

Treading on Python Series



EFFECTIVE POLARS

Optimized Data Manipulation

Matt Harrison

Effective Polars

Patterns for Data Manipulation

Effective Polars

Patterns for Data Manipulation

Matt Harrison

COPYRIGHT © 2024

While every precaution has been taken in the preparation of this book, the publisher and author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Contents

1 Getting Started with Polars	3
1.1 Introduction	3
1.2 Installing Polars	4
1.3 Polars Data Structures	5
1.4 Laziness	5
1.5 Contexts & Expressions	7
1.6 Reading CSV Files	9
1.7 Lazy CSV Reading	10
1.8 Data Type Inference and Manual Overrides	11
1.9 Automatic Type Shrinking	18
1.10 Float Conversion	19
1.11 Extracting Numbers from Strings	21
1.12 String Columns	28
1.13 Parsing Dates	32
1.14 Summary Statistics	36
1.15 Being Lazy	38
1.16 Chapter Functions and Methods	40
1.17 Summary	45
1.18 Exercises	45
2 Data Manipulation with Polars Using the Fuel Economy Dataset	47
2.1 Introduction	47
2.2 Getting the Data	47
2.3 Adding Columns	48
2.4 Simulating the Index	50
2.5 Removing Columns	51
2.6 Renaming Columns	53
2.7 Joining DataFrames	54
2.8 Right Joins	57
2.9 Inner Joins	58
2.10 Outer Joins	59
2.11 Semi Joins	60
2.12 Cross Joins	61
2.13 Anti Joins	62
2.14 Join Validation	63

Contents

2.15 Speeding up Joins with Sorting	65
2.16 Visualizing the Join	71
2.17 Adding Rows	72
2.18 Reshaping and Pivoting Data	73
2.19 Melting Data	77
2.20 Finding Duplicates	79
2.21 Finding Missing Values	82
2.22 Third-Party Libraries and Missing Values	86
2.23 Additional Column Selectors and Missing Values	88
2.24 Map and Apply	91
2.25 Chapter Functions and Methods	93
2.26 Summary	95
2.27 Exercises	96
3 String Manipulation	97
3.1 Twitter Data	97
3.2 Validating the Data	99
3.3 Extracting the Username	101
3.4 Extract the Username with a Regular Expression	107
3.5 Counting Words and Mentions	109
3.6 Checking for Emojis	114
3.7 Plotting Trends	116
3.8 Chapter Functions and Methods	119
3.9 Summary	121
3.10 Exercises	121
4 Aggregation with Polars	123
4.1 Introduction	123
4.2 Loading the Data	123
4.3 Aggregations	123
4.4 Lists	125
4.5 GroupBy Operations	128
4.6 Multiple Aggregations	131
4.7 Grouping By Multiple Columns	133
4.8 Pivoting with Multiple Aggregations	133
4.9 Aggregating to the Original Rows	137
4.10 Grouping to Lists	139
4.11 Chapter Methods and Functions	141
4.12 Summary	143
4.13 Exercises	143
5 Data Filtering and Selection	145
5.1 Introduction	145
5.2 Getting the Data	145
5.3 Filtering with Expressions	145
5.4 Filtering with Keywords	149

5.5	Filtering with Dates	150
5.6	Advanced Filtering Techniques	151
5.7	More Filtering with Window Expressions	152
5.8	Manipulating Data with Conditional Expressions	154
5.9	Handling Missing Data	161
5.10	Dropping Rows with Missing Values	165
5.11	Filling in Missing Values	170
5.12	Filling in Time Series Missing Values	171
5.13	Chapter Methods and Functions	175
5.14	Summary	177
5.15	Exercises	177
6	Sorting and Ordering in Polars	179
6.1	Introduction	179
6.2	Loading the Fuel Economy Dataset	179
6.3	Sorting by a Single Column	179
6.4	Sorting by Multiple Columns	184
6.5	Specifying Custom Ordering for Categorical Columns	186
6.6	Enums and Ordering	190
6.7	Group Ordering and <code>maintain_order</code>	195
6.8	Stable Sorting	196
6.9	Sorting and Filtering	197
6.10	Chapter Methods and Functions	198
6.11	Summary	199
6.12	Exercises	199
7	Time Series Analysis	201
7.1	Introduction	201
7.2	Loading the Dataset	201
7.3	Converting to Dates	203
7.4	Combining Columns to Create Dates	204
7.5	Changing Timezones	205
7.6	Time Aggregations	209
7.7	Time Intervals	211
7.8	Multiple Groupings with Timeseries	214
7.9	Window Functions in Polars	219
7.10	Interpolation	221
7.11	Upsampling and Downsampling	224
7.12	Joining Time Series Data	226
7.13	Visualizing the Merged Data	228
7.14	Chapter Methods and Functions	231
7.15	Summary	234
7.16	Exercises	234
8	Data Import and Export	237
8.1	Introduction	237

Contents

8.2 Importing Data	237
8.3 Exporting to CSV	238
8.4 Exporting to JSON	239
8.5 Reading JSON	241
8.6 Custom JSON Handling	243
8.7 Munging JSON	247
8.8 Exporting to Excel	250
8.9 Exporting to Parquet	252
8.10 Exporting to SQL	253
8.11 Using Arrow to Convert DataFrames	254
8.12 Converting to Pandas	258
8.13 Working with Other Libraries	259
8.14 Using XGBoost to Predict Mileage	259
8.15 Plotting Residuals with Matplotlib	261
8.16 PCA of the Autos Data	262
8.17 Configuration	265
8.18 Chapter Methods and Functions	266
8.19 Summary	275
8.20 Exercises	275
9 Being Lazy and Streaming	277
9.1 Loading Data	277
9.2 The replace Method	280
9.3 Lazy Version Take One	281
9.4 Fetching Data	283
9.5 Pandas Comparison	284
9.6 Viewing Plans	286
9.7 Streaming	288
9.8 Chapter Methods and Functions	289
9.9 Summary	291
9.10 Exercises	291
10 Porting from Pandas	293
10.1 The Code	293
10.2 Multiplication Diversion	297
10.3 Leveraging AI	299
10.4 Code Review	301
10.5 Enhancing Your Coding Process	302
10.6 Analyzing Raw Data	304
10.7 Creating a get_price Function	305
10.8 Creating a get_port Function	308
10.9 More Refactoring	311
10.10 Final Refactoring	320
10.11 To Port or Not to Port	325
10.12 Chapter Methods	328
10.13 Summary	328

10.14 Exercises	328
11 Extending Polars	331
11.1 Loans with Polars	331
11.2 Use Numba	333
11.3 Closed Form Solution	334
11.4 Adding a PCA API to Polars	336
11.5 Calculating the Principal Components	337
11.6 Scatter Plot of the Principal Components	339
11.7 Adding the API	341
11.8 Summary	344
11.9 Exercises	345
12 Conclusion	347
12.1 One Last Thing	347
About the Author	349
Technical Editors	349
Index	351

Foreword

“Came for the speed, stayed for the syntax.”

That’s a common refrain among Polars enthusiasts. And indeed, the Polars API is truly beautiful: not only does it make for very readable code, it also allows you to express complex aggregations which just aren’t expressible with the pandas API.

Matt Harrison is a world-renowned pandas expert, and I’m a Pandas maintainer. Yet we’ve both dedicated considerable time to Polars over the past year—not because we want to “kill Pandas” or anything like that, but because we’re so excited about the ground-breaking library that Polars is. It’s refreshing to see a project have the courage to do things differently and produce such an incredible result—its growing popularity is well-deserved!

When should you use Polars? I think the best time is when you’re starting a new project. Porting Pandas code to Polars is certainly possible, as Matt shows here, but it’s not necessarily easy. If you try thinking in Polars at the start of a new project, you’ll likely surprise yourself with how expressive its API truly is, you’ll use it idiomatically, and you’ll make full use of its amazing features.

I’m sure you’ll love learning about Polars while reading this book. When you start your next data science project, please join the Polars Discord to say hello! I would love to hear about your experience!

Marco Gorelli - Feb 2024

Chapter 1

Getting Started with Polars

1.1 Introduction

Polars is a Python library for data manipulation and analysis, focusing on performance and ease of use. It leverages Apache Arrow[^arrow], a columnar in-memory data format gaining popularity in the data science community. While Polars provides a DataFrame API similar to Pandas, some notable differences exist.

I assume you are familiar with (or at least have heard of) Pandas. It is the dominant library for manipulating tabular data among Python users and data scientists.

You might wonder why you want to use Polars instead of Pandas? The primary draw is performance. Unlike Pandas, Polars was written from the ground up with a query engine. The creator, Ritchie Venk, has described it as “a query engine with a dataframe API”. It can harness multicore processing, and with lazy evaluation and disk spillover, it can handle datasets larger than memory. However, one should know that this new library is still under active development. It might not integrate seamlessly with other libraries like Pandas or undergo breaking changes in future updates.

Here’s a table showcasing the differences between the Polars library and Pandas.

Feature	Polars	Pandas
Underlying Language	Rust	Python (NumPy / PyArrow)
Performance	Fast due to Rust’s performance and query engine	Faster than pure Python
Memory Management	More efficient due to Rust’s memory management and zero-copy functionality	Copy on write. Works well for small data.

1. Getting Started with Polars

Feature	Polars	Pandas
Lazy Evaluation	Yes, supports lazy evaluation, allowing for optimized execution plans and better performance with large datasets	No, operates eagerly, which can be less efficient for large or complex operations
Multicore Support	Native support for multicore processing, allowing for parallel computation and faster data processing	Limited multicore support; relies on Python's Global Interpreter Lock, though some operations can be parallelized with external libraries
Disk Spill-Over	Has built-in support for disk spill-over, beneficial for working with datasets larger than memory	No native support for disk spill-over
Ease of Use	User-friendly API but different from Pandas	Very user-friendly, especially for those familiar with Python; has a massive user base and extensive documentation
Community and Ecosystem	Growing community, but smaller compared to pandas	Very large community, extensive ecosystem, and a wealth of third-party libraries and tools

1.2 Installing Polars

Make sure you have Polars installed. I like using `pip install polars` inside of a virtual environment. You can also use `conda` to install Polars.

Note

It is a standard convention to alias the Polars library as `pl` when you import it. Like this:

```
import polars as pl
```

Note

In this book, I will generally write my code as if it were inside a Python REPL. So you will see the primary prompt, `>>>` and the secondary prompt `....`. I have followed this convention in many of my books. Some have complained about it. I really like to see the output of my operations, which lets me clearly see what was input and what was output. Note that unlike some of the complaints I have gotten, you can run code with these prompts in a Jupyter notebook.

Let's check the version of Polars we have installed.

```
>>> import polars as pl
>>> pl.__version__
'0.20.16'
```

1.3 Polars Data Structures

Like Pandas, Polars has both a Series and DataFrame data structure. Generally, we don't use the Series in Polars. Instead, Polars has a mechanism to represent a single column as an expression. This is typically done with the `pl.col` function. Unlike Pandas, which is *eager*, and evaluates every operation immediately, Polars can be lazy and leverage these expressions which enables lazy evaluation and the performance benefits that come with it.

You can also create a lazy dataframe that represents a dataframe that is not yet evaluated. For example, if we load a CSV file with the `pl.read_csv` function, we get a normal dataframe. However, if we use the `pl.scan_csv` function, we get a lazy dataframe. I will refer to the *normal* dataframe as an *eager* or *materialized* dataframe in this book.

1.4 Laziness

Often, when we are doing exploratory data analysis, we will work with materialized dataframes. However, when doing data engineering or wanting to speed up the code, we often work with lazy dataframes. It requires a little more typing to work with lazy dataframes. Usually, this code is at the start and end of an existing chain (using `scan_csv` instead of `read_csv` and adding `.collect` at the end). The rest of the code is generally the same. However, the speed benefits can be significant, especially when working with larger datasets. We also don't need to concern ourselves as much with the order of operations as the query engine can perform optimizations that would slow down eager mode (such as doing filtering after joining).

Not all operations are supported on lazy dataframes. The creator of Polars, Ritchie Vink, has described the library as a query engine with a dataframe API. If you have a long chain or pipeline of operations, it will try to determine the optimal order of operations. However, some operations

1. Getting Started with Polars

are *pipeline breaking* and will not work without the knowledge found in the materialized dataframe. For example, if you want to pivot the data (this puts the values from a single column into column headers), you need to know all the unique values in the column. You can't do this with a lazy dataframe, because it doesn't know the unique values until it is materialized.

Let's see what operations are not supported on lazy dataframes. I will use the Python `dir` function along with set operations to see what operations are supported on materialized dataframes but not on lazy dataframes.

```
>>> df_ops = set(x for x in dir(pl.DataFrame()) if not x.startswith('_'))
>>> lazy_ops = set(x for x in dir(pl.LazyFrame()) if not x.startswith('_'))
>>> print(sorted(df_ops - lazy_ops))
['apply', 'corr', 'drop_in_place', 'equals', 'estimated_size', 'extend',
 'find_idx_by_name', 'flags', 'fold', 'frame_equal', 'get_column',
 'get_column_index', 'get_columns', 'glimpse', 'hash_rows', 'height',
 'hstack', 'insert_at_idx', 'insert_column', 'is_duplicated',
 'is_empty', 'is_unique', 'item', 'iter_columns', 'iter_rows',
 'iter_slices', 'map_rows', 'max_horizontal', 'mean_horizontal',
 'min_horizontal', 'n_chunks', 'n_unique', 'partition_by', 'pivot',
 'plot', 'product', 'rechunk', 'replace', 'replace_at_idx',
 'replace_column', 'row', 'rows', 'rows_by_key', 'sample', 'shape',
 'shrink_to_fit', 'sum_horizontal', 'to_arrow', 'to_dict', 'to_dicts',
 'to_dummies', 'to_init_repr', 'to_numpy', 'to_pandas', 'to_series',
 'to_struct', 'transpose', 'unstack', 'upsample', 'vstack',
 'write_avro', 'write_csv', 'write_database', 'write_delta',
 'write_excel', 'write_ipc', 'write_ipc_stream', 'write_json',
 'write_ndjson', 'write_parquet']
```

Note

I'm a big fan of staying inside of Jupyter when possible. I like to use tools like `dir`, `?`, and `??` as a first source of information before resorting to using a search engine or AI to answer a question. In general, libraries in the Python data ecosystem tend to have good documentation that not only explains the parameters of functions and methods but also provides examples of using said functionality.

This might seem like a lot of missing functionality. But if you think about those operations, they require knowledge (i.e., materialization) of the underlying data.

Consider how many operations are supported by both eager and lazy dataframes. (I'll note it is odd that exporting JSON is available in lazy mode but not in other formats.)

```
>>> print(sorted(lazy_ops & df_ops))
['approx_n_unique', 'bottom_k', 'cast', 'clear', 'clone', 'columns',
```

```
'count', 'describe', 'drop', 'drop_nulls', 'dtypes', 'explode',
'fill_nan', 'fill_null', 'filter', 'gather_every', 'group_by',
'group_by_dynamic', 'group_by_rolling', 'groupby', 'groupby_dynamic',
'groupby_rolling', 'head', 'interpolate', 'join', 'join_asof', 'lazy',
'limit', 'max', 'mean', 'median', 'melt', 'merge_sorted', 'min',
'null_count', 'pipe', 'quantile', 'rename', 'reverse', 'rolling',
'schema', 'select', 'select_seq', 'set_sorted', 'shift',
'shift_and_fill', 'slice', 'sort', 'std', 'sum', 'tail', 'take_every',
'top_k', 'unique', 'unnest', 'update', 'var', 'width', 'with_columns',
'with_columns_seq', 'with_row_count', 'with_row_index']
```

1.5 Contexts & Expressions

In Polars, we generally don't work directly on a Series or DataFrame. Instead, we work with an expression. This expression is then evaluated in a *context*. There are many contexts available in Polars. Contexts are operations like `filter`, `select`, `group_by`, etc.

Inside these contexts, we can use operations to create columns. These operations generally don't work directly on the data. Instead, we build up an expression that is evaluated in the context.

Let's look at what is available in an expression. Often, expressions are created from the `pl.col` function, which is used to represent a column. This is a key thing to understand about Polars. When we use `pl.col`, we don't have access to the data directly. We describe the operation that we want to perform on the data. Polars will use this expression, build up an execution plan, and then execute the plan.

Here are the operations available on `pl.col`:

```
>>> col = pl.col('sample')
>>> col_ops = set(x for x in dir(col) if not x.startswith('_'))
>>> print(sorted(col_ops))
['abs', 'add', 'agg_groups', 'alias', 'all', 'and_', 'any', 'append',
 'apply', 'approx_n_unique', 'arccos', 'arccosh', 'arcsin', 'arcsinh',
 'arctan', 'arctanh', 'arg_max', 'arg_min', 'arg_sort', 'arg_true',
 'arg_unique', 'arr', 'backward_fill', 'bin', 'bottom_k', 'cast', 'cat',
 'cbrt', 'ceil', 'clip', 'clip_max', 'clip_min', 'cos', 'cosh', 'cot',
 'count', 'cum_count', 'cum_max', 'cum_min', 'cum_prod', 'cum_sum',
 'cumcount', 'cummax', 'cummin', 'cumprod', 'cumsum', 'cumulative_eval',
 'cut', 'degrees', 'deserialize', 'diff', 'dot', 'drop_nans',
 'drop_nulls', 'dt', 'entropy', 'eq', 'eq_missing', 'ewm_mean',
 'ewm_std', 'ewm_var', 'exclude', 'exp', 'explode', 'extend_constant',
 'fill_nan', 'fill_null', 'filter', 'first', 'flatten', 'floor',
 'floordiv', 'forward_fill', 'from_json', 'gather', 'gather_every',
 'ge', 'get', 'gt', 'hash', 'head', 'hist', 'implode', 'inspect',
 'interpolate', 'is_between', 'is_duplicated', 'is_finite', 'is_first',
 'is_first_distinct', 'is_in', 'is_infinite', 'is_last',
```

1. Getting Started with Polars

```
'is_last_distinct', 'is_nan', 'is_not', 'is_not_nan', 'is_not_null',
'is_null', 'is_unique', 'keep_name', 'kurtosis', 'last', 'le', 'len',
'limit', 'list', 'log', 'log10', 'log1p', 'lower_bound', 'lt', 'map',
'map_alias', 'map_batches', 'map_dict', 'map_elements', 'max', 'mean',
'median', 'meta', 'min', 'mod', 'mode', 'mul', 'n_unique', 'name',
'nan_max', 'nan_min', 'ne', 'ne_missing', 'neg', 'not_', 'null_count',
'or_', 'over', 'pct_change', 'peak_max', 'peak_min', 'pipe', 'pow',
'prefix', 'product', 'qcut', 'quantile', 'radians', 'rank', 'rechunk',
'register_plugin', 'reinterpret', 'repeat_by', 'replace', 'reshape',
'reverse', 'rle', 'rle_id', 'rolling', 'rolling_apply', 'rolling_map',
'rolling_max', 'rolling_mean', 'rolling_median', 'rolling_min',
'rolling_quantile', 'rolling_skew', 'rolling_std', 'rolling_sum',
'rolling_var', 'round', 'round_sig_figs', 'sample', 'search_sorted',
'set_sorted', 'shift', 'shift_and_fill', 'shrink_dtype', 'shuffle',
'sign', 'sin', 'sinh', 'skew', 'slice', 'sort', 'sort_by', 'sqrt',
'std', 'str', 'struct', 'sub', 'suffix', 'sum', 'tail', 'take',
'take_every', 'tan', 'tanh', 'to_physical', 'top_k', 'truediv',
'unique', 'unique_counts', 'upper_bound', 'value_counts', 'var',
'where', 'xor']
```

The expression construct is a powerful tool for lazily manipulating data. The result of most expressions is another expression. This allows us to build up a chain of operations. For example, I could cast the column to numbers, fill in the missing values with the mean, filter out the values greater than 100, sample 10 values, and then take the mean of those values. When we are running in lazy mode, this expression is evaluated by the query engine, and the execution plan is optimized. This can be much faster than running the operations on a materialized dataframe.

```
>>> (col
...     .cast(pl.Int32)
...     .fill_null(col.mean())
...     .clip(upper_bound=100)
...     .sample(10)
...     .mean()
... )
<Expr ['col("sample").strict_cast(Int3.') at 0x10416B8B0>
```

This style of writing code as continuous operations is called *method chaining*. It is a common pattern in Python and is used extensively in Pandas. However, it is not the only way to write code. We could also write the same code like this:

```
ex1 = col.cast(pl.Int64)
ex2 = ex1.fill_null(col.cast(pl.Int64))
ex3 = ex2.clip(upper_bound=100)
```

```
ex4 = ex3.sample(10)
ex5 = ex4.mean()
```

I prefer to surround the operations with parentheses, allowing us to ignore the normal Python line continuation rules. Now, we can put each operation on a separate line, making it easier to read. This makes the code look like a recipe. You will recognize this coding style if you are familiar with my *Effective Pandas* book. I strongly recommend that you use this style of coding when you are working with Polars. It will make your code easier to read and understand. But Polars will also be able to optimize the chain and make it as fast as possible.

Note that you can also create expressions using an attribute syntax. The documentation suggests that this is not preferred but requires slightly less typing and that the column names be valid Python attribute names.

```
>>> pl.col.a_column
<Expr ['col("a_column")'] at 0x163A46110>
```

1.6 Reading CSV Files

Enough theory. Let's see Polars in action.

We will start by reading a comma-separated value (CSV) file. CSV files are a common way to store tabular data. Polars can read most CSV files easily. We will read the file using the `pl.read_csv` function. I will load the Fuel Economy dataset from the US Department of Energy¹. This dataset contains information about the fuel economy of cars from 1984 to 2021. It is a relatively small dataset, but it is large enough to demonstrate the power of Polars.

The file that comes from the Fuel Economy website is a ZIP file. Unfortunately, Polars can't read ZIP files directly. We will use the Python `zipfile` library to extract the CSV file from the ZIP file. Then, we can read the CSV file with Polars.

```
>>> url = '../pylib/datasets/data/vehicles.csv.zip'
>>> import zipfile
>>> with zipfile.ZipFile(url) as z:
...     z.extractall('data/')
>>> path = 'data/vehicles.csv'
>>> df = pl.read_csv(path, null_values=['NA'])
```

Let's look at what the data looks like. Note that the `read_csv` function returns a materialized dataframe, so we can see the data immediately. When we print the dataframe, we will see a few of the first and last rows of the dataframe.

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

1. Getting Started with Polars

```
>>> print(df)
shape: (41_144, 83)
+-----+-----+-----+-----+-----+-----+-----+
| barrels0 | barrelsA | charge12 | . | phevCit | phevHwy | phevCom |
+-----+-----+-----+-----+-----+-----+-----+
| 8       | 08      | 0        | .   | y       | i64     | b       |
| f64    | f64     | f64     |     | i64    |          | i64    |
+-----+-----+-----+-----+-----+-----+-----+
| 15.70  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
| 29.96  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
| 12.21  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
| .       | .       | .       | .   | .       | .       | .       |
| 15.70  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
| 15.70  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
| 18.31  | 0.00    | 0.00    | .   | 0       | 0       | 0       |
+-----+-----+-----+-----+-----+-----+-----+
```

The number of rows in the dataframe is shown at the very top of the output: 41,144 rows and 83 columns. Next, you see a table with the column names and the data type in each column. The first column is the The first column is the *barrels0* column. This is the number of barrels of fuel the car uses yearly. It is a f64 type, a 64-bit floating-point number. A few more columns follow this. In the middle is an ellipsis (...) that indicates that there are more columns, but they are not shown. Note that there are 83 columns, but only a few are shown. We can configure how many columns are shown using the `pl.Config.set_tbl_cols` function. For this book, I set it to six columns.

Many of my students immediately ask how to update the number of columns shown. I suggest that you hold back on changing this setting. Humans have not evolved to scroll through 83 columns of data. Instead, I suggest you filter and limit the columns to view the data you want to see or use visualizations to see patterns in the data.

1.7 Lazy CSV Reading

Let's see what happens if we use the `pl.scan_csv` function to read the same file. This function returns a lazy dataframe. We can see that it is lazy because it doesn't show us the data. It shows a "plan".

```
>>> lazy = pl.scan_csv(path, null_values=['NA'])
>>> print(lazy)
naive plan: (run LazyFrame.explain(optimized=True) to see the optimized
plan)
```

```
Csv SCAN data/vehicles.csv
PROJECT */83 COLUMNS
```

When we print out the result, we see it as a lazy dataframe. It says the current plan is to read the CSV file and *project* all the columns. Project means

1.8. Data Type Inference and Manual Overrides

to select all the columns. Let's filter out some rows, select only a few columns, and then look at the plan again. I'm using the `pl.col` function to create an expression to filter rows and run it in the `.filter` context. Then, I will use the `.select` context to select only a few columns.

```
>>> print(lazy
...     .filter((pl.col('year') >= 1990) & (pl.col('year') < 2000))
...     .select(['year', 'make', 'model'])
... )
naive plan: (run LazyFrame.explain(optimized=True) to see the optimized
plan)

SELECT [col("year"), col("make"), col("model")] FROM
  FILTER [[(col("year")) >= (1990))] & [(col("year")) < (2000)]]] FROM

Csv SCAN data/vehicles.csv
PROJECT */83 COLUMNS
```

Now, it says the plan has a select and a filter operation. However, it does not show any query optimizations.

Polars can be intelligent when reading the CSV file. It will only read the needed columns and rows due to our filter and select operations. We can see that plan by running the `.explain` method.

```
>>> print(lazy
...     .filter((pl.col('year') >= 1990) & (pl.col('year') < 2000))
...     .select(['year', 'make', 'model'])
...     .explain()
... )
FAST_PROJECT: [year, make, model]

Csv SCAN data/vehicles.csv
PROJECT 3/83 COLUMNS
SELECTION: [[(col("year")) >= (1990))] & [(col("year")) < (2000)]]
```

The plan says it will only read the needed columns, `year`, `make`, and `model`. Then, it will filter out the rows that match the filter. This is a relatively simple optimization, but remember, we have 83 columns in the dataset. Reading a small percentage of the columns can yield a significant performance improvement.

1.8 Data Type Inference and Manual Overrides

CSV files have a big benefit: they are human-readable. However, that is about the extent of their benefits. They are not ideal for storing data. One of the

1. Getting Started with Polars

reasons is that they don't store data types. Instead, they store the data as text. Also, there is no optimization for storing data. For example, if you have a column of integers, you can store them as 32-bit integers, 64-bit integers, or even 8-bit integers. However, CSV files will store them as text. This means that the file will be larger than it needs to be. It also means that it will take longer to read the file.

Other file formats, such as Apache Parquet, can be much faster. However, due to the ubiquity of CSV files, we should learn how to deal with them.

When Polars reads a CSV file, it will try to infer the data types of each column. This is a common feature of many data analysis libraries. However, type inference is not always accurate. Or it might be a little sloppy. For example, a column that stores the year might be inferred as an integer. And it would probably be stored as a 64-bit integer. However, we know the year will never exceed four digits. So, we could store it as a 16-bit integer. This would save space. You want to be aware of space usage when working with datasets. Even though Polars can handle datasets larger than memory, working with datasets that fit in memory is still faster. Shrinking the dataset size can help process more data and do it faster.

Let's look at the data types that were inferred for the columns in the Fuel Economy dataset. We can use the `.dtypes` attribute to see the data types of each column.

```
>>> print(df.dtypes)
[Float64, Float64, Float64, Float64, Int64, Float64, Int64, Float64,
 Float64, Float64, Float64, Int64, Int64, Float64, Float64, Int64,
 Float64, Int64, Float64, Float64, Float64, Int64, Float64, String,
 Int64, String, Int64, Int64, Int64, String, String, String, Int64,
 Int64, Int64, Float64, Int64, Float64, Float64, Float64, Float64,
 Int64, Int64, Int64, Int64, String, String, String, Boolean,
 Int64, Int64, Int64, Float64, Float64, Float64, Float64, String,
 Float64, Float64, Float64, String, Int64, Int64, String,
 String, String, String, String, String, String, String, String,
 String, String, String, String, Int64, Int64, Int64]
```

A list of 83 types is not convenient to work with. Instead, I'll use the `.schema` attribute to get a mapping of column names to data types.

Let's look at the `.schema` property.

```
>>> print(df.schema)
OrderedDict([('barrels08', Float64), ('barrelsA08', Float64),
 ('charge120', Float64), ('charge240', Float64), ('city08', Int64),
 ('city08U', Float64), ('cityA08', Int64), ('cityA08U', Float64),
 ('cityCD', Float64), ('cityE', Float64), ('cityUF', Float64), ('co2',
 Int64), ('co2A', Int64), ('co2TailpipeAGpm', Float64),
 ('co2TailpipeGpm', Float64), ('comb08', Int64), ('comb08U', Float64),
 ('combA08', Int64), ('combA08U', Float64), ('combE', Float64),
```

1.8. Data Type Inference and Manual Overrides

```
('combinedCD', Float64), ('combinedUF', Float64), ('cylinders', Int64),  
('displ', Float64), ('drive', String), ('engId', Int64), ('eng_dscr',  
String), ('feScore', Int64), ('fuelCost08', Int64), ('fuelCostA08',  
Int64), ('fuelType', String), ('fuelType1', String), ('ghgScore',  
Int64), ('ghgScoreA', Int64), ('highway08', Int64), ('highway08U',  
Float64), ('highwayA08', Int64), ('highwayA08U', Float64),  
('highwayCD', Float64), ('highwayE', Float64), ('highwayUF', Float64),  
('hvl', Int64), ('hpv', Int64), ('id', Int64), ('lv2', Int64), ('lv4',  
Int64), ('make', String), ('model', String), ('mpgData', String),  
('phevBlended', Boolean), ('pv2', Int64), ('pv4', Int64), ('range',  
Int64), ('rangeCity', Float64), ('rangeCityA', Float64), ('rangeHwy',  
Float64), ('rangeHwyA', Float64), ('trany', String), ('UCity',  
Float64), ('UCityA', Float64), ('UHighway', Float64), ('UHighwayA',  
Float64), ('VClass', String), ('year', Int64), ('youSaveSpend', Int64),  
('guzzler', String), ('trans_dscr', String), ('tCharger', String),  
('sCharger', String), ('atvType', String), ('fuelType2', String),  
('rangeA', String), ('evMotor', String), ('mfrCode', String),  
('c240Dscr', String), ('charge240b', Float64), ('c240bDscr', String),  
('createdOn', String), ('modifiedOn', String), ('startStop', String),  
('phevCity', Int64), ('phevHwy', Int64), ('phevComb', Int64)])
```

When we read files that don't provide type information (like CSV and JSON files), Polars will try to infer the types. Numeric values will always use 64-bit numbers. This might be too generous for integer values. For example, the `year` column is an integer, but we don't need 64 bits to store the year values. Also, the `createdOn` column contains date information but comes back as a string of characters.

To make the book's examples more approachable, I will not examine all 83 columns of this dataset. Instead, I will select a few columns to work with that represent the typical data you will encounter in your work. This mixes numeric, string, categorical, and date data.

I will look at the cars' year, make, and model. This data also contains engine information, such as the engine displacement, number of cylinders, and the transmission type. It has a vehicle classification that describes the vehicle type, such as a car, truck, or SUV. It also has the fuel type, city, highway fuel economy, and the barrels of fuel used yearly. There is also a date column that reflects when the data was collected.

```
>>> cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany', 'drive',  
...           'VClass', 'fuelType', 'barrels08', 'city08', 'highway08',  
...           'createdOn']
```

I will use the `.select` context to view these columns. This method takes a list of column names and returns a new dataframe with only those columns.

If we only want to limit the columns but not perform other operations, we can use the `.select` method and pass a list of column names. We can also pass the names into `pl.col` if we want to chain operations.

1. Getting Started with Polars

Below, I'm just selecting the columns that I want to work with.

```
>>> print(df.select(cols))
```

```
shape: (41_144, 13)
```

year	make	model	.	city08	highway08	created0
i64	str	str	.	i64	i64	n
1985	Alfa Romeo	Spider Veloce 2000	.	19	25	Tue Jan 01 00:00:00 EST 2013
1985	Ferrari	Testarossa	.	9	14	Tue Jan 01 00:00:00 EST 2013
1985	Dodge	Charger	.	23	33	Tue Jan 01 00:00:00 EST 2013
.
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD Turbo	.	16	21	Tue Jan 01 00:00:00 EST 2013

Note

Remember, the examples in the book limit the columns displayed to six. However, looking at the top of the output, you will see the shape: (41_144, 13). This indicates that this dataset has 41,144 rows and 13 columns.

I'm doing the same thing here, but using the pl.col function to create the expressions.

```
>>> print(df.select(pl.col(cols)))
```

```
shape: (41_144, 13)
```

1.8. Data Type Inference and Manual Overrides

year	make	model	.	city08	highway08	created0
i64	str	str	.	i64	i64	n str
1985	Alfa Romeo	Spider Veloce 2000	.	19	25	Tue Jan 01 00:00:00 EST 2013
1985	Ferrari	Testarossa	.	9	14	Tue Jan 01 00:00:00 EST 2013
1985	Dodge	Charger	.	23	33	Tue Jan 01 00:00:00 EST 2013
.
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD Turbo	.	16	21	Tue Jan 01 00:00:00 EST 2013

Let's write some code to change the types. I will write the code throughout this chapter in the way I would create it in a notebook. I'll use chaining and parentheses to make the code easier to read. I will build up a chain of operations. In a typical notebook, I would run one operation, then go back and update the chain to add the next operation after I validate that the previous operation worked.

However, I will write each step in this book as a separate code block to show you my process.

My first goal is to examine the integer columns. Because I already limited the columns to the ones I want to work with, I will continue to work off of that method chain. I'll select the integer columns with the `.select` method and then use the `.describe` method to get the minimum and maximum values for each integer column.

```
>>> print(df
...     .select(pl.col(cols))
```

1. Getting Started with Polars

```
...     .select(pl.col(pl.Int64))
...     .describe()
...
shape: (9, 5)
  statistic      year    cylinders   city08   highway08
  str          f64        f64        f64        f64
  count      41144.00  40938.00  41144.00  41144.00
  null_count  0.00      206.00     0.00      0.00
  mean       2001.54     5.72      18.37     24.50
  .
  50%       2002.00     6.00      17.00     24.00
  75%       2011.00     6.00      20.00     28.00
  max       2020.00    16.00     150.00    124.00
```

Adding and Selecting Columns

```
(sales
  .with_columns(
    (pl.lit('UPC-') + pl.col('UPC')
      .cast(pl.Utf8)).alias('UPC'),
    Total=pl.col('Count') * pl.col('Cost'))
  )
  with_columns preserves and adds columns
  [UPC] [Count] [Date] [Cost] [Total]
  UPC-123 1.33 Mon 0.40 0.53
  UPC-123 1.00 Tue 0.50 0.50
  UPC-456 5.00 Mon 0.10 0.50
(sales
  .select(
    pl.col('UPC'),
    Total=pl.col('Count') * pl.col('Cost'))
  )
  select pulls out columns
  [UPC] [Total]
  123 0.53
  123 0.50
  456 0.50
```

Figure 1.1: Polars has a `.with_columns` method to add or update columns. The `.select` method pulls out or creates selected columns.

The `year` column can be a 16-bit integer, and the other columns can be 8-bit unsigned integers. You can update them by using the `.with_columns` datafram method. This method is used when you want to preserve all existing columns and potentially add or update existing ones. It takes a column expression and returns a new datafram with the column added or updated. Inside the `.with_columns` call, we can use the `pl.col` function to create an expression to cast the column to the desired type.

To drive the point, we aren't directly working with the data in the `year` column in the expression. We are indicating to Polars that there is a `year`

1.8. Data Type Inference and Manual Overrides

column in the data, and we want to cast it to a 16-bit integer. When the `.with_columns` context runs, Polars will read the data from the `year` column and cast it to a 16-bit integer.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                  pl.col('cylinders').cast(pl.UInt8),
...                  pl.col('highway08').cast(pl.UInt8),
...                  pl.col('city08').cast(pl.UInt8)))
... )
```

shape: (41_144, 13)

year	make	model	.	city08	highway08	createdOn
i16	str	str	.	u8	u8	str
1985	Alfa Romeo	Spider Veloce 2000	.	19	25	Tue Jan 01 00:00:00 EST 2013
1985	Ferrari	Testarossa	.	9	14	Tue Jan 01 00:00:00 EST 2013
1985	Dodge	Charger	.	23	33	Tue Jan 01 00:00:00 EST 2013
.
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD Turbo	.	16	21	Tue Jan 01 00:00:00 EST 2013

Note that if we try to change to an integer type that is too small, Polars will raise an error. For example, let's change the `year` column to an 8-bit integer.

```
>>> print(df
...     .select(pl.col(cols))
```

1. Getting Started with Polars

```
... .with_columns(pl.col('year').cast(pl.Int8))
...
Traceback (most recent call last)
...
ComputeError: conversion from `i64` to `i8` failed in column 'year' for
41144 out of 41144 values: [1985, 1985, . 1993]
```

I recommend using `.describe` to examine all the integer columns and determine if the column can be cast to a smaller integer type. This is especially relevant when you ingest data from a CSV file.

1.9 Automatic Type Shrinking

Polars includes a column expression to shrink numeric columns, the `.shrink_dtype` method. This convenience method can be handy. It seems to do a good job with our data.

```
>>> print(df
...     .select(pl.col(cols).shrink_dtype())
... )
shape: (41_144, 13)
   year      make       model      .    city08  highway08  created0
   i16        str        str      .    i16        i8          n
                                         str
   1985  Alfa Romeo      Spider    .    19        25      Tue Jan
                                         Veloce
                                         2000
                                         .          .
                                         01      00:00:00
                                         EST 2013
   1985  Ferrari      Testarossa  .    9         14      Tue Jan
                                         a
                                         .          .
                                         01      00:00:00
                                         EST 2013
   1985  Dodge       Charger    .    23        33      Tue Jan
                                         .
                                         .          .
                                         01      00:00:00
                                         EST 2013
   .
   .
   1993  Subaru      Legacy AWD    .    18        24      Tue Jan
                                         .
                                         .
                                         01      00:00:00
                                         EST 2013
   1993  Subaru      Legacy AWD    .    18        24      Tue Jan
                                         .
                                         .
                                         01      00:00:00
                                         EST 2013
```

1993	Subaru	Legacy	.	16	21	Tue Jan
		AWD Turbo				01
						00:00:00
						EST 2013

I would caution you to be careful about this method. I prefer artisanal column cleanup. Tools that do everything automatically seem to work well 80% of the time but cause issues at other times. For example, this does not use unsigned integers. In our data, `city08` can be an unsigned 8-bit integer, but this casts it as a 16-bit signed integer.

I find that many, upon learning about such conveniences, will wantonly apply them without checking the results.

I would suggest manual inspection and casting of columns. If I were to use this tool, I would run it and then take the types that it gives and manually apply them in the chain.

1.10 Float Conversion

Let's look at the float columns. We can use the same technique to find the minimum and maximum values, but we are more concerned with the number of decimal places.

Converting a float from 64-bit to 32-bit representation is a *type conversion*, commonly known as downcasting from `float64` to `float32`, which can be particularly beneficial when memory efficiency is crucial and precision is less of a concern. This is often seen in large datasets, where halving the size of floating-point numbers can significantly reduce overall memory usage. In machine learning and data analysis tasks, this downcasting can lead to faster computation and reduced memory load, especially when dealing with millions of data points.

However, it's important to note that `float32` has a lower precision than `float64`. This conversion might not be suitable for applications requiring high numerical precision, such as financial calculations or scientific simulations. Nonetheless, in many real-world applications, the precision offered by `float32` is sufficient, and the benefits of reduced memory footprint and computational speed are considerable.

Also, realize that if the data uses less memory, you can process more at once.

Rather than look at the first and last rows, I will use the `p1.sample` method to get a sample of the data. This context will return random rows from the dataframe. I will use the `n` parameter to get ten rows. I also use the `seed` parameter to set the random seed. This allows me to get the same random rows each time I run the code. This is useful when writing code and wanting to get the same results each time you run the code. It is also useful when you are writing tests. You can set the seed to a known value and test the results.

1. Getting Started with Polars

I like using sampling because the first rows are often not representative of the data. They might be test data or incomplete as data is often added to over time.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                  pl.col('cylinders').cast(pl.UInt8),
...                  pl.col('highway08').cast(pl.UInt8),
...                  pl.col('city08').cast(pl.UInt8))
...     .select(pl.col(pl.Float64))
...     .sample(n=10, seed=42)
... )
shape: (10, 2)
displ    barrels08
f64      f64
1.50    12.68
3.40    17.35
3.20    17.35
.
.
3.00    17.35
4.30    21.97
1.50    10.63
```

For these columns, I'm comfortable with the precision of 32-bit floats. Let's update the chain so that these columns will use 32-bit floats.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                  pl.col('cylinders').cast(pl.UInt8),
...                  pl.col('highway08').cast(pl.UInt8),
...                  pl.col('city08').cast(pl.UInt8),
...                  pl.col('displ').cast(pl.Float32),
...                  pl.col('barrels08').cast(pl.Float32),
...                  )
... )
shape: (41_144, 13)
year    make        model      .  city08  highway08  created0
i16     str         str        .   u8       u8          n
                    str
1985  Alfa        Spider     .   19       25        Tue Jan
                    Romeo      Veloce    2000
                                01
                                00:00:00
```

1.11. Extracting Numbers from Strings

1985	Ferrari	Testarossa	.	9	14	EST 2013
		a				Tue Jan 01 00:00:00
1985	Dodge	Charger	.	23	33	EST 2013
.	Tue Jan 01 00:00:00
1993	Subaru	Legacy AWD	.	18	24	EST 2013
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00
1993	Subaru	Legacy AWD Turbo	.	16	21	EST 2013
						Tue Jan 01 00:00:00
						EST 2013

You'll note that I'm not storing the intermediate dataframes. I'm just chaining the operations together, building up the chain as I go through my cleaning process. I recommend that you do the same. You don't need to copy the code each time you work with a new column. You can update the same code in a cell. However, I like to run it after each new link to ensure it works as expected.

1.11 Extracting Numbers from Strings

The *trany* column of this dataset shows a common occurrence in data: embedding two pieces of information in a single column. Look at the data.

```
>>> print(df
... .select('trany')
...
shape: (41_144, 1)
| trany
| str
|
| Manual 5-spd
| Manual 5-spd
| Manual 5-spd
| .
| .
```

1. Getting Started with Polars

Automatic 4-spd
Manual 5-spd
Automatic 4-spd

It looks to be of the form transmission-type number-of-gears. A column with two pieces of information in it may seem clever, but it is not a good idea. If this were my data, I would want to separate this information into two columns. We can't do math on the number of gears because it is stored as a string. If we wanted to do machine learning on the data, we would lose information if we treated it as a single column. It is better to split it into two columns.

I can create a new column called *automatic* by chaining an expression. First, because *trany* is a string, it has the `.str` accessor. Then, I can lowercase it and use the `.contains` method to see if the string contains the word "automatic". This will create a boolean column. Finally, I can name the column to *is_automatic*.

Let's try it out. I'll use `.select` to focus on the *trany* column.

We could convert this column to categorical data. But there is *ordinal* information in the speeds that we could not take advantage of if we converted the column to a category.

```
>>> print(df
...  .select('trany',
...           pl.col('trany')
...             .str.to_lowercase()
...             .str.contains('automatic')
...             .alias('automatic'))
... )
shape: (41_144, 2)
  trany          automatic
  str            bool
  Manual 5-spd  false
  Manual 5-spd  false
  Manual 5-spd  false
  .
  Automatic 4-spd  true
  Manual 5-spd  false
  Automatic 4-spd  true
```

Once I'm satisfied that the expression is correct, I can move it from my `.select` call to my `.with_columns` chain. Note that I am using the `.alias` method to set the column's name.

```
>>> print(df
...  .with_columns('trany',
```

1.11. Extracting Numbers from Strings

```
...     pl.col('trany')
...         .str.to_lowercase()
...         .str.contains('automatic')
...         .alias('is_automatic'))
...
shape: (41_144, 84)
  barrels0    barrelsA    charge12    .    phevHwy    phevCom    is_auto
  8          08           0           .      i64        b           i64        bool
  f64        f64          f64
  15.70      0.00        0.00       .      0          0        false
  29.96      0.00        0.00       .      0          0        false
  12.21      0.00        0.00       .      0          0        false
  .
  15.70      0.00        0.00       .      0          0        true
  15.70      0.00        0.00       .      0          0        false
  18.31      0.00        0.00       .      0          0        true
```

I'm not a huge fan of using `.alias` to set the name of a column. I prefer to use keyword arguments. This way, the column name acts as a hint to the reader. It makes the expression easier to understand. Note that keyword arguments need to follow positional arguments in the method call. This might be problematic if you are concerned with a specific order of columns. However, as I'll show later, I like to add an additional step at the end of my chain to explicitly set the order and names of the columns I want as the output of my chain.

Also, keyword arguments need to be valid Python identifiers. This means that they can't contain spaces or special characters. If you need special characters in your column names, use `.alias` or pass in the column name and expression as a dictionary and unpack it with the `**` operator.

