np
import pandas as pd
import pytest

import returns

@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )
@pytest.fixture
def orig_price_df(raw_price_df):
    return returns.orig_price_df(raw_price_df)

def test_basic(orig_price_df):
    assert len(orig_price_df) > 1
```

Now let's run the new code:

```
!pytest test_pd_refactor.py
============================= test session starts
     ==============================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition
plugins: anyio-4.2.0
collected 1 item
```

```
test_pd_refactor.py .
        [100%]

============================== warnings summary
     ==============================
test_pd_refactor.py::test_basic

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:109: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_cost = price_df[bm_ticker][0]

test_pd_refactor.py::test_basic

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:110: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_price = price_df[bm_ticker][-1]

-- Docs:
     https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 1 passed, 2 warnings in 0.73s
     ========================
```

Success!

Testing the Refactor

Now, let's add a test to ensure the refactoring works.

We will create the `test_refactor` function. This test ensures that our new `get_price` logic returns the same result as the previous code.

```
%%writefile test_pd_refactor.py
import numpy as np
import pandas as pd
#   from pandas.testing import assert_frame_equal
import pytest

import returns
```

```
@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )

@pytest.fixture
def raw_price_df_legacy(raw_price_df):
    return raw_price_df

@pytest.fixture
def orig_price_df(raw_price_df_legacy):
    return returns.orig_price_df(raw_price_df_legacy)

def test_basic(orig_price_df):
    assert len(orig_price_df) > 1

def test_refactor(orig_price_df, raw_price_df_legacy):
    price_df_rf = raw_price_df_legacy
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM','BRK-B',
                    'UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    port_rf = returns.get_portfolio(port_tickers, price_df_rf,
                                    price_df_rf[bm_ticker])
    new_price_df = returns.get_price(price_df_rf, bm_ticker, port_rf, K)
    pd.testing.assert_frame_equal(new_price_df, orig_price_df)
```

Let's run the tests with pytest.

```
!pytest test_pd_refactor.py
============================= test session starts
     ==============================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition
plugins: anyio-4.2.0
collected 2 items

test_pd_refactor.py ..
        [100%]

============================== warnings summary
     ==============================
test_pd_refactor.py::test_basic
test_pd_refactor.py::test_refactor
```

```
     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:109: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_cost = price_df[bm_ticker][0]

test_pd_refactor.py::test_basic
test_pd_refactor.py::test_refactor

     /Users/matt/Dropbox/work/books/EffectivePandas-SecondEdition/returns.
    py:110: FutureWarning: Series.__getitem__ treating keys as positions
     is deprecated. In a future version, integer keys will always be
     treated as labels (consistent with DataFrame behavior). To access a
     value by position, use `ser.iloc[pos]`
    bm_price = price_df[bm_ticker][-1]

-- Docs:
     https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 2 passed, 4 warnings in 0.82s
     =========================
```

Awesome! We now have a test that confirms that our refactoring gives the same results as the original code.

ipytest

In this section, we will delve into another testing mechanism called ipytest. Unlike other command line tools, ipytest is run from Jupyter.

To use IPyTest, ensure it's installed using `pip install IPyTest`. Next, you will need to import the library and run the `autoconfig` function that sets up the library to work in Jupyter.

```
import ipytest
ipytest.autoconfig()
```

Now we should have the `%%ipytest` cell magic available.

This cell magic allows us to create the tests and run them in the same cell. Instead of having to write a file with the test code and then invoke `pytest` in a

new cell, I can just stick `%%pytest` at the top of the cell with the test code and run that cell. The ipytest library will run the cell as if it were a pytest test.

```
%%ipytest
import pytest
import returns

@pytest.fixture
def raw_price_df():
    return pd.read_csv('data/tickers-raw.csv',
                       index_col='Date', parse_dates=['Date'],
                      dtype_backend='pyarrow', engine='pyarrow'
                      )

@pytest.fixture
def raw_price_df_legacy(raw_price_df):
    return raw_price_df.astype(float)

@pytest.fixture
def orig_price_df(raw_price_df_legacy):
    return returns.orig_price_df(raw_price_df_legacy)


def test_basic(orig_price_df):
    assert len(orig_price_df) > 1

def test_refactor(orig_price_df, raw_price_df_legacy):
    price_df_rf = raw_price_df_legacy
    K = 1_000_000
    port_tickers = ['QCOM','TSLA','NFLX','DIS','PG', 'MMM','IBM',
                    'BRK-B','UPS','F']
    bm_ticker= '^GSPC'
    ticker_list = [bm_ticker] + port_tickers
    period = '6mo'
    port_rf = returns.get_portfolio(port_tickers, price_df_rf,
                                    price_df_rf[bm_ticker])
    new_price_df = returns.get_price(price_df_rf, bm_ticker, port_rf, K)
    pd.testing.assert_frame_equal(new_price_df, orig_price_df)
```

A beneficial aspect of this approach is its flexibility. We can use ipytest anywhere in our notebook. This way, we can run a quick test to ensure it works. This method is particularly beneficial for collaborations and sanity checks.

Summary

In this chapter, we introduced pytest a powerful testing framework for Python, along with ipytest. This was only a surface-level introduction, as diving deep into these topics merits a book (or two).

This combination of tools provides the means to write, test, and debug Python code more efficiently, making it easier to ensure code quality and correctness. The capability to run tests directly in Jupyter is a boon for data scientists and those who often use these notebooks for coding.

Exercises

1. Install pytest and IPyTest using pip. Once installed, write a small script to confirm that pytest is working correctly on your machine.
2. Set up IPyTest in a Jupyter notebook and run a simple test to verify it's correctly configured.
3. Write a pytest test for your pandas code.
4. Execute the test using ipytest.

Conclusion

Congratulations! If you have read this far and start applying the techniques you have learned, you will have better pandas code than 90% of the pandas users. You will be able to write code that is more readable, more efficient, and more robust. You will be happy when you return to your code in six months and can still understand it.

If you are skeptical, I encourage you to try it out. Apply the patterns you have seen in this book:

- Chaining
- Leveraging *tweak* functions
- Using `.assign`
- Using `.pipe`
- Avoiding `.apply`
- Using PyArrow types
- Turn on copy on write
- Take advantage of Numba or Cython if necessary

I've had many readers comment to me over the years that they were skeptical about the patterns I was teaching, but after trying them out, they completely changed how they wrote pandas code. I hope you will have the same experience.

As an educator, I feel that I must also be open to learning from my students. If you have pandas patterns that you have discovered that are not in this book, please let me know. I would love to learn from you.

What's next?

Folks often ask what they should do after reading this book. I recommend practicing the patterns you have learned here. Try to apply them to your work or hobby projects. Practice will make you feel comfortable with these patterns, and you will have a deeper understanding of them once you apply them to your problems.

If you are looking for more pandas content, I have many courses at metasnake.com. These include courses on exploring analytics, refactoring and testing, sales reporting, and more.

Team Training

If you want to help your team learn these patterns, don't hesitate to contact me at matt@metasnake.com. I've helped some of the largest companies in the world improve their pandas code. I would be happy to help your team, too.

One More Thing

If you enjoyed this book, please leave a review on Amazon. This is the best way to say thank you to me and to help other people find this book. I don't have a large publisher behind me, so I rely on word of mouth to get the word out about this book. I would appreciate it if you took a moment to leave an honest review of this book.

I also appreciate any reviews or mentions on social media. Please tag me so I can thank you!