```
>>> print(df
...     .with_columns('trany',
...         is_automatic=pl.col('trany')
...             .str.to_lowercase()
...             .str.contains('automatic'))
...
shape: (41_144, 84)
  barrels0    barrelsA    charge12    .    phevHwy    phevCom    is_auto
  8          08           0           .      i64        b           i64        bool
  f64        f64          f64
  15.70      0.00        0.00       .      0          0        false
  29.96      0.00        0.00       .      0          0        false
  12.21      0.00        0.00       .      0          0        false
  .
  15.70      0.00        0.00       .      0          0        true
  15.70      0.00        0.00       .      0          0        false
  18.31      0.00        0.00       .      0          0        true)
```

1. Getting Started with Polars

15.70	0.00	0.00	.	0	0	true
15.70	0.00	0.00	.	0	0	false
18.31	0.00	0.00	.	0	0	true

Note

I'm lowercasing the *trany* column to *normalize* the values in case some values have different capitalization styles. I can make these column expressions as long as necessary, but there will be a computation penalty.

It might be better to do some exploration and see how many different entries there are for this column:

```
>>> print(df
...     .group_by('trany')
...     .len()
...     .sort('len', descending=True)
... )
```

shape: (38, 2)

trany	len
str	u32
Automatic 4-spd	11047
Manual 5-spd	8361
Automatic 3-spd	3151
.	.
Automatic (AM-S9)	3
Automatic (L4)	2
Automatic (L3)	2

In this case, it appears that it is spelled consistently. So, I really don't need to lowercase it. However, there are some missing values. We will need to decide what to do with those.

Note

Polars provides a `.value_counts` expression method that creates a column with a struct of value and count. You can unnest it to get the same results as above:

```
>>> print(df
...     .select(pl.col('trany')
...             .value_counts(sort=True))
...     .unnest('trany')
... )
```

shape: (38, 2)

trany	count
str	u32
Automatic 4-spd	11047
Manual 5-spd	8361
Automatic 3-spd	3151
.	.
Automatic (AM-S9)	3
Automatic (L4)	2
Automatic (L3)	2

Note

Here are the rows where the *trany* is missing:

```
>>> print(df
...     .filter(pl.col('trany').is_null())
...     .select('year', 'make', 'model', 'VClass')
... )
```

year	make	model	VClass
i64	str	str	str
2000	Nissan	Altra EV	Midsize Station Wagons
2000	Toyota	RAV4 EV	Sport Utility Vehicle - 2WD
2001	Toyota	RAV4 EV	Sport Utility Vehicle - 2WD
.	.	.	.
2003	Toyota	RAV4 EV	Sport Utility Vehicle - 2WD
1984	Ford	F150 Pickup 2WD	Standard Pickup Trucks 2WD
1984	GMC	C25 Pickup 2WD	Standard Pickup Trucks 2WD

I'm going to decide to make missing values *Automatic*. (Later code would show how to use the `pl.when` function to be more specific.)

Here's the code to handle the transmission type extraction:

```
>>> print(df
...     .select('trany',
...             is_automatic=pl.col('trany')
...                     .str.contains('Automatic')
...                     .fill_null(True)
...             )
... )
```

trany	is_automatic
str	bool

1. Getting Started with Polars

Manual	5-spd	false
Manual	5-spd	false
Manual	5-spd	false
.	.	.
Automatic	4-spd	true
Manual	5-spd	false
Automatic	4-spd	true

Now, let's pull out the other bit of information in the *trany* column, which is the number of gears. I will use a *regular expression* to extract the number of gears or speeds. I'll use the `.str` accessor again, but I'll use the `.extract` method this time. This method takes a regular expression and returns a new column with the extracted values. I'll use the regular expression `(\d+)` to extract the digits.

```
>>> print(df
... .select(num_gears=pl.col('trany')
...         .str.extract(r'(\d+)')
...         .cast(pl.UInt8))
... )
shape: (41_144, 1)
+-----+
| num_gears |
| u8          |
+-----+
| 5           |
| 5           |
| 5           |
| .           |
| 4           |
| 5           |
| 4           |
+-----+
```

That looks pretty good.

I'll chain a call to `.unique` to see what values are in the column. Regular expressions are notoriously difficult to get right. In this case, I want to ensure that I get things like 10-speed transmissions or rows that don't have a number of gears.

```
>>> print(df
... .select(num_gears=pl.col('trany')
...         .str.extract(r'(\d+)')
...         .cast(pl.UInt8)
...         .unique())
... )
shape: (10, 1)
```

num_gears
u8
null
1
3
.
8
9
10

That looks good. I'm going to also see where those values are missing.

```
>>> print(df
...     .select(num_gears=pl.col('trany')
...             .str.extract(r'(\d+)')
...             .cast(pl.UInt8))
...     .filter(pl.col('num_gears').is_null())
... )
shape: (837, 1)
| num_gears |
| u8         |
| null       |
| null       |
| null       |
| .          |
| null       |
| null       |
| null       |
```

I'll set missing gears to 6 and combine both new columns to the data:

```
>>> print(df
...     .with_columns(
...         is_automatic=pl.col('trany')
...             .str.contains('Automatic')
...             .fill_null(True),
...         num_gears=pl.col('trany')
...             .str.extract(r'(\d+)')
...             .cast(pl.UInt8)
...             .fill_null(6)
...     )
... )
shape: (41_144, 85)
| barrels0   barrelsA   charge12   .   phevCom   is_auto   num_gear |
```

1. Getting Started with Polars

8 f64	08 f64	0 f64	b i64	matic bool	rs u8
15.70	0.00	0.00	.	0	false
29.96	0.00	0.00	.	0	false
12.21	0.00	0.00	.	0	false
.
15.70	0.00	0.00	.	0	true
15.70	0.00	0.00	.	0	false
18.31	0.00	0.00	.	0	true

Now, instead of a column that looks pretty but has data encoded in a string, we have two columns that we can use for analysis.

1.12 String Columns

Let's look at the string columns in the dataset. You can't shrink a string column to a smaller size. However, you can change the data type from a string to a categorical type. This can be beneficial when you have a column with a small number of unique values.

Also, dates are often stored as strings when loaded from a CSV file. We will want to convert those to a date type.

I will use a column expression, `pl.col(pl.String)`, to select just the columns with the Utf-8 type to inspect them.

Note

In Polars, `pl.Utf8` and `pl.String` are synonyms. You can use either. `pl.String` is probably slightly easier to read.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                  pl.col('cylinders').cast(pl.UInt8),
...                  pl.col('highway08').cast(pl.UInt8),
...                  pl.col('city08').cast(pl.UInt8),
...                  pl.col('displ').cast(pl.Float32),
...                  pl.col('barrels08').cast(pl.Float32),
...                  )
...     .select(pl.col(pl.String)))
... )
shape: (41_144, 7)
   make      model      trany      .    VClass      fuelTyp      created
   str       str       str       str      str       e        str

```

1.12. String Columns

Alfa Romeo	Spider Veloce 2000	Manual 5-spd	.	Two Seaters	Regular	Tue Jan 01 00:00:00 EST 2013
Ferrari	Testarossa	Manual 5-spd	.	Two Seaters	Regular	Tue Jan 01 00:00:00 EST 2013
Dodge	Charger	Manual 5-spd	.	Subcompact Cars	Regular	Tue Jan 01 00:00:00 EST 2013
.	.	.	.	Compact Cars	Regular	.
Subaru	Legacy AWD	Automatic 4-spd	.	Compact Cars	Regular	Tue Jan 01 00:00:00 EST 2013
Subaru	Legacy AWD	Manual 5-spd	.	Compact Cars	Regular	Tue Jan 01 00:00:00 EST 2013
Subaru	Legacy AWD Turbo	Automatic 4-spd	.	Compact Cars	Premium	Tue Jan 01 00:00:00 EST 2013

The *make*, *model*, *VClass*, and *fueltype* columns look like categorical columns. They have a limited number of values. The *trans* column is a bit more complicated. It has two values: the transmission type and the number of gears. Finally, the *createdOn* column is a date. Let's look at each of these in turn.

I'll convert the categoricals in one swoop. I can use the `pl.Categorical` type. Until now, we have been working on individual columns. However, we can also work on multiple columns at once. I can pass in a list of columns I want to cast to the same type. I want to convert *make*, *model*, *VClass*, *drive*, and *fueltype* to categorical. I can pass them in as a list to the column expression and then call the `.cast` method to convert all of them to the `pl.Categorical` type.

1. Getting Started with Polars

I'll also update the code to mass convert the numeric columns by passing in a list of the columns I want to convert to unsigned 8-bit integers and 32-bit floats.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                 pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                 pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                 pl.col(['make', 'model', 'VClass', 'drive',
...                         'fuelType']).cast(pl.Categorical),
...                 )
...     )
shape: (41_144, 13)
```

year	make	model	.	city08	highway08	created0
i16	cat	cat	.	u8	u8	n str
1985	Alfa Romeo	Spider Veloce 2000	.	19	25	Tue Jan 01 00:00:00 EST 2013
1985	Ferrari	Testarossa	.	9	14	Tue Jan 01 00:00:00 EST 2013
1985	Dodge	Charger	.	23	33	Tue Jan 01 00:00:00 EST 2013
.
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD	.	18	24	Tue Jan 01 00:00:00 EST 2013
1993	Subaru	Legacy AWD Turbo	.	16	21	Tue Jan 01 00:00:00 EST 2013

We previously saw how to deal with the transmission column. Let's add that code here.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                 pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                 pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                 pl.col(['make', 'model', 'VClass', 'drive',
...                         'fuelType']).cast(pl.Categorical),
...                 is_automatic=pl.col('trany')
...                         .str.contains('Automatic')
...                         .fill_null(True),
...                 num_gears=pl.col('trany')
...                         .str.extract(r'(\d+)')
...                         .cast(pl.UInt8)
...                         .fill_null(6))
...     )
```

shape: (41_144, 15)

year	make	model	.	created0	is_autom	num_gear
i16	cat	cat	.	n	bool	u8
				str		
1985	Alfa Romeo	Spider Veloce 2000	.	Tue Jan 01 00:00:00 EST 2013	false	5
1985	Ferrari	Testarossa	.	Tue Jan 01 00:00:00 EST 2013	false	5
1985	Dodge	Charger	.	Tue Jan 01 00:00:00 EST 2013	false	5
.
1993	Subaru	Legacy AWD	.	Tue Jan 01 00:00:00 EST 2013	true	4
1993	Subaru	Legacy AWD	.	Tue Jan 01 00:00:00 EST 2013	false	5
1993	Subaru	Legacy AWD Turbo	.	Tue Jan 01 00:00:00 EST 2013	true	4

1. Getting Started with Polars

That chain is looking like a great sequence of operations to clean up this data.

1.13 Parsing Dates

Let's convert the *createdOn* column to a date. I'll use the `.str` accessor again, but this time I'll use the `.str.to_datetime` method. This method takes a special parameter, `format`, that describes the date format. Most date libraries use a format called *strftime*. This is an IEEE C language standard¹. It is a string that describes the format of the date. For example, the format string `%Y-%m-%d` describes a date in the format *year-month-day*. For this column, I'll use the format string `%a %b %d %H:%M:%S %Z %Y`.

Here is a table of the *strftime* format codes.

Table 1.2: Table of *strftime* format codes

Code	Meaning	Example
<code>%a</code>	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)
<code>%A</code>	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)
<code>%w</code>	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6
<code>%d</code>	Day of the month as a zero-padded decimal number.	01, 02, ..., 31
<code>%-d</code>	Day of the month as a decimal number.	1, 2, ..., 30
<code>%b</code>	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)
<code>%B</code>	Month as locale's full name.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)
<code>%m</code>	Month as a zero-padded decimal number.	01, 02, ..., 12
<code>%-m</code>	Month as a decimal number.	1, 2, ..., 12
<code>%y</code>	Year without century as a zero-padded decimal number.	00, 01, ..., 99
<code>%-y</code>	Year without century as a decimal number.	0, 1, ..., 99
<code>%Y</code>	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999

¹The *strftime* format is found at <https://pubs.opengroup.org/onlinepubs/9699919799/>

Code	Meaning	Example
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).	(empty), +0000, -0400, +1030
%Z	Time zone name (empty string if object is naive).	(empty), UTC, EST, CST
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366
%U	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, ..., 53
%W	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, ..., 53
%c	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE)
%x	Locale's appropriate date representation.	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)
%X	Locale's appropriate time representation.	21:30:00 (en_US); 21:30:00 (de_DE)
%%	A literal % character.	%

1. Getting Started with Polars

Let's parse the *createdOn* column.

```
>>> print(df
...     .select(pl.col(cols))
...     .with_columns(pl.col('year').cast(pl.Int16),
...                 pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                 pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                 pl.col(['make', 'model', 'VClass', 'drive',
...                         'fuelType']).cast(pl.Categorical),
...                 is_automatic=pl.col('trany')
...                         .str.contains('Automatic')
...                         .fill_null(True),
...                 num_gears=pl.col('trany')
...                         .str.extract(r'(\d+)')
...                         .cast(pl.UInt8)
...                         .fill_null(6))
...     .select(pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'))
... )
```

```
shape: (41_144, 1)
createdOn
datetime[μs]
+-----+
| 2013-01-01 00:00:00
| 2013-01-01 00:00:00
| 2013-01-01 00:00:00
| .
| 2013-01-01 00:00:00
| 2013-01-01 00:00:00
| 2013-01-01 00:00:00
+-----+
```

Awesome, that looks like it worked. You can see that the type is now `datetime[μs]`. Let's add it to our chain and also convert our chain into a function. Because I don't want to rename the *createdOn* column, I'll place it before the keyword arguments.

```
>>> def tweak_auto(df):
...     cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
...             'drive', 'VClass', 'fuelType', 'barrels08', 'city08',
...             'highway08', 'createdOn']
...     return (df
...             .select(pl.col(cols))
...             .with_columns(pl.col('year').cast(pl.Int16),
...                         pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                         pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                         pl.col(['make', 'model', 'VClass', 'drive',
```

```

...     'fuelType']).cast(pl.Categorical),
...     pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...     is_automatic=pl.col('trany')
...         .str.contains('Automatic')
...         .fill_null('Automatic'),
...     num_gears=pl.col('trany')
...         .str.extract(r'(\d+)')
...         .cast(pl.UInt8)
...         .fill_null(6))
... )

```

Let's try out the function.

```
>>> print(tweak_auto(df))
shape: (41_144, 15)
   year    make      model      .  createdOn  is_autom  num_gear
   i16      cat        cat      .      n      datetime      str      u8
                                         [μs]

  1985  Alfa Spider   .  2013-01-  false      5
        Romeo Veloce   .  01      00:00:00
        2000
  1985  Ferrari Testarosa   .  2013-01-  false      5
        sa      01      00:00:00
  1985  Dodge Charger   .  2013-01-  false      5
        01      00:00:00
  .
  1993  Subaru Legacy   .  2013-01-  true       4
        AWD      01      00:00:00
  1993  Subaru Legacy   .  2013-01-  false      5
        AWD      01      00:00:00
  1993  Subaru Legacy   .  2013-01-  true       4
        AWD      01      00:00:00

```

At this point, I'm feeling pretty confident that I have the data types correct. I've reviewed each column and ensured it is the correct type. I've also made sure that the data is correct. I've checked the minimum and maximum values for each column.

If I'm working in a notebook, I will move this function to the very top of the notebook. That way, I can use it easily the next time I come back to the notebook.

1. Getting Started with Polars

Note

I recommend that you follow this step when you work in notebooks. It is easy to get lost in the code. You might not remember the order of operations if you have strewn the code to clean up the data throughout the notebook. Putting this code in a single function and moving it to the top of the notebook lets you easily see what you did to clean up the data. When you return to the notebook, you can run the function and get the data in the correct format. You don't need to remember the order of operations or what you did to clean up the data. This will also make it easier to write tests for your code.

Your colleagues will thank you for this. They will be able to quickly see what you did to clean up the data, and they will be able to use the function in their notebooks as well.

Let's see how much memory we saved by converting the types.

```
>>> tweak_auto(df).estimated_size(unit='mb')  
2.446714401245117
```

It looks like it is now using 2.4MB of memory. Let's look at the original subset of columns before we converted the types.

```
>>> df.select(cols).estimated_size(unit='mb')  
5.914606094360352
```

It was using 5.9MB of memory. That's a nice reduction in memory usage. Not bad.

1.14 Summary Statistics

After I've cleaned the data, I like to look at some summary statistics again and try to understand the data. I'll use the `.describe` method first. This method will give me the count, null count, mean, standard deviation, minimum, 25th percentile, median, 75th percentile, and maximum for each column.

```
>>> print(tweak_auto(df).describe())  
shape: (9, 16)  
  statistic      year       make     .    createdOn   is_automat  
  str          f64        str      .      str        ic      num_gears  
                           .          .          .          .          .  
  
  count      41144.00  41144     .    41144      41144.00  41144.00  
  null_count  0.00      0       .      0      0.00      0.00  
  mean       2001.54  null     .  2013-08-1  0.69      5.04  
                           .          .          .          .          .  
                           3 08:22:0  
                           7.863115
```

std	11.14	null	.	null	null	1.36	
min	1984.00	null	.	2013-01-01 00:00:00	0.00	1.00	
25%	1991.00	null	.	2013-01-01 00:00:00	null	4.00	
50%	2002.00	null	.	2013-01-01 00:00:00	null	5.00	
75%	2011.00	null	.	2013-01-01 00:00:00	null	6.00	
max	2020.00	null	.	2019-06-04 00:00:00	1.00	10.00	

To see how two numeric columns interact, I can use the `.corr` method to compute the Pearson correlation coefficient. This will give me a number between -1 and 1. A value of 1 means that the two columns are perfectly correlated. A value of -1 implies that the two columns are perfectly negatively correlated. A value of 0 means that there is no correlation between the two columns.

Polars has a `.corr` method for computing the correlation between columns. However, this method only works with numeric columns, so I'll use the `cs.numeric` function to select the numeric columns.

Note

The `polars.selectors` module, commonly aliased as `cs`, contains many convenience functions to select columns based on type, regular expressions, and more. In the example below, I use `cs.numeric` to select integer and floating point columns of all sizes.

The `.corr` method returns a dataframe with the correlation coefficients. However, if a column has missing values, it will return `NaN` for that column. Also, it doesn't insert a column to indicate which columns are being correlated.

I'll use the `.pipe` method to add a column to show which columns are being connected. I can use the `.insert_column` method to insert a Series into a dataframe. This is one of the few times I will use a Series directly. I'll use the `pl.Series` constructor to create a Series with the column names. I'm using `.pipe` because I want to work on the current state of the dataframe, the correlation dataframe. I want to grab the columns from the correlation dataframe and insert them as the first column called *names*. The `.pipe` method expects a function that passes in the current state of the dataframe. I'll use a lambda function to pass in the current state of the dataframe. I'll use the

1. Getting Started with Polars

.insert_column method to insert the column. I'll use the pl.Series constructor to create a Series with the column names.

```
>>> import polars.selectors as cs
>>> print(tweak_auto(df)
...     .select(cs.numeric())
...     .corr()
...     .pipe(lambda df_: df_.insert_column(0, pl.Series('names', df_.columns)))
... )
```

shape: (7, 8)

names	year	displ	.	city08	highway08	num_gears
str	f64	f64	.	f64	f64	f64
year	1.00	NaN	.	0.18	0.27	0.70
displ	NaN	NaN	.	NaN	NaN	NaN
cylinders	NaN	NaN	.	NaN	NaN	NaN
.
city08	0.18	NaN	.	1.00	0.94	-0.03
highway08	0.27	NaN	.	0.94	1.00	0.09
num_gears	0.70	NaN	.	-0.03	0.09	1.00

Hmmm. There are many missing values. The Polars implementation of .corr doesn't handle missing values.

In practice, I'll use Pandas instead if I'm doing this in a notebook. It handles missing values better, has support for the Pearson, Spearman, and Kendall correlation coefficients, and I can color the cells to make it easier to see the correlation coefficients.

1.15 Being Lazy

The chaining process that we went through previously was *eager*. We were calculating the results as we went along. Let's make it lazy. We can make an eager dataframe lazy at any time by using the .lazy method. But we can also use pl.scan_csv to read the data lazily.

Why would you want to be lazy? Polars includes a query optimizer. If you have a chain of operations, the optimizer can determine the most efficient way to run the operations.

Here's my lazy version of the code. You can imagine I put all this at the top of my notebook.

```
>>> import polars as pl
>>> path = 'data/vehicles.csv'
>>> lazy = pl.scan_csv(path, null_values=['NA'])

>>> def tweak_auto(df):
...     cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
```

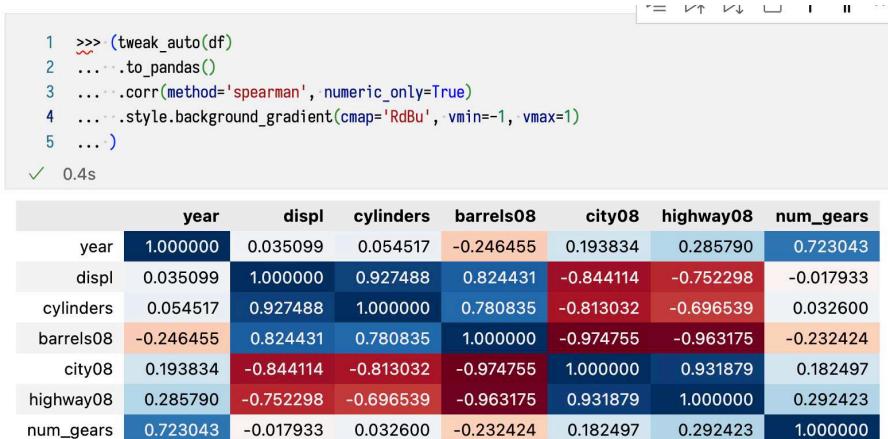


Figure 1.2: A correlation heatmap of numeric columns. In this case, I used Pandas to generate the values and visualize the correlation coefficients by adding a color gradient.

```

...     'drive', 'VClass', 'fuelType', 'barrels08', 'city08',
...     'highway08', 'createdOn']
...     return (df
...         .select(pl.col(cols))
...         .with_columns(pl.col('year').cast(pl.Int16),
...                     pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                     pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                     pl.col(['make', 'model', 'VClass', 'drive',
...                             'fuelType']).cast(pl.Categorical),
...                     pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...                     is_automatic=pl.col('trany')
...                         .str.contains('Automatic')
...                         .fill_null(True),
...                     num_gears=pl.col('trany')
...                         .str.extract(r'(\d+)')
...                         .cast(pl.UInt8)
...                         .fill_null(6))
...     )
... 
```

```

>>> print(tweak_auto(lazy).collect())
shape: (41_144, 15)
   year    make      model      .  created0  is_autom  num_gear
   i16     cat        cat        .    n       atic       s
                           datetime  bool      u8
                           [μs]

```

1. Getting Started with Polars

1985	Alfa Romeo	Spider Veloce 2000	.	2013-01-01	false	5
1985	Ferrari	Testarossa	.	2013-01-01	false	5
1985	Dodge	Charger	.	2013-01-01	false	5
.
1993	Subaru	Legacy AWD	.	2013-01-01	true	4
1993	Subaru	Legacy AWD	.	2013-01-01	false	5
1993	Subaru	Legacy AWD Turbo	.	2013-01-01	true	4
				00:00:00		

In this case, because all of the eager methods I used in my tweak function have a lazy equivalent, I can use the existing function and pass in a lazy dataframe and get a lazy dataframe back. I'll add a .collect method to the end to indicate that I want polars to optimize the chain and execute it. We explore laziness more in-depth in a later chapter. I just wanted you to be aware of it.

1.16 Chapter Functions and Methods

Table 1.3: Table of functions and methods used in this chapter

Function	Description
cs.numeric()	Select integer and float columns
pl.Int64	64-bit integer type
pl.Int32	32-bit integer type
pl.Int16	16-bit integer type
pl.Int8	8-bit integer type
pl.UInt64	64-bit unsigned integer type
pl.UInt32	32-bit unsigned integer type
pl.UInt16	16-bit unsigned integer type
pl.UInt8	8-bit unsigned integer type
pl.Float64	64-bit float type
pl.Float32	32-bit float type
pl.String pl.Utf8	UTF8 string type

Function	Description
<code>pl.Categorical(ordering='physical')</code>	Categorical strings. Ordering can be 'physical' (by appearance) or 'lexical' (alphabetical)
<code>pl.col(*args)</code>	Create a column expression for a column or a list of columns
<code>pl.read_csv(source, *, has_header=True, columns=None, new_columns=None, separator=',', comment_prefix=None, quote_char='"', skip_rows=0, dtypes=None, schema=None, null_values=None, missing_utf8_is_empty_string=False, limits=None, ignore_errors=False, try_parse_dates=False, n_threads=None, infer_schema_length=100, batch_size=8192, n_rows=None, encoding='utf8', low_memory=False, rechunk=True, use_pyarrow=False, storage_options=None, skip_rows_after_header=0, row_count_name=None, row_count_offset=0, sample_size=1024, eol_char='\n', raise_if_empty=True, truncate_ragged_lines=False)</code>	Reads a CSV file into a DataFrame. Includes options for specifying the file source, header presence, column selection and renaming, separators, comment handling, quoting, skipping rows, data types, schema, null value handling, UTF-8 settings, error handling, date parsing, threading, schema inference, batch size, row handling, and file format specifics.

1. Getting Started with Polars

Function	Description
<pre>pl.scan_csv(source, *, has_header=True, separator=',', comment_prefix=None, quote_char='''', skip_rows=0, dtypes=None, schema=None, null_values=None, missing_utf8_is_empty_string=False, ignore_errors=False, cache=True, with_column_names=None, infer_schema_length=100, n_rows=None, encoding='utf8', low_memory=False, rechunk=True, skip_rows_after_header=0, row_count_name=None, row_count_offset=0, try_parse_dates=False, eol_char='\n', new_columns=None, raise_if_empty=True, truncate_ragged_lines=False) df.describe(self, percentiles=(0.25, 0.5, 0.75))</pre>	Lazily reads a CSV file. Options include specifying the source path, header presence, separator, comment prefix, quote character, row skipping, data type definitions, schema, null value handling, UTF-8 settings, error tolerance, caching, column naming functions, schema inference, row limits, encoding, memory optimization, chunk reallocation, additional row skipping after the header, row count column, row count offset, date parsing attempts, end-of-line character, column renaming, empty file handling, and line truncation handling.
<pre>df.filter(*predicates, **constraints)</pre>	Provides summary statistics for a DataFrame. The percentiles parameter allows specifying one or more percentiles to include in the summary, with all values required to be in the range [0, 1].
<pre>df.select(*exprs, **named_exprs)</pre>	Filter by predicates. The constraints are keyword arguments that filter for equality.
<pre>df.with_columns(*exprs, **named_exprs)</pre>	Select columns from this DataFrame. Accepts expressions and literals. Keyword arguments are used to rename columns.
	Add columns to this DataFrame. Accepts expressions and literals. Keyword arguments are used to rename columns.

Function	Description
<code>df.corr(**kwargs)</code>	Computes the Pearson correlation coefficient for all pairs of columns in this DataFrame. Accepts keyword arguments for the numpy <code>corrcoef</code> method.
<code>df.insert_column(index, column)</code>	Inserts a column (Series) at the specified index.
<code>df.dtypes</code>	Property to return a list of datatypes of the columns in the DataFrame.
<code>df.schema</code>	Property to return a mapping of column names to datatypes.
<code>df.sample(n=None, *, fraction=None, with_replacement=False, shuffle=False, seed=None)</code>	Samples rows from the DataFrame. The <code>n</code> parameter specifies the number of rows to return. The <code>fraction</code> parameter specifies the fraction of rows to return. The <code>with_replacement</code> parameter specifies whether to sample with replacement. The <code>shuffle</code> parameter specifies whether to shuffle the rows before sampling. The <code>seed</code> parameter specifies the random seed to use.
<code>df.estimated_size(unit='b')</code>	Returns the estimated size of the DataFrame in the specified unit. The <code>unit</code> parameter can be 'b' (bytes), 'kb' (kilobytes), 'mb' (megabytes), 'gb' (gigabytes), or 'tb' (terabytes). Estimated because multiple columns may share buffers and bitmaps.
<code>df.to_pandas(*args, use_pyarrow_extension_array=False, **kwargs)</code>	Converts the DataFrame to a Pandas DataFrame. Can indicate if it uses pyarrow or numpy. <code>**kwargs</code> are passed to <code>pyarrow.Table.to_pandas</code> .
<code>df.lazy()</code>	Converts an eager DataFrame to a lazy DataFrame.
<code>lazy_df.explain(*, optimized=True, type_coercion=True, predicate_pushdown=True, projection_pushdown=True, simplify_expression=True, slice_pushdown=True, comm_subplan_elim=True, comm_subexpr_elim=True, streaming=True)</code>	Prints the query plan for the lazy DataFrame.

1. Getting Started with Polars

Function	Description
<code>lazy_df.collect(*, type_coercion=True, predicate_pushdown=True, projection_pushdown=True, simplify_expression=True, slice_pushdown=True, comm_subplan_elim=True, comm_subexpr_elim=True, no_optimization=False, streaming=False, background=False, _eager=False)</code>	Executes the lazy DataFrame and returns an eager DataFrame.
<code>col.cast(dtype, *, strict=True)</code> <code>col.shrink_dtype()</code>	Casts the column to the specified type. Shrink numeric columns to smallest signed variety.
<code>col.alias(name)</code> <code>col.unique(*, maintain_order=False)</code>	Renames the column. Returns a column with the unique values of this column. The <code>maintain_order</code> parameter specifies whether to maintain the order of the unique values.
<code>col.str.to_datetime(format, *, time_unit=None, time_zone=None, strict=True, exact=True, cache=True, use_earliest=None, ambiguous='raise')</code>	Converts the column to a datetime column. The <code>format</code> parameter specifies the format of the date. The <code>time_unit</code> parameter specifies the time unit to use ('us', 'ns', 'ms'). The <code>time_zone</code> parameter specifies the time zone to use. The <code>strict</code> parameter specifies whether to raise an error if the date is invalid. The <code>exact</code> parameter specifies whether to raise an error if the date is not exact. The <code>cache</code> parameter specifies whether to cache the results. The <code>use_earliest</code> parameter specifies whether to use the earliest date if the date is ambiguous (None - raise error, True, or False). The <code>ambiguous</code> parameter specifies how to handle ambiguous dates ('raise', 'earliest', 'latest').

Function	Description
<code>col.str.contains(pattern, *, literal=False, strict=True)</code>	Returns a boolean column indicating whether the string contains the specified pattern. The <code>literal</code> parameter specifies whether to interpret the pattern as a literal string. The <code>strict</code> parameter specifies whether to raise an error if the pattern is invalid.
<code>col.str.extract(pattern, *, group_index=1)</code>	Extracts the first group of the specified pattern. The <code>group_index</code> parameter specifies the group to extract.

1.17 Summary

Polars presents a powerful alternative to Pandas for data manipulation and analysis in Python. Its efficient memory management, performance advantages, and lazy evaluation capabilities make it a strong candidate for handling large datasets and complex data operations.

1.18 Exercises

1. What are some of the differences between Polars and Pandas?
2. Take a moderately sized dataset and experiment with changing data types, as shown in this chapter. Observe the changes in memory usage.
3. Compare the performance of lazy versus eager evaluation in Polars by executing a series of data transformations and aggregations on a large dataset. Measure the time taken for both approaches.

Chapter 2

Data Manipulation with Polars Using the Fuel Economy Dataset

2.1 Introduction

This chapter will guide you through managing and analyzing data effectively. You'll learn about various operations, such as adding, removing, and renaming columns, and more complex tasks, such as joining, concatenating, and appending data. These skills are crucial for extracting insights from datasets. By the end of this chapter, you'll be equipped with the knowledge to handle a wide range of data manipulation tasks using Polars, enhancing your data analysis capabilities.

2.2 Getting the Data

We are going to use the same dataset from the previous chapter. If I use this data, I won't repeat the `tweak_auto` function in subsequent chapters.

Let's load the data into a DataFrame.

```
>>> import polars as pl
>>> path = 'data/vehicles.csv'
>>> raw = pl.read_csv(path, null_values=['NA'])

>>> def tweak_auto(df):
...     cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
...             'drive', 'VClass', 'fuelType', 'barrels08', 'city08',
...             'highway08', 'createdOn']
...     return (df
...             .select(pl.col(cols))
...             .with_columns(pl.col('year').cast(pl.Int16),
...                         pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                         pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                         pl.col(['make', 'model', 'VClass', 'drive',
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
...         'fuelType']).cast(pl.Categorical),
...     pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...     is_automatic=pl.col('trany')
...         .str.contains('Automatic')
...         .fill_null('Automatic'),
...     num_gears=pl.col('trany')
...         .str.extract(r'(\d+)')
...         .cast(pl.UInt8)
...         .fill_null(6))
...
)
>>> autos = tweak_auto(raw)
```

In case you forgot, here's the data. It is a listing of car makes and models that are sold in the United States, along with their fuel efficiency ratings.

```
>>> print(autos)
```

```
shape: (41_144, 15)
```

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[tic	u8
				μs]	str	
1985	Alfa Romeo	Spider	.	2013-01-01 00:00:00	false	5
		Veloce	.	1		
		2000	.	00:00:00		
1985	Ferrari	Testarossa	.	2013-01-01 00:00:00	false	5
		a	.	1		
			.	00:00:00		
1985	Dodge	Charger	.	2013-01-01 00:00:00	false	5
			.	1		
			.	00:00:00		
.
1993	Subaru	Legacy	.	2013-01-01 00:00:00	true	4
		AWD	.	1		
			.	00:00:00		
1993	Subaru	Legacy	.	2013-01-01 00:00:00	false	5
		AWD	.	1		
			.	00:00:00		
1993	Subaru	Legacy	.	2013-01-01 00:00:00	true	4
		AWD Turbo	.	1		
			.	00:00:00		

2.3 Adding Columns

This section will cover the basic DataFrame operations necessary for practical data analysis. These include adding, removing, and renaming columns.

Let's assume that we want to create a new column for the fuel efficiency ratio of each vehicle. We can do this by dividing the highway MPG by the city MPG. The `.with_columns` method keeps the existing columns and lets us add or update columns. I'm using keyword arguments to specify the name of the new column, but I could also use the `.alias` method to name the columns instead. Again, I prefer to use keyword arguments when possible because it serves as a hint to the reader about what the column contains. It doesn't hurt that it's less typing, too.

My process will be to use `.select` to get the column I want, and then later, when I feel confident that I have the expression correct, I'll use `.with_columns` to add it to the DataFrame.

Here's my ratio calculation, along with the columns it was derived from. I want to inspect the results to ensure that they are correct.

```
>>> print(autos
...     .select(pl.col(['highway08', 'city08']),
...             mpg_ratio=(pl.col('highway08') / pl.col('city08')))
...
... )
shape: (41_144, 3)
  highway08    city08    mpg_ratio
  u8          u8        f64
  -----
  25          19       1.32
  14          9        1.56
  33          23       1.43
  .
  .
  24          18       1.33
  24          18       1.33
  21          16       1.31
```

Let's add the `mpg_ratio` column to the dataframe with `.with_columns`. This will preserve the existing dataframe columns and add `mpg_ratio` as a new column at the end.

```
>>> print(autos
...     .with_columns(mpg_ratio=pl.col('highway08') / pl.col('city08'))
... )
shape: (41_144, 16)
  year      make      model      .  is_automat
  i16      cat       cat      .  tic
                           num_gears  mpg_ratio
                           u8        f64
                           str
  -----
  1985  Alfa Romeo  Spider   .  false      5       1.32
                           Veloce
                           2000
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

1985	Ferrari	Testaross	.	false	5	1.56
1985	Dodge	Charger	.	false	5	1.43
.
1993	Subaru	Legacy	.	true	4	1.33
		AWD				
1993	Subaru	Legacy	.	false	5	1.33
		AWD				
1993	Subaru	Legacy	.	true	4	1.31
		AWD Turbo				

2.4 Simulating the Index

Polars does not have an index column like Pandas. If you are a Pandas user and relied on a numeric monotonic index, you can create a column to simulate the index. The `.with_row_index` method does this.

>>> print(autos						
... .with_row_index('index')						
...)						
shape: (41_144, 16)						
index year make . createdOn is_automati num_gears						
u32 i16 cat . datetime[μs] c u8						
]						
0 1985 Alfa Romeo . 2013-01-01 false 5						
			00:00:00			
1 1985 Ferrari . 2013-01-01 false 5						
			00:00:00			
2 1985 Dodge . 2013-01-01 false 5						
			00:00:00			
.
41141 1993 Subaru . 2013-01-01 true 4						
			00:00:00			
41142 1993 Subaru . 2013-01-01 false 5						
			00:00:00			
41143 1993 Subaru . 2013-01-01 true 4						
			00:00:00			

Note

I thought the whole purpose of Polars was to eliminate Pandasisms. Now, you are suggesting that I simulate an index? What gives?

This can be useful for retaining order in the data, especially if you are going to be doing a lot of parallel operations. You can use this column

to preserve the order of the data and then remove it when you are done. It's a bit of a hack, but it works.

2.5 Removing Columns

To remove a column from a DataFrame, we can use the `.drop` method. This context takes a list of column names as input and returns a new DataFrame with the specified columns removed. However, rather than focus on the columns that I don't want, I prefer to focus on the columns that I do want. The columns that I will be working with. This also hints to the reader and the query planner about what columns I will use.

In practice, it can be a chore to explicitly list out all of the columns we want to keep. I'm a lazy programmer, so I'll minimize the typing I must do. To do that, I will use the `.drop` method and then print out the columns I want to keep. Then, I'll copy and paste that list into `.select`.

Let's assume we don't want the `createdOn` column in our DataFrame. We can remove it using the `.drop` method. I'll chain on the operation to inspect the columns as well. This will show me the columns I am interested in keeping.

```
>>> print(autos
...   .drop('createdOn')
...   .columns
... )
['year', 'make', 'model', 'displ', 'cylinders', 'trany', 'drive',
 'VClass', 'fuelType', 'barrels08', 'city08', 'highway08',
 'is_automatic', 'num_gears']
```

Now, I have the columns I want to keep. I'll copy them. Often, I will make a variable to hold the list of columns I want to keep. I'll drop that list into `.select`.

```
>>> final_cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
...   'drive', 'VClass', 'fuelType', 'barrels08', 'city08', 'highway08',
...   'is_automatic', 'num_gears']

>>> print(autos
...   .select(final_cols)
... )
```

year	make	model	highway08	is_automati	num_gear
i16	cat	cat	u8	tic	u8
1985	Alfa Romeo	Spider	.	25	false
		Veloce			5

2. Data Manipulation with Polars Using the Fuel Economy Dataset

1985	Ferrari	Testaross	.	14	false	5
1985	Dodge	Charger	.	33	false	5
.
1993	Subaru	Legacy	.	24	true	4
		AWD				
1993	Subaru	Legacy	.	24	false	5
		AWD				
1993	Subaru	Legacy	.	21	true	4
		AWD Turbo				

Polars also has a `.exclude` method on a column expression. This is probably more flexible than the `.drop` context because you can perform further operations on the columns that are not excluded.

```
>>> print(autos
...     .select(pl.all()
...             .exclude(['createdOn', 'barrels08'])
...             .name.suffix('_auto'))
... )
```

year_auto	make_auto	model_au	.	highway08_auto	is_automat	num_gear
i16	cat	to		u8	o	u8
1985	Alfa Romeo	Spider Veloce	.	25	false	5
		2000				
1985	Ferrari	Testarossa	.	14	false	5
1985	Dodge	Charger	.	33	false	5
.
1993	Subaru	Legacy	.	24	true	4
		AWD				
1993	Subaru	Legacy	.	24	false	5
		AWD				
1993	Subaru	Legacy	.	21	true	4
		AWD Turbo				

Future sections will show how to use column selectors to filter the columns as well.

Note

Being positive about what columns you want to preserve (rather than indicating the columns to drop) can aid the Polars query planner when operating in lazy mode.

2.6 Renaming Columns

You might want to rename a column for various reasons. For example, you might want to rename a column to make it easier to type or more descriptive. You might also want to rename a column to make joining another DataFrame easier. Unlike Pandas, Polars will not let you have duplicate column names. If you derive a column from another, it will use the same name as the original column.

We can rename a column using the `.alias` method or keyword arguments. We can combine an expression using `.alias` with both the `.with_columns` and `.select` methods.

The `.with_columns` method takes an expression and returns a new DataFrame with the expression added as a column. If the expression has the same name as an existing column, the existing column is replaced. If the expression has a different name than an existing column, a new column is added. It keeps the original columns in the DataFrame.

Let's rename `city08` to `city_mpg` and `highway08` to `highway_mpg`. Note that this output will have both the original and new columns. If you want to remove the original columns, we can use a `.select` call that doesn't include the original columns. I'll give an example of using both `.alias` and keyword arguments to rename columns below.

```
>>> final_cols = ['year', 'make', 'model', 'city_mpg', 'highway_mpg']
>>> print(autos
...     .with_columns(pl.col('city08').alias('city_mpg'),
...                  highway_mpg=pl.col('highway08'))
...     .select(final_cols)
... )
```

shape: (41_144, 5)

year	make	model	city_mpg	highway_mpg
i16	cat	cat	u8	u8
1985	Alfa Romeo	Spider Veloce	2000	19
1985	Ferrari	Testarossa		25
1985	Dodge	Charger		14
.	.	.	23	33
1993	Subaru	Legacy AWD		.
1993	Subaru	Legacy AWD	18	24
1993	Subaru	Legacy AWD Turbo	18	24
			16	21

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Note

In Python, non-keyword arguments (or positional) must be before keyword arguments. If you use a positional argument after a keyword argument, you will get a `SyntaxError`.

```
>>> print(autos
...     .with_columns(highway_mpg=pl.col('highway08'),
...                     pl.col('city08').alias('city_mpg'))
...     .select(final_cols)
... )
Traceback (most recent call last)
...
SyntaxError: positional argument follows keyword argument
(3173993774.py, line 3)
```

There is also a `dataframe` method, `.rename`, that accepts a dictionary mapping for renaming columns.

```
>>> final_cols = ['year', 'make', 'model', 'city_mpg', 'highway_mpg']
>>> print(autos
...     .rename({'city08':'city_mpg',
...               'highway08':'highway_mpg'})
...     .select(final_cols)
... )
shape: (41_144, 5)
+-----+-----+-----+-----+-----+
| year | make | model | city_mpg | highway_mpg |
| i16  | cat   | cat   | u8       | u8         |
+-----+-----+-----+-----+-----+
| 1985 | Alfa Romeo | Spider Veloce | 2000 | 19      |
| 1985 | Ferrari    | Testarossa    |       | 9       |
| 1985 | Dodge       | Charger       |       | 23      |
| .    | .          | .            |       | .       |
| 1993 | Subaru      | Legacy AWD    |       | 18      |
| 1993 | Subaru      | Legacy AWD    |       | 18      |
| 1993 | Subaru      | Legacy AWD Turbo |       | 16      |
+-----+-----+-----+-----+-----+
```

2.7 Joining DataFrames

Joining dataframes is a typical operation in data analysis. It allows us to combine data from multiple sources into a single dataframe. This section will cover the basics of joining `DataFrames` in Polars. There are a few types of joins:

- Inner Joins
- Outer Joins

- Left Joins
- Right Joins
- Cross Joins
- Anti Joins

Let's start with the *left join*. A left join is a common type that preserves all the rows from the left dataframe (possibly duplicating them) and adds the columns from the right DataFrame. If there are rows in the left dataframe that don't have a match in the right dataframe, the columns from the right dataframe will be filled with null values. If there are rows in the left dataframe that match multiple rows in the right, you will get a row for each match.

I will make two simple dataframes to demonstrate the different types of joins. One dataframe has information about trucks, and another has manufacturer information. Pay attention to the fact that the *trucks* dataframe has two *Ford* trucks and also includes a custom truck that doesn't have a manufacturer. Note that the *manufacturer* dataframe has a *Toyota* entry, but there are no *Toyota* trucks in the *trucks* dataframe.

```
>>> trucks = pl.DataFrame(
...     {'make': ['Ford', 'Tesla', 'Chevy', 'Custom', 'Ford'],
...      'model': ['F150', 'Cybertruck', 'Silverado', 'HotRod', 'F250'],
...      'year': [2018, 2024, 2019, 1967, 2017],
...      'city_mpg': [19, None, 17, 12, 18],
...     })
>>> print(trucks)
shape: (5, 4)
make      model      year  city_mpg
str        str       i64      i64
+-----+-----+-----+-----+
Ford      F150      2018    19
Tesla    Cybertruck  2024    null
Chevy    Silverado   2019    17
Custom   HotRod     1967    12
Ford      F250      2017    18

>>> manufacturer = pl.DataFrame(
...     {'name': ['Ford', 'Tesla', 'Chevy', 'Toyota'],
...      'country': ['USA', 'USA', 'USA', 'Japan'],
...      'founded': [1903, 2003, 1911, 1937],
...      'employees': [199_000, 48_000, 225_000, 370_000],
...      'vehicles': [80, 3, 45, 30],
...     })
>>> print(manufacturer)
shape: (4, 5)
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

name	country	founded	employees	vehicles
str	str	i64	i64	i64
Ford	USA	1903	199000	80
Tesla	USA	2003	48000	3
Chevy	USA	1911	225000	45
Toyota	Japan	1937	370000	30

Left Join

df1

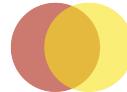
name	pet
Fred	Dog
Suzy	Dog
Suzy	Cat
Bob	Fish

df2

Name	Color
Suzy	Black
Suzy	Blue
Suzy	Red
Fred	Green
Joe	Yellow
Joe	Blue

```
(df1
    .join(df2, left_on='name', right_on='Name', how='left')
)
```

name	pet	Color
Fred	Dog	Green
Suzy	Dog	Black
Suzy	Dog	Blue
Suzy	Dog	Red
Suzy	Cat	Black
Suzy	Cat	Blue
Suzy	Cat	Red
Bob	Fish	None



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

Figure 2.1: Left joining with Polars.

Let's do a left join with this data. By default, Polars will use the `on` parameter to specify columns with the same name in both DataFrames and use those columns to join the DataFrames.

```
>>> print(manufacturer.join(trucks, how='left'))
Traceback (most recent call last)
...
ValueError: must specify `on` OR `left_on` and `right_on`
```

This threw an error because we didn't tell it what column to join on. It's nice that the error message tells us that. Let's try again.

Our DataFrames don't share any column name, so we'll have to specify the columns to join on with the `left_on` and `right_on` parameters. These let us indicate which columns to merge on the DataFrame we are calling the method

on (`left_on`) and the dataframe we are joining (`right_on`). In the `manufacturer` dataframe, the column is named `name`, and in the `trucks` dataframe, the column is named `make`. We'll use those columns to join the dataframes.

```
>>> print(manufacturer.join(trucks, how='left', left_on='name',
...                               right_on='make'))
```

shape: (5, 8)

name	country	founded	.	model	year	city_mpg
str	str	i64	.	str	i64	i64
Ford	USA	1903	.	F150	2018	19
Ford	USA	1903	.	F250	2017	18
Tesla	USA	2003	.	Cybertruck	2024	null
Chevy	USA	1911	.	Silverado	2019	17
Toyota	Japan	1937	.	null	null	null

Notice that every row from the left dataframe is included in the output. If there is a match in the right dataframe, the columns from the right dataframe are included. The `Ford` entry matches each of the `Ford` trucks in the `trucks` dataframe. The `Toyota` row doesn't have a match in the right dataframe, so the `model` and `year` columns are null. Also, the custom truck didn't have an entry in the `manufacturer` dataframe, so it was not included in the output.

2.8 Right Joins

A right join is the opposite of a left join. It keeps all the rows from the right dataframe and adds the columns from the left dataframe. If there are rows in the right dataframe that don't have a match in the left dataframe, the columns from the left dataframe will be filled with null values.

Polars doesn't support `how='right'` in the `.join` method. Instead, we can use `how='left'` and then swap the dataframes. (We will also need to swap the `left_on` and `right_on` parameters.)

```
>>> print(trucks.join(manufacturer, how='left', right_on='name',
...                               left_on='make'))
```

shape: (5, 8)

make	model	year	.	founded	employees	vehicles
str	str	i64	.	i64	i64	i64
Ford	F150	2018	.	1903	199000	80
Tesla	Cybertruck	2024	.	2003	48000	3
Chevy	Silverado	2019	.	1911	225000	45
Custom	HotRod	1967	.	null	null	null
Ford	F250	2017	.	1903	199000	80

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Right Join

df1

name	pet
Fred	Dog
Suzy	Dog
Suzy	Cat
Bob	Fish

df2

Name	Color
Suzy	Black
Suzy	Blue
Suzy	Red
Fred	Green
Joe	Yellow
Joe	Blue

```
(df2
    .join(df1, left_on='Name', right_on='name', how='left')
)
```

Name	Color	pet
Suzy	Black	Dog
Suzy	Black	Cat
Suzy	Blue	Dog
Suzy	Blue	Cat
Suzy	Red	Dog
Suzy	Red	Cat
Fred	Green	Dog
Joe	Yellow	None
Joe	Blue	None



Note there is no right join, we just flip a left

Figure 2.2: Right joins in Polars are done with left joins and flipping the dataframe the you call `.join` on.

How do you tell when to use a left join and when to use a right join? It depends on the data that you want to keep. Place the data that you want to keep on the left.

2.9 Inner Joins

An inner join keeps only the rows that have a match in both DataFrames. It does not keep any rows that don't have a match in both DataFrames. This is the default join type in Polars. You use this when you want to preserve the data that is common to both DataFrames. It will drop rows that are unique to either DataFrame. If multiple rows match, it will include each match.

This join will often simplify your data analysis because you don't have to worry about null values that result from the join (there could still be null values in the original data).

In this case, it will drop both the *Toyota* and the custom truck rows.

```
>>> print(manufacturer.join(trucks, left_on='name', right_on='make'))
```

shape: (4, 8)

name	country	founded	.	model	year	city_mpg	
str	str	i64		str	i64	i64	
BMW	Germany	1916	.	3 Series	1980	22	
BMW	Germany	1916	.	5 Series	1980	22	
BMW	Germany	1916	.	7 Series	1980	22	
BMW	Germany	1916	.	Z4	2002	28	
BMW	Germany	1916	.	Z8	2003	31	
BMW	Germany	1916	.	X5	2003	24	
BMW	Germany	1916	.	X6	2004	24	
BMW	Germany	1916	.	Z4 M Coupe	2006	30	
BMW	Germany	1916	.	Z4 M Roadster	2006	30	
BMW	Germany	1916	.	Z8	2006	30	
BMW	Germany	1916	.	X5 M	2007	33	
BMW	Germany	1916	.	X6 M	2007	33	
BMW	Germany	1916	.	Z4 M Coupe	2009	33	
BMW	Germany	1916	.	Z4 M Roadster	2009	33	
BMW	Germany	1916	.	Z4 G39	2018	33	
BMW	Germany	1916	.	Z4 G29	2019	33	
BMW	Germany	1916	.	Z4 G29	2020	33	
BMW	Germany	1916	.	Z4 G29	2021	33	
BMW	Germany	1916	.	Z4 G29	2022	33	
BMW	Germany	1916	.	Z4 G29	2023	33	
BMW	Germany	1916	.	Z4 G29	2024	33	
BMW	Germany	1916	.	Z4 G29	2025	33	
BMW	Germany	1916	.	Z4 G29	2026	33	
BMW	Germany	1916	.	Z4 G29	2027	33	
BMW	Germany	1916	.	Z4 G29	2028	33	
BMW	Germany	1916	.	Z4 G29	2029	33	
BMW	Germany	1916	.	Z4 G29	2030	33	
BMW	Germany	1916	.	Z4 G29	2031	33	
BMW	Germany	1916	.	Z4 G29	2032	33	
BMW	Germany	1916	.	Z4 G29	2033	33	
BMW	Germany	1916	.	Z4 G29	2034	33	
BMW	Germany	1916	.	Z4 G29	2035	33	
BMW	Germany	1916	.	Z4 G29	2036	33	
BMW	Germany	1916	.	Z4 G29	2037	33	
BMW	Germany	1916	.	Z4 G29	2038	33	
BMW	Germany	1916	.	Z4 G29	2039	33	
BMW	Germany	1916	.	Z4 G29	2040	33	
BMW	Germany	1916	.	Z4 G29	2041	33	
BMW	Germany	1916	.	Z4 G29	2042	33	
BMW	Germany	1916	.	Z4 G29	2043	33	
BMW	Germany	1916	.	Z4 G29	2044	33	
BMW	Germany	1916	.	Z4 G29	2045	33	
BMW	Germany	1916	.	Z4 G29	2046	33	
BMW	Germany	1916	.	Z4 G29	2047	33	
BMW	Germany	1916	.	Z4 G29	2048	33	
BMW	Germany	1916	.	Z4 G29	2049	33	
BMW	Germany	1916	.	Z4 G29	2050	33	
BMW	Germany	1916	.	Z4 G29	2051	33	
BMW	Germany	1916	.	Z4 G29	2052	33	
BMW	Germany	1916	.	Z4 G29	2053	33	
BMW	Germany	1916	.	Z4 G29	2054	33	
BMW	Germany	1916	.	Z4 G29	2055	33	
BMW	Germany	1916	.	Z4 G29	2056	33	
BMW	Germany	1916	.	Z4 G29	2057	33	
BMW	Germany	1916	.	Z4 G29	2058	33	
BMW	Germany	1916	.	Z4 G29	2059	33	
BMW	Germany	1916	.	Z4 G29	2060	33	
BMW	Germany	1916	.	Z4 G29	2061	33	
BMW	Germany	1916	.	Z4 G29	2062	33	
BMW	Germany	1916	.	Z4 G29	2063	33	
BMW	Germany	1916	.	Z4 G29	2064	33	
BMW	Germany	1916	.	Z4 G29	2065	33	
BMW	Germany	1916	.	Z4 G29	2066	33	
BMW	Germany	1916	.	Z4 G29	2067	33	
BMW	Germany	1916	.	Z4 G29	2068	33	
BMW	Germany	1916	.	Z4 G29	2069	33	
BMW	Germany	1916	.	Z4 G29	2070	33	
BMW	Germany	1916	.	Z4 G29	2071	33	
BMW	Germany	1916	.	Z4 G29	2072	33	
BMW	Germany	1916	.	Z4 G29	2073	33	
BMW	Germany	1916	.	Z4 G29	2074	33	
BMW	Germany	1916	.	Z4 G29	2075	33	
BMW	Germany	1916	.	Z4 G29	2076	33	
BMW	Germany	1916	.	Z4 G29	2077	33	
BMW	Germany	1916	.	Z4 G29	2078	33	
BMW	Germany	1916	.	Z4 G29	2079	33	
BMW	Germany	1916	.	Z4 G29	2080	33	
BMW	Germany	1916	.	Z4 G29	2081	33	
BMW	Germany	1916	.	Z4 G29	2082	33	
BMW	Germany	1916	.	Z4 G29	2083	33	
BMW	Germany	1916	.	Z4 G29	2084	33	
BMW	Germany	1916	.	Z4 G29	2085	33	
BMW	Germany	1916	.	Z4 G29	2086	33	
BMW	Germany	1916	.	Z4 G29	2087	33	
BMW	Germany	1916	.	Z4 G29	2088	33	
BMW	Germany	1916	.	Z4 G29	2089	33	
BMW	Germany	1916	.	Z4 G29	2090	33	
BMW	Germany	1916	.	Z4 G29	2091	33	
BMW	Germany	1916	.	Z4 G29	2092	33	
BMW	Germany	1916	.	Z4 G29	2093	33	
BMW	Germany	1916	.	Z4 G29	2094	33	
BMW	Germany	1916	.	Z4 G29	2095	33	
BMW	Germany	1916	.	Z4 G29	2096	33	
BMW	Germany	1916	.	Z4 G29	2097	33	
BMW	Germany	1916	.	Z4 G29	2098	33	
BMW	Germany	1916	.	Z4 G29	2099	33	
BMW	Germany	1916	.	Z4 G29	2100	33	

Inner Join

df1	
name	pet
Fred	Dog
Suzy	Dog
Suzy	Cat
Bob	Fish

df2	
Name	Color
Suzy	Black
Suzy	Blue
Suzy	Red
Fred	Green
Joe	Yellow
Joe	Blue

```
(df1
    .join(df2, left_on='name', right_on='Name', how='inner')
)
```

Name	Color	pet
Suzy	Black	Dog
Suzy	Black	Cat
Suzy	Blue	Dog
Suzy	Blue	Cat
Suzy	Red	Dog
Suzy	Red	Cat
Fred	Green	Dog



Note there is no right join, we just flip a left

Figure 2.3: Inner joins in Polars.

Ford	USA	1903	.	F150	2018	19
Tesla	USA	2003	.	Cybertruck	2024	null
Chevy	USA	1911	.	Silverado	2019	17
Ford	USA	1903	.	F250	2017	18

2.10 Outer Joins

An outer join keeps all of the rows from both DataFrames. If there is a match in the other dataframe, the columns from the other dataframe are included. If there is no match in the other dataframe, the columns from the other DataFrame are filled with null values. Use an outer join to keep all of the data from both dataframes. You will likely have to deal with null values that result from the join.

The result has both the *Toyota* and the custom truck rows. The *Toyota* row has null values for the *model* and *year* columns. The custom truck row has null values for the *country* column.

```
>>> print(manufacturer.join(trucks, how='outer', left_on='name',
...                                     right_on='make'))
```

shape: (6, 9)

name	country	founded	.	model	year	city_mpg
str	str	i64		str	i64	i64
Ford	USA	1903	.	F150	2018	19
Tesla	USA	2003	.	Cybertruck	2024	null
Chevy	USA	1911	.	Silverado	2019	17
Toyota	Japan	1935	.	Sienna	2017	18

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Outer Join

df1

name	pet
Fred	Dog
Suzy	Dog
Suzy	Cat
Bob	Fish

df2

Name	Color
Suzy	Black
Suzy	Blue
Suzy	Red
Fred	Green
Joe	Yellow
Joe	Blue

```
(df1
    .join(df2, left_on='name', right_on='Name', how='outer')
)
```

Name	Color	name	pet
Suzy	Black	Suzy	Dog
Suzy	Black	Suzy	Cat
Suzy	Blue	Suzy	Dog
Suzy	Blue	Suzy	Cat
Suzy	Red	Suzy	Dog
Suzy	Red	Suzy	Cat
Fred	Green	Fred	Dog
Joe	Yellow	None	None
Joe	Blue	None	None
None	None	Bob	Fish



Note that Bob and Joe have missing (null) values!

Figure 2.4: Outer joins in Polars.

Ford	USA	1903	.	F150	2018	19
Tesla	USA	2003	.	Cybertruck	2024	null
Chevy	USA	1911	.	Silverado	2019	17
null	null	null	.	HotRod	1967	12
Ford	USA	1903	.	F250	2017	18
Toyota	Japan	1937	.	null	null	null

2.11 Semi Joins

A *semi-join* keeps only the rows from the left dataframe that have a match in the right dataframe. It does not add any columns from the right dataframe. This is useful when you want to filter the rows in the left dataframe based on the rows in the right dataframe. It differs from a left join because it doesn't add any columns from the right dataframe. It is different from an inner join because it doesn't keep any rows that don't have a match in the right dataframe.

If we want to keep all of the rows from *manufacturer* that have a match in *trucks*, we can use a semi-join. This will keep the *Ford*, *Tesla*, and *Chevy* rows from *manufacturer*. It drops the *Toyota* row.

```
>>> print(manufacturer.join(trucks, how='semi', left_on='name',
...                               right_on='make'))
```

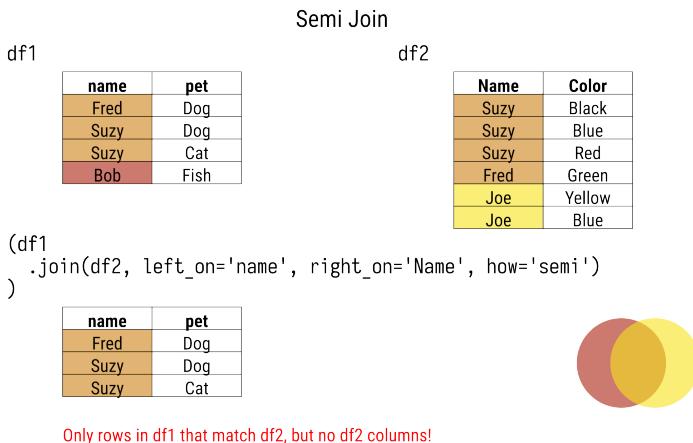


Figure 2.5: Illustration of a semi-join. This keeps rows from the left dataframe that have a match in the right dataframe.

shape: (3, 5)				
name	country	founded	employees	vehicles
str	str	i64	i64	i64
Ford	USA	1903	199000	80
Tesla	USA	2003	48000	3
Chevy	USA	1911	225000	45

2.12 Cross Joins

A *cross-join* differs from the joins we have seen. It doesn't match up the rows based on column values. A cross-join is a join that combines every row from the left DataFrame with every row from the right DataFrame. It is also known as a Cartesian product. It is rarely used in practice because it can result in a very large DataFrame. However, it can be helpful for testing purposes or generating data.

I'm not going to use a cross-join with the truck data because it would not make sense. Instead, I'll give a simple example.

Let's assume you work in a store with colored pants. If you have three sizes of pants and they each come in two colors, you can use a cross join to generate a DataFrame with all possible combinations of sizes and colors.

```
>>> sizes = pl.DataFrame({'size': ['small', 'medium', 'large'],})
>>> colors = pl.DataFrame({'color': ['red', 'green'],})
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Cross Join

df1

name	pet
Suzy	Dog
Suzy	Cat
Bob	Fish

df2

Name	Color
Suzy	Black
Joe	Blue

```
(df1
    .join(df2, how='cross')
)
```

name	pet	Name	Color
Suzy	Dog	Suzy	Black
Suzy	Dog	Joe	Blue
Suzy	Cat	Suzy	Black
Suzy	Cat	Joe	Blue
Bob	Fish	Suzy	Black
Bob	Fish	Joe	Blue

Every row from the left is matched with every right!

Figure 2.6: A cross join doesn't match up the rows based on column values. It just make a combination of every row on the left and right.

```
>>> print(sizes.join(colors, how='cross'))
```

```
shape: (6, 2)
size      color
str       str
-----
small     red
small     green
medium   red
medium   green
large    red
large    green
```

2.13 Anti Joins

An anti-join keeps all of the rows from the left dataframe that don't have a match in the right dataframe. It does not keep any rows that have a match in the right dataframe. This is useful when you want to find the rows in one dataframe but not another dataframe. This is very useful for debugging.

An anti-join can be used to determine which manufacturers are not in the trucks DataFrame.

```
>>> print(manufacturer.join(trucks, how='anti', left_on='name',
...                               right_on='make'))
```

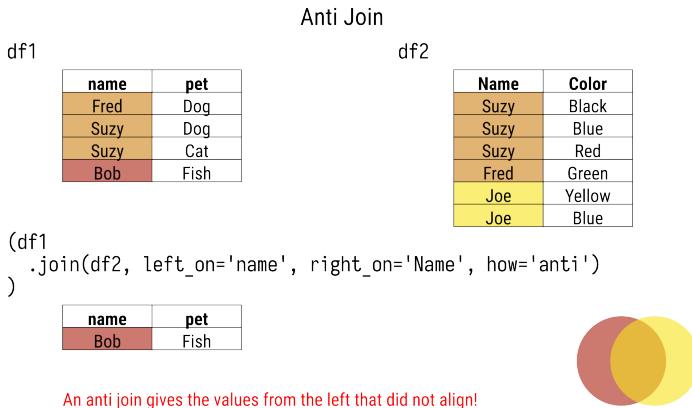


Figure 2.7: An anti-join returns rows from the right that did not align with the left.

shape: (1, 5)				
name	country	founded	employees	vehicles
str	str	i64	i64	i64
Toyota	Japan	1937	370000	30

Conversely, if we want to know which trucks are not in the *manufacturers* dataframe, we can put the *manufacturers* dataframe on the right side of the anti-join.

```
>>> print(trucks.join(manufacturer, how='anti', right_on='name',
...                     left_on='make'))
shape: (1, 4)
  make      model     year   city_mpg
  str       str      i64      i64
  Custom   HotRod  1967      12
```

2.14 Join Validation

When you join two dataframes, you want to understand if the join is a one-to-one join, a one-to-many join, or a many-to-many join. This will help you understand the structure of the data and how to use it in your analysis. It can also prevent errors. For example, if you run an online store and allow customers to have multiple shipping addresses, you want to ensure that a shipment only goes to one address. If you had an orders table and a shipping table, you would want to ensure that the join is a one-to-one join.

2. Data Manipulation with Polars Using the Fuel Economy Dataset

In our truck data, there are two entries for *Ford*. If we join based on that column, we need to realize that it is either a one-to-many join or a many-to-many join. Let's validate if the inner join is a one-to-one join.

```
>>> print(manufacturer.join(trucks, left_on='name', right_on='make',
...                               validate='1:1'))
Traceback (most recent call last)
...
ComputeError: the join keys did not fulfil 1:1 validation
```

This failed with a `ComputeError`. The error message tells us that the join was not a one-to-one join. Let's try again with a many-to-one validation.

```
>>> print(manufacturer.join(trucks, left_on='name', right_on='make',
...                               validate='m:1'))
Traceback (most recent call last)
...
ComputeError: the join keys did not fulfil m:1 validation
```

This also failed with a `ComputeError`. It would be nice if it indicated which rows were duplicated.

The join should validate if we use '`1:m`' (a one-to-many join).

```
>>> print(manufacturer.join(trucks, left_on='name', right_on='make',
...                               validate='1:m'))
shape: (4, 8)
+-----+
| name   country   founded   .   model      year   city_mpg |
| str    str        i64       .   str        i64    i64          |
+-----+
| Ford    USA        1903     .   F150      2018    19          |
| Tesla   USA        2003     .   Cybertruck 2024    null        |
| Chevy   USA        1911     .   Silverado  2019    17          |
| Ford    USA        1903     .   F250      2017    18          |
+-----+
```

A validation fails when duplicated values are on the `1` side. To find the offending rows in a failed '`m:1`' validation, we want to look for duplicate values in the right dataframe.

```
>>> print(trucks
...     .filter(pl.col('make').is_duplicated())
... )
shape: (2, 4)
+-----+
| make   model   year   city_mpg |
| str    str     i64    i64          |
+-----+
| Ford   F150   2018    19          |
| Ford   F250   2017    18          |
+-----+
```

2.15 Speeding up Joins with Sorting

Remember, Polars has a query engine, so it can be smart about joins. Polars can use a faster algorithm to join the data if your data is sorted. Let's explore this a little bit.

I will load another dataset¹, the Open Energy Information (OpenEI) dataset. This dataset has information about cars, trucks, and other vehicle types, showing fuel efficiency and trends over time. This dataset is a little larger than the previous one but has fewer columns. I'll aggregate the city mileage from the previous dataset and join it with the OpenEI dataset. This will allow us to see how sorting can speed up the join.

Loading this data was a little bit of a pain. It is stored as an XLSB file, a binary format for Excel files. I tried loading it with the default Polars but didn't successfully get the data off the *City* sheet. With a little help from the Polars discord, I installed the `fastexcel`² library and ingested it with the `engine='calamine'` parameter to `pl.read_excel`.

The headers are on the second row of the *City* sheet. So we will add options for that. Note that the options are not in the Polars library but on the `fastexcel` page.

```
url = 'https://data.openei.org/files/907/' \
    '2016cityandcountyhighdutyvehicleinventory.xlsb'
inv_raw = pl.read_excel(url, engine='calamine',
    read_options=dict(header_row=1), sheet_name='City')
```

Here's the data: It has city-level information about fuel efficiency for different vehicle types. I will aggregate the city mileage for each vehicle type and then join it with the autos dataframe.

```
>>> print(inv_raw
...     .select(['state_abr', 'gisjoin', 'city_id', 'city_name',
...             'fuel_type_org', 'fuel_type', 'class', '2000', '2001'])
... )
shape: (307_828, 9)
  state_abr      gisjoin      city_id      class      2000      2001
  br            str          i64          str          str          str
  str
  AL      G0100012      100124      .      Car      0.00013
        4                      5208220
                                6598161
        3
  AL      G0100012      100124      .      Car      0.000270
        4                      41644131
```

¹<https://data.openei.org/submissions/907>

²<https://fastexcel.toucantoco.dev/fastexcel.html>

2. Data Manipulation with Polars Using the Fuel Economy Dataset

AL	G0100012	100124	.	Truck	963225
	4				0.001622
					49864791
					2915089
					77934

WY	G5608666	5686665	.	Unknown	.
	5				.
WY	G5608666	5686665	.	Car	.
	5				.
WY	G5608666	5686665	.	Unknown	.
	5				.

The year columns contain the percentage of cars purchased in the county with that gas type. Sadly, when loading the data, the percent was not converted into a number. We will fix that by replacing '' with 0 (note that you need to use .replace, which matches the whole string, and not str.replace, which matches substrings). Then, I will *melt* the data.

Melting allows us to move the column names into the values of a column. In this case, we want to have a column with just the year in it. The id_vars specifies column names we want to preserve in the melted data. It will put the values that were below the year columns into a new column that we are naming *year* (variable_name='year'). If you aren't familiar with melting, we will jump into it in the next chapter.

```
years = [str(i) for i in range(2000, 2019)]  
  
inv_yr = (inv_raw  
    .with_columns(  
        [pl.col(year).replace('', 0).cast(pl.Float32)  
            for year in years])  
    .melt(variable_name='year', value_vars=years,  
          id_vars=['state_abbr', 'gisjoin', 'city_id', 'city_name',  
                   'fuel_type_org', 'fuel_type', 'class',]  
    )  
    .select('state_abbr', 'city_name', 'fuel_type_org', 'fuel_type', 'class',  
           year=pl.col('year').cast(pl.Int16),  
           percent=pl.col('value')*100)  
)
```

The resulting data looks like this:

```
>>> print(inv_yr)  
shape: (5_848_732, 7)  
| state_abbr  city_name   fuel_type   .  class   year   percent |  
| r           str         _org        str     i16     f32    |  
| str          str         str        str     int16    float |
```

AL	Abbeville	BI	.	Car	2000	0.00
AL	Abbeville	DIES	.	Car	2000	0.03
AL	Abbeville	DIES	.	Truck	2000	0.16
.
WY	Yoder	GAS	.	Unknown	2018	0.00
WY	Yoder	HYBRID	.	Car	2018	0.00
		ELECTRIC				
		VEHICLE				
WY	Yoder	UNK	.	Unknown	2018	0.00

Let's join this with the autos data. I want the mean city mileage for each year, vehicle type, and fuel type. I'll convert `VClass` to a string, create a new column called `simple_class` that generalizes `VClass`, cast `fuel_type` to a string, then join by `year`, `simple_class`, and `fuel_type`. I'll aggregate the city mileage with the mean function. Again, I will cover melting and aggregating in the next chapter. (Sometimes, there are circular dependencies in learning.)

```
# make simple cat
gas_mapping = {
    'Diesel': 'Diesel vehicle',
    'Regular': 'Gasoline vehicle',
    'Premium': 'Gasoline vehicle',
    'Midgrade': 'Gasoline vehicle',
    'Gasoline or E85': 'Flex fuel vehicle',
    'Premium or E85': 'Flex fuel vehicle',
    'Premium Gas or Electricity': 'Plug-in hybrid electric vehicle',
    'Regular Gas or Electricity': 'Plug-in hybrid electric vehicle',
    'Premium and Electricity': 'Hybrid electric vehicle',
    'Regular Gas and Electricity': 'Hybrid electric vehicle',
    'Electricity': 'Electric vehicle',
    'Gasoline or natural gas': 'Other/Unknown',
    'Gasoline or propane': 'Other/Unknown',
    'CNG': 'Other/Unknown',
}

agg_yr = (autos
    .with_columns(VClass=pl.col('VClass').cast(pl.String))
    .with_columns(
        simple_class=pl.when(pl.col('VClass')
            .str.to_lowercase().str.contains('car'))
            .then(pl.lit('Car'))
        .when(pl.col('VClass')
            .str.to_lowercase().str.contains('truck'))
            .then(pl.lit('Truck'))
        .otherwise(pl.lit('Other'))),
    )
)
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
fuel_type=pl.col('fuelType').cast(pl.String)
    .replace(gas_mapping, default='Missing'))
    .group_by(['year', 'simple_class', 'fuel_type'])
    .agg(pl.col('city08').mean().alias('mean_mpg'))
)
```

Here's what the aggregated mileage data looks like:

```
>>> print(agg_yr)
```

shape: (361, 4)

year	simple_class	fuel_type	mean_mpg
i16	str	str	f64
1995	Other	Gasoline vehicle	15.39
1995	Truck	Diesel vehicle	14.67
1996	Car	Other/Unknown	15.00
.	.	.	.
2017	Truck	Gasoline vehicle	16.64
2020	Car	Gasoline vehicle	22.84
1991	Truck	Gasoline vehicle	13.98

I'm going to join these two datasets by year, vehicle class, and fuel type.

```
>>> print (agg_yr
...     .join(inv_yr, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
```

shape: (3_234_418, 8)

year	simple_cl	fuel_type	.	city_nam	fuel_typ	percent
i16	ass	str	.	e	e_org	f32
	str			str	str	
2000	Car	Other/Unk	.	Abbevill	BI	0.00
		nown		e		
2000	Car	Diesel	.	Abbevill	DIES	0.03
		vehicle		e		
2000	Car	Flex fuel	.	Abbevill	FFV	0.07
		vehicle		e		
.
2018	Car	Gasoline	.	Yoder	GAS	0.00
		vehicle				
2018	Truck	Gasoline	.	Yoder	GAS	0.00
		vehicle				
2018	Car	Hybrid	.	Yoder	HYBRID	0.00
		electric			ELECTRIC	
		vehicle			VEHICLE	

Let's do some quick benchmarking to see if sorting or using categoricals has an impact on joining. Note that `inv_yr` has over 5 million rows.

```
agg_yr_shuf = agg_yr.sample(len(agg_yr), with_replacement=False, seed=42)
inv_yr_shuf = inv_yr.sample(len(inv_yr), with_replacement=False, seed=42)

>>> %%timeit
>>> (agg_yr_shuf
...     .join(inv_yr_shuf, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
89.7 ms ± 6.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

That's our baseline.

Note

Anytime you include a benchmark, someone appears out of nowhere to state why it is incorrect. Here's my general take on benchmarking code like Polars.

Make sure you benchmark code with real-world data sizes. For example, if you will be processing 10 million rows of data, don't benchmark code against ten rows of data. Code that runs fast at scale often runs slower with few rows of data.

Let's sort each dataframe by `year` and time the join again:

```
# sort by year
agg_yr_sort = agg_yr.sort('year')
inv_yr_sort = inv_yr.sort('year')

>>> %%timeit
>>> (agg_yr_sort
...     .join(inv_yr_sort, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
81.9 ms ± 2.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

That looks like it didn't buy us anything. Going back to my benchmark rant, that doesn't necessarily mean that sorting won't help a join. In this example, we are joining based on three columns, so we probably want to sort all three. But if you are only joining on a single column, sorting will likely make the code run more efficiently.

Let's try again sorting all of the joining columns.

```
# sort by year and class
agg_yr_sort2 = agg_yr.sort('year', 'simple_class', 'fuel_type')
inv_yr_sort2 = inv_yr.sort('year', 'class', 'fuel_type')
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
>>> %%timeit
>>> (agg_yr_sort2
...     .join(inv_yr_sort2, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
73.3 ms ± 1.92 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

That gave us a little boost.

Let's see if converting the strings to categorical values helps. In this case, we need to use the `pl.StringCache` context manager and convert all of the columns to categories inside of the same context. When you are working with multiple dataframes that have categoricals with the same values, you can use the `StringCache` to ensure that the categories are the same across the dataframes. This can help with joins and aggregations.

```
with pl.StringCache():
    agg_yr_cat = agg_yr_sort2.with_columns(
        pl.col('simple_class', 'fuel_type').cast(pl.Categorical))
    inv_yr_cat = inv_yr_sort2.with_columns(
        pl.col('class', 'fuel_type').cast(pl.Categorical))

>>> %%timeit
>>> (agg_yr_cat
...     .join(inv_yr_cat, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
56.4 ms ± 1.11 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

For this example, we get a big boost using categories.

Let's compare this to a Pandas sort.

```
>>> agg_pd = agg_yr_sort2.to_pandas().sort_values(['year', 'simple_class',
...                                                 'fuel_type'])
>>> inv_pd = inv_yr_sort2.to_pandas().sort_values(['year', 'class',
...                                                 'fuel_type'])

>>> %%timeit
>>> (agg_pd
...     .merge(inv_pd, left_on=['year', 'simple_class', 'fuel_type'],
...           right_on=['year', 'class', 'fuel_type'])
... )
1.02 s ± 29.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Depending on your data and needs, it might make sense to sort your data or convert your data to categoricals to speed up your joins.

2.16 Visualizing the Join

Let's complete this example by visualizing the results of the join. I'll make a line for each *simple class* vehicle type over the years for Salt Lake City.

To do this, I need a dataframe aggregated by year. I'm going to use the `.pivot` method, which we will cover shortly. Once I have the year in a column and other columns for each line, I can make a plot.

Here's the data prepped for the plot.

```
>>> print(agg_yr
...   .join(inv_yr, left_on=['year', 'simple_class', 'fuel_type'],
...         right_on=['year', 'class', 'fuel_type'])
...   .filter(city_name='Salt Lake City')
...   .pivot(index='year', columns='simple_class', values='mean_mpg',
...         aggregate_function='mean')
... )
shape: (19, 3)
   year    Car      Truck
   i16    f64     f64
   2000  20.47   12.56
   2001  19.97   12.69
   2002  20.83   12.62
   .
   2016  34.48   17.66
   2017  43.26   17.58
   2018  41.72   17.95
```

I'll use the `.plot.line` method to visualize the results.

```
>>> (agg_yr
...   .join(inv_yr, left_on=['year', 'simple_class', 'fuel_type'],
...         right_on=['year', 'class', 'fuel_type'])
...   .filter(city_name='Salt Lake City')
...   .pivot(index='year', columns='simple_class', values='mean_mpg',
...         aggregate_function='mean')
...   .plot.line(x='year', y=['Car', 'Truck'],
...             title='Salt Lake City Vehicle Mileage')
... )
```

```
:NdOverlay [Variable]
:Curve [year] (value)
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

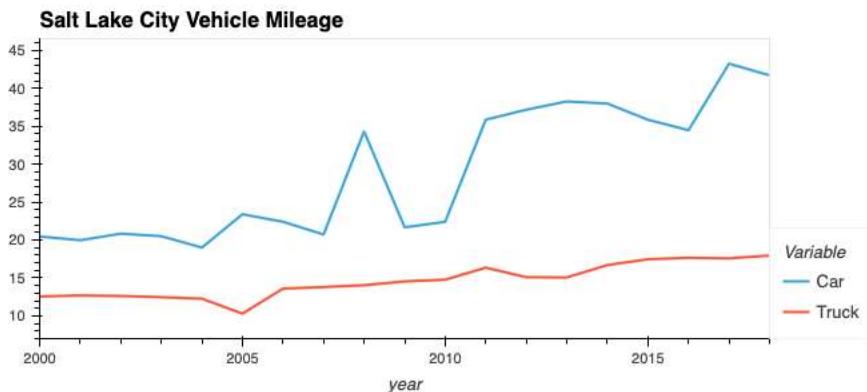


Figure 2.8: Visualizing the car and truck mileage over the years for Salt Lake City.

2.17 Adding Rows

We already learned how to add columns to a dataframe using `.with_columns`. The `.with_columns` method takes a column name and an expression and returns a new dataframe with the column added. If the column name already exists, it will be replaced.

To add a row to a dataframe, we can use the `.vstack` method. The `.vstack` method takes a dataframe and returns a new dataframe with the rows from the original dataframe and the rows from the dataframe that was passed in. It does not modify the original dataframe.

I will remove the last ten rows of the autos dataframe and add the first ten rows to the end of the dataframe.

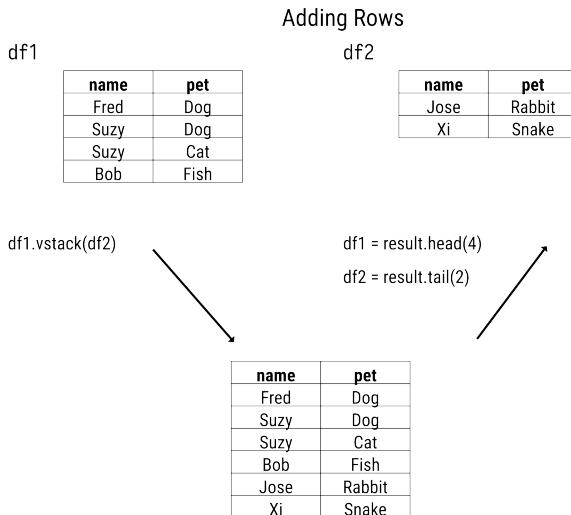


Figure 2.9: Adding rows.

1993	Toyota	Corolla	.	2013-01-01 00:00:00	true	3
1993	Toyota	Corolla	.	2013-01-01 00:00:00	false	5
1993	Toyota	Corolla	.	2013-01-01 00:00:00	true	4

You could use `.head` and `.tail` to undo a simple `.vstack` operation if you know how many rows you added.

2.18 Reshaping and Pivoting Data

One of the most powerful features of Polars is the ability to reshape and pivot data. This allows you to transform data from a wide format to a long format and vice versa. This is a typical operation in data analysis. It is often used to prepare data for plotting or machine learning.

Let's filter out the top 5 car manufacturers. Let's pull the values from the `make` column into the columns and leave the `year` column as the first column (Polars will call this the `index` even though Polars doesn't have indices like Pandas). We will take the median of the `city08` column for each `year` and `make` combination. This will give us the median city fuel efficiency for each `year` and `make` of car.

First, we need to determine the top five car manufacturers.

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
>>> top_n = (autos
...     .group_by('make')
...     .agg(pl.col('city08').count())
...     .sort('city08')
...     .tail(5)
... )
```

```
>>> print(top_n)
```

shape: (5, 2)

make	city08
cat	u32
Toyota	2071
GMC	2494
Dodge	2583
Ford	3371
Chevrolet	4003

Pivoting & Aggregating Data

tall

year	model	trany	city08
1984	Tempo	Manual 5-spd	31
1984	Tempo	Automatic 3-spd	21
1985	Tempo	Automatic 3-spd	21
1985	Tempo	Manual 5-spd	21
1986	Tempo	Manual 5-spd	31
1986	Tempo	Manual 5-spd	22
1987	Tempo	Automatic 3-spd	19
1987	Tempo	Manual 5-spd	22

```
(tall
    .pivot(index='year', columns='model', values='city08',
           aggregate_function='median')
)
```

year	Tempo
1984	26.00
1985	21.00
1986	26.50
1987	20.50

Figure 2.10: Pivoting data.

Let's use the top five car manufacturers to filter the *autos* DataFrame and then pivot the data. To pivot the data, we specify what goes into the leftmost column (Polars calls this the *index*). We must also indicate the row values we want to pull into the column(s) (using *columns*). This will define the matrix of the pivot. The values that go in the cells are specified with *values*. If duplicate

values exist, we must pass an `aggregate_function` parameter to combine them into a single value.

We filter the data to the top five car manufacturers in our example. We put the year down the leftmost column and the make across the top. We want the city fuel efficiency in the cells. Because there are multiple vehicles for each year and make, we need to aggregate them. We will use the median to reduce the multiple values into a single value. I'm using the median because it is less sensitive to outliers than the mean, and I know that some manufacturers have electric vehicles that appear as outliers in the data. This table will better represent the typical fuel efficiency for each *year* and *make* of car.

```
>>> print(autos
...     .filter(pl.col('make').is_in(top_5['make']))
...     .pivot(index='year', columns='make', values='city08',
...            aggregate_function='median')
... )
shape: (37, 6)
+-----+-----+-----+-----+-----+-----+
| year | Dodge | Toyota | Chevrolet | Ford | GMC |
+-----+-----+-----+-----+-----+-----+
| i16  | f64   | f64    | f64       | f64  | f64  |
+-----+-----+-----+-----+-----+-----+
| 1985 | 17.00 | 22.00 | 15.00    | 15.00 | 15.00 |
| 1993 | 14.00 | 16.00  | 15.00    | 15.00 | 14.00 |
| 1994 | 14.50 | 17.00  | 15.00    | 15.00 | 14.00 |
| .    | .      | .      | .        | .    | .    |
| 2018 | 16.00 | 22.00 | 19.50    | 19.00 | 16.00 |
| 2019 | 16.00 | 22.00 | 18.00    | 19.00 | 15.00 |
| 2020 | null   | 29.00 | 29.00    | 21.00 | null  |
+-----+-----+-----+-----+-----+-----+
```

In the above example, we used '`median`' as the aggregate function. We can supply an expression that will aggregate the values or one of a set of predefined strings. The predefined strings are:

- '`first`' - returns the first value from the index group
- '`last`' - returns the last value from the index group
- '`sum`' - returns the sum of the values from the index group
- '`max`' - returns the maximum value from the index group
- '`min`' - returns the minimum value from the index group
- '`mean`' - returns the mean of the values from the index group
- '`median`' - returns the median of the values from the index group
- '`count`' - returns the count of the values from the index group

Looking closely, you will see that the *year* column is not sorted. This is because Polars runs the group by operation in parallel and then combines the results. You would get different row orders if you ran the same code multiple times. We can address this by sorting the *year* column.

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
>>> pivoted = (autos
...     .filter(pl.col('make').is_in(top_n['make']))
...     .pivot(index='year', columns='make', values='city08',
...           aggregate_function='median')
...     .sort('year')
... )
```

```
>>> print(pivoted)
```

```
shape: (37, 6)
```

year	Dodge	Toyota	Chevrolet	Ford	GMC
i16	f64	f64	f64	f64	f64
1984	18.00	22.00	16.00	14.50	15.00
1985	17.00	22.00	15.00	15.00	15.00
1986	14.50	19.00	15.00	16.00	15.00
.
2018	16.00	22.00	19.50	19.00	16.00
2019	16.00	22.00	18.00	19.00	15.00
2020	null	29.00	29.00	21.00	null

Alternatively, we can tell Polars to preserve the order of the rows by first sorting the *year* column and then indicating that the column is sorted with `.set_sorted`.

```
>>> pivoted = (autos
...     .filter(pl.col('make').is_in(top_n['make']))
...     .sort('year')
...     .set_sorted('year')
...     .pivot(index='year', columns='make', values='city08',
...           aggregate_function='median')
... )
```

```
>>> print(pivoted)
```

```
shape: (37, 6)
```

year	Chevrolet	Ford	Toyota	Dodge	GMC
i16	f64	f64	f64	f64	f64
1984	16.00	14.50	22.00	18.00	15.00
1985	15.00	15.00	22.00	17.00	15.00
1986	15.00	16.00	19.00	14.50	15.00
.
2018	19.50	19.00	22.00	16.00	16.00
2019	18.00	19.00	22.00	16.00	15.00
2020	29.00	21.00	29.00	null	null

There is also an option in the `.pivot` call to indicate that results should be in the order of the index column by using `maintain_order=True`.

```

>>> pivoted = (autos
...     .filter(pl.col('make').is_in(top_n['make']))
...     .sort('year')
...     .pivot(index='year', columns='make', values='city08',
...             aggregate_function='median', maintain_order=True)
... )

>>> print(pivoted)
shape: (37, 6)
   year Chevrolet Ford Toyota Dodge GMC
   i16      f64    f64    f64    f64    f64
1984  16.00  14.50  22.00  18.00  15.00
1985  15.00  15.00  22.00  17.00  15.00
1986  15.00  16.00  19.00  14.50  15.00
...
2018  19.50  19.00  22.00  16.00  16.00
2019  18.00  19.00  22.00  16.00  15.00
2020  29.00  21.00  29.00  null   null

```

2.19 Melting Data

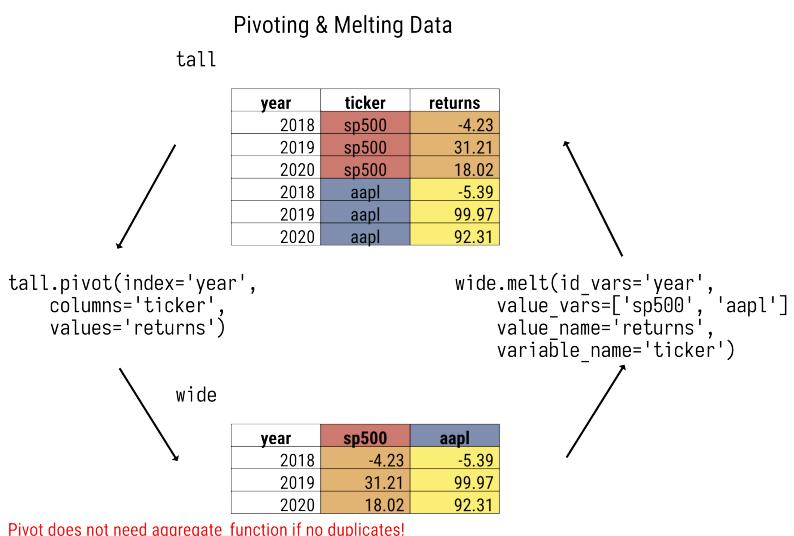


Figure 2.11: Pivoting and melting data.

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Melting data is the opposite of pivoting data. It takes data from a wide format to a long format.

Let's melt the data from the previous example. We want to put the *make* headers back into a *make* column and the *city08* values into a *city08* column. Here's the pivoted data:

```
>>> print(pivoted)
```

```
shape: (37, 6)
```

year	Chevrolet	Ford	Toyota	Dodge	GMC
i16	f64	f64	f64	f64	f64
1984	16.00	14.50	22.00	18.00	15.00
1985	15.00	15.00	22.00	17.00	15.00
1986	15.00	16.00	19.00	14.50	15.00
.
2018	19.50	19.00	22.00	16.00	16.00
2019	18.00	19.00	22.00	16.00	15.00
2020	29.00	21.00	29.00	null	null

We must pass at least two parameters to the `.melt` method. The `id_vars` parameter is a list of columns we want to keep. The `value_vars` parameter lists columns we want to melt. In this case, we want to keep the *year* column and melt the *Ford*, *Chevrolet*, *Dodge*, *Toyota*, and *Honda* columns.

```
>>> print(pivoted)
... .melt(id_vars='year', value_vars=['Chevrolet', 'Ford', 'GMC',
...                                         'Toyota', 'Dodge'])
... )
```

```
shape: (185, 3)
```

year	variable	value
i16	str	f64
1984	Chevrolet	16.00
1985	Chevrolet	15.00
1986	Chevrolet	15.00
.	.	.
2018	Dodge	16.00
2019	Dodge	16.00
2020	Dodge	null

You'll notice that the resulting dataframe has a *variable* column and a *value* column. Those names are not particularly descriptive. We can change them with the `variable_name` and `value_name` parameters. The `variable_name` parameter is the name of the column that contains the melted column names. In this case, it is the *make* column. The `value_name` parameter is the name of the column that contains the melted values. In this case, it is the *city08* column.

```
>>> print(pivoted
... .melt(id_vars='year', value_vars=['Chevrolet', 'Ford', 'GMC',
...                                         'Toyota', 'Dodge'],
...         value_name='median_city_mpg', variable_name='make')
... )
shape: (185, 3)
   year      make    median_city_mpg
   i16      str        f64
1984 Chevrolet  16.00
1985 Chevrolet  15.00
1986 Chevrolet  15.00
.
2018 Dodge     16.00
2019 Dodge     16.00
2020 Dodge     null
```

That looks better.

2.20 Finding Duplicates

We saw that the `.join` method allows us to validate the join type. This can help us catch and deal with errors where we have a one-to-many or many-to-many join when we expect a one-to-one join.

However, perhaps you have data from an external source and want to validate that the data is correct and that there are no duplicates. There exists an `.is_duplicated` method on both the DataFrame and the column expression. Generally, we want to combine an expression that returns boolean values with the `.filter` method to filter out the rows that are duplicates.

Let's see if there are duplicate rows in the `autos` dataset.

```
>>> print(autos
... .select(pl.col('highway08').is_duplicated())
... )
shape: (41_144, 1)
  highway08
  bool
true
true
true
.
true
true
true
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

It looks like the `highway08` column has many duplicate values.

We can view the rows with duplicate values using the `.filter` method. We pass in the boolean mask we created with the `.is_duplicated` method.

```
>>> print(autos
...     .filter(pl.col('highway08').is_duplicated())
... )
```

shape: (41_137, 15)

year	make	model	.	created0	is_autom	num_gear
i16	cat	cat	.	n	atic	s
			.	datetime	str	u8
1985	Alfa Romeo	Spider Veloce	.	2013-01-01 00:00:00	false	5
1985	Ferrari	Testarossa	.	2013-01-01 00:00:00	false	5
1985	Dodge	Charger	.	2013-01-01 00:00:00	false	5
.
1993	Subaru	Legacy AWD	.	2013-01-01 00:00:00	true	4
1993	Subaru	Legacy AWD	.	2013-01-01 00:00:00	false	5
1993	Subaru	Legacy AWD Turbo	.	2013-01-01 00:00:00	true	4

This isn't particularly interesting because many cars have the same highway fuel efficiency.

Let's look at a more realistic example. Let's assume we want to know if there are any duplicate `year model` combinations. Sadly, we can't include both columns in the `pl.col` expression like `pl.col('year', 'model').is_duplicated()`. That will throw an error.

If we want to determine where a combination of columns is duplicated as a group, we need to combine them. We do this with a `struct` expression.

In Polars, a `struct` is a single column that contains a sequence of values (the collection of columns that are combined).

Let's check for duplicates of `year model` combinations.

First, I will combine the two columns into a single column using the `pl.struct` function.

```
>>> print(autos
... .select(pl.struct(pl.col('year'), pl.col('model'))))
... )
shape: (41_144, 1)
| year
| struct[2]
+-----+
{1985,"Spider Veloce 2000"}
{1985,"Testarossa"}
{1985,"Charger"}
.
{1993,"Legacy AWD"}
{1993,"Legacy AWD"}
{1993,"Legacy AWD Turbo"}
```

Now I can use the `.is_duplicated` method to find the duplicate values.

```
>>> print(autos
... .select(pl.struct(pl.col('year'), pl.col('model')).is_duplicated()))
... )
shape: (41_144, 1)
| year
| bool
+-----+
false
false
true
.
true
true
true
```

We can find the duplicate rows by replacing `.select` with `.filter`.

```
>>> print(autos
... .filter(pl.struct(pl.col('year'), pl.col('model')).is_duplicated())
... .sort('year', 'model')
... )
shape: (32_294, 15)
| year   make    model      .  createdOn    is_automat  num_gears
| i16     cat     cat        .  datetime[    tic          u8
|                  µs]           str
+-----+
1984   Alfa   Spider     .  2013-01-0  false       5
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

	Romeo	Veloce	1			
		2000	00:00:00			
1984	Alfa Romeo	Spider Veloce	.	2013-01-0	false	5
		1				
		2000	00:00:00			
1984	Dodge	Charger	.	2013-01-0	false	4
		1				
			00:00:00			
.
2020	BMW	M2 Competitor Coupe	.	2019-04-0	true	7
		2				
2020	Jeep	Gladiator Pickup	.	2019-04-0	true	8
		2				
2020	Jeep	Gladiator Pickup	.	2019-04-0	false	6
		2				
		4WD	00:00:00			

We can see that there are multiple *year model* combinations that are duplicated.

2.21 Finding Missing Values

One big difference between Pandas and Polars is that Polars distinguishes between NULL and NaN values. A NULL value is a value that is missing. A NaN value is a value that is not a number and may result from a mathematical operation. Non-numerical data types cannot be NaN values. For example, a string cannot be a NaN value. Polars has both an `.is_null` method and an `.is_nan` method that are found on a column expression. When reading a CSV file with Polars, it is possible to get both NULL and NaN values in a single column. I'll demonstrate this below.

I generally avoid synthetic data, but it is useful for demonstrating the difference between NULL and NaN values.

```
>>> missing_df = pl.DataFrame({'val': [-1.1, 0, 2.3, None, 5.7, 7]})  
>>> print(missing_df)  
shape: (6, 1)  
| val  
| f64  
|---  
| -1.10  
| 0.00  
| 2.30  
| null
```

5.70	
7.00	

You can see that there is a `null` value in the `val` column. Let's see what happens when we try to divide the `val` column by itself.

```
>>> print(missing_df
... .with_columns(val2=pl.col('val')/pl.col('val'))
... )
shape: (6, 2)
  val      val2
  f64      f64
  -1.10    1.00
  0.00     NaN
  2.30    1.00
  null     null
  5.70    1.00
  7.00    1.00
```

When 0 is divided by 0, the result is `NaN`. The `val2` column has both `null` and `NaN` values. Again, `NaN` is different from `null` in Polars. The former means not a number (the result of an invalid math operation), and the latter means a missing value.

There are a few expression methods that deal with missing values and `NaN`s: `.is_finite`, `.is_nan`, and `.is_null`.

Let's use the predicate methods that deal with missing values and `nans` to see how they work on a column that has both `NaN` and `null` values.

```
>>> print(missing_df
... .with_columns(val2=pl.col('val')/pl.col('val'))
... .with_columns(is_null2=pl.col('val2').is_null(),
...                 is_nan2=pl.col('val2').is_nan(),
...                 is_finite2=pl.col('val2').is_finite(),
...                 interpolate=pl.col('val2').interpolate()
...   )
... )
shape: (6, 6)
  val      val2  is_null2  is_nan2  is_finite2  interpolate
  f64      f64    bool       bool      bool        f64
  -1.10    1.00  false     false     true       1.00
  0.00     NaN   false     true     false      NaN
  2.30    1.00  false     false     true       1.00
  null     null  true      null     null       1.00
  5.70    1.00  false     false     true       1.00
  7.00    1.00  false     false     true       1.00
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Notice that the `.is_nan` and `.is_finite` methods return `null` for NULL values. However, `.is_null` returns `false` for `NaN`.

Let's use the `.fill_null` method to replace the NULL values in the `val` column with 0.

```
>>> print(missing_df
... .with_columns(
...     forward=pl.col('val').fill_null(strategy='forward'),
...     backward=pl.col('val').fill_null(strategy='backward'),
...     min=pl.col('val').fill_null(strategy='min'),
...     max=pl.col('val').fill_null(strategy='max')
...   )
... )
shape: (6, 5)
```

val	forward	backward	min	max
f64	f64	f64	f64	f64
-1.10	-1.10	-1.10	-1.10	-1.10
0.00	0.00	0.00	0.00	0.00
2.30	2.30	2.30	2.30	2.30
null	2.30	5.70	-1.10	7.00
5.70	5.70	5.70	5.70	5.70
7.00	7.00	7.00	7.00	7.00

Here is an example of using the `.fill_null` method and two examples of the `.interpolate` method:

```
>>> print(missing_df
... .with_columns(
...     mean=pl.col('val').fill_null(pl.col('val').mean()),
...     interpolate=pl.col('val').interpolate(),
...     nearest=pl.col('val').interpolate('nearest')
...   )
... )
shape: (6, 4)
```

val	mean	interpolate	nearest
f64	f64	f64	f64
-1.10	-1.10	-1.10	-1.10
0.00	0.00	0.00	0.00
2.30	2.30	2.30	2.30
null	2.78	4.00	5.70
5.70	5.70	5.70	5.70
7.00	7.00	7.00	7.00

Note that if you are doing machine learning and using `.fill_null` to replace missing values, you should be careful not to replace missing values

with values from the future. For example, if you are predicting the price of a house, you should not replace missing values with the median price of a house in the future. This would be a data leak. You would want to avoid `.fill_null(strategy='backward')` and `.interpolate()` in this case, and use `.fill_null(strategy='forward')` or `.fill_null(value)` instead.

For “fun”, let’s round trip this to a CSV file and see what happens.

```
>>> (missing_df
...     .with_columns(val2=pl.col('val')/pl.col('val'))
...     .write_csv('missing.csv')
... )
```

I’ll use a Jupyter *cell magic*, `%pfile`, to look at the contents of the file.

```
%pfile missing.csv
val, val2
-1.1, 1.0
0.0, NaN
2.3, 1.0
,
5.7, 1.0
7.0, 1.0
```

Notice that the last column, `val2`, has `NaN` and a missing value. This is how Polars supports both `NaN` and `null` values in a single column in a CSV file.

Let’s read this and see what happens.

```
>>> print(pl.read_csv('missing.csv'))
shape: (6, 2)
[[{"val": -1.1, "val2": 1.0}, {"val": 0.0, "val2": null}, {"val": 2.3, "val2": 1.0}, {"val": null, "val2": null}, {"val": 5.7, "val2": 1.0}, {"val": 7.0, "val2": 1.0}]]
```

Polars is able to parse this correctly. It can distinguish between `NaN` and `null` values. If we read this with Pandas, it would convert both of the values to `NaN`.

Now let’s convert the dataframe to Pandas with the `.to_pandas` method.

```
>>> print(missing_df
...     .with_columns(val2=pl.col('val')/pl.col('val'))
...     .to_pandas())
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
val  val2
0 -1.1  1.0
1  0.0  NaN
2  2.3  1.0
3  NaN  NaN
4  5.7  1.0
5  7.0  1.0
```

Note that Polars can export both NULL and NaN values to CSV. However, Pandas cannot import NULL values from CSV. Pandas will convert NULL values to NaN values. This is a limitation of Pandas and something you should be aware of when working with both Polars and Pandas.

2.22 Third-Party Libraries and Missing Values

Let's see how libraries like XGBoost and NumPy deal with Polars data that has NaN and null values.

I'm going to use the XGBoost library to make a regression model. I'll train it on the synthetic data we created that have both NaN and null. XGBoost supports missing values in Pandas data. Let's see how it works with Polars. Here's our X data:

```
>>> X = (missing_df
... .with_columns(val2=pl.col('val')/pl.col('val')))

>>> print(X)
shape: (6, 2)
val      val2
f64      f64
|-|
-1.10  1.00
0.00  NaN
2.30  1.00
null  null
5.70  1.00
7.00  1.00
```

Here's the labels for the regression model.

```
>>> y = pl.Series(range(6))
>>> print(y)
shape: (6,)
Series: '' [i64]
[
    0
    1
```

```

2
3
4
5
]
```

Let's train the model:

```

>>> import xgboost as xgb
>>> reg = xgb.XGBRegressor()
>>> reg.fit(X, y)
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None,
             early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None,
             feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=None,
             max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=None, max_leaves=None,
             min_child_weight=None, missing=nan,
             monotone_constraints=None,
             multi_strategy=None, n_estimators=None, n_jobs=None,
             num_parallel_tree=None, random_state=None, ...)
```

It looks like it trained without issue.

Let's make a prediction using the training data.

```

>>> print(reg.predict(X))
[1.1198493e-03 1.0000179e+00 1.9999869e+00 3.0000448e+00 3.9999456e+00
 4.9988852e+00]
```

Cool! It appears that XGBoost works fine with Polars numeric values.
Note that it returns a NumPy array.

Let's try the NumPy library. We will take the sine of the X variable:

```

>>> import numpy as np
>>> np.sin(X)
array([[-0.89120736,  0.84147098],
       [ 0.          ,         nan],
       [ 0.74570521,  0.84147098],
       [         nan,         nan],
       [-0.55068554,  0.84147098],
       [ 0.6569866 ,  0.84147098]])
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

NumPy treats the `NaN` and `null` as `nan`. Polars represents missing values using a bitmask to indicate that values are missing. NumPy doesn't have this concept.

2.23 Additional Column Selectors and Missing Values

Let's look at the numeric rows from the `autos` data where any value is `null`. I will use the `selectors` module to select the numeric columns. It has a `numeric` function that returns all numeric columns in a DataFrame.

```
>>> import polars.selectors as cs
```

```
>>> print(autos
... .select(cs.numeric().is_null().sum())
... )
```

```
shape: (1, 7)
| year    displ   cylinders   .   city08   highway08   num_gears |
| u32     u32      u32          .   u32       u32          u32      |
| 0       204     206           .   0         0            0        |
```

The `selectors` module has many convenience functions for selecting columns. For example, if you want to select all integer columns, you can use the `integer` function.

Here are the other selectors that are available.

Table 2.1: Column Selector functions from the `polars.selectors` module.

Selector	Description
<code>all</code>	Select all columns
<code>binary</code>	Select all binary columns
<code>boolean</code>	Select all boolean columns
<code>categorical</code>	Select all categorical columns
<code>date</code>	Select all date columns
<code>datetime</code>	Select all datetime columns
<code>float</code>	Select all float columns
<code>integer</code>	Select all integer columns
<code>numeric</code>	Select all numeric columns
<code>object</code>	Select all object columns
<code>string</code>	Select all string columns
<code>temporal</code>	Select all temporal columns
<code>time</code>	Select all time columns
<code>timezone</code>	Select all timezone columns

You can also perform set operations on the selectors. You can use `+` and `|` to perform a *union* operation. You can use `-` to perform a *set difference* operation.

2.23. Additional Column Selectors and Missing Values

You can use `&` to perform an *intersection* operation. You can use `^` to perform a *symmetric difference* operation.

For example, if you want to select all columns that are not dates or booleans, use this code. It starts with all the columns, removes any date columns, and finally removes boolean columns using set difference.

```
>>> no_dates_or_bools = (cs.all() - cs.date() - cs.boolean())
>>> print(no_dates_or_bools)
SELECTOR
```

You can debug the selectors with the `expand_selector` function. This will show you which columns are selected by the selector.

```
>>> print(cs.expand_selector(autos, no_dates_or_bools))
('year', 'make', 'model', 'displ', 'cylinders', 'trany', 'drive',
 'VCClass', 'fuelType', 'barrels08', 'city08', 'highway08', 'createdOn',
 'is_automatic', 'num_gears')
```

Let's fill in the missing values of `displ` and `cylinders` with the mean of each year's `make` column. First, let's create an expression to perform that calculation:

```
>>> print(autos
... .select(pl.col('displ').mean().over(['year', 'make']))
... )
shape: (41_144, 1)
+-----+
| displ |
| f32   |
+-----+
| 2.25  |
| 3.57  |
| 3.33  |
| .     |
| 1.95  |
| 1.95  |
| 1.95  |
```

Now, let's debug this by looking at the rows where the `displ` column is null. I will convert the `.select` to a `.with_columns` so that I'm adding a new column with the mean of the `displ` column for each `make` and `year` combination. Then, I will use the `.filter` method to select the rows where the `displ` column is null. Finally, I will use the `.select` method to limit the columns to the `make`, `year`, `displ`, and `mean_displ` columns.

```
>>> print(autos
... .with_columns(
...     pl.col('displ').mean().over(['year', 'make']).alias('mean_displ')))
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

```
... .filter(pl.col('displ').is_null())
... .select(['year', 'make', 'displ', 'mean_displ'])
...
shape: (204, 4)
| year   make    displ  mean_displ |
| i16    cat     f32    f32   |
|-----|
| 2000   Nissan  null   2.84  |
| 2000   Toyota  null   2.64  |
| 2001   Toyota  null   2.74  |
| .      .       .      .      |
| 2019   Tesla   null   null   |
| 2019   Tesla   null   null   |
| 2019   Tesla   null   null   |
```

The *mean_displ* column is `null` for *Tesla* cars. This is because no *Tesla* cars have internal combustion engines. Let's address this by calling `.fill_null(0)` on the *mean_displ* column. I will also pull the expression out into a variable because it is getting a little long (and I will use it again later).

```
>>> mean_displ = (pl.col('displ')
...                 .mean().over(['year', 'make']))
...                 .fill_null(0)
...                 .alias('mean_displ')
... )
>>> print(autos
... .with_columns(mean_displ)
... .filter(pl.col('displ').is_null())
... .select(['year', 'make', 'displ', 'mean_displ'])
... )
shape: (204, 4)
| year   make    displ  mean_displ |
| i16    cat     f32    f32   |
|-----|
| 2000   Nissan  null   2.84  |
| 2000   Toyota  null   2.64  |
| 2001   Toyota  null   2.74  |
| .      .       .      .      |
| 2019   Tesla   null   0.00  |
| 2019   Tesla   null   0.00  |
| 2019   Tesla   null   0.00  |
```

Finally, to create a new dataframe without missing *displ* values, we will fill in the missing values with the *mean_displ* column. I need to pass the expression I stored in the `mean_displ` variable to the `.fill_null` method.

```
>>> print(autos
... .with_columns(pl.col('displ').fill_null(mean_displ))
... .select(['year', 'make', 'model', 'displ'])
... )
shape: (41_144, 4)
| year   make      model      displ |
| i16    cat       cat       f32  |
|-----|
| 1985   Alfa Romeo Spider Veloce 2000  2.00 |
| 1985   Ferrari   Testarossa        4.90 |
| 1985   Dodge     Charger          2.20 |
| .           .          .          .          |
| 1993   Subaru    Legacy AWD        2.20 |
| 1993   Subaru    Legacy AWD        2.20 |
| 1993   Subaru    Legacy AWD Turbo  2.20 |
```

This example showed you how I debug and work out the expressions I need to use to manipulate data. I generally start with a `.select` and then convert it to a `.with_columns` to add new columns. I can also use `.filter` to select the values I want to explore with boolean expressions.

2.24 Map and Apply

If you need to integrate Polars with a library that doesn't understand Polars expressions, you can use the `.map_batches` method to pass a series to a custom function.

Let's explore how this works. I'm going to make a `debug` function that prints out the type of the object passed to it. This will help us understand what happens when passing a Polars series to a custom function.

```
>>> def debug(thing):
...     print(type(thing))
...     return thing

>>> print(autos
... .select(pl.col('make').map_batches(debug))
... )
<class 'polars.series.series.Series'>
shape: (41_144, 1)
| make |
| cat  |
|-----|
| Alfa Romeo |
| Ferrari   |
| Dodge     |
| .          |
```

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Subaru
Subaru
Subaru

We see the Polars Series make an appearance. You can think of this as a materialized column that has many of the same operations as expressions.

Let's make a NumPy function that standardizes a column. This will be a function that subtracts the mean and divides it by the standard deviation. We could do this with Polars expressions, but we want to demonstrate how to use the `.map_batches` method. I need to convert the Polars series into NumPy using `.to_numpy`.

```
>>> import numpy as np
>>> def standardize(arr_pl):
...     arr = arr_pl.to_numpy()
...     return (arr - np.mean(arr)) / np.std(arr)

>>> print(autos
...     .select(city08_standardized=pl.col('city08').map_batches(standardize))
... )
shape: (41_144, 1)
city08_standardized
f64
|-----|
0.08
-1.19
0.59
.
-0.05
-0.05
-0.30
```

As I said, I prefer to do this in Polars directly. For completeness, here is the Polars code to standardize the `comb08` column using expressions.

```
>>> def standardize_pl(col):
...     return (col - col.mean()) / col.std()

>>> print(autos
...     .select(city08_standardized=standardize_pl(pl.col('city08')))
... )
shape: (41_144, 1)
city08_standardized
f64
|-----|
0.08
```

```
-1.19
0.59
.
-0.05
-0.05
-0.30
```

2.25 Chapter Functions and Methods

Table 2.2: Functions and methods from chapter

Function	Description
<code>df.select(*exprs, **named_exprs)</code>	Select columns from this DataFrame. Accepts expressions and literals. Keyword arguments are used to rename columns.
<code>df.with_columns(*exprs, **named_exprs)</code>	Add columns to this DataFrame. Accepts expressions and literals. Keyword arguments are used to rename columns.
<code>df.with_row_index(name='index', offset=0)</code>	Insert a new first column with monotonically increasing row numbers starting at offset.
<code>df.drop(columns, *more_columns)</code>	Remove columns from this DataFrame. You can pass in a list or each column as a parameter. You can also use a selector to remove columns.
<code>col.exclude(columns, *more_columns)</code>	Exclude columns from a multi-column expression. Accepts regular expression input that starts with ^ and ends with \$.
<code>df.rename(mapping)</code>	Rename columns in this DataFrame. The mapping is a dictionary of old column names to new column names.
<code>df.join(other, on=None, how='inner', *, left_on=None, right_on=None, suffix='_right', validate='m:m', join_nulls=False)</code>	Join this DataFrame with another DataFrame. The <code>on</code> parameter is a list of columns to join on. The <code>how</code> parameter is the type of join. The <code>left_on</code> and <code>right_on</code> parameters are used to specify the columns to join on. The <code>suffix</code> parameter is used to add a suffix to the columns from the right DataFrame. The <code>validate</code> parameter is used to validate the join type. The <code>join_nulls</code> parameter is used to specify if null values should be joined.
<code>df.vstack(other, *, in_place=False)</code>	Add rows from another DataFrame to this DataFrame. The <code>in_place</code> parameter is used to specify if the rows should be added to this DataFrame or a new DataFrame should be returned.

2. Data Manipulation with Polars Using the Fuel Economy Dataset

Function	Description
<code>df.group_by(by, *more_by, maintain_order=False) grouped.agg(*aggs, **named_aggs)</code>	Group this DataFrame by the specified columns.
<code>df.sort(by, *more_by, descending=False, nulls_last=False)</code>	Aggregate the grouped DataFrame. The aggs parameter(s) is an expression(s). The named_aggs parameter are key value pairs of column names and expressions.
<code>df.pivot(values=None, index=None, columns=None, aggregate_function=None, maintain_order=True, sort_columns=False, separator=' '_')</code>	Sort this DataFrame by the specified columns. descending is a sequence of booleans for each column.
<code>df.melt(id_vars, values_vars, variable_name, value_name)</code>	Pivot the DataFrame from long to wide format. Place values from columns into the header. The index can specify values to group by. The values parameter specifies the values to aggregate. If there are no duplicate values, you can omit the aggregate_function parameter. If there are duplicate values, you must specify an aggregate function. The maintain_order parameter specifies if the order of the columns should be maintained. The sort_columns parameter specifies if the columns should be sorted. The separator parameter specifies the separator to use between the column names.
<code>col.is_duplicated() pl.struct(*exprs, schema=None, eager=False) col.alias(name)</code>	Melt the DataFrame from wide to long format. The id_vars parameter is a list of columns to keep. The values_vars parameter is a list of columns to melt. The variable_name parameter is the name of the column that contains the melted column names. The value_name parameter is the name of the column that contains the melted values.
<code>col.is_finite()</code>	Return a boolean mask of duplicate values.
<code>col.is_nan()</code>	Create a single expression (column) from a combination of exprs (expressions).
<code>col.is_finite()</code>	Return an expression with boolean values indicating if the value is finite. NaN evaluates to false. null evaluates to null.
<code>col.is_nan()</code>	Return an expression with boolean values indicating if the value is NaN. null evaluates to null.

Function	Description
<code>col.is_null()</code>	Return an expression with boolean values indicating if the value is null. NaN evaluates to false.
<code>col.fill_nan(value)</code>	Replace NaN values with value.
<code>col.fill_null(value=None, strategy=None, limit=None)</code>	Replace null values with value. The <code>strategy</code> parameter can be 'forward', 'backward', 'min', 'max', 'zero', or 'one'. The <code>limit</code> parameter is the number of consecutive null values to replace. If <code>limit</code> is None, all null values will be replaced.
<code>col.interpolate(method='linear')</code>	Replace null values with interpolated values. The <code>method</code> parameter can be 'linear', or 'nearest'.
<code>cs.expand_selector(df, selector)</code>	Return a list of column names that are selected by the selector.
<code>col.over(expr, *more_exprs, mapping_strategy = 'group_to_rows')</code>	Create a window expression. Group over the specified expressions. The <code>mapping_strategy</code> parameter can be 'group_to_rows', 'join', or 'explode'. Generally, an aggregation function is used before the <code>.over</code> method.
<code>col.map_batches(function, return_dtype=None, *, skip_nulls=True, pass_name=False, strategy='thread_local')</code>	Apply a function to each batch of data. The <code>return_dtype</code> parameter is used to specify the return type. The <code>skip_nulls</code> parameter is used to specify if null values should be skipped. The <code>pass_name</code> parameter is used to specify if the column name should be passed to the function. The <code>strategy</code> parameter is used to specify the parallelization strategy. 'thread_local' runs on a single thread. 'threaded' runs on multiple threads. (if function releases GIL)
<code>col.explode()</code>	Explode a column of lists into a row for each value in the list.

2.26 Summary

This chapter has explored adding and removing columns, renaming them for clarity, and manipulating DataFrames through various joining methods. The techniques learned here are fundamental in transforming raw data into actionable insights. The power of data manipulation lies in its ability to reshape information into a form that answers specific questions and reveals hidden patterns. Continue to practice and apply these techniques to diverse

2. Data Manipulation with Polars Using the Fuel Economy Dataset

datasets, and you'll grow more proficient and confident in your data analysis skills.

2.27 Exercises

1. Create a new column in the Fuel Economy dataset that calculates the average MPG (miles per gallon) by averaging the `city08` and `highway08` columns.
2. Remove the `createdOn` column and explain why you might consider doing this for machine learning.
3. Perform an inner join using the Fuel Economy dataset and a new DataFrame you create, which includes additional information about car makes (e.g., country of origin).
4. Reshape the data to have the car makes as columns and years as rows, showing the average city MPG for each make per year.
5. Using the concept of lazy evaluation in Polars, attempt to optimize the query plan for a complex operation, such as calculating the average MPG per make and model for cars after the year 2000. Compare the performance of the eager and lazy approaches and discuss the differences.
6. Try to integrate the dataset with external APIs to enrich the data. For example, you could fetch additional data about fuel prices over the years and correlate it with fuel efficiency trends.

Chapter 3

String Manipulation

Polars has powerful string manipulation capabilities. This chapter will explore some of the most common string manipulation operations. If you are familiar with Python, you will find that Polars string manipulation is similar to Python's.

3.1 Twitter Data

I will use an export of my Twitter/X data¹ to demonstrate string manipulation. Let's load the data from the CSV file.

```
>>> url = 'https://github.com/mattharrison/datasets/raw/' \
...     'master/data/___mharrison__2020-2021.csv'
>>> import polars as pl
>>> raw = pl.read_csv(url)
>>> print(raw)
shape: (5_791, 40)
| Tweet id    Tweet      Tweet      . promoted promoted promoted |
| i64          permalink  text      . dial      media     media   |
|                         str       str      phone    views     engage-  |
|                         str       str      str      str      nts      |
|                         str       str      str      str      str      |
| 121258051  https://t Sounds   . null      null      null   |
| 790578073  witter.co like a   . null      null      null   |
| 7           m/___mharr great   . null      null      null   |
|                      ison_. topic!  . null      null      null   |
|                      https.  . null      null      null   |
| 121258249  https://t @FogleBi . null      null      null   |
| 482803609  witter.co rd Looks  . null      null      null   |
```

¹Sadly, it looks like the ability to export this data has disappeared following the Elon takeover.

3. String Manipulation

7	m/_mharr ison_.	like SLC. I can.			
121261373	https://t witter.co	@afilina . null null null			
569869004		That's			
9	m/_mharr ison_.	really amount an.			
.
147589144	https://t witter.co	@posco . null null null			
124302540		Visiting			
8	m/_mharr ison_.	Hawaii for the h.			
147645381	https://t witter.co	@johndsa . null null null			
975187865		unders			
6	m/_mharr ison_.	My son just built.			
147702472	https://t witter.co	@tunguz . null null null			
205115801		Xgboost			
8	m/_mharr ison_.				

This data is exported from the Twitter analytics page. There are many columns in that file that are empty. I'm going to limit the columns to the ones that are relevant to this demonstration. Here are the columns with a description:

- *Tweet id*: The unique identifier of the tweet.
- *Tweet permalink*: The URL of the tweet.
- *Tweet text*: The text of the tweet.
- *time*: The time the tweet was created.
- *impressions*: The number of times the tweet was seen.
- *engagements*: The number of times the tweet was interacted with.
- *engagement rate*: The percentage of engagements per impression.
- *retweets*: The number of times the tweet was retweeted.
- *replies*: The number of times the tweet was replied to.
- *likes*: The number of times the tweet was liked.
- *user profile clicks*: The number of times the user profile was clicked from the tweet.

```
>>> def tweak_twit(df):
...     return (df
...         .select(['Tweet id', 'Tweet permalink', 'Tweet text', 'time',
...                 'impressions', 'engagements', 'engagement rate',
```

```

...
    'retweets', 'replies', 'likes', 'user profile clicks'])
...
)

>>> twit = tweak_twit(raw)
>>> print(twit)
shape: (5 791, 11)
   Tweet id      Tweet      Tweet      .  replies  likes  user
   i64          permalink  text      f64      f64  profile
                   str        str
                                         clicks
                                         f64

 1212580517  https://tw itter.com/_mharrisohn. Sounds   .  0.00   3.00   3.00
 905780737
 1212582494  https://tw itter.com/_mharrisohn. @FogleBir d Looks   .  0.00   1.00   1.00
 828036097
 1212613735  https://tw itter.com/_mharrisohn. @afilina That's   .  0.00   1.00   2.00
 698690049
 1475891441  .           .           .  0.00   4.00   4.00
 243025408
 1476453819  https://tw itter.com/_mharrisohn. @johndsau nders My   .  0.00   2.00   4.00
 751878656
 1477024722  https://tw itter.com/_mharrisohn. @tunguz Xgboost   .  3.00   23.00  8.00
 051158018

```

3.2 Validating the Data

The column expressions have an `.str` accessor, much like Pandas. This accessor provides access to string manipulation methods. These methods are similar to Python's and Pandas' string manipulation methods.

3. String Manipulation

Let's look at the methods available on the `.str` accessor. (Note that some of these are deprecated.)

```
>>> col = pl.col('Tweet permalink')
>>> print([m for m in dir(col.str)
...       if not m.startswith('_')])
['concat', 'contains', 'contains_any', 'count_match', 'count_matches',
 'decode', 'encode', 'ends_with', 'explode', 'extract', 'extract_all',
 'extract_groups', 'find', 'json_decode', 'json_extract',
 'json_path_match', 'len_bytes', 'len_chars', 'lengths', 'ljust',
 'lstrip', 'n_chars', 'pad_end', 'pad_start', 'parse_int', 'replace',
 'replace_all', 'replace_many', 'reverse', 'rjust', 'rstrip', 'slice',
 'split', 'split_exact', 'splith', 'starts_with', 'strip',
 'strip_chars', 'strip_chars_end', 'strip_chars_start', 'strip_prefix',
 'strip_suffix', 'strptime', 'to_date', 'to_datetime', 'to_decimal',
 'to_integer', 'to_lowercase', 'to_time', 'to_titlecase',
 'to_uppercase', 'zfill']
```

Let's see the overlap of Python and Polars functionality. I'll do this using `set` functionality.

```
>>> print(sorted([m for m in
...      set(dir(col.str)) & set(dir(''))
...      if not m.startswith('_')]))
['encode', 'find', 'ljust', 'lstrip', 'replace', 'rjust', 'rstrip',
 'split', 'strip', 'zfill']
```

Here are the extra methods (or methods that are spelled differently like `.to_uppercase`).

```
>>> print(sorted([m for m in
...      set(dir(col.str)) - set(dir(''))
...      if not m.startswith('_')]))
['concat', 'contains', 'contains_any', 'count_match', 'count_matches',
 'decode', 'ends_with', 'explode', 'extract', 'extract_all',
 'extract_groups', 'json_decode', 'json_extract', 'json_path_match',
 'len_bytes', 'len_chars', 'lengths', 'n_chars', 'pad_end', 'pad_start',
 'parse_int', 'replace_all', 'replace_many', 'reverse', 'slice',
 'split_exact', 'splith', 'starts_with', 'strip_chars',
 'strip_chars_end', 'strip_chars_start', 'strip_prefix', 'strip_suffix',
 'strptime', 'to_date', 'to_datetime', 'to_decimal', 'to_integer',
 'to_lowercase', 'to_time', 'to_titlecase', 'to_uppercase']
```

There is a lot of functionality there. In general, I don't like writing exhaustive documentation covering every method. I prefer to showcase functionality that is used frequently. I will demonstrate some of these methods with my tweet data.

3.3. Extracting the Username

Let's validate that the URLs in the Tweet permalink column are valid. We can check if the URLs start with `https://twitter.com/`. We will use `.str.starts_with` to do this. This is analogous to Python's `.startswith` method (it is just spelled differently).

```
>>> 'https://metasnake.com'.startswith('https://twitter.com')
False
```

I'll try to see if any links don't start with the correct prefix in the column. Polars will apply this logic to every value in the column.

```
>>> print(twit
...     .filter(~col.str.starts_with('https://twitter.com/'))
... )
shape: (0, 11)
```

Tweet id	Tweet	Tweet text	.	replies	likes	user
i64	permalink	str		f64	f64	profile
		str				clicks

It looks like every link starts with `https://twitter.com/`. This is good.

3.3 Extracting the Username

Our URLs look like this: HOST/USER/status/ID

Let's write an expression that extracts the username from the URL. We can do this in a few ways. One is by splitting the URL on the / character. The other is by using a regular expression. We will show both methods.

The `.str.split` method is analogous to Python's `.split` method.

```
>>> print('https://metasnake.com/effective-polars'.split('/'))
['https:', '', 'metasnake.com', 'effective-polars']
```

We will use the `.str.split` method to split the URL column on the / character. Look at the return type of this method. This returns a *list* column. Look a little closer. It isn't just a list, but a list that has strings. It is a typed list. Polars likes to track such information because it helps the query planner make intelligent choices.

```
>>> print(twit
...     .select(col.str.split('/'))
... )
shape: (5_791, 1)
| Tweet permalink
| list[str]
```

3. String Manipulation

```
["https:", "", . "12125805179057.  
["https:", "", . "12125824948280.  
["https:", "", . "12126137356986.  
. .  
["https:", "", . "14758914412430.  
["https:", "", . "14764538197518.  
["https:", "", . "14770247220511.
```

Just as a string column has an `.str` accessor, lists also have a `.list` accessor to access list methods.

Here are the methods.

```
>>> print([m for m in dir(col.list)
...     if not m.startswith('_')])
['all', 'any', 'arg_max', 'arg_min', 'concat', 'contains',
 'count_match', 'count_matches', 'diff', 'drop_nulls', 'eval',
 'explode', 'first', 'gather', 'gather_every', 'get', 'head', 'join',
 'last', 'len', 'lengths', 'max', 'mean', 'median', 'min', 'n_unique',
 'reverse', 'sample', 'set_difference', 'set_intersection',
 'set_symmetric_difference', 'set_union', 'shift', 'slice', 'sort',
 'std', 'sum', 'tail', 'take', 'to_array', 'to_struct', 'unique', 'var']
```

There is a bunch of functionality here as well.

Let's get the length of the list. This will give us the number of parts in the URL.

```
>>> print(twit
... .select(col.str.split('/'))
... .list.len())
...
shape: (5_791, 1)
| Tweet permalink
| u32
|-
| 6
| 6
| 6
| .
| 6
| 6
| 6
```

If I wanted to convert each part of the list into a column, I need to convert the list into a *struct*. Then I can use the `.unnest` method. This method puts every item from a struct into its own column.

The `.list.to_struct` method converts each list into a struct. You can think of a *struct* datatype as record or structured type data (similar to a Python tuple).

```
>>> print(twit
... .select(col.str.split('/'))
... .list.to_struct()
... )
shape: (5_791, 1)
| Tweet permalink
| struct[6]
+-----+
{"https:","","twitter.com","__mh.
>{"https:","","twitter.com","__mh.
>{"https:","","twitter.com","__mh.
|.
>{"https:","","twitter.com","__mh.
>{"https:","","twitter.com","__mh.
>{"https:","","twitter.com","__mh.
```

Now we can use the dataframe's `.unnest` method to convert the struct into columns.

```
>>> print(twit
... .select(col.str.split('/'))
... .list.to_struct()
... .unnest('Tweet permalink')
... )
shape: (5_791, 6)
| field_0   field_1   field_2   field_3   field_4   field_5
| str       str       str       str       str       str
+-----+
https:      twitter.com  __mharrison status  12125805179
            .          .           .         .        05780737
https:      twitter.com  __mharrison status  12125824948
            .          .           .         .        28036097
https:      twitter.com  __mharrison status  12126137356
            .          .           .         .        98690049
.
https:      twitter.com  __mharrison status  14758914412
            .          .           .         .        43025408
https:      twitter.com  __mharrison status  14764538197
            .          .           .         .        51878656
https:      twitter.com  __mharrison status  14770247220
            .          .           .         .        51158018
```

Now, we have a column for each position in the struct.

The `.to_struct` dataframe method *nests* the columns of a dataframe into a single series.

3. String Manipulation

```
>>> print(twit
... .select(col.str.split('/'))
... .list.to_struct()
... .unnest('Tweet permalink')
... .to_struct()
... )
shape: (5_791,)
Series: '' [struct[6]]
[
  {"https:"", "twitter.com", "__mharrison__", "status", "1212580517905780737"
  }
  {"https:"", "twitter.com", "__mharrison__", "status", "1212582494828036097"
  }
  {"https:"", "twitter.com", "__mharrison__", "status", "1212613735698690049"
  }
  .
  {"https:"", "twitter.com", "__mharrison__", "status", "1475891441243025408"
  }
  {"https:"", "twitter.com", "__mharrison__", "status", "1476453819751878656"
  }
  {"https:"", "twitter.com", "__mharrison__", "status", "1477024722051158018"
  }
]
]
```

Note

If we wanted to *undo* the split (i.e., convert a list of strings into a single string), we could use the `.list.join` method:

```
>>> print(twit
... .select(col.str.split('/'))
... .list.join('/')
... )
... )
shape: (5_791, 1)
Tweet permalink
str
|
https://twitter.com/__mharrison_.
https://twitter.com/__mharrison_.
https://twitter.com/__mharrison_.
.
https://twitter.com/__mharrison_.
https://twitter.com/__mharrison_.
https://twitter.com/__mharrison_.
```

Note

We can also convert a struct column to a list column using `.map_elements`. Note that this method is generally slower than Polars expressions. Don't reach for it unless you really need it:

```
>>> print(twit
...     .select(col.str.split('/'))
...     .list.to_struct()
...     .select(pl.all().map_elements(lambda elem: list(elem)))
... )
shape: (5_791, 1)
| Tweet permalink
| list[str]
|
| ["field_0", "field_1", . "field_.
| ["field_0", "field_1", . "field_.
| ["field_0", "field_1", . "field_.
| .
| ["field_0", "field_1", . "field_.
| ["field_0", "field_1", . "field_.
| ["field_0", "field_1", . "field_.
```

The result of the unnest shows that the username is in the 4th part (Python is zero-based) of the URL. We can use the `.list.get` method to get index 3 (the 4th part) of the list.

```
>>> print(twit
...     .with_columns(username=col.str.split('/'))
...     .list.get(3))
... )
shape: (5_791, 12)
| Tweet id      Tweet      Tweet      .   likes    user      username
| i64          permalink  text       f64      profile   str
|                 str        str
|
| 121258051  https://t Sounds   .  3.00    3.00    __mharris
| 790578073  witter.co like a   .  3.00    3.00    on__
| 7           m/_mharr great   .  3.00    3.00    on__
|                   ison_. topic!   .  3.00    3.00    on__
|                   https.   .  3.00    3.00    on__
| 121258249  https://t @FogleBir .  1.00    1.00    __mharris
| 482803609  witter.co d Looks   .  1.00    1.00    on__
| 7           m/_mharr like SLC. .  1.00    1.00    on__
|                   ison_. I can.  .  1.00    1.00    on__
```

3. String Manipulation

121261373	https://t	@afilina	.	1.00	2.00	__mharris
569869004	witter.co	That's				on__
9	m/_mharr	really				
	ison_.	amount				
		an.				
.
147589144	https://t	@posco	.	4.00	4.00	__mharris
124302540	witter.co	Visiting				on__
8	m/_mharr	Hawaii				
	ison_.	for the				
		h.				
147645381	https://t	@johndsau	.	2.00	4.00	__mharris
975187865	witter.co	nders My				on__
6	m/_mharr	son just				
	ison_.	built.				
147702472	https://t	@tunguz	.	23.00	8.00	__mharris
205115801	witter.co	Xgboost				on__
8	m/_mharr					
	ison_.					

Note

We can also use *index access* to get the fourth element. For those familiar with Python and lists, this should seem normal. For those who are new to Python, this syntax can be a little confusing because it does not use parentheses. This is not a method call (with parentheses) but pulling out items with position 3 with index access using square brackets.

```
>>> print(twit
... .with_columns(username=col.str.split('/'))
... .list[3])
... )
```

shape: (5_791, 12)

	Tweet id	Tweet	Tweet	.	likes	user	username
	i64	permalink	text		f64	profile	str
		str	str			clicks	
						f64	
121258051	https://t	Sounds	.	3.00	3.00	__mharris	
790578073	witter.co	like a				on__	
7	m/_mharr	great					
	ison_.	topic!					
		https.					
121258249	https://t	@FogleBir	.	1.00	1.00	__mharris	
482803609	witter.co	d Looks				on__	

3.4. Extract the Username with a Regular Expression

7	m/_mharr	like SLC.				
	ison_.	I can.				
121261373	https://t	@afilina	.	1.00	2.00	_mharris
569869004	witter.co	That's				on__
9	m/_mharr	really				
	ison_.	amount				
		an.				
.
147589144	https://t	@posco	:	4.00	4.00	_mharris
124302540	witter.co	Visiting				on__
8	m/_mharr	Hawaii				
	ison_.	for the				
		h.				
147645381	https://t	@johndsau	.	2.00	4.00	_mharris
975187865	witter.co	nders My				on__
6	m/_mharr	son just				
	ison_.	built.				
147702472	https://t	@tunguz	.	23.00	8.00	_mharris
205115801	witter.co	Xgboost				on__
8	m/_mharr					
	ison_.					

3.4 Extract the Username with a Regular Expression

We can use a regular expression to extract the username from the URL.

I have wrapped the regular expression with the URL portion containing the username in parentheses. This will allow us to extract the username from the URL using a group capture.

Using the regular expression, the `.str.extract` method will extract the username from the URL. We can pass in a `group_index` argument to specify which group to extract. In this case, we want to extract the first group, so we pass in 1. (Unlike Python, regular expression group indexes are 1-based. Unlike Pandas (1.x), we also need to pass in a capture group (using parentheses) to extract the group.)

```
>>> regex = r'^https://twitter\.com/([a-zA-Z0-9_]+)/status/(\d+)$'
>>> print(twitt
... .select(user=col.str.extract(regex, group_index=1))
...
shape: (5_791, 1)
| user
| str
|
| __mharrison__
| __mharrison__
| __mharrison__
```

3. String Manipulation

```
.  
--mharrison--  
--mharrison--  
--mharrison--
```

Note

I'm not going to go deep into regular expressions. That topic deserves its own book. However, given the world we live in, I will give some advice. Regular expressions (regexes) are handy in that almost every programming language (that a normal person would want to use) supports the same syntax. However, regexes are notoriously difficult to create. They are also painful to read (after you or someone else created them). Although I've used them for over 25 years, I don't use them every day, and there are parts of the syntax that I still need to look up.

This is an area where AI does well. It can both craft and explain regular expressions. (I have no shame. I'll admit it. I asked AI to create that regex for me.) AI is also well suited to explaining a regular expression. Here's a sample of the explanation of the previous regex

This regular expression (regex) is designed to match and extract information from Twitter status URLs. Let's break it down piece by piece:

- ^: Asserts the start of the string. This ensures that the match must occur at the very beginning of the text being searched.
- https:\\/: Matches the literal string "https: / " in the URL. The backslash \ is used as an escape character to allow the forward slashes / and colon : to be included in the pattern as literal characters rather than special regex symbols.
- twitter\\.com\\/: Matches the literal string "twitter.com/". Similar to the previous section, the backslash \ is used to escape the period . and the forward slash / so they are treated as literal characters. The period . is a special character in regex that matches any single character (except newline), so it needs to be escaped to match a literal period.
- ([a-zA-Z0-9]+): This is a capturing group that matches one or more characters that are either uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), or underscores (_). This part matches and captures the Twitter username in the URL.

- `\/status\/:` Matches the literal string “/status/”. The forward slashes are escaped to ensure they are treated as literal characters.
- `(\d+):` This is another capturing group that matches one or more digits (`\d` is shorthand for `[0-9]`). This part of the regex matches and captures the numerical ID of the Twitter status (tweet).
- `$:` Asserts the end of the string. This ensures that the regex matches only if the pattern concludes exactly at the end of the searched text.

In summary, this regex is used to validate and extract the username and status ID from a Twitter URL, ensuring that the URL follows the format: “`https://twitter.com/username/status/statusID`”. The extracted parts are the username and the numerical ID of the tweet, which are captured by the parentheses () in the regex.

I want to work smarter. I’m perfectly happy that my AI assistant is taking the lead on regex duties.

3.5 Counting Words and Mentions

I want to explore the list functionality a little more. Let’s count the number of words in a tweet. If we use `.str.split`, we can split on a space character.

```
>>> tweet_col = pl.col('Tweet text')
>>> print(twit
... .select(tweet_col.str.split(' '))
... )
shape: (5_791, 1)
| Tweet text
| list[str]
|-
| ["Sounds", "like", . "topic!"]
| ["@FogleBird", "Looks", . "I"]
| ["@afilina", "That's", . "and"]
|
| ["@posco", "Visiting", . "the"]
| ["@johndsaunders", "My", . "buil.
| ["@tunguz", "Xgboost"]
```

Note the type of the result here. It is a list with strings inside of it. In Pandas, we need to loop over each row and sum up the list (or use `.apply` to do that). Polars has extra list functionality to perform many of these operations as expressions.

3. String Manipulation

For example, there is a `.list.len` method that calculates the length of the list. This tells us the number of words.

```
>>> print(tweet
...     .select(tweet_col.str.split(' ').list.len())
... )
shape: (5_791, 1)
Tweet text
u32
[6, 9, 33, ..., 35, 6, 2]
```

After I have built up this expression, I can throw it back into `.with_columns` to get an updated dataframe with the new column.

```
>>> print(twit
... .with_columns(word_count=tweet_col.str.split(' ').list.len())
... )
shape: (5_791, 12)
   Tweet id      Tweet      Tweet      .    likes    user    word_coun
   i64          permalink    text      .    f64      profile    t
   str           str         str      .    f64      clicks    u32
                                         f64

   121258051  https://t Sounds     .    3.00    3.00    6
   790578073  witter.co like a
   7          m/_mharr great
   ison_.      topic!
   https.
   121258249  https://t @FogleBir  .    1.00    1.00    9
   482803609  witter.co d Looks
   7          m/_mharr like SLC.
   ison_.      I can.
   121261373  https://t @afilina  .    1.00    2.00   33
   569869004  witter.co That's
   9          m/_mharr really
   ison_.      amount
   an.
   .
   147589144  https://t @posco   .    4.00    4.00    .
```

124302540	witter.co	Visiting				
8	m/_mharr	Hawaii				
	ison_.	for the				
		h.				
147645381	https://t	@johndsa	.	2.00	4.00	6
975187865	witter.co	nders My				
6	m/_mharr	son just				
	ison_.	built.				
147702472	https://t	@tunguz	.	23.00	8.00	2
205115801	witter.co	Xgboost				
8	m/_mharr					
	ison_.					

Let's count the number of mentions in a tweet. A mention is when you use an @ followed by the username. To count the mentions, we need to check if each word starts with @ and sum them up. This is where .list.eval comes in. This allows you to run an expression against an element in the list. You can chain off of pl.element to build up the element expression. We will tack on .str.starts_with('@') to evaluate the expression that checks if every string in the list starts with @.

```
>>> print(twit
... .select(tweet_col.str.split(' ').list.eval(
...     pl.element().str.starts_with('@'))
... )
shape: (5_791, 1)
Tweet text
list[bool]
[[false, false, . false]
 [true, false, . false]
 [true, false, . false]
 .
 [true, false, . false]
 [true, false, . false]
 [true, false]]
```

Note

If we try to chain the element expression off of the list rather than the element, we get a ComputeError. It doesn't make sense to try and check if a list starts with '@':

```
>>> print(twit
... .select(tweet_col.str.split(' ').str.starts_with('@'))
... )
```

3. String Manipulation

```
Traceback (most recent call last)
...
InvalidOperationError: cannot cast List type (inner: 'String', to:
'String')
```

We need to embed the element computation inside of `.list.eval`.

After we have the list with booleans, we need to count the number of trues. We can use the `.list.sum` method to count them because Python treats `True` as 1 and `False` as 0:

```
>>> print(twit
...   .select(tweet_col.str.split(' '))
...       .list.eval(pl.element().str.starts_with('@'))
...       .list.sum()
... )
shape: (5_791, 1)
Tweet text
u32
=====
0
1
1
.
1
1
1
```

Make sure you have your parentheses situated correctly. We don't want to sum up the lists inside of the `.eval` (that would be summing each boolean). Trying to do so gives us a `SchemaError`.

```
>>> print(twit
...   .select(tweet_col.str.split(' '))
...       .list.eval(pl.element()
...           .str.starts_with('@')
...           .list.len())
... )
Traceback (most recent call last)
...
SchemaError: invalid series dtype: expected `List`, got `bool`
```

Now that I've debugged both the `word_count` and `num_mentions` expressions, I will use them with `.with_columns` to get a dataframe with the new columns.

```
>>> print(twit
... .with_columns(word_count=tweet_col.str.split(' ').list.len(),
...     num_mentions=tweet_col.str.split(' ')
...             .list.eval(pl.element().str.starts_with('@'))
...             .list.sum())
... )
```

shape: (5_791, 13)

Tweet id	Tweet permalink str	Tweet text str	.	user profile clicks	word count u32	num mentions u32
i64				f64		
121258051	https://t.witter.co/m/_mharrison_.	Sounds like a great topic! https.	.	3.00	6	0
790578073	https://t.witter.co/m/_mharrison_.	@FogleBird Looks like SLC. I can.	.	1.00	9	1
482803609	https://t.witter.co/m/_mharrison_.	@afilina That's really amount an.	.	2.00	33	1
569869004	https://t.witter.co/m/_mharrison_.	@posco Visiting Hawaii for the h.	.	4.00	35	1
121261373	https://t.witter.co/m/_mharrison_.	@johndsa unders	.	4.00	6	1
147589144	https://t.witter.co/m/_mharrison_.	My son just built.	.	8.00	2	1
124302540	https://t.witter.co/m/_mharrison_.	@tunguz Xgboost	.			
6						
975187865						
147645381						
147702472						
205115801						
8						

3. String Manipulation

3.6 Checking for Emojis

I have heard a rumor that emojis drive engagement. If you use more emojis, the algorithm is tricked into showing your tweets to more folks. Let's see if that's true. I just so happened to have used a bunch of emojis in some of my tweets.

We can use the `.str.contains` method and loop over all emojis to check if a tweet contains an emoji. That would be slow. Let's see if we can create a regular expression matching any non-ASCII character.

```
>>> non_ascii = r'[^\\x00-\\x7F]'  
>>> tweet_col = pl.col('Tweet text')  
>>> print(twit  
... .select(tweet_col, has_emoji=tweet_col.str.contains(non_ascii))  
... )  
shape: (5_791, 2)  
| Tweet text | has_emoji |  
| str | bool |  
  
| Sounds like a great topic! https. | false |  
| @FogleBird Looks like SLC. I can. | true |  
| @afilina That's really amount an. | true |  
| . | . |  
| @posco Visiting Hawaii for the h. | true |  
| @johndsaunders My son just built. | false |  
| @tunguz Xgboost | false |
```

I'm curious now. Is there a correlation between emojis in a tweet and the number of engagements? (We'll include tweet length as well.) Let's find out.

```
>>> print(twit  
... .select('engagements',  
...         tweet_len=tweet_col.str.split(' ').list.len(),  
...         has_emoji=tweet_col.str.contains(non_ascii))  
... .corr()  
... )  
shape: (3, 3)  
| engagements | tweet_len | has_emoji |  
| f64 | f64 | f64 |  
  
| 1.00 | 0.06 | 0.04 |  
| 0.06 | 1.00 | 0.05 |  
| 0.04 | 0.05 | 1.00 |
```

The Pearson correlation coefficient reported here for emojis and engagements is 0.04. That is a very weak correlation. It looks like the rumor is false.

I'll visualize a scatterplot of this to ensure my intuition is correct. Because there are only two values for `has_emoji`, I will *jitter* the values by a little so they don't overlap as much. Polars has no random number functionality available. I will use numpy for that. When I use numpy, I break out of the preferred Polars style of doing everything with expressions. I use `.pipe` to pass the current state of the dataframe and then extract the actual column values as a series and pass them into my jitter function.

```
>>> import numpy as np
>>> def jitter(col, amount=.5):
...     return col + np.random.uniform(-amount, amount, len(col))

>>> (twit
...     .select('engagements',
...             has_emoji=tweet_col.str.contains(non_ascii).cast(pl.Int8))
...     .pipe(lambda df: df.with_columns(jitter(df['has_emoji'], amount=.4)))
...     .plot.scatter(x='engagements', y='has_emoji', alpha=.1)
... )
```

:Scatter [engagements] (has_emoji)

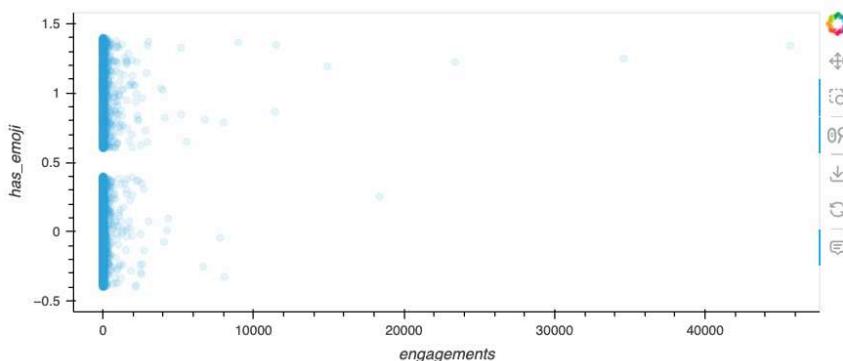


Figure 3.1: Jittered scatterplot of emoji status in tweet and engagements.

From inspecting the scatterplot, it appears that emojis have little impact on engagement. (The values are still pretty stacked up, even with jittering. One might consider applying a log transformation to pull them apart even more. Or, if you use the Bokeh or Plotly backend for plotting, you can manually zoom into the dense area.)

3. String Manipulation

3.7 Plotting Trends

Let's plot the impressions over time for both tweets and replies. The data does not indicate whether the tweet is a reply or not. We can use the `.str.starts_with` method to check if the tweet text starts with @ to determine if the tweet is a reply.

```
>>> print(twit
... .select('time', 'engagements', tweet_col,
...         reply=tweet_col.str.starts_with('@'))
... )
```

shape: (5_791, 4)

time	engagements	Tweet text	reply
str	f64	str	bool
2020-01-02 03:44:00+00:00	7.00	Sounds like a great topic! https.	false
2020-01-02 03:52:00+00:00	3.00	@FogleBird Looks like SLC. I can.	true
2020-01-02 05:56:00+00:00	6.00	@afilina That's really amount an.	true
.	.	.	.
2021-12-28 18:08:00+00:00	12.00	@posco Visiting Hawaii for the h.	true
2021-12-30 07:23:00+00:00	8.00	@johndsaunders My son just built.	true
2021-12-31 21:11:00+00:00	63.00	@tunguz Xgboost	true

Let's convert the `time` column to a datetime type and plot the impressions over time for both tweets and replies. If we pass the correct format string, the `.str.to_datetime` method will convert the string to a date.

Note

This is another area where AI seems to be sufficiently competent. Unless you have all of the strftime format codes memorized, you can pass in a sample date string and have the AI generate the format code for you.

```
>>> print(twit
... .select(pl.col('time').str.to_datetime('%Y-%m-%d %H:%M:%S%z'),
...         'engagements',
...         reply=tweet_col.str.starts_with('@'))
... .pivot(index='time', columns='reply', values='engagements',
...        aggregate_function='sum')
... .rename({'false': 'original', 'true': 'reply'})
... )
```

time datetime[µs, UTC]	original f64	reply f64
2020-01-02 03:44:00 UTC	7.00	null
2020-01-02 03:52:00 UTC	null	3.00
2020-01-02 05:56:00 UTC	null	6.00
.	.	.
2021-12-28 18:08:00 UTC	null	12.00
2021-12-30 07:23:00 UTC	null	8.00
2021-12-31 21:11:00 UTC	null	63.00

We now have a column for the x-axis (*time*) and two columns for lines on the y-axis (*original* and *reply*). We can use the `plot` method to plot the data.

Note

Polars uses the hvPlot library for plotting. You can install that library with pip, or install Polars using this command to enable the optional plotting dependencies:

```
$ pip install 'polars[plot]'
```

```
>>> (twit
... .select(pl.col('time').str.to_datetime('%Y-%m-%d %H:%M:%S%z'),
...         'engagements',
...         reply=tweet_col.str.starts_with('@'))
... .pivot(index='time', columns='reply', values='engagements',
...        aggregate_function='sum')
... .rename({'false': 'original', 'true': 'reply'})
... .plot(kind='line', x='time', y=['original', 'reply'])
... )

:NdOverlay [Variable]
:Curve [time] (value)
```

I'm not super happy with this plot. Some outliers are making it hard to see the trend. We could try plotting the log of the engagement values. However, I will aggregate the data to the weekly level and plot the weekly average engagements. (The time series chapter will cover this in more detail.)

```
>>> (twit
... .select(pl.col('time').str.to_datetime('%Y-%m-%d %H:%M:%S%z'),
...         'engagements',
...         reply=tweet_col.str.starts_with('@'))
... .pivot(index='time', columns='reply', values='engagements',
...        aggregate_function='sum')
```

3. String Manipulation

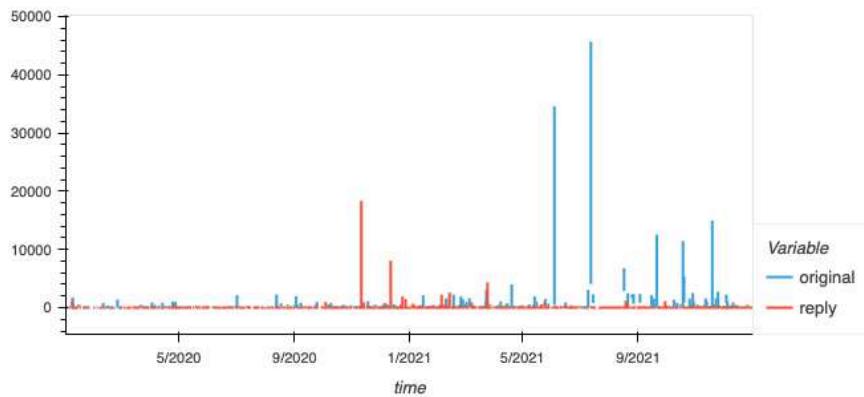


Figure 3.2: Visualization of engagements for original tweets and replies

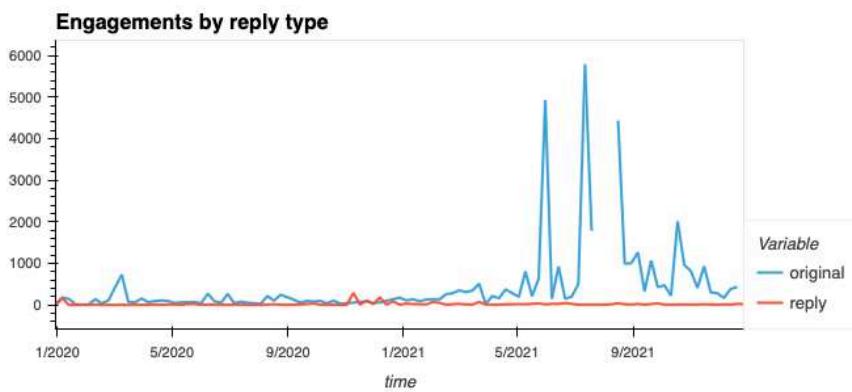


Figure 3.3: Visualization of engagements for original tweets and replies by weekly average.

```

... .set_sorted('time')
... .group_by_dynamic('time', every='1w')
... .agg(pl.col('true', 'false').mean())
... .rename({'false': 'original', 'true': 'reply'})
... .plot(kind='line', x='time', y=['original', 'reply'],
...       title='Engagements by reply type')
...
:NdOverlay [Variable]
:Curve [time] (value)

```

This shows that the number of engagements for original tweets is much higher than for replies, and it seems to be trending upward.

Polars made it relatively pain-free to analyze my tweet data and ask questions about it. Because we can process strings and numbers, we can quickly derive insight. The fact that Polars also has plotting is a nice bonus that lets us validate some of our conclusions.

3.8 Chapter Functions and Methods

Table 3.1: Chapter Functions and Methods

Function/Method	Description
.str.concat(delimiter=None, *, ignore_null=True)	Concatenate strings.
.str.contains(pattern, *, literal=False, strict=True)	Check if a string contains a pattern.
.str.contains_any(patterns, *, ascii_case_insensitive=False)	Check if a string contains any of the patterns.
.str.count_matches(pattern, *, literal=False)	Count the number of matches of a pattern.
.str.decode(encoding, *, strict=True)	Decode a string from 'hex' or 'base64'
.str.encode(encoding, *, strict=True)	Encode a string to 'hex' or 'base64'
.str.ends_with(suffix)	Check if strings end with a suffix.
.str.explode()	Explode lists of strings into separate rows.
.str.extract(pattern, group_index=1)	Extract groups matching a regular expression.
.str.extract_all(pattern)	Extract all occurrences of a pattern.
.str.extract_groups(pattern)	Extract groups from strings based on a regular expression.
.str.json_decode(dtype=None, infer_schema_length=100)	Decode JSON strings.

3. String Manipulation

Function/Method	Description
<code>.str.json_path_match(json_path)</code>	Match JSON paths and return the matched elements.
<code>.str.len_bytes()</code>	Get the length of strings in bytes.
<code>.str.len_chars()</code>	Get the number of characters in strings.
<code>.str.ljust(width, fill_char=' ')</code>	Left-justify strings in a field of given width.
<code>.str.pad_end(length, fill_char=' ')</code>	Pad strings on the right to a certain length.
<code>.str.pad_start(length, fill_char=' ')</code>	Pad strings on the left to a certain length.
<code>.str.replace(pattern, value, *, literal=False, n=1)</code>	Replace a pattern with a value.
<code>.str.replace_all(pattern, value, *, literal=False)</code>	Replace all occurrences of a pattern with a value.
<code>.str.replace_many(patterns, replace_with, *, ascii_case_insensitive=False)</code>	Replace multiple patterns with a value.
<code>.str.reverse()</code>	Reverse each string.
<code>.str.slice(offset, length=None)</code>	Slice substrings from strings.
<code>.str.split(pattern, *, inclusive=False)</code>	Split strings based on a pattern.
<code>.str.split_exact(by, n, *, inclusive=False)</code>	Split strings into exactly n substrings based on a pattern.
<code>.str.splitn(by, n)</code>	Split strings into at most n substrings based on a pattern.
<code>.str.starts_with(prefix)</code>	Check if strings start with a prefix.
<code>.str.strip_chars(characters=None)</code>	Remove specified characters from strings.
<code>.str.strip_chars_end(characters)</code>	Remove specified characters from the end of strings.
<code>.str.strip_chars_start(characters)</code>	Remove specified characters from the start of strings.
<code>.str.strip_prefix(prefix)</code>	Remove a prefix from strings, if present.
<code>.str.strip_suffix(suffix)</code>	Remove a suffix from strings, if present.
<code>.str.strptime(dtype, format, *, strict=True, exact=True, cache=True, ambiguous='raise')</code>	Parse strings to datetime objects.
<code>.str.to_date(format, *, strict=True, exact=True, cache=True)</code>	Convert strings to date objects.

Function/Method	Description
<code>.str.to_datetime(format, *, time_unit=None, time_zone=None, strict=True, exact=True, cache=True, ambiguous='raise')</code>	Convert strings to datetime objects.
<code>.str.to_decimal(inference_length=100)</code>	Convert strings to decimal objects.
<code>.str.to_integer(*, base=10, strict=True)</code>	Convert strings to integer objects.
<code>.str.to_lowercase()</code>	Convert strings to lowercase.
<code>.str.to_time(format, *, strict=True, cache=True)</code>	Convert strings to time objects.
<code>.str.to_titlecase()</code>	Convert strings to title case.
<code>.str.to_uppercase()</code>	Convert strings to uppercase.
<code>.str.zfill(length)</code>	Pad strings on the left with zeros to fill a width.

3.9 Summary

This chapter explored many string manipulation methods. We used these methods to validate the data, extract the username from a URL, check for emojis, and plot trends. We also used the `.str` accessor to access list methods. We used the `.list` accessor to convert a list into a struct and then used the `.unnest` method to convert the struct into columns. Polars is equipped with many string manipulation methods that are similar to Python’s string manipulation methods and Pandas’ string manipulation methods.

3.10 Exercises

1. Extract the domain from the `Tweet permalink` column.
2. Extract the tweet ID from the `Tweet permalink` column.
3. Uppercase the `Tweet text` column.
4. Lowercase the `Tweet text` column.
5. Replace the word “the” with “THE” in the `Tweet text` column.
6. Plot the engagement rate for tweets with and without emojis over time.

Chapter 4

Aggregation with Polars

4.1 Introduction

One of the most common tasks in data analysis is to perform aggregation and grouping operations on datasets. I like to say that aggregation is calculating the summarizations your boss wants to see. When the boss asks, “How are things going?” They don’t want to see the raw data; they want a summary.

This is especially true for large datasets, where summarizing and aggregating data is often necessary to gain insights and make sense of the information. In the previous chapter, we saw how to pivot, and in this chapter, we will learn how to perform aggregation and grouping operations in Polars.

4.2 Loading the Data

We will again use the Fuel Economy dataset we used in the previous chapter. Refer to previous chapters for more details on the dataset.

4.3 Aggregations

Aggregation is the process of summarizing data. From a sequence of values, we reduce them to a single value that describes the sequence. For example, the mean, median, and standard deviation are all aggregations. The input is a sequence, and the output is a scalar value. But there are many more like min, max, sum, count, any, all, etc.

Most operations on a column in Polars should be performed using the expression syntax. Many of the aggregations are available as expressions.

Let’s create a column expression and inspect the aggregation methods available.

```
>>> col = pl.col('make')
>>> print(sorted(att for att in dir(col) if not att.startswith('_')))
```

['abs', 'add', 'agg_groups', 'alias', 'all', 'and_', 'any', 'append',

4. Aggregation with Polars

```
'apply', 'approx_n_unique', 'arccos', 'arccosh', 'arcsin', 'arcsinh',
'arctan', 'arctanh', 'arg_max', 'arg_min', 'arg_sort', 'arg_true',
'arg_unique', 'arr', 'backward_fill', 'bin', 'bottom_k', 'cast', 'cat',
'cbrt', 'ceil', 'clip', 'clip_max', 'clip_min', 'cos', 'cosh', 'cot',
'count', 'cum_count', 'cum_max', 'cum_min', 'cum_prod', 'cum_sum',
'cumcount', 'cummax', 'cummin', 'cumprod', 'cumsum', 'cumulative_eval',
'cut', 'degrees', 'deserialize', 'diff', 'dot', 'drop_nans',
'drop_nulls', 'dt', 'entropy', 'eq', 'eq_missing', 'ewm_mean',
'ewm_std', 'ewm_var', 'exclude', 'exp', 'explode', 'extend_constant',
'fill_nan', 'fill_null', 'filter', 'first', 'flatten', 'floor',
'floordiv', 'forward_fill', 'from_json', 'gather', 'gather_every',
'ge', 'get', 'gt', 'hash', 'head', 'hist', 'implode', 'inspect',
'interpolate', 'is_between', 'is_duplicated', 'is_finite', 'is_first',
'is_first_distinct', 'is_in', 'is_infinite', 'is_last',
'is_last_distinct', 'is_nan', 'is_not', 'is_not_nan', 'is_not_null',
'is_null', 'is_unique', 'keep_name', 'kurtosis', 'last', 'le', 'len',
'limit', 'list', 'log', 'log10', 'log1p', 'lower_bound', 'lt', 'map',
'map_alias', 'map_batches', 'map_dict', 'map_elements', 'max', 'mean',
'median', 'meta', 'min', 'mod', 'mode', 'mul', 'n_unique', 'name',
'nan_max', 'nan_min', 'ne', 'ne_missing', 'neg', 'not_', 'null_count',
'or_', 'over', 'pct_change', 'peak_max', 'peak_min', 'pipe', 'pow',
'prefix', 'product', 'qcut', 'quantile', 'radians', 'rank', 'rechunk',
'register_plugin', 'reinterpret', 'repeat_by', 'replace', 'reshape',
'reverse', 'rle', 'rle_id', 'rolling', 'rolling_apply', 'rolling_map',
'rolling_max', 'rolling_mean', 'rolling_median', 'rolling_min',
'rolling_quantile', 'rolling_skew', 'rolling_std', 'rolling_sum',
'rolling_var', 'round', 'round_sig_figs', 'sample', 'search_sorted',
'set_sorted', 'shift', 'shift_and_fill', 'shrink_dtype', 'shuffle',
'sign', 'sin', 'sinh', 'skew', 'slice', 'sort', 'sort_by', 'sqrt',
'std', 'str', 'struct', 'sub', 'suffix', 'sum', 'tail', 'take',
'take_every', 'tan', 'tanh', 'to_physical', 'top_k', 'truediv',
'unique', 'unique_counts', 'upper_bound', 'value_counts', 'var',
'where', 'xor']
```

Of these attributes, many of them are aggregation methods, including `.all`, `.any`, `.len`, `.max`, `.mean`, `.median`, `.min`, `.n_unique`, `.product`, `.quantile`, `.std`, `.sum`, and `.var`.

Let's try a few of these methods on the `comb08` column.

```
>>> city = pl.col('city08')
>>> print(autos
...     .select(mean_city=city.mean(),
...             std_city=city.std(),
...             var_city=city.var(),
...             q99_city=city.quantile(.99)
...     )
```

```
... )
shape: (1, 4)
| mean_city    std_city    var_city    q99_city |
| f64          f64         f64        f64      |
| 18.37       7.91       62.50      40.00   |
```

If we add a new aggregated column to an existing dataframe, the new column will be appended to the right of the existing columns. The aggregated value will be repeated for each row in the dataframe.

```
>>> print(autos
... .with_columns(mean_city=city.mean())
... )
shape: (41_144, 16)
| year      make           model      .  is_automati num_gears  mean_city |
| i16       cat            cat        .  tic          i8        f64      |
|                           cat          bool      |
| 1985     Alfa Romeo    Spider     .  false        5       18.37
|                   Veloce
|                   2000
| 1985     Ferrari      Testaross  .  false        5       18.37
|                   a
| 1985     Dodge        Charger    .  false        5       18.37
|                   .
| 1993     Subaru       Legacy     .  true         4       18.37
|                   AWD
| 1993     Subaru       Legacy     .  false        5       18.37
|                   AWD
| 1993     Subaru       Legacy     .  true         4       18.37
|                   AWD Turbo |
```

4.4 Lists

One feature of Polars is the ability to have typed lists in a column. Pandas can do this, but it stores the lists as Python lists. Polars stores the lists as typed arrays, which allows Polars to perform operations on the lists in a vectorized manner.

Imagine if you had columns of student test scores and wanted to drop the two lowest scores for each student. There isn't an easy way to do this with columns.

However, we can combine the test columns into a single column of lists. Then, we can use list operations to sort the lists and remove the two lowest values. Finally, we can sum up the list. These list operations are methods found off of the column's `.list` attribute.

4. Aggregation with Polars

```
>>> tests = pl.DataFrame({'name':['Tom', 'Sally', 'Jose'],
...                         'test1':[99, 98, 95],
...                         'test2':[92, None, 99],
...                         'test3':[91, 93, 95],
...                         'test4':[94, 92, 99]})

>>> print(tests)
shape: (3, 5)

```

name	test1	test2	test3	test4
str	i64	i64	i64	i64
Tom	99	92	91	94
Sally	98	null	93	92
Jose	95	99	95	99

Below, I create a dataframe with a column for each step. First, we combine the test scores into a list with the `pl.concat_list` function. We will name that column *scores*. Then, we can use the `.list` attribute, which has a `.sort` method, to sort values in each cell in the column. Then, we can use the `.slice` method to remove the two lowest values. Finally, we can sum up the list to get the total score for each student. Because each of these columns depends on the previous column, we need to put them in their own `.with_columns` call; otherwise, Polars would try to evaluate the columns in parallel.

```
>>> import polars.selectors as cs
>>> print(tests
...     .select(scores=pl.concat_list(cs.matches(r'test\d+')))
...     .with_columns(sorted_scores=pl.col('scores').list.sort())
...     .with_columns(slice_scores=pl.col('sorted_scores').list.slice(2,4))
...     .with_columns(sum_scores=pl.col('slice_scores').list.sum())
... )
shape: (3, 4)

```

scores	sorted_scores	slice_scores	sum_scores
list[i64]	list[i64]	list[i64]	i64
[99, 92, . 94]	[91, 92, . 99]	[94, 99]	193
[98, null, . 92]	[null, 92, . 98]	[93, 98]	191
[95, 99, . 99]	[95, 95, . 99]	[99, 99]	198

Note

Polars has many ways to select columns. In the previous example, I used a regular expression. I can also use the `.exclude` method:

```
>>> print(tests.select(pl.all().exclude('name')))
shape: (3, 4)
```

test1	test2	test3	test4
i64	i64	i64	i64
99	92	91	94
98	null	93	92
95	99	95	99

There are also set operations available. We could use *set difference* to start with all of the columns and remove the *name* column. However, the expression syntax doesn't support this directly:

```
>>> (tests.select(pl.all() - pl.col('name')))

Traceback (most recent call last)
...
ComputeError: arithmetic on string and numeric not allowed, try an
explicit cast first
```

We need to use `cs.all` (from `import polars.selectors as cs`) instead of `pl.all` to use set operations:

```
>>> print(tests.select(cs.all() - pl.col('name')))

shape: (3, 4)
+-----+
| test1  test2  test3  test4 |
| i64    i64    i64    i64   |
+-----+
| 99     92     91     94   |
| 98     null   93     92   |
| 95     99     95     99   |
+-----+
```

Finally, you could use `cs.starts_with` to filter the columns:

```
>>> print(tests.select(cs.starts_with('test')))

shape: (3, 4)
+-----+
| test1  test2  test3  test4 |
| i64    i64    i64    i64   |
+-----+
| 99     92     91     94   |
| 98     null   93     92   |
| 95     99     95     99   |
+-----+
```

If we want to calculate horizontal aggregations, we can also use the list functionality. Let's calculate the maximum test score for each student.

```
>>> print(tests
... .select(scores=pl.concat_list(cs.matches(r'test\d+')))
... .with_columns(max=pl.col('scores').list.max())
... )
```

4. Aggregation with Polars

shape: (3, 2)

scores	max
list[i64]	i64
[99, 92, . 94]	99
[98, null, . 92]	98
[95, 99, . 99]	99

We will see many examples of using both list and struct column types in this book.

4.5 GroupBy Operations

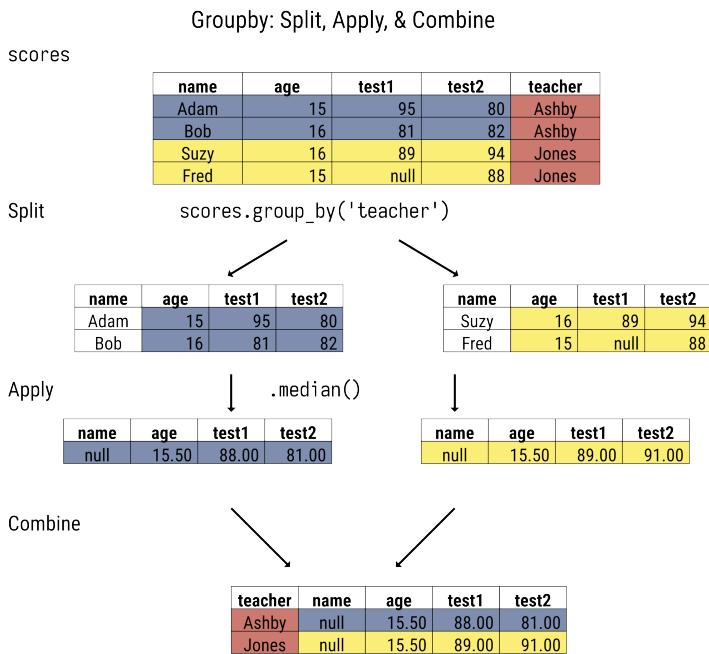


Figure 4.1: A groupby operation splits the data into groups based on some criteria. You can apply a function (or functions) to each group independently. Then, the results are combined. The column we are grouping by is placed in the first column of the result.

Grouping is a general term for a class of operations that involves splitting up a dataset based on some criteria, applying a function to each group independently, and then combining the results into a data structure. It is the

underpinning of a pivot table. You will want to master the groupby operation because it is a powerful tool for data analysis.

Suppose your boss asked you to “summarize mileage data by make.” When you hear the term “by make,” you should immediately think of a groupby operation.

Let’s do a basic groupby operation on the *make* column. We will group the data by the *make* column and then calculate each group’s average *city08* value. This will involve two steps: the first step is to group the data (using the `.group_by` method), and the second step is to aggregate the data (using the `.agg` method on the result of the groupby). This is a case where even though we are working in eager mode, the first step doesn’t materialize in the final result. It doesn’t know what to do with the data until the second step, the aggregation, is performed.

```
>>> print(autos
...     .group_by('make')
... )
<polars.dataframe.groupby.GroupBy object at 0x13ba9b910>
```

Let’s use a column expression to create the aggregation. We will use the `.mean` method on the *comb08* column.

```
>>> print(autos
...     .group_by('make')
...     .agg(mean_city=pl.col('city08').mean())
... )
shape: (136, 2)
+-----+-----+
| make      | mean_city |
| cat       | f64       |
+-----+-----+
| JBA Motorcars, Inc. | 15.00   |
| Ruf Automobile GmbH | 11.00   |
| Ferrari          | 11.32   |
| .               | .        |
| McEvoy Motors    | 18.00   |
| Vector           | 7.25    |
| VPG              | 12.20   |
+-----+-----+
```

Note

You’ll note that (unlike Pandas) the *make* column is not sorted in the output. Again, this is because Polars wants to run this operation as efficiently as possible, and sorting the data would be an unnecessary step. You can sort the data using the `.sort` method.

4. Aggregation with Polars

A `maintain_order` parameter can be set to `True` when calling the `.group_by` method. This will maintain the order of the data. In this case, the output will be in the order that the `make` values were preset in the original data.

```
>>> print(autos
... .group_by('make', maintain_order=True)
... .agg(mean_city=pl.col('city08').mean())
... )
```

shape: (136, 2)

make	mean_city
cat	f64
Alfa Romeo	18.31
Ferrari	11.32
Dodge	15.46
.	.
Goldacre	16.00
Isis Imports Ltd	16.00
PAS Inc - GMC	14.00

We will use the `.sort` method to sort the data by the `make` column.

```
>>> print(autos
... .group_by('make')
... .agg(mean_city=pl.col('city08').mean())
... .sort('make')
... )
```

shape: (136, 2)

make	mean_city
cat	f64
Alfa Romeo	18.31
Ferrari	11.32
Dodge	15.46
.	.
Goldacre	16.00
Isis Imports Ltd	16.00
PAS Inc - GMC	14.00

Note that the `make` column is not alphabetically sorted. This is because the `make` column is a categorical column, and the categories are sorted by the order in which they appear in the original dataframe. We can see this by inspecting the categories of the `make` column.

We can specify the ordering when casting to `pl.Categorical`. Let's cast the values to set the ordering of the `make` column to alphabetical (Polars calls this '`lexical`').

```
>>> print(autos
... .with_columns(pl.col('make').cast(pl.Categorical('lexical')))
... .group_by('make')
... .agg(mean_city=pl.col('city08').mean())
... .sort('make')
... )
shape: (136, 2)
  make          mean_city
  cat           f64
  AM General    15.17
  ASC Incorporated 14.00
  Acura          19.23
  .
  Wallace Environmental 12.44
  Yugo            23.00
  smart            68.42
```

4.6 Multiple Aggregations

In the previous example, we aggregated the `city08` column. We can pass multiple expressions to the `.agg` method to perform multiple aggregations. Let's calculate the average and median values for both the `city08` and `highway08` columns.

```
>>> print (autos
... .with_columns(pl.col('make').cast(pl.Categorical('lexical')))
... .group_by('make')
... .agg(mean_city=pl.col('city08').mean(),
...       mean_highway=pl.col('highway08').mean(),
...       median_city=pl.col('city08').median(),
...       median_highway=pl.col('highway08').median()
...     )
... .sort('make')
... )
shape: (136, 5)
  make          mean_city  mean_highway  median_city  median_highway
  cat           f64        f64          f64          f64
  AM General    15.17      15.00        14.50        15.00
  ASC Incorporated 14.00      21.00        14.00        21.00
  Acura          19.23      26.23        19.00        27.00
  .
  Wallace Environmental 12.44      16.00        11.00        15.00
```

4. Aggregation with Polars

Multiple Aggregations with Groupby

auto

make	year	cylinders	city08
Alfa Romeo	1985	4	19
Ferrari	1985	12	9
Dodge	1985	4	23
Dodge	1985	8	10
Subaru	1993	4	17
Subaru	1993	4	19
Subaru	1993	4	20
Subaru	1993	4	18
Subaru	1993	4	18
Subaru	1993	4	16

```
(auto
    .group_by('make')
    .agg(pl.all().mean().name.suffix('_mean'),
        pl.col(['cylinders', 'city08']).min().name.suffix('_min'),
        pl.col('city08').max().name.suffix('_max'),
    )
    )  
    Using .suffix because Polars cannot repeat column names
    .sort('make')
)
```

make	year_mean	cylinders_mean	city08_mean	cylinders_min	city08_min	city08_max
AM General	1984.33	5.00	15.17	4	13	18
ASC Incorp	1987.00	6.00	14.00	6	14	14
Acura	2005.48	5.24	19.23	4	13	38
Alfa Romeo	1998.58	5.10	18.31	4	11	24
American M	1984.48	5.41	16.30	4	15	19
Volkswagen	2002.81	4.55	21.82	4	11	126
Volvo	2002.35	4.86	18.30	4	13	29
Wallace En	1991.50	7.81	12.44	4	8	17
Yugo	1988.38	4.00	23.00	4	21	24
smart	2013.95	3.00	68.42	3	31	124

Figure 4.2: A grouping with multiple aggregations. Note that I'm using `.name.suffix` to ensure that the column names are unique.

Environmental

Yugo	23.00	28.25	23.00	28.50
smart	68.42	60.97	34.00	41.00

Polars does not like having multiple columns with the same name. In the above example, I aggregated the `city08` column twice and the `highway08` column twice. Polars would happily preserve the name of the original column in the output, except that the names would be repeated. I'm using `.alias` to rename the columns to avoid this. You could also use `.name.suffix` or `.name.prefix` to ensure the column names are unique.

4.7 Grouping By Multiple Columns

We can also group by multiple columns. Let's group by the *make* and *year* columns and calculate each group's average *city08* value. I'm going to use `.cast` to convert the *make* column to an alphabetically ordered categorical column. You pass in 'lexical' to the `Categorical` constructor to specify that the categories should be ordered alphabetically.

```
>>> print(autos
... .with_columns(pl.col('make').cast(pl.Categorical('lexical')))
... .group_by(['make', 'year'])
... .agg(mean_city08=pl.col('city08').mean())
... .sort(['make', 'year'])
... )
shape: (1_749, 3)
   make          year  mean_city08
   cat           i16    f64
   -----
AM General      1984     15.50
AM General      1985     14.50
ASC Incorporated 1987     14.00
.
smart           2017     60.67
smart           2018    118.00
smart           2019    118.00
```

4.8 Pivoting with Multiple Aggregations

The `.pivot` method can only be used with a single aggregation. If we want to perform multiple aggregations, we can use the `.group_by` method and then use the `.pivot` method on the result of the `groupby`. Let's calculate the minimum and maximum *city08* values by year for the top three makes.

First, we will calculate the top three makes (the value counts in Pandas speak). We can do this by grouping by the *make* column and aggregating with the `.len` method. Remember, we will need to sort the data by the aggregated column before taking the top three rows.

```
>>> top3 = (autos
... .group_by('make')
... .len()
... .sort(pl.col('len'), descending=True)
... .head(3)
... )

>>> print(top3)
shape: (3, 2)
```

4. Aggregation with Polars

make	len
cat	u32
Chevrolet	4003
Ford	3371
Dodge	2583

Now we can perform the groupby operation.

```
>>> print(autos
... .filter(pl.col('make').is_in(top3['make']))
... .group_by(['year', 'make'])
... .agg(min_city08=pl.col('city08').min(),
...       max_city08=pl.col('city08').max())
... .sort(['year', 'make'])
... )
```

shape: (110, 4)

year	make	min_city08	max_city08
i16	cat	u8	u8
1984	Dodge	8	31
1984	Chevrolet	10	33
1984	Ford	9	35
.	.	.	.
2019	Ford	11	43
2020	Chevrolet	22	30
2020	Ford	20	24

If we want the *make* column to be pulled into the columns, we can use the `.pivot` method.

```
>>> print(autos
... .filter(pl.col('make').is_in(top3['make']))
... .group_by(['year', 'make'])
... .agg(min_city08=pl.col('city08').min(),
...       max_city08=pl.col('city08').max())
... .sort(['year', 'make'])
... .pivot(index='year', columns='make',
...       values=['min_city08', 'max_city08'])
... )
```

shape: (37, 7)

year	min_city08	min_city0	.	max_city0	max_city0	max_city0
i16	_make_Dodg	8_make_Ch		8_make_Do	8_make_Ch	8_make_Fo
	e	evrolet		dge	evrolet	rd
u8	u8	u8		u8	u8	u8

4.8. Pivoting with Multiple Aggregations

1984	8	10	.	31	33	35
1985	8	9	.	31	39	36
1986	9	9	.	31	44	34
.
2018	13	13	.	19	128	118
2019	13	12	.	19	128	43
2020	null	22	.	null	30	24

A recent addition to Polars is the `.plot` attribute. Previously I would create my plots using Pandas.

Line plots are straightforward in Pandas. It will plot the index on the x-axis and each column as a line in the chart. The legend will be automatically generated from the column names.

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(figsize=(10, 6))
>>> (autos
...     .filter(pl.col('make').is_in(top3['make']))
...     .group_by(['year', 'make'])
...     .agg(min_city08=pl.col('city08').min(),
...          max_city08=pl.col('city08').max())
...     .sort(['year', 'make'])
...     .pivot(index='year', columns='make',
...            values=['min_city08', 'max_city08'])
...     .select(['year', 'min_city08_make_Dodge', 'max_city08_make_Dodge',
...             'min_city08_make_Ford', 'max_city08_make_Ford',
...             'max_city08_make_Chevrolet', 'min_city08_make_Chevrolet'])
...     .to_pandas()
...     .set_index('year')
...     .plot(ax=ax, color=['#55c667', '#2f8738', '#2c4279', '#141f38',
...                         '#d4e116', '#afba12'])
... )
```

<Figure size 1000x600 with 1 Axes>

Now, Polars will use the `hvplot` library for plotting. Make sure you have installed it. Let's recreate this plot using that library.

```
>>> res = (autos
...     .filter(pl.col('make').is_in(top3['make']))
...     .group_by(['year', 'make'])
...     .agg(min_city08=pl.col('city08').min(),
...          max_city08=pl.col('city08').max())
...     .sort(['year', 'make'])
...     .pivot(index='year', columns='make',
```

4. Aggregation with Polars

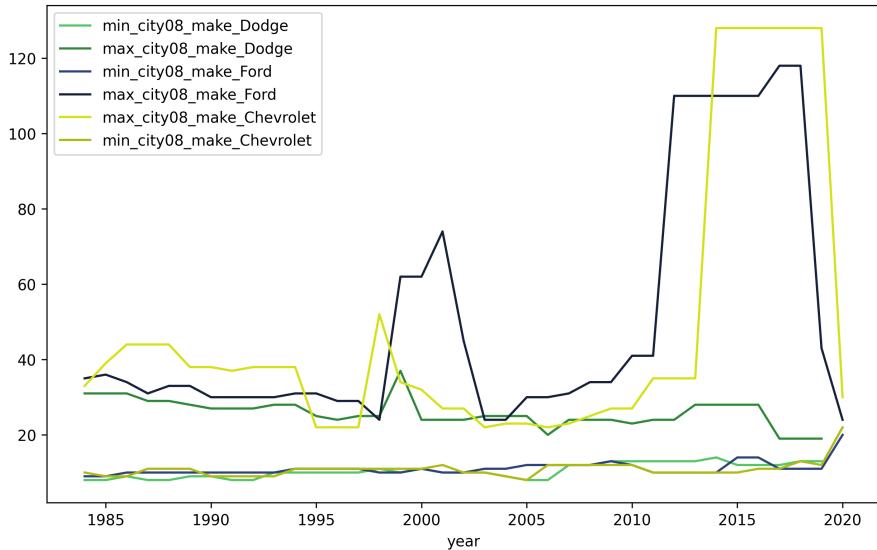


Figure 4.3: Plot of minimum and maximum mileage over the years.

```
...     values=['min_city08', 'max_city08'])
... .select(['year', 'min_city08_make_Dodge', 'max_city08_make_Dodge',
...         'min_city08_make_Ford', 'max_city08_make_Ford',
...         'max_city08_make_Chevrolet', 'min_city08_make_Chevrolet'])
... .plot(x='year', color=['#55c667', '#2f8738', '#2c4279', '#141f38',
...                     '#d4e116', '#afba12'])
... )
```

This code is very similar to the Pandas plotting code. In the Pandas code, we stuck the date into the index. Because Polars doesn't have indexes, we specify `x='year'` to plot the `year` column in the x-axis. The rest of the code is the same.

One cool thing about the hvPlot library is that it abstracts Matplotlib, Plotly, and Bokeh. It uses the Bokeh library by default. This plotting library uses JavaScript to render the plot so it is interactive. However, if you want to save a static PNG image of the plot programmatically, you will need to install a scriptable browser environment like PhantomJS or Selenium. (You can manually click the download button without such an install.)

```
import hvplot
# saving requires selenium or phantomjs
hvplot.save(res, 'img/hvplot1.png')
```

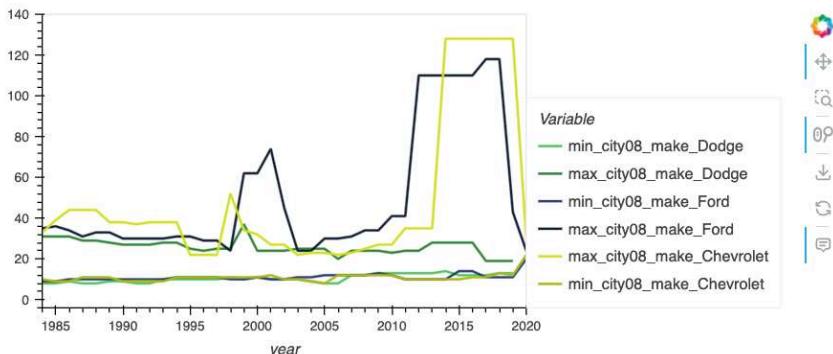


Figure 4.4: Plot created using Polars.

4.9 Aggregating to the Original Rows

In Pandas, a `.transform` method allows you to aggregate the data and then return the aggregated values to the original rows. Polars has no `.transform` method, but you can achieve the same result using window functions.

Let's calculate the *make* average *city08* value and return it to the original rows. The construction for this is as follows. Use `pl.col` to refer to the column you want to aggregate. Then do the aggregation (in this case `.mean()`). Follow that with `.over` and pass in the column you want to group by ('*make*' in this case).

```
>>> print(autos
...     .with_columns(mean_city08=pl.col('city08')
...                   .mean()
...                   .over('make'))
... )
```

year	make	model	is_automatic	num_gears	mean_city08
1985	Alfa Romeo	Spider	false	5	18.31
		Veloce			
		2000			
1985	Ferrari	Testarossa	false	5	11.32
		a			
1985	Dodge	Charger	false	5	15.46
.
1993	Subaru	Legacy AWD	true	4	20.32

4. Aggregation with Polars

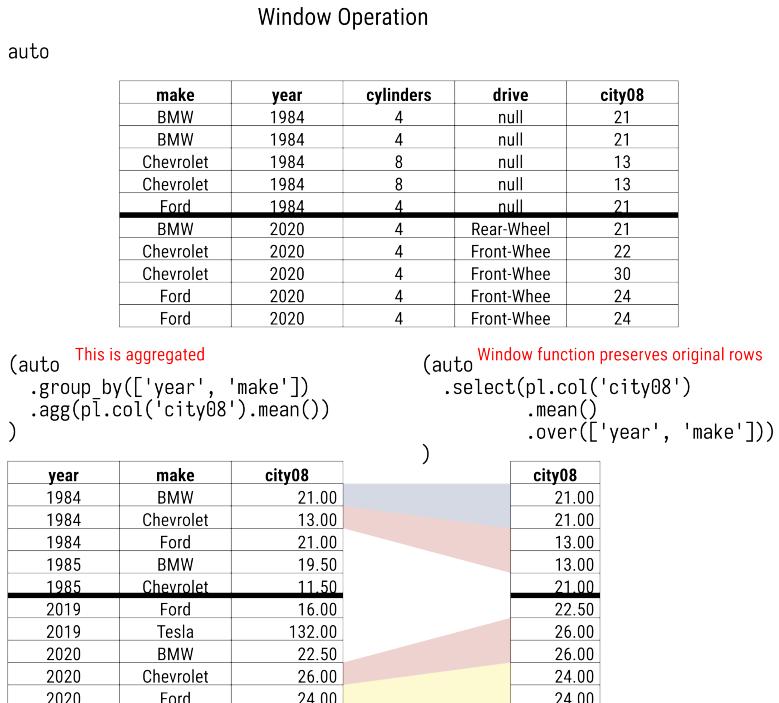


Figure 4.5: The window functions allow us to perform aggregations on groups but return the resulting aggregation in terms of the original rows.

1993	Subaru	Legacy	.	false	5	20.32
1993	Subaru	Legacy	.	true	4	20.32

Let's quickly validate the results for *Alfa Romeo*, *Ferrari*, and *Dodge*.

```
>>> print(autos
...  .group_by('make')
...  .agg(mean_city08=pl.col('city08').mean())
...  .filter(pl.col('make').cast(pl.String)
...        .is_in(['Alfa Romeo', 'Ferrari', 'Dodge']))
... )
shape: (3, 2)
| make      | mean_city08 |
| cat       | f64        |
```

Dodge	15.46
Ferrari	11.32
Alfa Romeo	18.31

A cool realization is that we can use window functions to do multiple groupings and aggregations in a single operation.

In the next chapter, I will demonstrate filtering with window functions.

4.10 Grouping to Lists

I'm going to show an example of the power of Polars. Recreating this with Pandas or SQL would be a pain. You can write powerful code in short order once you get the hang of expressions, window functions, and lists.

Let's assume I wanted the count of unique values for every column by make.

First, you can aggregate to a column by passing `pl.all` to the `.agg` context:

```
>>> print(autos
...     .group_by('make')
...     .agg(pl.all())
... )
shape: (136, 15)
   make      year      model      .  created0  is_autom  num_gear
   cat      list[i16]  list[cat]
                  ]          list[dat
                                list[boo
                                  etime[μs
                                    l]
                                    ]]
                                    ]]

Alfa Romeo  [1985, 1994, 1993]  ["Spider", "Veloce", "2000", "Spider", ..]  [2013-01-01 00:00:00, 2013-01-01.., 2013-01-01.., 2013-01-01.., ..]  [false, false, .., false, ..]  [5, 5, .., 5]
Ferrari    [1985, 1994, 1993]  ["Testarossa", "Ferrari 348", "Ferrari 348 TB/T." ]  [2013-01-01 00:00:00, 2013-01-01.., 2013-01-01.., 2013-01-01.., ..]  [false, false, .., true]  [5, 5, .., 5]
Dodge      [1985, 1985, 1985]  ["Charge r", "B15 0/B250 Wagon 2WD."]  [2013-01-01 00:00:00, 2013-01-01.., 2013-01-01.., 2013-01-01.., ..]  [false, true, .., true]  [5, 3, .., 3]
Goldacre   [1991]                ["Goldac .."]  [2013-01-01 00:00:00, ..]  [false]  [5]
```

4. Aggregation with Polars

			re Limit	-01 00:0		
			ed"]	0:00]		
Isis	[1992]	["Morgan	.	[2013-01	[false]	[5]
Imports		Plus 8"]		-01 00:0		
Ltd				0:00]		
PAS Inc -	[1992,	["Pas-Sy	.	[2013-01	[true,	[4, 4]
GMC	1992]	clone",		-01 00:0	true]	
		"Pas-typ		0:00,		
		hoon"]		2013-01-		
				01.		

I will limit this to *make*, *model*, and *cylinders** so the data is a little easier to see. To get the unique values for each list, we can chain up expressions to remove duplicates, drop nulls, get the length, and add a suffix:

```
>>> print(autos
... .select('make', 'model', 'cylinders')
... .group_by('make')
... .agg(pl.all().unique().drop_nulls().len().name.suffix('_len'))
... )
shape: (136, 3)
```

make	model_len	cylinders_len
cat	u32	u32
Alfa Romeo	12	3
Ferrari	64	2
Dodge	116	4
.	.	.
Goldacre	1	1
Isis Imports Ltd	1	1
PAS Inc - GMC	2	1

These are the aggregated values. There are only 136 makes, so this dataframe has 136 rows. What if we wanted to have these *_len* columns added to the original dataframe? That's where window functions come in. We need to move the `.group_by('make')` to an `.over('make')` to our previous code. We also need to use `.with_columns` instead of `.select`.

```
>>> print(autos
... .with_columns(pl.col('model', 'cylinders')
...     .unique().drop_nulls().len()
...     .over('make')
...     .name.suffix('_len'))
... )
shape: (41_144, 17)
| year   make      model      .  num_gears  model_len  cylinders |
```

4.11. Chapter Methods and Functions

i16	cat	cat	i8	u32	_len	u32
1985	Alfa Romeo	Spider Veloce 2000	.	5	12	3
1985	Ferrari	Testarossa	.	5	64	2
1985	Dodge	Charger	.	5	116	4
.
1993	Subaru	Legacy AWD	.	4	71	3
1993	Subaru	Legacy AWD	.	5	71	3
1993	Subaru	Legacy AWD Turbo	.	4	71	3

To get the count of every column, change the selector to use all of the columns and remove just the *make* column:

```
>>> print(autos
... .select((cs.all() - pl.col('make'))
...         .unique().drop_nulls().len()
...         .over('make')
...         .name.suffix('_len'))
... )
```

year_len	model_len	displ_len	.	created0_n_len	is_automatc_len	num_gear_s_len
u32	u32	u32	.	u32	u32	u32
18	12	6	.	11	2	5
35	64	16	.	21	2	4
36	116	30	.	38	2	6
.
37	71	11	.	39	2	6
37	71	11	.	39	2	6
37	71	11	.	39	2	6

I'm pretty stoked about this because these are operations that Polars makes painless. I'm not aware of other tools that can do this with such ease.

4.11 Chapter Methods and Functions

4. Aggregation with Polars

Table 4.1: Functions and Methods for chapter

Method	Description
<code>col.all(ignore_nulls=False)</code>	Returns True if all elements are True.
<code>col.any(ignore_nulls=False)</code>	Returns True if any element is True.
<code>col.argmax()</code>	Returns the index of the maximum value.
<code>col.argmin()</code>	Returns the index of the minimum value.
<code>col.implode()</code>	Combines values into a list.
<code>col.len()</code>	Counts the number of non-null values.
<code>col.max()</code>	Maximum value in a column.
<code>col.mean()</code>	Mean of non-null values in a column.
<code>col.median()</code>	Median of non-null values in a column.
<code>col.min()</code>	Minimum value in a column.
<code>col.n_unique()</code>	Number of unique values.
<code>col.nan_max()</code>	Maximum value in a column, ignoring NULL values.
<code>col.nan_min()</code>	Minimum value in a column, ignoring NULL values.
<code>col.product()</code>	Product of non-null values in a column.
<code>col.quantile(quantile, interpolation='nearest')</code>	Quantile of non-null values in a column. interpolation can be one of nearest, lower, higher, midpoint, linear.
<code>col.std(ddof=1)</code>	Standard deviation of a column. ddof is the delta degrees of freedom.
<code>col.sum()</code>	Sum of non-null values in a column.
<code>col.var()</code>	Variance of a column.
<code>cs.matches(pattern)</code>	Checks if the values in a column match a specified pattern.
<code>pl.concat_list(exprs, *more_exprs)</code>	Concatenates lists within a column. See also <code>pl.struct</code>
<code>col.list.sort(*, descending=False)</code>	Sorts a list within a column.
<code>col.list.slice(offset, length)</code>	Slices each list within a column.
<code>col.list.sum()</code>	Calculates the sum of values within each list in a column.
<code>df.sort(by, *more_by, descending=False, nulls_last=False)</code>	Sort this DataFrame by the specified columns. descending is a sequence of booleans for each column.
<code>df.groupby(by, *more_by, maintain_order=False)</code>	Group this DataFrame by the specified columns.
<code>grouped.agg(*aggs, **named_aggs)</code>	Aggregate the grouped DataFrame. The aggs parameter(s) is an expression(s). The named_aggs parameters are key-value pairs of column names and expressions.

Method	Description
<code>df.pivot(values=None, index=None, columns=None, aggregate_function=None, maintain_order=True, sort_columns=False, separator='_')</code>	Pivot the DataFrame from long to wide format. Place values from <code>columns</code> into the header. The <code>index</code> can specify values to group by. The <code>values</code> parameter specifies the values to aggregate. You can omit the <code>aggregate_function</code> parameter if there are no duplicate values. If there are duplicate values, you must specify an aggregate function. The <code>maintain_order</code> parameter specifies if the order of the columns should be maintained. The <code>sort_columns</code> parameter specifies if the columns should be sorted. The <code>separator</code> parameter specifies the separator between the column names.
<code>df.filter(*predicates, **constraints)</code> <code>col.cat.set_ordering(ordering='physical')</code>	Filter by predicates. The constraints are keyword arguments that filter for equality. Sets the ordering of categories within a categorical column. <code>ordering</code> defaults to the order found, can also be lexical.
<code>col.over(expr, *more_exprs, mapping_strategy ='group_to_rows')</code>	Create a window expression. Group over the specified expressions. The <code>mapping_strategy</code> parameter can be ' <code>group_to_rows</code> ', ' <code>join</code> ', or ' <code>explode</code> '. Generally, an aggregation function is used before the <code>.over</code> method.
<code>col.is_in(other)</code>	Checks if each element in a column is in a given set.
<code>col.str.extract(pattern, group_index=1)</code>	Extracts substrings from a string column based on a regex pattern.

4.12 Summary

We've explored data aggregation with Polars, uncovering its robust capabilities in processing and summarizing large datasets. From basic aggregations to advanced groupby operations, Polars provides a powerful toolkit for summarizing data. Understanding these concepts enriches our analytical skills and empowers us to handle real-world data challenges with greater proficiency.

4.13 Exercises

1. Compute the mean of the `city08` column using Polars.
2. Calculate the standard deviation of the `highway08` column.
3. Find the maximum value in the `barrels08` column.

4. Aggregation with Polars

4. Determine the number of unique values in the `make` column.
5. Calculate the median of `city08` for cars with automatic transmission.
6. Find the minimum and maximum `city08` values for each `fuelType`.
7. Compute the sum of `barrels08` for each `VClass`.
8. Group the dataset by `make` and calculate the average `city08`.
9. Group by year and find the median `highway08`.
10. For each `fuelType`, compute the count of vehicles.
11. For each `make`, find the average and median `city08`.
12. Calculate both the sum and standard deviation of `barrels08` for each `VClass`.
13. Create a custom aggregation that calculates the range (max - min) of `city08` for each `make`.
14. Define a function to compute the weighted average of `city08` and `highway08`, with weights of 0.6 and 0.4, respectively.
15. Apply the learned aggregation techniques on the Fuel Economy dataset to find the most fuel-efficient `make` in 2020.
16. Analyze trends in fuel consumption (`barrels08`) over the years, summarizing findings with aggregation methods.
17. Transform the `make` column into a categorical type and perform a basic aggregation.
18. Convert the `year` column into a decade grouping (e.g., 1990s, 2000s) and aggregate `highway08`.
19. Using grouping and aggregation, explore the relationship between `cylinders` and fuel efficiency (`city08` and `highway08`).
20. Investigate how `fuelType` influences the average `barrels08` consumption.
21. Group by both `make` and `year`, and calculate the mean `city08`.
22. Perform a multi-level groupby on `make`, `year`, and `fuelType`, and compute the median `highway08`.

Chapter 5

Data Filtering and Selection

5.1 Introduction

A ubiquitous aspect of data analysis is filtering and selecting specific rows and columns from a dataset. This is often necessary to extract relevant information, filter outliers, remove erroneous data, or perform other data manipulation tasks. This chapter will cover the basics of data filtering and selection in Polars, including conditional statements, Boolean indexing, and query expressions. It will also introduce advanced techniques for efficient data selection, such as using masks and filters, working with null values, and applying dynamic queries based on external criteria.

5.2 Getting the Data

Again, we will be using our friend, the Fuel Economy dataset.

5.3 Filtering with Expressions

The most basic way to filter data in Polars is by creating conditionals using expressions and Boolean operators. Then, we combine these expressions with the `.filter()` method to select the rows that satisfy the condition.

Any boolean expression that returns True or False can be used to filter data. When you pass it to the `.filter()` method, it will return a new DataFrame with only the rows that satisfy the condition (are True). Here is the expression: `pl.col('city08') > 40`

For example, we can use the following code to select all cars with a fuel efficiency of 40 MPG or higher.

```
>>> print(autos
...     .filter(pl.col('city08') > 40)
... )
shape: (375, 15)
| year   make    model      .  createdOn      is_automati  num_gears |
```

5. Data Filtering and Selection

The `.filter` Method

`mpg`

make	year	city08	highway08
Alfa Romeo	1985	19	25
Ferrari	1985	9	14
Dodge	1985	23	33
Dodge	1985	10	12
Subaru	1993	17	23
Subaru	1993	19	26
Subaru	1993	20	28
Subaru	1993	18	24
Subaru	1993	18	24
Subaru	1993	16	21

```
makes = ['Ford', 'Toyota']           cast to string so .is_in works
(mpq
    .with_columns(pl.col('make').cast(pl.String))
    .filter(pl.col.make.is_in(makes) &
            (pl.col.city08 > 50))
)
```

make	year	city08	highway08
Toyota	2000	81	64
Toyota	2001	81	64
Ford	2001	74	58
Toyota	2002	87	69
Toyota	2003	87	69
Toyota	2019	52	48
Toyota	2019	58	53
Toyota	2019	55	53
Toyota	2020	53	52
Toyota	2020	55	53

Figure 5.1: The `.filter` method is a convenient way to select rows that satisfy a given condition.

i16	cat	cat	datetime[μs]	c	i8
			[]	bool	
1994	Geo	Metro XFI	2013-01-01 00:00:00	false	5
2000	Honda	Insight	2013-01-01 00:00:00	false	5
2000	Nissan	Altra EV	2013-01-01 00:00:00	null	null
.
1991	Geo	Metro XFI	2013-01-01 00:00:00	false	5
1992	Geo	Metro XFI	2013-01-01 00:00:00	false	5
1993	Geo	Metro XFI	2013-01-01 00:00:00	false	5

We can combine multiple conditions by using the `&` and `|` operators. The `&` operator is an *and* operation that combines two conditions and returns True

only if both conditions are True. The `|` operator is the *or* operation that returns True if either of the conditions is True. We can negate a condition by using the `~` operator, the *not* operation.

We can also chain calls to `.filter`. For example, to select all Toyotas with a fuel efficiency of 40 MPG or higher, we can use the following code:

```
>>> print(autos
...     .filter(pl.col('city08') > 40)
...     .filter(pl.col('make') == 'Toyota')
... )
shape: (67, 15)
+-----+-----+-----+-----+-----+-----+
| year | make | model |      | createdOn | is_automat | num_gears |
| i16  | cat   | cat    |      | datetime[μ | ic          | i8         |
|      |       |        |      | s]           | bool        |             |
+-----+-----+-----+-----+-----+-----+
| 2000 | Toyota | RAV4 EV | . | 2013-01-01 | null        | null       |
|      |          |          | . | 00:00:00   |             |             |
| 2001 | Toyota | Prius   | . | 2013-01-01 | true        | null       |
|      |          |          | . | 00:00:00   |             |             |
| 2001 | Toyota | RAV4 EV | . | 2013-01-01 | null        | null       |
|      |          |          | . | 00:00:00   |             |             |
| .   | .       | .       | . | .          | .           | .          |
| 2020 | Toyota | Corolla Hybrid | . | 2019-02-06 | true        | null       |
|      |          |          | . | 00:00:00   |             |             |
| 2019 | Toyota | RAV4 Hybrid AWD | . | 2019-03-26 | true        | 6          |
|      |          |          | . | 00:00:00   |             |             |
| 2020 | Toyota | Prius Prime | . | 2019-06-04 | true        | null       |
|      |          |          | . | 00:00:00   |             |             |
+-----+-----+-----+-----+-----+-----+
```

We can also put multiple conditions in the `.filter` method.

```
>>> print(autos
...     .filter(pl.col('city08') > 40,
...             pl.col('make') == 'Toyota')
... )
shape: (67, 15)
+-----+-----+-----+-----+-----+-----+
| year | make | model |      | createdOn | is_automat | num_gears |
| i16  | cat   | cat    |      | datetime[μ | ic          | i8         |
|      |       |        |      | s]           | bool        |             |
+-----+-----+-----+-----+-----+-----+
| 2000 | Toyota | RAV4 EV | . | 2013-01-01 | null        | null       |
|      |          |          | . | 00:00:00   |             |             |
| 2001 | Toyota | Prius   | . | 2013-01-01 | true        | null       |
|      |          |          | . | 00:00:00   |             |             |
| 2001 | Toyota | RAV4 EV | . | 2013-01-01 | null        | null       |
|      |          |          | . | 00:00:00   |             |             |
+-----+-----+-----+-----+-----+-----+
```

5. Data Filtering and Selection

.
2020	Toyota	Corolla	.	2019-02-06	true	null	
		Hybrid	.	00:00:00			
2019	Toyota	RAV4 Hybrid	.	2019-03-26	true	6	
		AWD	.	00:00:00			
2020	Toyota	Prius Prime	.	2019-06-04	true	null	
			.	00:00:00			

I'm not a huge fan of either of the two previous methods. It is not immediately obvious what kind of filtering is being done. Is it doing both conditions? Is it doing one or the other? I want to be more explicit about what I'm doing. That's why we want to use boolean operators like `&`, `|`, and `~` to combine conditions.

Here is how I would write the previous example:

```
>>> print(autos
...     .filter((pl.col('make') == 'Toyota') & (pl.col('city08') > 40))
... )
```

shape: (67, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μs]	ic	i8
2000	Toyota	RAV4 EV	.	2013-01-01 00:00:00	null	null
2001	Toyota	Prius	.	2013-01-01 00:00:00	true	null
2001	Toyota	RAV4 EV	.	2013-01-01 00:00:00	null	null
.
2020	Toyota	Corolla	.	2019-02-06 00:00:00	true	null
		Hybrid	.	00:00:00		
2019	Toyota	RAV4 Hybrid	.	2019-03-26 00:00:00	true	6
		AWD	.	00:00:00		
2020	Toyota	Prius Prime	.	2019-06-04 00:00:00	true	null

Note

When filtering with multiple conditions, make sure you wrap each condition with parentheses. In Python, the `&` operator has a higher precedence than `==` or `>` and will likely error out if you leave out the parentheses.

```
>>> print(autos
...     .filter(pl.col('make') == 'Toyota' & pl.col('city08') > 40)
```

```

...
Traceback (most recent call last)
...
TypeError: the truth value of an Expr is ambiguous

You probably got here by using a Python standard library function
instead of the native expressions API.
Here are some things you might want to try:
- instead of `pl.col('a') and pl.col('b')`, use `pl.col('a') &
  pl.col('b')`
- instead of `pl.col('a') in [y, z]`, use `pl.col('a').is_in([y, z])`
- instead of `max(pl.col('a'), pl.col('b'))`, use
  `pl.max_horizontal(pl.col('a'), pl.col('b'))`
```

5.4 Filtering with Keywords

Polars provides syntactic sugar for equality comparisons. It does this with keyword arguments. If we want all the *make* rows that are *Ford* we can do this:

```

>>> print(autos
...     .filter(make='Ford')
... )
shape: (3_371, 15)
+-----+-----+-----+-----+-----+-----+-----+
| year | make | model |      | createdOn | is_automati | num_gears |
| i16  | cat   | cat   |      | datetime[us]| c           | i8        |
+-----+-----+-----+-----+-----+-----+-----+
| 1993 | Ford  | Taurus | .    | 2013-01-01 | true        | 4         |
|       |        |          |      | 00:00:00   |             |            |
| 1993 | Ford  | Taurus SHO | .   | 2013-01-01 | false       | 5         |
|       |        |          |      | 00:00:00   |             |            |
| 1993 | Ford  | Taurus | .    | 2013-01-01 | true        | 4         |
|       |        |          |      | 00:00:00   |             |            |
| .    | .    | .    | .    | .    | .    | .    |
| 1993 | Ford  | Tempo | .    | 2013-01-01 | false       | 5         |
|       |        |          |      | 00:00:00   |             |            |
| 1993 | Ford  | Tempo | .    | 2013-01-01 | true        | 3         |
|       |        |          |      | 00:00:00   |             |            |
| 1993 | Ford  | Tempo | .    | 2013-01-01 | false       | 5         |
|       |        |          |      | 00:00:00   |             |            |
```

This is a nice convenience, but it is very limited in what it can do. It only works for equality comparisons.

5. Data Filtering and Selection

5.5 Filtering with Dates

Let's filter on *createdOn*, a date column. We can compare dates to specific Python datetime objects. Here, we select all the dates from March 2018 forward:

```
>>> from datetime import datetime
>>> mar2018 = datetime(2018, 3, 1)
>>> print(autos
... .filter(pl.col('createdOn') >= mar2018)
... .sort('createdOn')
... )
shape: (1_577, 15)
```

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μs]	ic	i8
2019	Genesis	G70 AWD	.	2018-03-02 00:00:00	true	8
2019	Genesis	G70 RWD	.	2018-03-02 00:00:00	true	8
2019	Genesis	G70 AWD	.	2018-03-02 00:00:00	true	8
.
2020	Ford	Explorer AWD	.	2019-06-04 00:00:00	true	10
2020	Lincoln	Aviator AWD	.	2019-06-04 00:00:00	true	10
2020	Toyota	Prius Prime	.	2019-06-04 00:00:00	true	null

We can also use the `.is_between` method to select dates between two dates. Note that this method works for any ordinal type, not just dates. For example, we can use it to select all cars with miles per gallon between 40 and 50.

```
>>> print(autos
... .filter(pl.col('city08').is_between(40, 50))
... .sort('city08')
... .select(['year', 'make', 'model', 'VClass', 'city08'])
... )
shape: (185, 5)
```

year	make	model	VClass	city08
i16	cat	cat	cat	u8
1985	Honda	Civic CRX HF	Two Seaters	40
2003	Honda	Civic Hybrid	Compact Cars	40

5.6. Advanced Filtering Techniques

2006	Honda	Civic Hybrid	Compact Cars	40
.
2019	Chevrolet	Malibu Hybrid	Midsize Cars	49
1999	Ford	Ranger Pickup 2WD	Standard Pickup	50
			Trucks 2WD	
2015	Toyota	Prius	Midsize Cars	50

Below we select all the cars with a *createdOn* date between March 2018 and five years prior:

```
>>> from datetime import timedelta
>>> mar2013 = mar2018 - timedelta(days=5*365)
>>> print(autos
... .filter(pl.col('createdOn').is_between(mar2013, mar2018))
... .sort('createdOn')
... )
```

shape: (5_350, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μ	ic	i8
				s]	bool	
2014	Lexus	GS 350	.	2013-10-29 00:00:00	true	8
2014	Nissan	Frontier 2WD	.	2013-10-29 00:00:00	true	5
2014	Nissan	Frontier 2WD	.	2013-10-29 00:00:00	false	5
.
2018	Land Rover	Range Rover LWB SVA	.	2018-02-26 00:00:00	true	8
2018	Land Rover	Range Rover Sport	.	2018-02-26 00:00:00	true	8
2018	Land Rover	Range Rover Sport SVR	.	2018-02-26 00:00:00	true	8

5.6 Advanced Filtering Techniques

Let's find the high outliers for the *city08* column. First, I will filter out all values with a Z-score greater than 3. (The Z-score is the number of standard deviations away from the mean.) This is a common technique for finding outliers when the data is normally distributed. Then, I will select the columns that I want to view.

```
>>> def standardize(col):
...     return (col - col.mean()) / col.std()
```

5. Data Filtering and Selection

```
>>> print(autos
...     .filter(stdize(pl.col('city08')) > 3)
...     .select(['year', 'make', 'model', 'VClass', 'city08'])
... )
```

shape: (334, 5)

year	make	model	VClass	city08
i16	cat	cat	cat	u8
1994	Geo	Metro XFI	Subcompact Cars	43
2000	Honda	Insight	Two Seaters	49
2000	Nissan	Altra EV	Midsized Station Wagons	81
.
1991	Geo	Metro XFI	Subcompact Cars	43
1992	Geo	Metro XFI	Subcompact Cars	43
1993	Geo	Metro XFI	Subcompact Cars	43

We can also get fancy and combine these with window functions. Here, we find the outliers for each *make* and *year* combination. We do this by tacking on a `.over(['year', 'make'])` to our standard deviation calculation.

```
>>> print(autos
...     .filter(stdize(pl.col('city08')).over(['year', 'make']) > 3)
...     .select(['year', 'make', 'model', 'VClass', 'city08'])
... )
```

shape: (624, 5)

year	make	model	VClass	city08
i16	cat	cat	cat	u8
1985	Ferrari	Testarossa	Two Seaters	9
1993	CX Automotive	XM v6	Midsized Cars	13
1993	CX Automotive	XM v6a	Midsized Cars	13
.
1993	Plymouth	Colt	Subcompact Cars	28
1993	Lexus	ES 300	Compact Cars	16
1993	Lexus	ES 300	Compact Cars	16

5.7 More Filtering with Window Expressions

Here's a more complicated example. I want to find cars in production for over 10 years. To do that, I first need to calculate the *model_age*. I can't group by the *model* because that will aggregate all the rows for each model. I want a *model_age* value for each row. To do that, I need to use a *window expression*. Window expressions apply an aggregation to a group of rows. We can use the `.over()` method to apply our expression to a group of rows. In our case, we will use `.over('model')` to apply the expression to each model.

Let's use `.with_columns` to try it out and see what we get.

```
>>> print(autos
... .with_columns(
...     model_age=(pl.col('year').max() - pl.col('year').min())
...         .over('model'))
... )
shape: (41_144, 16)
+-----+
| year   make      model      .  is_automati num_gears model_age |
| i16    cat        cat        .  bool          i8       i16      |
+-----+
| 1985   Alfa Romeo Spider     .  false        5        1      |
|           Veloce               |
|           2000                  |
| 1985   Ferrari   Testarossa .  false        5        8      |
|           a                   |
| 1985   Dodge     Charger     .  false        5        35     |
|           .                   .  .           .        .      |
| 1993   Subaru    Legacy     .  true         4        27     |
|           AWD                 |
| 1993   Subaru    Legacy     .  false        5        27     |
|           AWD                 |
| 1993   Subaru    Legacy     .  true         4        2      |
|           AWD Turbo            |
+-----+
```

That looks pretty good. Now, let's filter out the rows where *model_age* is less than 10. I just need to add `.filter(col('model_age') > 10)` to the end of the previous expression.

```
>>> print(autos
... .with_columns(
...     model_age=(pl.col('year').max() - pl.col('year').min())
...         .over('model'))
... .filter(pl.col('model_age') > 10)
... )
shape: (16_337, 16)
+-----+
| year   make      model      .  is_automati num_gears model_age |
| i16    cat        cat        .  bool          i8       i16      |
+-----+
| 1985   Dodge   Charger     .  false        5        35     |
| 1993   Toyota  Corolla    .  true         3        36     |
| 1993   Toyota  Corolla    .  false        5        36     |
|           .                   .  .           .        .      |
| 1993   Saturn   SL         .  false        5        11     |
| 1993   Subaru   Legacy AWD .  true         4        27     |
| 1993   Subaru   Legacy AWD .  false        5        27     |
+-----+
```

5. Data Filtering and Selection

5.8 Manipulating Data with Conditional Expressions

A very common operation (and one that Pandas make quite awkward) is to use the equivalent of an if-else statement to update a column based on a condition.

Let's assume we want to simplify the fuel type to natural gas, gas, diesel, and electric. First, let's look at our values for *fuelType*.

```
>>> print(autos
... .group_by('fuelType')
... .len()
... )
shape: (14, 2)
fuelType          len
cat              u32

```

Premium or E85	127
Gasoline or propane	8
Regular Gas and Electricity	41
.	.
CNG	60
Premium Gas or Electricity	36
Midgrade	106

Note

There is a `.value_counts` expression method, but it returns a struct. Because the is an expression and not a context, it is not dependent on having a dataframe around. It also provides the opportunity for the query engine to perform optimizations.

```
>>> print(autos
... .select(pl.col('fuelType').value_counts(sort=True))
... )
shape: (14, 1)
fuelType          struct[2]

```

{"Regular",26447}
{"Premium",11542}
{"Gasoline or E85",1317}
.
{"Gasoline or natural gas",20}
{"Gasoline or propane",8}
{"Regular Gas or Electricity",4}

5.8. Manipulating Data with Conditional Expressions

If we want this in two columns, we can use the `.unnest` method.

```
>>> print(autos
... .select(pl.col('fuelType').value_counts(sort=True))
... .unnest('fuelType')
... )
shape: (14, 2)
fuelType          count
cat              u32
Regular           26447
Premium           11542
Gasoline or E85   1317
.
Gasoline or natural gas    20
Gasoline or propane       8
Regular Gas or Electricity 4
```

We can pull off just one column by combining the `.struct` attribute with an index operation.

```
>>> print(autos
... .select(pl.col('fuelType')
...         .value_counts(sort=True)
...         .struct[0]))
... )
shape: (14, 1)
fuelType
cat
Regular
Premium
Gasoline or E85
.
Gasoline or natural gas
Gasoline or propane
Regular Gas or Electricity
```

I want to make a new column called `fuelTypeSimple` and set it to the value of `fuelType`. I will build up an expression. Polars doesn't have an if else statement, but it does have `.when`, `.then`, and `.otherwise` methods. I will use these to build up the expression.

I want the logic to be similar to this Python if statement:

```
if 'Diesel' in fuel_type:
    fuel_type_simple = 'Diesel'
```

5. Data Filtering and Selection

```
elif 'CNG' in fuel_type:  
    fuel_type_simple = 'CNG'  
elif 'Electricity' in fuel_type:  
    fuel_type_simple = 'Electric'  
else:  
    fuel_type_simple = 'Gasoline'
```

The format in Polars looks like this:

```
(pl.when(COND_EXPR)  
 .then(EXPR)  
.when(COND_EXPR2)  
 .then(EXPR2)  
... # more when's  
.otherwise(EXPRN) # optional
```

Let's try to create the *fuelTypeSimple* column using these methods. To construct this with Polars expressions, we need to replace if and elif with the pl.when function. The body of the if statement is replaced with the pl.then function. The else statement replaces the pl.otherwise function. It looks like this:

```
>>> fuel_str = pl.col('fuelType').cast(pl.String)  
>>> simple = (pl.when(fuel_str.str.contains('Diesel')).then(pl.lit('Diesel'))  
...     .when(fuel_str.str.contains('CNG')).then(pl.lit('CNG'))  
...     .when(fuel_str.str.contains('Electricity')).then(pl.lit('Electric'))  
...     .otherwise(pl.lit('Gasoline')))  
  
>>> print(autos  
... .with_columns(simple.alias('fuelTypeSimple'))  
... )  
shape: (41_144, 16)
```

year	make	model	.	is_automat	num_gears	fuelTypeS
i16	cat	cat	.	tic	i8	imple
				bool		str
1985	Alfa Romeo	Spider Veloce	.	false	5	Gasoline
		2000				
1985	Ferrari	Testarossa	.	false	5	Gasoline
		a				
1985	Dodge	Charger	.	false	5	Gasoline
.
1993	Subaru	Legacy AWD	.	true	4	Gasoline
1993	Subaru	Legacy	.	false	5	Gasoline

5.8. Manipulating Data with Conditional Expressions

1993	Subaru	Legacy	AWD	true	4	Gasoline
		AWD Turbo				

Note

The above code uses `pl.lit` to create a scalar value for a column. If we were to write:

```
(pl.when(fuel_str.str.contains('Diesel')).then(pl.lit('Diesel')))
```

as:

```
(pl.when(fuel_str.str.contains('Diesel')).then('Diesel'))
```

it is assumed that the 'Diesel' value is the column's name. It is a shorthand for this code:

```
(pl.when(fuel_str.str.contains('Diesel')).then(pl.col('Diesel')))
```

Now that we have validated that the expression works, we can filter on the column. Let's look at the CNG rows.

```
>>> fuel_str = pl.col('fuelType').cast(pl.String)
>>> simple = (pl.when(fuel_str.str.contains('Diesel')).then(pl.lit('Diesel'))
...     .when(fuel_str.str.contains('CNG')).then(pl.lit('CNG'))
...     .when(fuel_str.str.contains('Electricity')).then(pl.lit('Electric'))
...     .otherwise(pl.lit('Gasoline')))

>>> print(autos
... .with_columns(fuelTypeSimple=simple)
... .filter(pl.col('fuelTypeSimple') == 'CNG')
... )
```

year	make	model	.	is_automatic	num_gears	fuelTypeSimple
i16	cat	cat	.	bool	i8	str
1996	Ford	Crown Victoria	.	true	4	CNG
		CNG				
1997	Ford	Crown Victoria	.	true	4	CNG
		CNG				
1998	Honda	Civic	CNG	.	true	4
.

5. Data Filtering and Selection

2014	Honda	Civic	.	true	5	CNG
		Natural				
		Gas				

2014	Mobility Ventures LLC	MV-1	CNG	.	true	4	CNG

2015	Honda	Civic	.	true	5	CNG
		Natural				
		Gas				

To filter, we don't even need to create a new column first. We can inline the conditional expression in the `.filter` method. (Although, often it is useful to create the column to see the data and inspect that it is correct.)

```
>>> fuel_str = pl.col('fuelType').cast(pl.String)
>>> simple = (pl.when(fuel_str.str.contains('Diesel')).then(pl.lit('Diesel'))
...     .when(fuel_str.str.contains('CNG')).then(pl.lit('CNG'))
...     .when(fuel_str.str.contains('Electricity')).then(pl.lit('Electric'))
...     .otherwise(pl.lit('Gasoline')))
```

```
>>> print(autos
... .filter(simple == 'CNG')
... )
```

shape: (60, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μs]	tic	i8
1996	Ford	Crown	.	2013-01-01 00:00:00	true	4
		Victoria	.			
		CNG	.			
1997	Ford	Crown	.	2013-01-01 00:00:00	true	4
		Victoria	.			
		CNG	.			
1998	Honda	Civic	CNG	2013-01-01 00:00:00	true	4
			.			
.
2014	Honda	Civic	.	2014-02-17 00:00:00	true	5
		Natural	.			
		Gas	.			
2014	Mobility Ventures LLC	MV-1	CNG	2014-09-10 00:00:00	true	4
			.			
2015	Honda	Civic	.	2014-11-05	true	5

5.8. Manipulating Data with Conditional Expressions

```
Natural      6  
Gas         00:00:00
```

In the above examples, I'm using scalars (`pl.lit` expressions) for the values of `.then` and `.otherwise`. You can also use expressions to use the existing column (or a derivation from the column).

For example, I often want to generalize categorical values to keep only the top N values. If the value is one of the top 10 makes, keep it; otherwise, use the scalar `Other`.

Let's make a new column, `simple_make`, with only the top 10 makes, giving the remaining values the value of `Other`. First, we will create an expression to give us the top 10 makes.

```
>>> top_n = (pl.col('make')  
...     .value_counts(sort=True)  
...     .struct[0]  
...     .head(10)  
... )  
  
>>> top_expr = (pl.when(pl.col('make').is_in(top_n))  
...     .then(pl.col('make'))  
...     .otherwise(pl.lit('Other'))  
... )  
  
>>> print(autos  
...     .with_columns(simple_make=top_expr)  
... )
```

year	make	model	.	is_automat	num_gears	simple_ma
i16	cat	cat	.	tic	i8	ke
1985	Alfa Romeo	Spider	.	false	5	Other
		Veloce				
		2000				
1985	Ferrari	Testarossa	.	false	5	Other
		a				
1985	Dodge	Charger	.	false	5	Dodge
.
1993	Subaru	Legacy	.	true	4	Other
		AWD				
1993	Subaru	Legacy	.	false	5	Other
		AWD				
1993	Subaru	Legacy	.	true	4	Other
		AWD Turbo				

5. Data Filtering and Selection

In fact, I use the above so often that I create a helper function to limit categories.

```
>>> def limit(col_name, n=10, default='other'):
...     col = pl.col(col_name)
...     top_n = (col
...             .value_counts(sort=True)
...             .struct[0]
...             .head(n)
...     )
...     return (pl.when(col.is_in(top_n))
...             .then(col)
...             .otherwise(pl.lit(default)))
... )

>>> print(autos
...       .with_columns(simple_make=limit('make'))
...   )
shape: (41_144, 16)
+-----+-----+-----+-----+-----+-----+
| year | make | model | . | is_automat... | num_gears | simple_ma...
| i16  | cat   | cat    | . | ic bool    | i8        | ke cat      |
+-----+-----+-----+-----+-----+-----+
```

year	make	model	.	is_automat...	num_gears	simple_ma...
i16	cat	cat	.	ic bool	i8	ke cat
1985	Alfa Romeo	Spider	.	false	5	other
		Veloce				
		2000				
1985	Ferrari	Testarossa	.	false	5	other
		a				
1985	Dodge	Charger	.	false	5	Dodge
.
1993	Subaru	Legacy	.	true	4	other
		AWD				
1993	Subaru	Legacy	.	false	5	other
		AWD				
1993	Subaru	Legacy	.	true	4	other
		AWD Turbo				

Note

The above code has a `CategoricalRemappingWarning`. This is because Polars uses a dictionary to encode categorical values. The value for the `other` parameter was not in the dictionary. To overcome this, we can:

- Use `pl.enable_string_cache` to enable string caching. I would insert this at the top of the code right after the import statements.

- Use the `pl.StringCache` context manager when you create the `Dataframe`.

```
>>> with pl.StringCache():
...     autos2 = tweak_auto(raw)
...     print(autos2
...         .select(simple_make=limit('make', 20, 'other')))

shape: (41_144, 1)
+-----+
| simple_make |
| cat          |
+-----+
| other        |
| other        |
| Dodge        |
| .            |
| Subaru       |
| Subaru       |
| Subaru       |
+-----+
```

Looking up strings in a cache has a slight overhead. There is one more option. Using the `Enum` type. Knowing the categories beforehand, you can use the `Enum` type to create a categorical column. This will be faster than using the string cache. However, in this case, I don't know the top values for the category ahead of time. If I knew these values, I wouldn't have to call `.value_counts`. Also, hard-coding the values would return invalid results if the distribution changed in the future.

5.9 Handling Missing Data

You will often encounter missing data in your datasets. If you want to use machine learning on your data, you probably have to deal with missing values. Many ML algorithms do not work with missing data. Polars has a few methods for handling missing data.

First, let's quantify how much data is missing from our dataset. We can use the `.is_null` method. I will select all the columns using `pl.col("*")`.

Note

The expression `pl.col('*')` is equivalent to `pl.all()`. Both of them will select all of the columns.

```
>>> print(autos
...     .select(pl.col('*').is_null())
... )
shape: (41_144, 15)
| year      make      model      .      createdOn      is_automatic      num_gears |
```

5. Data Filtering and Selection

bool	bool	bool	bool	bool	bool
false	false	false	.	false	false
false	false	false	.	false	false
false	false	false	.	false	false
.
false	false	false	.	false	false
false	false	false	.	false	false
false	false	false	.	false	false

This gives us back a dataframe with boolean values.

Note

In Python, a `True` value evaluates to `1`, and a `False` evaluates to `0`.

```
>>> int(True)  
1
```

In Pandas, we would sum these up or take their mean to calculate the count or percentage of missing values. This works in Polars as well.

```
>>> print(autos  
... .select(pl.all().is_null().sum())  
... )  
shape: (1, 15)  
| year   make   model   .   createdOn   is_automatic   num_gears |  
| u32     u32     u32      .     u32          u32           u32    |  
| 0       0       0       .     0           11            837   |
```

However, in Polars, there is also a `.null_count` method that will give us the count of null values:

```
>>> print(autos  
... .select(pl.all().null_count())  
... )  
shape: (1, 15)  
| year   make   model   .   createdOn   is_automatic   num_gears |  
| u32     u32     u32      .     u32          u32           u32    |  
| 0       0       0       .     0           11            837   |
```

It turns out that there is a shortcut for this. We can use the `.null_count` method on the DataFrame directly.

```
>>> print(autos
... .null_count()
...
)
shape: (1, 15)
| year    make    model    .   createdOn  is_automatic  num_gears |
| u32     u32     u32      .   u32       u32          u32      |
| 0       0       0       .   0         11          837      |
```

This is also included in the `.describe` method. There is a `null_count` row.

```
>>> print(autos
... .describe()
...
)
shape: (9, 16)
| statistic    year      make      .   createdOn  is_automati  num_gears |
| str          f64       str       .   str        f64          f64      |
|               |           |           |           |           |           |           |
| count        41144.00  41144    .   41144     41133.00   40307.00
| null_count   0.00      0        .   0         11.00      837.00
| mean         2001.54   null     .   2013-08-1  0.69      5.02
|             |           |           |           |           |           |
|             |           |           |           |           |           |           |
| std          11.14     null     .   null      null       1.36
| min          1984.00   null     .   2013-01-0  0.00      1.00
|             |           |           |           |           |           | |
|             |           |           |           |           |           |           |
|             |           |           |           |           |           |           |
| 25%          1991.00   null     .   2013-01-0  null      4.00
|             |           |           |           |           |           | |
|             |           |           |           |           |           |           |
|             |           |           |           |           |           |           |
| 50%          2002.00   null     .   2013-01-0  null      5.00
|             |           |           |           |           |           | |
|             |           |           |           |           |           |           |
|             |           |           |           |           |           |           |
| 75%          2011.00   null     .   2013-01-0  null      6.00
|             |           |           |           |           |           |
|             |           |           |           |           |           |           |
| max          2020.00   null     .   2019-06-0  1.00      10.00
|             |           |           |           |           |           |
|             |           |           |           |           |           |           |
```

Polars does not have a `.null_percent` method, but we can easily calculate it using the `.mean` method.

```
>>> print(autos
... .select(pl.all().is_null().mean() * 100)
```

5. Data Filtering and Selection

```
... )
shape: (1, 15)
| year   make    model    .   createdOn   is_automatic   num_gears |
| f64    f64     f64      .   f64        f64          f64       |
| 0.00   0.00   0.00   .   0.00       0.03        2.03    |
```

Let's filter out the columns with values of 0 so we can see which columns have missing values. This operation is a little more complicated than many. I'm using `.pipe` to pass the current DataFrame to the `.select` method. I want to select the floating-point columns with a value greater than 0. I need to use `col.all()` to get a boolean value for each column. If I don't include that, it will return all of the columns because `df_.select(pl.col(pl.Float64) > 0)` is a DataFrame with all of the columns with true or false values. I only want the columns where all of the values are true.

```
>>> print(autos
...   .select(pl.all().is_null().cast(pl.Int32).mean() * 100)
...   .pipe(lambda df_: df_.select([col.name
...       for col in df_.select(pl.col(pl.Float64)> 0)
...       if col.all()]))
... )
shape: (1, 6)
| displ   cylinders   trany   drive   is_automatic   num_gears |
| f64     f64         f64     f64     f64          f64       |
| 0.50   0.50        0.03   2.89   0.03        2.03    |
```

We might consider repeating this exercise to look for NaN values because Polars has both NULL and NaN values. Only numeric columns can have NaN values, so we will need to filter on the numeric columns. I'll use the `selectors` module to get the numeric columns.

```
>>> import polars.selectors as cs
>>> print(autos
...   .select(cs.numeric().is_nan().cast(pl.Int32).mean() * 100)
...   .pipe(lambda df_: df_.select([col.name
...       for col in df_.select(pl.col(pl.Float64)> 0)
...       if col.all()]))
... )
shape: (0, 0)
```



It looks like there are no NaN values in the numeric columns. (This is not particularly surprising since most CSV files don't generally distinguish

between NULL and NaN values. Certainly, the Fuel Economy dataset doesn't.)

Note

When parsing CSV files, Polars treats NaN as nan and the empty string as null.

5.10 Dropping Rows with Missing Values

NULL values can be problematic for machine learning algorithms. Many algorithms will not work with NULL values. Some more modern algorithms like XGBoost can handle NULL values. I like to have a plan to deal with missing values because I will often use Principal Component Analysis (PCA) to explore my data. PCA will not work with NULL values.

```
>>> from sklearn import decomposition
>>> import polars.selectors as cs
>>> pca = decomposition.PCA()
>>> pca.fit(autos.select(cs.numeric()))
Traceback (most recent call last)
...
ValueError: Input X contains NaN.
PCA does not accept missing values encoded as NaN natively. For
supervised learning, you might want to consider
sklearn.ensemble.HistGradien...
```

Let's go nuclear and drop all the rows that have missing values. We can use the `.drop_nulls` method.

```
>>> print(autos
... .drop_nulls()
... )
shape: (38_934, 15)
| year   make      model      .  createdOn  is_automat...  num_gears |
| i16    cat       cat        .  datetime[...]
|                  |          |           |           |
|                  |          |           |           |
| 1985   Alfa Romeo Spider   .  2013-01-0  false     5
|                  Veloce      |  1
|                  2000       |  00:00:00
| 1985   Ferrari   Testaross  .  2013-01-0  false     5
|                  a          |  1
|                  |          |  00:00:00
| 1985   Dodge     Charger   .  2013-01-0  false     5
|                  |          |  1
|                  |          |  00:00:00
```

5. Data Filtering and Selection

.
1993	Subaru	Legacy	.	2013-01-0	true	4	
		AWD	.	1			
			.	00:00:00			

We can be more selective and only drop rows with missing values in specific columns. We do this by passing a list of columns to the `subset` parameter. Let's remove all rows with missing values in the `cylinders` and `displ` columns.

>>> print(autos							
... .drop_nulls(subset=['cylinders', 'displ']))							
...)							
shape: (40_938, 15)							
year	make	model	.	createdOn	is_automat	num_gears	
i16	cat	cat	.	datetime[tic	i8	
			.	μs]	bool		
1985	Alfa Romeo	Spider	.	2013-01-0	false	5	
		Veloce	.	1			
		2000	.	00:00:00			
1985	Ferrari	Testarossa	.	2013-01-0	false	5	
		a	.	1			
			.	00:00:00			
1985	Dodge	Charger	.	2013-01-0	false	5	
			.	1			
			.	00:00:00			
.	
1993	Subaru	Legacy	.	2013-01-0	true	4	
		AWD	.	1			
			.	00:00:00			
1993	Subaru	Legacy	.	2013-01-0	false	5	
		AWD	.	1			
			.	00:00:00			
1993	Subaru	Legacy	.	2013-01-0	true	4	
		AWD Turbo	.	1			
			.	00:00:00			

Note

We can also use a pass in a column selector to the `.drop_nulls` subset parameter. This code drops all missing numeric values.

```
>>> print(autos
... .drop_nulls(subset=cs.numeric())
... )
shape: (40_115, 15)
```

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[ic	i8
				μs]	bool	
1985	Alfa Romeo	Spider	.	2013-01-0	false	5
		Veloce		1		
		2000		00:00:00		
1985	Ferrari	Testarossa	.	2013-01-0	false	5
		a		1		
				00:00:00		
1985	Dodge	Charger	.	2013-01-0	false	5
				1		
				00:00:00		
.
1993	Subaru	Legacy	.	2013-01-0	true	4
		AWD		1		
				00:00:00		
1993	Subaru	Legacy	.	2013-01-0	false	5
		AWD		1		
				00:00:00		
1993	Subaru	Legacy	.	2013-01-0	true	4
		AWD Turbo		1		
				00:00:00		

Let's run PCA now.

```
>>> pca = decomposition.PCA()
>>> pca.fit(autos.select(cs.numeric()).drop_nulls())
PCA()
```

Much better. It runs now that we don't have missing values.

Note

Dropping all rows with missing values is a blunt tool. Tread with caution.

5. Data Filtering and Selection

I have a whole textbook whose focus is statistical analysis with missing data. Rather than take pages to discuss that theory here, I will give some advice that I have found useful in my career.

If your data has missing values, find a subject matter expert who can help explain why the data is missing. Those conversations are invaluable for determining the appropriate actions to take, whether dropping rows or columns or replacing missing values.

We can also remove rows with missing values. For example, we might want to find rows where both `num_gears` and `is_automatic` are missing. If we throw both columns into the column expression and use the `.is_null` method, Polars complains that this is an ambiguous operation. It doesn't know if we want to do an `and` or an `or` operation.

```
>>> (autos
... .filter(pl.col('num_gears', 'is_automatic').is_null())
... )
Traceback (most recent call last)
...
ComputeError: The predicate passed to 'LazyFrame.filter' expanded to
multiple expressions:

col("num_gears").is_null(),
col("is_automatic").is_null(),
This is ambiguous. Try to combine the predicates with the 'all' or `any'
expression.
```

Error originated just after this operation:

```
DF ["year", "make", "model", "displ"]; PROJECT */15 COLUMNS; SELECTION:
"None"
```

We can write out the expression in a more explicit way. We can use the `&` operator to combine the two conditions.

```
>>> print(autos
... .filter(pl.col('num_gears').is_null() &
...         pl.col('is_automatic').is_null())
... )
shape: (11, 15)
  year   make      model      .  createdOn    is_automat  num_gears
  i16     cat        cat        .  datetime[μ
                                s]          ic           bool
                                |
  2000  Nissan   Altra EV    .  2013-01-01    null       null
                                00:00:00
  2000  Toyota   RAV4 EV    .  2013-01-01    null       null
```

5.10. Dropping Rows with Missing Values

2001	Toyota	RAV4 EV	.	00:00:00 2013-01-01	null	null
.	.	.	.	00:00:00	.	.
2003	Toyota	RAV4 EV	.	2013-01-01	null	null
				00:00:00		
1984	Ford	F150 Pickup	.	2013-01-01	null	null
		2WD		00:00:00		
1984	GMC	C25 Pickup	.	2013-01-01	null	null
		2WD		00:00:00		

Alternatively, we can use the `all_horizontal` function to perform row-wise and operations. (There is also an `any_horizontal` function to perform row-wise or operations.)

```
>>> print(autos
... .filter(pl.all_horizontal(
...     pl.col('num_gears', 'is_automatic').is_null())))
... )
```

shape: (11, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μs]	ic	i8
2000	Nissan	Altra EV	.	2013-01-01 00:00:00	null	null
2000	Toyota	RAV4 EV	.	2013-01-01 00:00:00	null	null
2001	Toyota	RAV4 EV	.	2013-01-01 00:00:00	null	null
.
2003	Toyota	RAV4 EV	.	2013-01-01 00:00:00	null	null
1984	Ford	F150 Pickup	.	2013-01-01 2WD 00:00:00	null	null
1984	GMC	C25 Pickup	.	2013-01-01 2WD 00:00:00	null	null

To filter the rows where both `num_gears` and `is_automatic` are missing, we invert the filter by adding a `~` in front of the expression.

```
>>> print(autos
... .filter(~pl.all_horizontal(
...     pl.col('num_gears', 'is_automatic').is_null())))
... )
shape: (41_133, 15)
```

5. Data Filtering and Selection

year i16	make cat	model cat	.	createdOn datetime[μs]	is_automat- ic bool	num_gears i8
1985	Alfa Romeo	Spider	.	2013-01-01 00:00:00	false	5
		Veloce	.			
		2000	.			
1985	Ferrari	Testarossa	.	2013-01-01 00:00:00	false	5
		a	.			
1985	Dodge	Charger	.	2013-01-01 00:00:00	false	5
.
1993	Subaru	Legacy	.	2013-01-01 00:00:00	true	4
		AWD	.			
1993	Subaru	Legacy	.	2013-01-01 00:00:00	false	5
		AWD	.			
1993	Subaru	Legacy	.	2013-01-01 00:00:00	true	4
		AWD Turbo	.			

5.11 Filling in Missing Values

Let's fill in the missing values in the *cylinders* column with 0. We can use the `.fill_null` method to do this.

```
>>> print(autos
... .with_columns(pl.col('cylinders').fill_null(0))
... )
```

year i16	make cat	model cat	.	createdOn datetime[μs]	is_automat- ic bool	num_gears i8
1985	Alfa Romeo	Spider	.	2013-01-01 00:00:00	false	5
		Veloce	.			
		2000	.			
1985	Ferrari	Testarossa	.	2013-01-01 00:00:00	false	5
		a	.			
1985	Dodge	Charger	.	2013-01-01 00:00:00	false	5

5.12. Filling in Time Series Missing Values

.
1993	Subaru	Legacy	.	2013-01-0	true	4	
		AWD		1			
				00:00:00			
1993	Subaru	Legacy	.	2013-01-0	false	5	
		AWD		1			
				00:00:00			
1993	Subaru	Legacy	.	2013-01-0	true	4	
		AWD Turbo		1			
				00:00:00			

Note that arbitrarily filling in missing values can be problematic. For example, the distribution of the data will likely change. Here, you can see that the mean value changes.

```
>>> print(autos
... .select(cyl_mean=pl.col('cylinders').mean(),
...         cyl_fill0_mean=pl.col('cylinders').fill_null(0).mean())
... )
```

shape: (1, 2)

cyl_mean	cyl_fill0_mean
f64	f64
5.72	5.69

You should have a good reason for replacing missing values. I recommend talking to a domain expert if you are not sure what to do.

5.12 Filling in Time Series Missing Values

When you have ordered data with missing values, you may want to fill in the missing values based on the previous or next value. This is called forward or backward filling. Polars has a `strategy` parameter that allows you to do this. Let's create a dataframe to represent snow on the ground over time.

Because we are using Polars, I will include both NULL and NaN values. (Thinking about snowfall, I'm not sure how it can be NaN. NULL makes sense, but NaN doesn't. But I'm going to include it anyway.)

```
>>> import numpy as np
>>> snow = pl.DataFrame({'depth': [0, 0, np.nan, 9.1, 11.3, None, 7.8,
...                           15, 20]})
>>> print(snow)
shape: (9, 1)
| depth |
| f64  |
```

5. Data Filtering and Selection

0.00
0.00
NaN
.
7.80
15.00
20.00

I will demonstrate forward fill, backward fill, linear interpolation, and filling in the missing values with a constant value (0) and the mean.

```
>>> print(snow
... .with_columns(pl.col('depth'),
...     depth_ffill=pl.col('depth').fill_null(strategy='forward'),
...     depth_bfill=pl.col('depth').fill_null(strategy='backward'),
...     depth_interp=pl.col('depth').interpolate(),
...     depth_0fill=pl.col('depth').fill_null(0),
...     depth_mean=pl.col('depth').fill_null(strategy='mean'),
... )
... )
```

shape: (9, 6)

depth	depth_ffil	depth_bfil	depth_inte	depth_0fil	depth_mean
f64	l	l	rp	l	n
	f64	f64	f64	f64	f64
0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00
NaN	NaN	NaN	NaN	NaN	NaN
.
7.80	7.80	7.80	7.80	7.80	7.80
15.00	15.00	15.00	15.00	15.00	15.00
20.00	20.00	20.00	20.00	20.00	20.00

Notice that the NaN values are not filled in with `.fill_null`. Also, note that filling in

with the mean appeared to fail. This is because the mean calculation does return a real number when there are NaN values. (It will, however, work with NULL values.)

Note that this dataset also had NumPy NaN values. Polars treats these as different from null values. Let's use the `.describe()` method to see the difference. It is not counted as a null value; it also causes the mean and standard deviation to be NaN.

```
>>> print(snow
... .describe()
...
shape: (9, 2)
  statistic    depth
  str          f64
  count      8.00
  null_count 1.00
  mean        NaN
  .
  50%       11.30
  75%       15.00
  max       20.00
```

Polars provides an `.is_nan` and `.fill_nan` method to handle these values.

```
>>> print(snow
... .with_columns('depth',
...     fill_nan=pl.col('depth').fill_nan(None),
...     depth_ffill=pl.col('depth').fill_null(strategy='forward'),
...     depth_bfill=pl.col('depth').fill_null(strategy='backward'),
...     depth_interp=pl.col('depth').interpolate(),
...     depth_mean=pl.col('depth').fill_null(strategy='mean'))
...
...
shape: (9, 6)
  depth   fill_nan  depth_ffill  depth_bfil  depth_inte  depth_mean
  f64     f64        f64           l          rp          f64
  .
  f64     f64        f64           f64         f64         f64
  0.00   0.00       0.00        0.00       0.00       0.00
  0.00   0.00       0.00        0.00       0.00       0.00
  NaN    null       NaN         NaN        NaN        NaN
  .
  7.80   7.80       7.80        7.80       7.80       7.80
  15.00  15.00      15.00       15.00      15.00      15.00
  20.00  20.00      20.00       20.00      20.00      20.00
```

I tried to fill in the missing `NaN` values, but the fill operations didn't work. This is because Polars is trying to run these operations in parallel. It doesn't know that the `depth` column was updated.

It fails if we refer to the `fill_nan` column inside the same `.with_columns` call.

```
>>> print(snow
... .with_columns('depth',
```

5. Data Filtering and Selection

```
...     fill_nan=pl.col('depth').fill_nan(None),
...     depth_ffill=pl.col('fill_nan').fill_null(strategy='forward'),
...     depth_bfill=pl.col('fill_nan').fill_null(strategy='backward'),
...     depth_interp=pl.col('fill_nan').interpolate(),
...     depth_mean=pl.col('fill_nan').fill_null(strategy='mean')
... )
...
)
Traceback (most recent call last)
...
ColumnNotFoundError: fill_nan
```

Error originated just after this operation:
DF ["depth"]; PROJECT */1 COLUMNS; SELECTION: "None"

Let's add synchronization to the previous example. We can use two calls to `.with_columns` so that the second can use the updated `depth` column.

```
>>> imp_pl = (snow
...   .with_columns(fill_nan=pl.col('depth').fill_nan(None))
...   .with_columns(
...     depth_ffill=pl.col('fill_nan').fill_null(strategy='forward'),
...     depth_bfill=pl.col('fill_nan').fill_null(strategy='backward'),
...     depth_interp=pl.col('fill_nan').interpolate(),
...     depth_mean=pl.col('fill_nan').fill_null(strategy='mean')
... )
...
)
>>> print(imp_pl)
```

```
shape: (9, 6)
+-----+
| depth    fill_nan    depth_ffill    depth_bfil    depth_inte    depth_mean |
| f64      f64          f64           l             rp            f64          |
+-----+
| 0.00    0.00        0.00        0.00        0.00        0.00        |
| 0.00    0.00        0.00        0.00        0.00        0.00        |
| NaN     null        0.00        9.10        4.55        9.03        |
| .         .          .           .           .           .           |
| 7.80    7.80        7.80        7.80        7.80        7.80        |
| 15.00   15.00       15.00       15.00       15.00       15.00       |
| 20.00   20.00       20.00       20.00       20.00       20.00       |
+-----+
```

I'm going to create a Matplotlib with Polars to visualize these different fill types.

```
>>> import hvplot
>>> hvplot.extension('matplotlib')
>>>
```

```

>>> shift = .8
>>> plot=(imp_pl
...     .select(Original=pl.col('depth'),
...             Forward_fill=pl.col('depth_ffill')+shift,
...             Backward_fill=pl.col('depth_bfill')+shift * 2,
...             Interpolate=pl.col('depth_interp')+shift * 3,
...             Mean_fill=pl.col('depth_mean')+shift * 4,
...         )
...     .plot(title='Missing Values Demo', width=1000, height=500)
... )

```

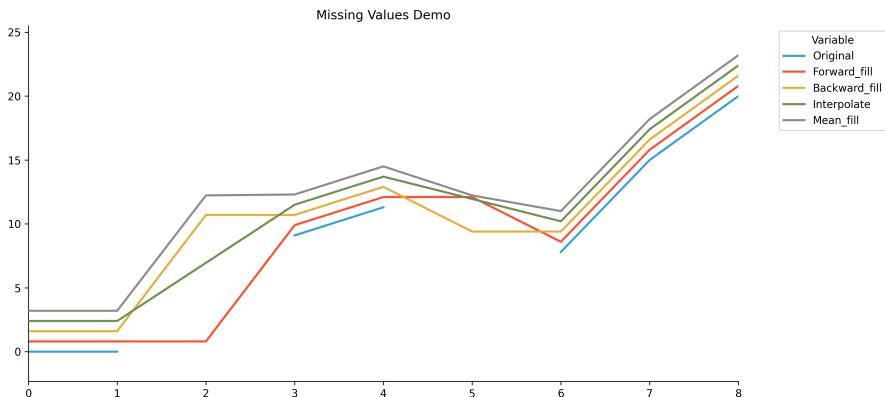


Figure 5.2: A visualization of different techniques for dealing with missing data.

Note

Note that if you are doing machine learning, you will probably not want to use a forward fill or interpolation. This is because those methods use data from the future to fill in the missing values. This is called *data leakage*. You will want to use a backward fill or another method that doesn't use future data.

5.13 Chapter Methods and Functions

Table 5.1: Chapter methods and functions

Method	Description
<code>df.filter(expr)</code>	Returns a new DataFrame with only the rows that satisfy the condition.

5. Data Filtering and Selection

Method	Description
<code>df.describe(percentiles=[0.25, 0.5, 0.75])</code>	Returns a DataFrame with descriptive statistics for each column.
<code>df.drop_nulls(subset=None)</code>	Returns a new DataFrame with rows that have null values removed. If subset is specified, only the columns in the list will be checked for null values.
<code>col.is_between(lower_bound, upper_bound, closed='both')</code>	Returns a boolean expression that is true if the column value is between the lower and upper bounds. The closed parameter can be set to both, left, right, or none to control whether the bounds are included or excluded.
<code>col.is_null()</code>	Returns a boolean expression that is true if the column value is null.
<code>col.null_count()</code>	Returns the number of null values in the column.
<code>col.fill_null(value, strategy=None, limit=None)</code>	Returns a new column with null values filled in with the specified value. If strategy is specified, it can be set to forward, backward, linear, or mean. If limit is specified, it will limit the number of consecutive null values that are filled in.
<code>col.fill_nan(value)</code>	Returns a new column with NaN values filled in with the specified value.
<code>col.interpolate(method='linear')</code>	Returns a new column with null values filled in using linear interpolation. method can also be 'nearest'.
<code>col.over(expr, *more_exprs, mapping_strategy ='group_to_rows')</code>	Create a window expression. Group over the specified expressions. The mapping_strategy parameter can be 'group_to_rows', 'join', or 'explode'. Generally, an aggregation function is used before the .over method.
<code>group.count()</code>	Returns the number of rows in each group.
<code>pl.lit(value, dtype=None, *, allow_object=False)</code>	Returns a literal value.
<code>pl.when(*predicates, **constraints)</code>	Start a when, then, otherwise expression. Follow this with .then and optionally another .when expression. Can add .otherwise for an else.
<code>when.then(statement)</code>	Value to use when the when predicate is true.
<code>then.otherwise(statement)</code>	Value to use when the when predicate is false.

5.14 Summary

This chapter has looked at data filtering and selection using Polars. Starting with Boolean indexing and conditional statements, we progressed to constructing and employing query expressions for more complex data retrieval. We also covered efficient row and column selection methods, handling null values, and applying dynamic filters. As you move forward, remember these techniques are pivotal in data manipulation and can significantly impact the insights derived from your datasets.

5.15 Exercises

1. How can you use conditional statements to select specific rows in a DataFrame using Polars?
2. Create a query expression to find all vehicles from 2000 onwards that use electric fuel.
3. Write a Polars expression to filter out vehicles with null values in the 'city08' column.
4. Demonstrate how to select cars in production for over 15 years using window expressions.
5. Using the given dataset, write a script to categorize vehicles into 'Electric', 'Gasoline', 'Diesel', and 'Hybrid' based on their fuel type.
6. How can missing data be handled in Polars, and what are the implications of different methods like forward fill or interpolation?
7. Using the dataset, find the average number of gears in cars produced after the year 2010.
8. Write a script to convert the 'createdOn' column to a datetime format and filter records to show only vehicles created in the last five years.
9. Create a boolean expression to identify and select all SUVs with a highway mileage greater than 30 MPG.
10. Using filtering techniques, find the top 10 most fuel-efficient models in the dataset.

Chapter 6

Sorting and Ordering in Polars

6.1 Introduction

In this chapter, we will learn about sorting and ordering techniques in Polars. Sorting is a fundamental operation in data analysis, and it is essential to understand how data can be sorted and ordered. We will learn about sorting criteria and how to implement them in Polars. We will also discuss the performance aspects of sorting and how to optimize sorting operations for large datasets.

6.2 Loading the Fuel Economy Dataset

Again, we will load the fuel economy dataset for this chapter.

6.3 Sorting by a Single Column

The primary method for sorting data in Polars is the `.sort` method. It can be used to sort data by single or multiple columns. Let's start by sorting the data by a single column.

The interface for the `.sort` method differs from Python sorting methods like `sorted` or `list.sort`.

Unlike Python, which has a `key` argument for specifying a custom sorting function, Polars uses a different approach. Instead of passing a function, we pass a column or expression to sort by. For example, in Python, to sort a list of strings by their length, we would do the following:

```
>>> names = ['Al', 'Bob', 'Charlie', 'Dan', 'Edith', 'Frank']
```

We can sort this lexically by calling the `sorted` function:

```
>>> sorted(names)
['Al', 'Bob', 'Charlie', 'Dan', 'Edith', 'Frank']
```

6. Sorting and Ordering in Polars

If we wanted to sort this list by the length of the strings, we would pass a function to the `key` argument:

```
>>> sorted(names, key=len)
['Al', 'Bob', 'Dan', 'Edith', 'Frank', 'Charlie']
```

In Polars, instead of using `key`, we use the `by` parameter to provide a parallel expression to sort the data by. For example, to sort the `autos` dataframe by the length of the `make` column, we would first need to convert the `make` column from a categorical column to a string column. Then we can use the `.sort` method and pass in the length of the `make` column as the `by` parameter:

```
>>> print(autos
... .with_columns(pl.col('make').cast(pl.String))
... .sort(by=pl.col('make').str.len_chars())
... )
shape: (41_144, 15)
+-----+-----+-----+-----+-----+-----+
| year | make | model | . | createdOn | is_autom | num_gears |
| i16  | str   | cat    | . | datetime[ | bool     | i8       |
|      |        |        |   | μs]       |          |           |
+-----+-----+-----+-----+-----+-----+
| 1993 | BMW   | 740i   | . | 2013-01-0 | true     | 5         |
|      |        |        |   | 1          |          |           |
|      |        |        |   | 00:00:00  |          |           |
| 1993 | BMW   | 740il  | . | 2013-01-0 | true     | 5         |
|      |        |        |   | 1          |          |           |
|      |        |        |   | 00:00:00  |          |           |
| 1993 | BMW   | 750il  | . | 2013-01-0 | true     | 4         |
|      |        |        |   | 1          |          |           |
|      |        |        |   | 00:00:00  |          |           |
| .    | .     | .     | . | .         | .       | .         |
| 1984 | Superior | Funeral | . | 2013-01-0 | true     | 3         |
|       | Coaches   | Coach   |   | 1          |          |           |
|       | Div E.p.  | 2WD     |   | 00:00:00  |          |           |
|       | Dutton    |          |   |           |          |           |
| 1989 | Environmen | ERD 1  | . | 2013-01-0 | true     | 4         |
|       | tal Rsch   |          |   | 1          |          |           |
|       | and Devp   |          |   | 00:00:00  |          |           |
|       | Corp      |          |   |           |          |           |
| 1984 | S and S   | Funeral | . | 2013-01-0 | true     | 3         |
|       | Coach     | Coach   |   | 1          |          |           |
|       | Company   | 2WD     |   | 00:00:00  |          |           |
|       | E.p. Dutt. |          |   |           |          |           |
+-----+-----+-----+-----+-----+-----+
```

We can also sort the `autos` dataframe by `year` without providing a custom sorting expression.

```
>>> print(autos
... .sort('year')
... )
shape: (41_144, 15)
   year      make      model      .  createdOn      is_automat  num_gears
   i16        cat        cat        .  datetime[    tic          i8
                                         μs]           bool

1984  Alfa Romeo  Spider       .  2013-01-0  false        5
      Veloce      1  00:00:00
      2000         1  00:00:00
1984  Bertone     X1/9       .  2013-01-0  false        5
      1  00:00:00
1984  Chevrolet   Corvette   .  2013-01-0  true         4
      1  00:00:00
      .
2020  Ford        Explorer   .  2019-06-0  true        10
      AWD          4  00:00:00
2020  Lincoln     Aviator    .  2019-06-0  true        10
      AWD          4  00:00:00
2020  Toyota      Prius     .  2019-06-0  true      null
      Prime        4  00:00:00
```

Let's sort the *autos* dataframe by the average of city and highway miles per gallon.

```
>>> print(autos
... .sort((pl.col('city08') + pl.col('highway08'))/2)
... )
shape: (41_144, 15)
   year      make      model      .  createdOn      is_automat  num_gears
   i16        cat        cat        .  datetime[    tic          i8
                                         μs]           bool

2018  Tesla     Model 3       .  2018-01-11  true        1
      Long Range   00:00:00
2019  Tesla     Model 3       .  2019-01-29  true        1
      Long Range   00:00:00
2019  Tesla     Model 3       .  2019-05-02  true        1
      Standard     00:00:00
      Range
```

6. Sorting and Ordering in Polars

.
2016	BMW	i3 BEV	.	2016-01-05	true	1	
			.	00:00:00			
2017	BMW	i3 BEV (60 Amp-hour battery)	.	2016-08-10	true	1	
			.	00:00:00			
2017	Tesla	Model 3 Long Range	.	2017-11-28	true	1	
			.	00:00:00			

Let's sort by a categorical column, `VClass`.

```
>>> print(autos
... .sort(pl.col('VClass'))
... )
```

shape: (41_144, 15)

year	make	model	.	createdOn	is_automatic	num_gears
i16	cat	cat	.	datetime[μs]	bool	i8
1985	Alfa Romeo	Spider	.	2013-01-01	false	5
		Veloce	.	1		
		2000	.	00:00:00		
1985	Ferrari	Testarossa	.	2013-01-01	false	5
		a	.	00:00:00		
1994	Acura	NSX	.	2013-01-01	true	4
			.	00:00:00		
.
2020	Hyundai	Santa Fe FWD	.	2019-06-04	true	8
			.	00:00:00		
2020	Lexus	NX 300	.	2019-06-04	true	6
			.	00:00:00		
2020	Mitsubishi	Outlander 2WD	.	2019-06-04	true	6
			.	00:00:00		

Let's sort by the average of the `city08` for each make. First, I will check that I can compute the average of `city08` for each make. Because I don't want the results to be aggregated by `make`, I will use the `.over` method to compute for the makes but preserve the original rows.

```
>>> print(autos
... .with_columns(make_avg=pl.col('city08').mean().over(pl.col('make'))))
```

6.3. Sorting by a Single Column

...)

shape: (41_144, 16)

year	make	model	.	is_automat	num_gears	make_avg
i16	cat	cat	.	tic	i8	f64
				bool		
1985	Alfa Romeo	Spider	.	false	5	18.31
		Veloce				
		2000				
1985	Ferrari	Testarossa	.	false	5	11.32
1985	Dodge	Charger	.	false	5	15.46
.
1993	Subaru	Legacy AWD	.	true	4	20.32
1993	Subaru	Legacy AWD	.	false	5	20.32
1993	Subaru	Legacy AWD	.	true	4	20.32
		Turbo				

That looks like it worked. Now, let's sort by the average of *city08* for each make.

```
>>> print(autos
... .sort(pl.col('city08').mean().over(pl.col('make'))))
... )
```

shape: (41_144, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[μ s]	ic bool	i8
1996	Vector	Avtech SC /	.	2013-01-01 00:00:00	false	5
		M12				
1997	Vector	Avtech SC /	.	2013-01-01 00:00:00	false	5
		M12				
1992	Vector	W8	.	2013-01-01 00:00:00	true	3
.
2019	Tesla	Model S	.	2019-05-02 00:00:00	true	1
		Long Range				
2019	Tesla	Model S	.	2019-05-02 00:00:00	true	1
		Performance				
		(19"				
		Wheels)				
2019	Tesla	Model S	.	2019-05-02 00:00:00	true	1
		Performance				
		(21"				
		Wheels)				

6. Sorting and Ordering in Polars

6.4 Sorting by Multiple Columns

In the previous section, we learned how to sort by a single column. However, we saw that the interface for the `.sort` method supports an arbitrary number of columns to sort by, with the `*more_by` argument. Let's see how to sort by multiple columns.

Let's sort by `year` and then by `make`.

```
>>> print(autos
... .sort(['year', 'make'])
... )
shape: (41_144, 15)
| year      make      model      .  createdOn      is_automotive      num_gears |
| i16       cat        cat        .  datetime[μs]    bool          i8           |
1984     Alfa Romeo   Spider     .  2013-01-0   false         5
                  Veloce     1
                  2000      00:00:00
1984     Alfa Romeo   GT V6 2.5  .  2013-01-0   false         5
                  1
                  00:00:00
1984     Alfa Romeo   Spider     .  2013-01-0   false         5
                  Veloce     1
                  2000      00:00:00
.
2020     Genesis      G80 RWD   .  2019-06-0   true          8
                  4
                  00:00:00
2020     Genesis      G80 AWD   .  2019-06-0   true          8
                  4
                  00:00:00
2020     Genesis      G80 RWD   .  2019-06-0   true          8
                  4
                  00:00:00
```

We'd probably want to include `model` as well. Let's sort by `year`, `make`, and `model`.

```
>>> print(autos
... .sort(['year', 'make', 'model'])
... )
shape: (41_144, 15)
| year      make      model      .  createdOn      is_automotive      num_gears |
| i16       cat        cat        .  datetime[μs]    bool          i8           |
```

6.4. Sorting by Multiple Columns

1984	Alfa Romeo	Spider	.	2013-01-0	false	5
		Veloce	1			
		2000	00:00:00			
1984	Alfa Romeo	Spider	.	2013-01-0	false	5
		Veloce	1			
		2000	00:00:00			
1984	Alfa Romeo	GT V6 2.5	.	2013-01-0	false	5
		1	00:00:00			
.
2020	Genesis	G70 RWD	.	2019-04-1	true	8
		6	00:00:00			
2020	Genesis	G70 RWD	.	2019-04-1	false	6
		6	00:00:00			
2020	Genesis	G70 RWD	.	2019-05-0	true	8
		2	00:00:00			

Let's sort by *year* in descending order and then by *make* and *model* in ascending order. We need to pass a sequence of booleans to the *descending* argument. We would pass [True, False, False] for our example.

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[tic	i8
				us]	bool	
2020	Subaru	Ascent	.	2019-05-1	true	8
		6	00:00:00			
2020	Subaru	Ascent	.	2019-05-1	true	8
		Limited/T	6	00:00:00		
		ouring	00:00:00			
2020	Toyota	Corolla	.	2019-02-0	true	10
		6	00:00:00			
.
1984	Import Foreign	1fas 410	.	2013-01-0	false	4
		1	00:00:00			

6. Sorting and Ordering in Polars

	Auto Sales			00:00:00		
	Inc					
1984	S and S	Funeral	.	2013-01-0	true	3
	Coach	Coach		1		
	Company	2WD		00:00:00		
	E.p. Dutt.					
1984	Superior	Funeral	.	2013-01-0	true	3
	Coaches	Coach		1		
	Div E.p.	2WD		00:00:00		
	Dutton					

That looks like it didn't work. The problem is that the `make` and `model` columns are categorical but not ordered. We need to set the order for them. We can cast it to an ordered categorical with `cast(pl.Categorical('lexical'))` to set the ordering to *lexical ordering* (alphabetic ordering). Let's try again.

```
>>> print(autos
... .with_columns(pl.col('make').cast(pl.Categorical('lexical')),
...                 pl.col('model').cast(pl.Categorical('lexical'))))
... .sort(['year', 'make', 'model'],
...       descending=[True, False, False])
... )
```

shape: (41_144, 15)

year	make	model	.	createdOn	is_automatic	num_gears
i16	cat	cat	.	datetime[μs]	bool	i8
2020	Acura	RDX AWD	.	2019-05-02 00:00:00	true	10
2020	Acura	RDX AWD	.	2019-05-02 A-SPEC 00:00:00	true	10
2020	Acura	RDX FWD	.	2019-05-02 00:00:00	true	10
.
1984	Volvo	760 GLE	.	2013-01-01 00:00:00	false	5
1984	Volvo	760 GLE	.	2013-01-01 00:00:00	true	4
1984	Volvo	760 GLE	.	2013-01-01 00:00:00	true	4

Much better.

6.5 Specifying Custom Ordering for Categorical Columns

In the previous section, we saw how to set the ordering for categorical columns. We used the `cast(pl.Categorical('lexical'))` to set the ordering to

6.5. Specifying Custom Ordering for Categorical Columns

lexical ordering. You can also specify `ordering='physical'` to set the ordering to the physical ordering of the categories in the column. This is the default ordering for categorical columns.

Let's assume we have a column with month names, and we want to sort by month.

```
>>> print(autos
... .with_columns(month=pl.col('createdOn').dt.strftime('%B'))
... .sort('month')
... )
shape: (41_144, 16)
| year      make      model      .  is_automat  num_gears  month
| 116      cat        cat        .    bool        18          str
|               .           .           .           .           .
| 2014      Porsche   Panamera S  .  true         8          April
|           .           E-Hybrid
| 2014      MINI      Cooper     .  true         6          April
|           .           (3-doors)
| 2014      MINI      Cooper     .  false        6          April
|           .           (3-doors)
|           .
|           .
| 2019      Ford       Explorer   .  true         6          September
|           .           AWD
| 2019      Jeep       Grand     .  true         8          September
|           .           Cherokee
|           .           SRT 4WD
| 2019      Chrysler   Pacifica  .  true         null        September
|           .           Hybrid
```

A naive approach would be to sort by the month names. That does not work because it orders the months alphabetically. We need them to be in the order of the months, January, February, March, etc.

The process to do this is a little unintuitive. We need to create a *string cache*. A string cache can be used in a context manager that allows us to temporarily create categoricals with a custom ordering. We need to create a Series with a categorical type and pass in the categories in the order to sort them. This is the 'physical' ordering and the default ordering that the categorical type uses for sorting.

Next, while inside the context manager, we can cast a column with those same values to a categorical column. This will use the ordering we specified in the string cache.

```
>>> month_order = ['January', 'February', 'March', 'April', 'May', 'June',
...                   'July', 'August', 'September', 'October', 'November',
...                   'December']
```

6. Sorting and Ordering in Polars

```
>>> with pl.StringCache():
...     autos = pl.Series(month_order).cast(pl.Categorical)
...     print(autos
...         .with_columns(month=pl.col('createdOn').dt.strftime('%B')
...                     .cast(pl.Categorical))
...         .sort('month'))
...
shape: (41_144, 16)
   year    make      model      .  is_automati num_gears  month
   i16      cat        cat      .      tic       i8      cat
                                         bool
1985  Alfa Romeo  Spider      .  false        5  January
                                         Veloce
                                         2000
1985  Ferrari    Testarossa  .  false        5  January
1985  Dodge      Charger     .  false        5  January
...
2019  Lincoln    Navigator   .  true         10 December
                                         4WD
2019  Fiat        500e       .  true         1  December
2019  Hyundai    Sonata     .  true         6  December
                                         Plug-in
                                         Hybrid
```

Alternatively, we can use `@pl.StringCache` as a decorator to create a string cache. If we create a categorical column while in this decorated function, it will use the ordering we specified.

```
>>> month_order = ['January', 'February', 'March', 'April', 'May', 'June',
...                  'July', 'August', 'September', 'October', 'November',
...                  'December']
>>> @pl.StringCache()
... def create_month_order():
...     s = pl.Series(month_order).cast(pl.Categorical)
...     return (autos
...             .with_columns(month=pl.col('createdOn').dt.strftime('%B')
...                         .cast(pl.Categorical)))
...
...
>>> print(create_month_order().sort('month'))
shape: (41_144, 16)
   year    make      model      .  is_automati num_gears  month
   i16      cat        cat      .      tic       i8      cat
                                         bool
```

6.5. Specifying Custom Ordering for Categorical Columns

1985	Alfa Romeo	Spider	.	false	5	January	
		Veloce	.				
		2000	.				
1985	Ferrari	Testarossa	.	false	5	January	
1985	Dodge	Charger	.	false	5	January	
.	
2019	Lincoln	Navigator	.	true	10	December	
		4WD	.				
2019	Fiat	500e	.	true	1	December	
2019	Hyundai	Sonata	.	true	6	December	
		Plug-in					
		Hybrid					

From looking at the code, this seems a little weird. How are we tying the `s` series back into the `month` column? `StringCache` is doing the magic for us. Because the new column has the same value as the series, polars uses the series order.

With this in mind, I could go back to my initial `tweak_auto` function and create a new series that has the makes in alphabetic order and then cast the `make` column to a category. But it is probably clearer (if I'm just doing alphabetic order) to set the ordering to '`lexical`' on the `.cast` call.

```
>>> import polars as pl
>>> path = 'data/vehicles.csv'
>>> raw = pl.read_csv(path, null_values=['NA'])

>>> @pl.StringCache()
... def tweak_auto(df):
...     cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
...             'drive', 'VClass', 'fuelType', 'barrels08', 'city08',
...             'highway08', 'createdOn']
...     return (df
...             .select(pl.col(cols))
...             .with_columns(pl.col('year').cast(pl.Int16),
...                         pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                         pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                         pl.col('make').cast(pl.Categorical('lexical')),
...                         pl.col(['model', 'VClass', 'drive', 'fuelType'])
...                             .cast(pl.Categorical),
...                         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...                         is_automatic=pl.col('trany').str.contains('Auto'),
...                         num_gears=pl.col('trany').str.extract(r'(\d+)').cast(pl.Int8)
...                     )
...             )
```

6. Sorting and Ordering in Polars

```
>>> autos = tweak_auto(raw)
```

Let's make sure that worked.

```
>>> print(tweak_auto(raw))
... .sort('make'))
```

shape: (41_144, 15)

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[tic	i8
				μs]	bool	
1985	AM General	Post	.	2013-01-0	true	3
		Office	.	1		
		DJ5 2WD	.	00:00:00		
1985	AM General	Post	.	2013-01-0	true	3
		Office	.	1		
		DJ8 2WD	.	00:00:00		
1984	AM General	FJ8c Post	.	2013-01-0	true	3
		Office	.	1		
			.	00:00:00		
2018	smart	fortwo	.	2017-12-0	true	1
		electric	.	5		
		drive con	.	00:00:00		
2019	smart	EQ fortwo	.	2019-01-2	true	1
		(coupe)	.	9		
			.	00:00:00		
2019	smart	EQ fortwo	.	2019-01-2	true	1
		(convertible)	.	9		
			.	00:00:00		

6.6 Enums and Ordering

If you happen to know the ordering of categorical columns ahead of time and have multiple dataframes with the same categories, this is a case where you can use an `Enum`.

Enums are slightly more efficient than categorical types because the encoding is more straightforward. We need to define them using `pl.Enum`.

A good pedagogical example to demonstrate order and categories is the months of the year. Generally, when we sort them, we want them to be in chronological order (i.e., January, February, March, etc.), and not in alphabetic order (i.e., April, August, February, etc.).

Let's make a dataset of birthdays to explore enums:

```
>>> import io
>>> data = '''Name,Birthday
... Brianna Smith,2000-02-16
... Alex Johnson,2001-01-15
... Carlos Gomez,2002-03-17
... Diana Clarke,2003-04-18
... Ethan Hunt,2002-05-19
... Fiona Gray,2005-06-20
... George King,2006-07-21
... Hannah Scott,2007-08-22
... Ian Miles,2008-09-23
... Julia Banks,2009-10-24'''
```

```
>>> students = pl.read_csv(io.StringIO(data))
>>> print(students)
```

shape: (10, 2)

Name	Birthday
str	str
Brianna Smith	2000-02-16
Alex Johnson	2001-01-15
Carlos Gomez	2002-03-17
.	.
Hannah Scott	2007-08-22
Ian Miles	2008-09-23
Julia Banks	2009-10-24

Let's add a month column and sort by month. First, we will need to convert the string data from the *Birthday* column to a date. Then, we can use `.dt.strptime` to retrieve the month name. Finally, we will use the `.sort` method.

```
>>> bday = pl.col('Birthday')
>>> print(students)
... .with_columns(bday.str.to_datetime('%Y-%m-%d'))
... .with_columns(month=bday.dt.strftime('%B'))
... .sort('month')
... )
```

shape: (10, 3)

Name	Birthday	month
str	datetime[μs]	str
Diana Clarke	2003-04-18 00:00:00	April
Hannah Scott	2007-08-22 00:00:00	August
Brianna Smith	2000-02-16 00:00:00	February
.	.	.

6. Sorting and Ordering in Polars

Ethan Hunt	2002-05-19 00:00:00	May
Julia Banks	2009-10-24 00:00:00	October
Ian Miles	2008-09-23 00:00:00	September

Note that the order of the `month` column is alphabetical. We want them in chronological order. Let's create an `Enum` with the months in the desired order.

```
>>> month_type = pl.Enum(['January', 'February', 'March', 'April', 'May',
...     'June', 'July', 'August', 'September', 'October', 'November',
...     'December'])
```

After we have defined an enum, we can cast to it. If we sort them by month, they will be in the chronological order we specified above.

```
>>> print(students
...     .with_columns(bday.str.to_datetime('%Y-%m-%d'))
...     .with_columns(month=bday.dt.strftime('%B').cast(month_type))
...     .sort('month')
... )
```

shape: (10, 3)

Name str	Birthday datetime[μs]	month enum
Alex Johnson	2001-01-15 00:00:00	January
Brianna Smith	2000-02-16 00:00:00	February
Carlos Gomez	2002-03-17 00:00:00	March
.	.	.
Hannah Scott	2007-08-22 00:00:00	August
Ian Miles	2008-09-23 00:00:00	September
Julia Banks	2009-10-24 00:00:00	October

Here's another dataset with holidays. Let's assume we want to merge a holiday for each birthday month.

```
>>> holiday_data = '''Month,Holiday
... January,New Year's Day
... February,Valentine's Day
... March,St. Patrick's Day
... April,April Fools' Day
... May,Memorial Day
... June,Juneteenth
... July,Independence Day
... August,Labor Day
... September,Patriot Day
... October,Halloween
... November,Thanksgiving
... December,Christmas Day'''
```

```
>>> holidays = pl.read_csv(io.StringIO(holiday_data))
>>> print(holidays)
shape: (12, 2)
+-----+
| Month      Holiday |
| str        str    |
+-----+
| January    New Year's Day |
| February   Valentine's Day |
| March      St. Patrick's Day |
| .          .           |
| October    Halloween     |
| November   Thanksgiving |
| December   Christmas Day |
+-----+
```

If we try and merge the month columns as strings and sort by month, we are back to lexical ordering.

```
>>> print(students
...     .with_columns(bday.str.to_datetime('%Y-%m-%d'))
...     .with_columns(month=bday.dt.strftime('%B'))
...     .join(holidays, left_on='month', right_on='Month')
...     .sort('month'))
shape: (10, 4)
+-----+
| Name       Birthday            month      Holiday   |
| str        datetime[μs]        str        str       |
+-----+
| Diana Clarke 2003-04-18 00:00:00 April     April Fools' Day |
| Hannah Scott  2007-08-22 00:00:00 August    Labor Day      |
| Brianna Smith 2000-02-16 00:00:00 February  Valentine's Day |
| .          .           .           .           .           |
| Ethan Hunt    2002-05-19 00:00:00 May      Memorial Day    |
| Julia Banks   2009-10-24 00:00:00 October   Halloween     |
| Ian Miles     2008-09-23 00:00:00 September Patriot Day    |
+-----+
```

If one of the columns is a categorical, we will see a `ComputeError`:

```
>>> print(students
...     .with_columns(bday.str.to_datetime('%Y-%m-%d'))
...     .with_columns(month=bday.dt.strftime('%B').cast(pl.Categorical))
...     .join(holidays, left_on='month', right_on='Month')
...     .sort('month'))
Traceback (most recent call last)
...
ComputeError: datatypes of join keys don't match - `month`: cat on left
                  does not match `Month`: str on right
```

6. Sorting and Ordering in Polars

If both merge columns are categoricals defined outside of a string cache, we get a `CategoricalRemappingWarning` warning. Also, the ordering is based on the appearance of the categories in the `students` data. The February date came first in the data, the first month in the physical ordering.

```
>>> print(students
...     .with_columns(bday.str.to_datetime('%Y-%m-%d'))
...     .with_columns(month=bdy.dt.strftime('%B').cast(pl.Categorical))
...     .join(holidays.with_columns(pl.col('Month').cast(pl.Categorical)),
...           left_on='month', right_on='Month')
...     .sort('month')))
shape: (10, 4)
```

Name str	Birthday datetime[μs]	month cat	Holiday str
Brianna Smith	2000-02-16 00:00:00	February	Valentine's Day
Alex Johnson	2001-01-15 00:00:00	January	New Year's Day
Carlos Gomez	2002-03-17 00:00:00	March	St. Patrick's Day
.	.	.	.
Hannah Scott	2007-08-22 00:00:00	August	Labor Day
Ian Miles	2008-09-23 00:00:00	September	Patriot Day
Julia Banks	2009-10-24 00:00:00	October	Halloween

If we cast both columns to the month enum, we get the desired behavior. The months are now sorted in chronological order.

```
>>> print(students
...     .with_columns(bday.str.to_datetime('%Y-%m-%d'))
...     .with_columns(month=bdy.dt.strftime('%B').cast(month_type))
...     .join(holidays.with_columns(pl.col('Month').cast(month_type)),
...           left_on='month', right_on='Month')
...     .sort('month')))
shape: (10, 4)
```

Name str	Birthday datetime[μs]	month enum	Holiday str
Alex Johnson	2001-01-15 00:00:00	January	New Year's Day
Brianna Smith	2000-02-16 00:00:00	February	Valentine's Day
Carlos Gomez	2002-03-17 00:00:00	March	St. Patrick's Day
.	.	.	.
Hannah Scott	2007-08-22 00:00:00	August	Labor Day
Ian Miles	2008-09-23 00:00:00	September	Patriot Day
Julia Banks	2009-10-24 00:00:00	October	Halloween

The enum functionality available in Polars is the preferred mechanism for handling low cardinality categorical data when you are concerned with the order, and you know the values ahead of time.

6.7 Group Ordering and maintain_order

Note that if we group by `make` (which we just set the order on), the results don't come back with `make` ordered. In fact, if you run this multiple times, the order of the rows will change.

```
>>> print(autos
... .group_by(pl.col('make'))
... .len()
...
shape: (136, 2)
| make           len
| cat            u32
|-
| Federal Coach 14
| Qvale          1
| Scion          84
| .
| Panther Car Company Limited 4
| Fisker          1
| Genesis         43
```

You can make the order consistent by passing `maintain_order=True`. However, this The results are sorted based on the order of the rows in the initial data, not on the order of the categoricals that we grouped by.

```
>>> print(autos
... .group_by(pl.col('make'), maintain_order=True)
... .len()
...
shape: (136, 2)
| make           len
| cat            u32
|-
| Alfa Romeo    62
| Ferrari       229
| Dodge          2583
| .
| Goldacre       1
| Isis Imports Ltd 1
| PAS Inc - GMC 2
```

If you want the results in the order of the categorical, you need to sort the results after.

```
>>> print(autos
... .group_by(pl.col('make'))
```

6. Sorting and Ordering in Polars

```
... .len()
... .sort('make')
...
shape: (136, 2)
+-----+
| make          len |
| cat           u32 |
+-----+
| AM General    6  |
| ASC Incorporated 1 |
| Acura         347 |
| .              .  |
| Wallace Environmental 32 |
| Yugo          8  |
| smart         38 |
+-----+
```

6.8 Stable Sorting

Many sort algorithms have the property of being *stable*. A stable sort algorithm maintains the relative order of equal elements. For example, let's say we have a list of students and their grades, and this list is sorted by first name. If we sort the list by grade, a stable sort algorithm will maintain the relative order of students with the same grade. An unstable sort algorithm will not guarantee this.

This makes sorting more predictable. You can do multiple sorts and the relative order of equal elements will be maintained.

```
>>> students = pl.DataFrame({
...     'name': ['Alice', 'Bob', 'Charlie', 'Dana', 'Eve'],
...     'age': [25, 20, 25, 21, 24],
...     'grade': [88, 92, 95, 88, 60],
... })
```

```
>>> print(students)
shape: (5, 3)
+-----+
| name      age   grade |
| str       i64   i64  |
+-----+
| Alice    25    88  |
| Bob     20    92  |
| Charlie  25    95  |
| Dana    21    88  |
| Eve     24    60  |
+-----+
```

Let's sort the students by *age*. Notice that *Alice* and *Charlie* have the same age. In the original data, *Alice* comes before *Charlie*. Let's see if that relative order is maintained after sorting by *age*.

```
>>> print(students
...     .sort('age')
... )
shape: (5, 3)
+-----+
| name    age   grade |
| str      i64   i64  |
+-----+
| Bob     20    92   |
| Dana   21    88   |
| Eve    24    60   |
| Alice   25    88   |
| Charlie 25    95   |
+-----+
```

Let's try again. This time will sort by *grade*. Both *Alice* and *Dana* have the same grade. Let's see if the relative order of *Alice* and *Dana* is maintained after sorting by *grade*.

```
>>> print(students
...     .sort('grade')
... )
shape: (5, 3)
+-----+
| name    age   grade |
| str      i64   i64  |
+-----+
| Eve     24    60   |
| Alice   25    88   |
| Dana   21    88   |
| Bob     20    92   |
| Charlie 25    95   |
+-----+
```

We can see by these examples that Polars uses a stable sort algorithm.

6.9 Sorting and Filtering

Because Polars has a query planner, it can do smart things if it knows a column is sorted. For example, if we know that a column is sorted, we can use binary search to find the first and last index of a value. This can be useful for filtering. Let's try it out.

First, we will see if we can find all of the 1994 cars without sorting the *year* column.

```
>>> %%timeit
>>> (autos.filter(pl.col('year') == 1994))
329 µs ± 3.89 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Now, let's make a new dataframe that is sorted by *year* and try to find all of the 1994 cars.

6. Sorting and Ordering in Polars

```
>>> autos_year_sorted = autos.sort('year')
```

Let's time the same filter as before but with the *year* column sorted.

```
>>> %%timeit
>>> (autos_year_sorted.filter(pl.col('year') == 1994))
60.9 µs ± 4.2 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

With this small dataset, we see a pretty big difference. The difference could be more pronounced with a larger dataset.

Let's try searching for all Ford cars with three dataframes:

- Categorical *make* column
- String *make* column
- Sorted string *make* column

```
>>> cat_make = autos
>>> string_make = autos.with_columns(make=pl.col('make').cast(pl.String))
>>> sorted_make = (autos
...     .with_columns(make=pl.col('make').cast(pl.String))
...     .sort('make'))
```

Now, let's time searching for all Ford cars with the three dataframes.

```
>>> %%timeit
>>> cat_make.filter(pl.col('make') == 'Ford')
84.2 µs ± 1.35 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
each)

>>> %%timeit
>>> string_make.filter(pl.col('make') == 'Ford')
135 µs ± 2.17 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

>>> %%timeit
>>> sorted_make.filter(pl.col('make') == 'Ford')
109 µs ± 2.62 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

These results speak for themselves. If you need to filter a column multiple times, it might be worth it to sort the column first. Alternatively, if the string column has duplicate values, it might be worth converting it to a categorical column to save both memory and speed up filtering.

6.10 Chapter Methods and Functions

Table 6.1: Chapter functions and methods

Method	Description
<code>df.sort(by, *more_by, descending=False, nulls_last=False)</code>	Sort this DataFrame by the specified columns. <code>descending</code> is a sequence of booleans for each column. If <code>more_by</code> is specified, sort by those as well. If <code>descending</code> is True, sort in descending order. If <code>nulls_last</code> is True, sort nulls last.
<code>col.str.len_chars()</code>	Compute the length of each string in this column.
<code>col.over(expr, *more_exprs, mapping_strategy ='group_to_rows')</code>	Create a window expression. Group over the specified expressions. The <code>mapping_strategy</code> parameter can be ' <code>group_to_rows</code> ', ' <code>join</code> ', or ' <code>explode</code> '. Generally, an aggregation function is used before the <code>.over</code> method.
<code>pl.StringCache()</code>	Context manager and decorator for creating a string cache.
<code>col.dt.strptime(format)</code>	Format the datetime column using the specified format.

6.11 Summary

In this chapter, we've explored sorting and ordering data in Polars. We started with basic sorting techniques and gradually moved towards more complex scenarios, including custom sorting functions and optimizing sorting operations for large datasets. The fuel economy dataset served as a practical guide, helping us understand real-world applications of these sorting methods. Remember, efficient data sorting is not just about arranging data; it's about uncovering insights and making data-driven decisions.

6.12 Exercises

1. Why is sorting an important operation in data analysis?
2. How does sorting data by multiple columns help in data analysis? Give an example using the fuel economy dataset.
3. Explain the difference between stable and unstable sorting. Why is this distinction important?
4. How can custom sorting functions be implemented in Polars? Create a function to sort electric cars by battery range.
5. Using the fuel economy dataset, sort the data by year in descending order and then by highway mpg in ascending order. Analyze the output.
6. Create a custom sorting criteria to rank vehicles based on a composite index of fuel efficiency and emissions.

6. Sorting and Ordering in Polars

-
7. For a given year, list all car models in ascending order of CO2 emission.
 8. Compare the sorting performance on different sizes of datasets. What observations can you make?

Chapter 7

Time Series Analysis

7.1 Introduction

Time series analysis is a key component of data analysis used in various applications, including finance, economics, and business. This chapter will explore how to analyze time series using Polars, with practical examples drawn from the hydrological data.

7.2 Loading the Dataset

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 river¹.

This data is a tab-delimited ASCII file in detail described here².

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.

¹https://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>

7. Time Series Analysis

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. I'll write some Python code to handle that for us because the `read_csv` function won't.

```
>>> import urllib.request

>>> def download_and_modify_url(url, local_filename):
...     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)
```

Let's load the data.

```
>>> import polars as pl

>>> def tweak_river(df_):
...     return (df_
...             .select('agency_cd', 'site_no', 'tz_cd',
...                     pl.col('datetime').str.to_datetime(),
...                     cfs=pl.col('144166_00060'),
...                     gage_height=pl.col('144167_00065').cast(pl.Float64)
...                 )
...             )

>>> raw = pl.read_csv('data/devilclean.txt', separator='\t')
>>> dd = tweak_river(raw)
>>> print(dd)
shape: (539_305, 6)
+-----+
| agency_cd    site_no    tz_cd    datetime      cfs    gage_height |
| str          i64        str      datetime[us]  f64    f64           |
+-----+
| USGS        9333500    MDT     2001-05-07   71.00  null          |
+-----+
```

			01:00:00			
USGS	9333500	MDT	2001-05-07	71.00	null	
			01:15:00			
USGS	9333500	MDT	2001-05-07	71.00	null	
			01:30:00			
.	
USGS	9333500	MDT	2020-09-28	9.20	6.15	
			09:00:00			
USGS	9333500	MDT	2020-09-28	9.20	6.15	
			09:15:00			
USGS	9333500	MDT	2020-09-28	9.20	6.15	
			09:30:00			

7.3 Converting to Dates

There are a few mechanisms for converting strings to dates in Polars: doing it while loading the data (in `pl.read_csv`), `.str.to_date`, `.str.to_datetime`, and `.str.strptime`.

Let's illustrate these functions with some examples.

First, let's try to read the data as a date. We use `try_parse_dates=True` in the `read_csv` function. This will try to convert all the columns to dates. If it can't, it will leave them as strings.

```
>>> print(pl.read_csv('data/devilclean.txt', separator='\t',
...                     try_parse_dates=True))
```

agency_cd	site_no	datetime	.	144166_00	144167_0	144167_0
str	i64	datetime[.	060_cd	0065	0065_cd
		μs]	.	str	str	str
USGS	9333500	2001-05-0	.	A:[91]	null	null
		7				
		01:00:00				
USGS	9333500	2001-05-0	.	A:[91]	null	null
		7				
		01:15:00				
USGS	9333500	2001-05-0	.	A:[91]	null	null
		7				
		01:30:00				
.
USGS	9333500	2020-09-2	.	P	6.15	P
		8				
		09:00:00				
USGS	9333500	2020-09-2	.	P	6.15	P
		8				
		09:15:00				

7. Time Series Analysis

USGS	9333500	2020-09-2 8 09:30:00	.	P	6.15	P
------	---------	----------------------------	---	---	------	---

Generally, I like to be more explicit and ensure that my dates are properly converted. However, converting dates while reading the data can be faster than converting them after the fact.

Note that casting the string to a pl.Datetime object using .cast will fail.

```
>>> format = '%Y-%m-%d %H:%M'
>>> print(raw)
... .select(original=pl.col('datetime'),
...         to_datetime=pl.col('datetime').str.to_datetime(format),
...         to_date=pl.col('datetime').str.to_date(format),
...         strftime=pl.col('datetime').str.strptime(pl.Datetime, format),
...         # line below fails
...         #cast=pl.col('datetime').cast(pl.Datetime)
...         )
...
... )
```

shape: (539_305, 4)

original str	to_datetime datetime[μs]	to_date date	strftime datetime[μs]
2001-05-07 01:00	2001-05-07 01:00:00	2001-05-07	2001-05-07 01:00:00
2001-05-07 01:15	2001-05-07 01:15:00	2001-05-07	2001-05-07 01:15:00
2001-05-07 01:30	2001-05-07 01:30:00	2001-05-07	2001-05-07 01:30:00
.	.	.	.
2020-09-28 09:00	2020-09-28 09:00:00	2020-09-28	2020-09-28 09:00:00
2020-09-28 09:15	2020-09-28 09:15:00	2020-09-28	2020-09-28 09:15:00
2020-09-28 09:30	2020-09-28 09:30:00	2020-09-28	2020-09-28 09:30:00

7.4 Combining Columns to Create Dates

Let's assume we want to create a date from two string columns, *month* and *year*. Since we don't have a time component in this example (ignore that the *to_datetime* column already exists), so let's create a day for the first of each month.

```
>>> print(raw)
... .select(to_datetime=pl.col('datetime').str.to_datetime(format))
```

```
... .with_columns(month=pl.col('to_datetime').dt.strftime('%m'),
...                  year=pl.col('to_datetime').dt.strftime('%Y'))
...
shape: (539_305, 3)
+-----+-----+-----+
| to_datetime      | month | year  |
| datetime[μs]    | str   | str   |
+-----+-----+-----+
| 2001-05-07 01:00:00 | 05   | 2001  |
| 2001-05-07 01:15:00 | 05   | 2001  |
| 2001-05-07 01:30:00 | 05   | 2001  |
| .               | .     | .     |
| 2020-09-28 09:00:00 | 09   | 2020  |
| 2020-09-28 09:15:00 | 09   | 2020  |
| 2020-09-28 09:30:00 | 09   | 2020  |
+-----+-----+-----+
```

We will use the `pl.date` function to combine the existing values into a date.

```
>>> print(raw
... .select(to_datetime=pl.col('datetime').str.to_datetime(format))
... .with_columns(month=pl.col('to_datetime').dt.strftime('%m'),
...                  year=pl.col('to_datetime').dt.strftime('%Y'))
... .select(pl.date(pl.col('year'), pl.col('month'), 1))
...
shape: (539_305, 1)
+-----+
| date |
| date |
+-----+
| 2001-05-01 |
| 2001-05-01 |
| 2001-05-01 |
| .       |
| 2020-09-01 |
| 2020-09-01 |
| 2020-09-01 |
```

7.5 Changing Timezones

If a column is a date type, you can inspect the current time zone by pulling off the `.time_zone` attribute of the `.dtype` property:

```
>>> print(tweak_river(raw)
... ['datetime']
... .dtype.time_zone
)
None
```

7. Time Series Analysis

The `datetime` column has no timezone information currently.

We can add a time zone to naive dates using the `.dt.replace_time_zone` method found on the column expression. Below, I will convert the times to UTC (which, to be pedantic, is not a time zone but a time standard). Once it is in UTC, we can convert it to a different time zone with `.dt.convert_time_zone`.

Note

See the `pytz.all_timezones` attribute from the `pytz` library for a list of valid time zones.

```
>>> import pytz
>>> pytz.all_timezones[:5]
['Africa/Abidjan',
 'Africa/Accra',
 'Africa/Addis_Ababa',
 'Africa/Algiers',
 'Africa/Asmara']
```

This data is from the *America/Denver* time zone, so let's try to get it in that format. Below are some examples. The correct time is the *Denver2* column. We need the information from the `tz_cd` column to get the correct time zone. Also, Polars has issues parsing the `tz_cd` column, so we need string manipulation to convert the names to offsets.

I will convert the `tz_cd` column to an offset and tack that onto the end of the `datetime` column. Then, I'll convert that to a date time that includes offset information. Finally, I'll convert that to the *America/Denver* time zone. See the *Denver2* column below.

```
>>> format = '%Y-%m-%d %H:%M'
>>> print(raw
... .select(original=pl.col('datetime'),
...         naive=pl.col('datetime').str.to_datetime(format),
...         utc=pl.col('datetime').str.to_datetime(format)
...             .dt.replace_time_zone('UTC'),
...         Denver=pl.col('datetime').str.to_datetime(format)
...             .dt.replace_time_zone('UTC')
...             .dt.convert_time_zone('America/Denver'),
...         Denver2=(pl.col('datetime') + ' ' + (pl.col('tz_cd')
...             .str.replace('MST', '-0700').str.replace('MDT', '-0600')))
...             .str.to_datetime('%Y-%m-%d %H:%M %z')
...             .dt.convert_time_zone('America/Denver'),
...         Denver3=(pl.col('datetime').str.to_datetime(format,
...             time_zone='America/Denver', ambiguous='earliest'))
...     )
... )
```

shape: (539_305, 6)					
original str	naive datetime[μs]	utc datetime[μs, UTC]	Denver datetime[μs, America/Denver]	Denver2 datetime[μs, America/Denver]	Denver3 datetime[μs, America/Denver]
2001-05-07 01:00	2001-05-07	2001-05-07	2001-05-07	2001-05-07	2001-05-07
	01:00:00	01:00:00	19:00:00	01:00:00	01:00:00
		UTC	MDT	MDT	MDT
2001-05-07 01:15	2001-05-07	2001-05-07	2001-05-07	2001-05-07	2001-05-07
	01:15:00	01:15:00	19:15:00	01:15:00	01:15:00
		UTC	MDT	MDT	MDT
2001-05-07 01:30	2001-05-07	2001-05-07	2001-05-07	2001-05-07	2001-05-07
	01:30:00	01:30:00	19:30:00	01:30:00	01:30:00
		UTC	MDT	MDT	MDT
.
2020-09-28 09:00	2020-09-28	2020-09-28	2020-09-28	2020-09-28	2020-09-28
	09:00:00	09:00:00	03:00:00	09:00:00	09:00:00
		UTC	MDT	MDT	MDT
2020-09-28 09:15	2020-09-28	2020-09-28	2020-09-28	2020-09-28	2020-09-28
	09:15:00	09:15:00	03:15:00	09:15:00	09:15:00
		UTC	MDT	MDT	MDT
2020-09-28 09:30	2020-09-28	2020-09-28	2020-09-28	2020-09-28	2020-09-28
	09:30:00	09:30:00	03:30:00	09:30:00	09:30:00
		UTC	MDT	MDT	MDT

Note

With this dataset, I can also do a non-failing date conversion with this code, passing in the `time_zone` and setting the `ambiguous` parameter.

If you use the `ambiguous` parameter, double-check your results. For example, I could also use `ambiguous='latest'`, and it would give me back results, but they would be erroneous. For this reason, I prefer the `Denver2` calculation we did above. I feel more comfortable with the results.

Here's some code to demonstrate how to double-check the results. You can see that the early morning times cause problems during the daylight savings transition.

7. Time Series Analysis

```
>>> format = '%Y-%m-%d %H:%M'
>>> print(raw
... .select(
...     Denver2=(pl.col('datetime') + ' ' + (pl.col('tz_cd')
...         .str.replace('MST', '-0700').str.replace('MDT', '-0600'))))
...     .str.to_datetime('%Y-%m-%d %H:%M %z')
...     .dt.convert_time_zone('America/Denver'),
...     Denver3=(pl.col('datetime').str.to_datetime(format,
...         time_zone='America/Denver', ambiguous='latest'))
...     )
... .filter(pl.col('Denver3') != pl.col('Denver2'))
... )
```

shape: (56, 2)

Denver2 datetime[μs, America/Denver]	Denver3 datetime[μs, America/Denver]
2003-10-26 01:00:00 MDT	2003-10-26 01:00:00 MST
2003-10-26 01:15:00 MDT	2003-10-26 01:15:00 MST
2003-10-26 01:30:00 MDT	2003-10-26 01:30:00 MST
.	.
2019-11-03 01:15:00 MDT	2019-11-03 01:15:00 MST
2019-11-03 01:30:00 MDT	2019-11-03 01:30:00 MST
2019-11-03 01:45:00 MDT	2019-11-03 01:45:00 MST

I will add this time zone conversion to the `tweak` function I wrote above. I'll also add code to convert the discharge and gage height columns to numeric values. I will parameterized the `cfs` and `gage_height` names.

```
import polars as pl

def tweak_river(df_, cfs_col, gage_height_col):
    return (df_
        .select(
            'agency_cd', 'site_no',
            cfs=pl.col(cfs_col),
            gage_height=pl.col(gage_height_col).cast(pl.Float64),
            datetime=(pl.col('datetime') + ' ' + (pl.col('tz_cd')
                .str.replace('MST', '-0700').str.replace('MDT', '-0600'))))
                .str.to_datetime('%Y-%m-%d %H:%M %z')
                .dt.convert_time_zone('America/Denver')
        )
    )

dd = tweak_river(raw, cfs_col='144166_00060',
                  gage_height_col='144167_00065')
```

7.6 Time Aggregations

We can get basic aggregations by time amounts using the `group_by` method. Below, I will make a new column called `year` and group by that. Then, I will aggregate by the mean of the floating point columns.

```
>>> print(dd
... .with_columns(year=pl.col('datetime').dt.year())
... .group_by('year')
... .agg(pl.col(pl.Float64).mean())
... )
shape: (20, 3)
+-----+
| year   cfs      gage_height |
| i32    f64      f64          |
+-----+
| 2008   80.60    3.58        |
| 2014   71.79    5.52        |
| 2010   81.93    3.14        |
| .       .         .           |
| 2009   53.67    3.15        |
| 2007   95.31    3.64        |
| 2006   413.24   null        |
+-----+
```

Note that the result is not sorted by year. Polars is multi-threaded, and the order of the results is not guaranteed.

Because this data is already sorted by date, we can use the `maintain_order` argument to `group_by` to keep the order.

```
>>> print(dd
... .with_columns(year=pl.col('datetime').dt.year())
... .group_by('year', maintain_order=True)
... .agg(pl.col(pl.Float64).mean())
... )
shape: (20, 3)
+-----+
| year   cfs      gage_height |
| i32    f64      f64          |
+-----+
| 2001   74.30    NaN         |
| 2002   116.21   NaN        |
| 2003   86.96    NaN        |
| .       .         .           |
| 2018   71.07    6.42        |
| 2019   92.89    6.67        |
| 2020   58.43    6.47        |
+-----+
```

7. Time Series Analysis

Polars also has a `.group_by_dynamic` method that allows you to group by a column that is specifically for grouping based on time. This allows you to specify windows based on a start location and period size.

Let's repeat the above code using `.group_by_dynamic`. If we just run it naively, we get an `InvalidOperationException`.

```
>>> (dd
...     .group_by_dynamic(index_column='datetime', every='1y')
...     .agg(pl.col(pl.Float64).mean())
... )
Traceback (most recent call last)
...
InvalidOperationException: argument in operation 'group_by_dynamic' is not
explicitly sorted
```

- If your data is ALREADY sorted, set the sorted flag with:
`'.set_sorted()'`.
- If your data is NOT sorted, sort the 'expr/series/column' first.

The query planner needs a hint to know that the `datetime` column is sorted. We can do this in two ways. First, we can use the `sort` method to sort the data by `datetime`.

```
>>> print(dd
...     .sort('datetime')
...     .group_by_dynamic(index_column='datetime', every='1y')
...     .agg(pl.col(pl.Float64).mean())
... )
shape: (20, 3)
+-----+-----+-----+
| datetime           | cfs    | gage_height |
| datetime[μs, America/Denver] | f64    | f64        |
+-----+-----+-----+
| 2001-01-01 00:00:00 MST | 74.30  | null       |
| 2002-01-01 00:00:00 MST | 116.21 | null      |
| 2003-01-01 00:00:00 MST | 86.96  | null      |
| .                 | .     | .          |
| 2018-01-01 00:00:00 MST | 71.07  | 6.42      |
| 2019-01-01 00:00:00 MST | 92.89  | 6.67      |
| 2020-01-01 00:00:00 MST | 58.43  | 6.47      |
+-----+-----+-----+
```

The second way is to give a hint to the query planner using the `.set_sorted` method. This method takes an argument indicating the column that is sorted. Because `datetime` is already sorted, I prefer to use this method.

```
>>> print(dd
...     .set_sorted('datetime')
...     .group_by_dynamic(index_column='datetime', every='1y')
```

```

... .agg(pl.col(pl.Float64).mean())
...
)
shape: (20, 3)
+-----+-----+-----+
| datetime           | cfs    | gage_height |
| datetime[μs, America/Denver] | f64    | f64        |
+-----+-----+-----+
| 2001-01-01 00:00:00 MST | 74.30  | null       |
| 2002-01-01 00:00:00 MST | 116.21 | null       |
| 2003-01-01 00:00:00 MST | 86.96  | null       |
| .                   | .      | .          |
| 2018-01-01 00:00:00 MST | 71.07  | 6.42       |
| 2019-01-01 00:00:00 MST | 92.89  | 6.67       |
| 2020-01-01 00:00:00 MST | 58.43  | 6.47       |
+-----+-----+-----+

```

Note that the column that we grouped by is a `datetime` column, not the integer year value. This is nice because we can do further date manipulations on the column.

7.7 Time Intervals

In the above example, I used the '`1y`' interval to group by years. Here is a table of the valid intervals:

Table 7.1: Intervals for the `every` and `period` parameters of the `group_by_dynamic` method.

Interval	Description
'ns'	nanoseconds
'us'	microseconds
'ms'	milliseconds
's'	seconds
'm'	minutes
'h'	hours
'd'	days
'w'	weeks
'mo'	months
'q'	quarters
'y'	years

Let's look at every 2 months. We will use the '`2mo`' interval:

```

>>> print(dd
... .set_sorted('datetime')
... .group_by_dynamic(index_column='datetime', every='2mo')
... .agg(pl.col(pl.Float64).mean())
... )

```

7. Time Series Analysis

shape: (117, 3)

datetime	cfs	gage_height
datetime[μs, America/Denver]	f64	f64
2001-05-01 00:00:00 MDT	42.00	null
2001-07-01 00:00:00 MDT	165.86	null
2001-09-01 00:00:00 MDT	43.78	null
.	.	.
2020-05-01 00:00:00 MDT	16.79	6.38
2020-07-01 00:00:00 MDT	14.58	5.91
2020-09-01 00:00:00 MDT	4.90	5.84

Let's do every 3 hours, 4 minutes, and 5 seconds:

```
>>> print(dd
... .set_sorted('datetime')
... .group_by_dynamic(index_column='datetime', every='3h4m5s')
... .agg(pl.col(pl.Float64).mean())
... )
```

shape: (44_796, 3)

datetime	cfs	gage_height
datetime[μs, America/Denver]	f64	f64
2001-05-06 23:03:20 MDT	70.60	null
2001-05-07 02:07:25 MDT	67.92	null
2001-05-07 05:11:30 MDT	64.46	null
.	.	.
2020-09-28 02:25:20 MDT	9.67	6.16
2020-09-28 05:29:25 MDT	9.51	6.16
2020-09-28 08:33:30 MDT	9.20	6.15

Let's do every seven days starting on Monday, but make the window size 5 to only get weekdays. Let's also calculate the range of *cfs* for each window:

```
>>> print(dd
... .set_sorted('datetime')
... .group_by_dynamic(index_column='datetime', every='7d', period='5d',
...                 start_by='monday', check_sorted=False)
... .agg(pl.col(pl.Float64).mean(),
...       cfs_range=(pl.col('cfs').max() - pl.col('cfs').min()))
... )
```

shape: (877, 4)

datetime	cfs	gage_height	cfs_range
datetime[μs, America/Denver]	f64	f64	f64
2001-05-07 00:00:00 MDT	62.19	null	29.00

2001-05-14 00:00:00 MDT	50.53	null	20.00
2001-05-21 00:00:00 MDT	43.37	null	17.00
.	.	.	.
2020-09-14 00:00:00 MDT	5.81	6.03	2.94
2020-09-21 00:00:00 MDT	8.32	6.12	6.13
2020-09-28 00:00:00 MDT	9.56	6.16	1.00

And here is the weekend flow:

```
>>> print(dd
... .set_sorted('datetime')
... .group_by_dynamic(index_column='datetime', every='7d', period='2d',
...                 start_by='saturday', check_sorted=False)
... .agg(pl.col(pl.Float64).mean(),
...       cfs_range=(pl.col('cfs').max() - pl.col('cfs').min()))
... )
shape: (837, 4)
```

datetime datetime[μs, America/Denver]	cfs f64	gage_height f64	cfs_range f64
2001-05-12 00:00:00 MDT	52.48	null	15.00
2001-05-19 00:00:00 MDT	49.18	null	18.00
2001-05-26 00:00:00 MDT	42.14	null	12.00
.	.	.	.
2020-09-12 00:00:00 MDT	5.93	6.00	3.83
2020-09-19 00:00:00 MDT	6.30	6.05	3.48
2020-09-26 00:00:00 MDT	10.63	6.19	5.87

Pandas has a `.transform` method to create a column that is the same length as the original DataFrame but based on the aggregation. Polars does not have this method, but we can use the `.over` method to create the same result. Sadly, `.over` does not work with time interval strings (but `.rolling`, which we will see soon, does). We will need to create the groups manually.

Let's do the average yearly flow but retain the original rows. We will also make a new column that is the percentage of the yearly average, called `pct_of_avg`. This will need to be in a separate context since the expressions in the context are evaluated at the same time, and we need to use the average from the first context in the second context.

```
>>> print(dd
... .with_columns(year=pl.col('datetime').dt.year())
... .with_columns(year_mean_cfs=pl.col('cfs').mean().over('year'))
... .with_columns(pct_of_avg=(pl.col('cfs') / pl.col('year_mean_cfs'))
...               .mul(100).round(2))
... )
shape: (539_305, 8)
```

7. Time Series Analysis

agency_cd	site_no	cfs	.	year	year_mean_cfs	pct_of_avg
str	i64	f64	.	i32	f64	f64
USGS	9333500	71.00	.	2001	74.30	95.55
USGS	9333500	71.00	.	2001	74.30	95.55
USGS	9333500	71.00	.	2001	74.30	95.55
.
USGS	9333500	9.20	.	2020	58.43	15.75
USGS	9333500	9.20	.	2020	58.43	15.75
USGS	9333500	9.20	.	2020	58.43	15.75

We can calculate multiple aggregations simultaneously by passing a list of aggregations to `pl.col`. In this example, we will calculate the average of the `cfs` and `gage_height` by year and quarter. I use the `.suffix` method to add a suffix to the column names to tell which column is which.

```
>>> print(dd
... .with_columns(year=pl.col('datetime').dt.year(),
...                 quarter=pl.col('datetime').dt.quarter())
... .with_columns(pl.col(['cfs', 'gage_height']).mean().over('year')
...                 .name.suffix('_mean_year'),
...                 pl.col(['cfs', 'gage_height']).mean().over('quarter')
...                 .name.suffix('_mean_quarter'))
... )
shape: (539_305, 11)
```

agency_cd	site_no	cfs	.	gage_height	cfs_mean_q	gage_height
str	i64	f64	.	t_mean_yea	quarter	ht_mean_q
			.	r	f64	quarter
USGS	9333500	71.00	.	null	73.40	4.65
USGS	9333500	71.00	.	null	73.40	4.65
USGS	9333500	71.00	.	null	73.40	4.65
.
USGS	9333500	9.20	.	6.47	90.29	4.46
USGS	9333500	9.20	.	6.47	90.29	4.46
USGS	9333500	9.20	.	6.47	90.29	4.46

7.8 Multiple Groupings with Timeseries

The `.group_by` method can take a list of columns to group by, but it doesn't work with time intervals. Upon first look, the `.group_by_dynamic` method doesn't seem to work with multiple columns. However, that is not the case; the functionality is hidden in the `by` parameter.

I will load the gage height data for the Snake River¹ in Wyoming. Then, I can compare the flow of the Dirty Devil River to the gage height of the Snake River.

```
>>> snake_raw = (pl.read_csv('data/nwis.waterservices.usgs.gov.txt',
...                           skip_rows=27, separator='\t',
...                           skip_rows_after_header=1))
>>> snake = (tweak_river(snake_raw
...                        .with_columns(cfs=pl.lit(None).cast(pl.Float64)),
...                        cfs_col='cfs', gage_height_col='319803_00065')
... )
```

If both dataframes have the same columns, I can use the `.vstack` method to stack them on top of each other. I'm also going to add a *source* column to each dataframe so that we can tell which data is which.

Note

While performing this operation, I discovered that there is no guarantee that `.with_columns` will maintain the order of the columns. I had to use the `.select` method to ensure that the columns were in the correct order. The `.vstack` method will fail if the columns are not in the same order.

```
>>> abc = pl.DataFrame({'b': [1,2,3],
...                      'c': [4,5,6],
...                      'a': [7,8,9]})

>>> print(abc.with_columns(['a', 'b', 'c']))
shape: (3, 3)
| b    c    a |
| i64  i64  i64 |
|-----|
| 1    4    7 |
| 2    5    8 |
| 3    6    9 |
```



```
>>> print(abc.select(['a', 'b', 'c']))
shape: (3, 3)
| a    b    c |
| i64  i64  i64 |
|-----|
| 7    1    4 |
| 8    2    5 |
| 9    3    6 |
```

Here's the stacked data.

¹<https://waterdata.usgs.gov/monitoring-location/13022500/>

7. Time Series Analysis

```
>>> common_cols = ['agency_cd', 'site_no', 'datetime',
...                 'cfs', 'gage_height', 'source']
>>> print(snake
...     .with_columns(source=pl.lit('snake'))
...     .select(common_cols)
...     .vstack(dd
...             .with_columns(source=pl.lit('devil'))
...             .select(common_cols)
...             )
... )
```

shape: (1_112_112, 6)

agency_cd	site_no	datetime	cfs	gage_height	source
str	i64	datetime[μs, America/Denver]	f64	f64	str
USGS	13022500	2007-10-01 01:00:00 MDT	null	4.48	snake
USGS	13022500	2007-10-01 01:15:00 MDT	null	4.50	snake
USGS	13022500	2007-10-01 01:30:00 MDT	null	4.47	snake
.
USGS	9333500	2020-09-28 09:00:00 MDT	9.20	6.15	devil
USGS	9333500	2020-09-28 09:15:00 MDT	9.20	6.15	devil
USGS	9333500	2020-09-28 09:30:00 MDT	9.20	6.15	devil

If I just group the data by the *datetime* column, I will get the aggregated average of both locations for *gage_height*. I want each location to be separate, so I will need to group by both the *datetime* and *source* columns. But the *index_column* parameter of the *group_by_dynamic* method only takes one column. I can use the *by* parameter add additional columns to group by.

```
>>> common_cols = ['agency_cd', 'site_no', 'datetime',
...                 'cfs', 'gage_height', 'source']
>>> print(snake
...     .with_columns(source=pl.lit('snake'))
...     .select(common_cols)
...     .vstack(dd
...             .with_columns(source=pl.lit('devil'))
...             .select(common_cols)
...             )
...     .sort('datetime')
...     .group_by_dynamic(index_column='datetime', every='1mo',
```

```

...           by='source')
...   .agg(pl.col('gage_height').mean())
...
shape: (423, 3)
+-----+
| source      datetime          gage_height |
| str         datetime[μs, America/Denver]  f64   |
+-----+
| devil      2001-05-01 00:00:00 MDT    null   |
| devil      2001-06-01 00:00:00 MDT    null   |
| devil      2001-07-01 00:00:00 MDT    null   |
| .          .                   .       .     |
| snake      2023-12-01 00:00:00 MST    2.95   |
| snake      2024-01-01 00:00:00 MST    2.89   |
| snake      2024-02-01 00:00:00 MST    2.65   |
+-----+

```

Note

It turns out that we can actually perform date aggregated grouping with `.group_by` and multiple columns. Instead of grouping by the date column, we use `.dt.truncate` to truncate the date to the desired level of aggregation.

```

>>> common_cols = ['agency_cd', 'site_no', 'datetime',
...                  'cfs', 'gage_height', 'source']
>>> print(snake
...       .with_columns(source=pl.lit('snake'))
...       .select(common_cols)
...       .vstack(dd
...                 .with_columns(source=pl.lit('devil'))
...                 .select(common_cols)
...                 )
...       .sort('datetime')
...       .group_by([pl.col('datetime').dt.truncate('1mo'), 'source'])
...       .agg(pl.col('gage_height').mean())
...     )
shape: (423, 3)
+-----+
| datetime          source  gage_height |
| datetime[μs, America/Denver]  str     f64   |
+-----+
| 2001-11-01 00:00:00 MST    devil   null   |
| 2001-12-01 00:00:00 MST    devil   null   |
| 2003-09-01 00:00:00 MDT    devil   null   |
| .                   .       .     .     |
| 2022-12-01 00:00:00 MST    snake   2.73   |
+-----+

```

7. Time Series Analysis

2023-05-01 00:00:00 MDT	snake	6.66
2024-01-01 00:00:00 MST	snake	2.89

You might wonder why we would use `.group_by_dynamic` at all. Two reasons. First, it is faster. Second, it retains the order of the date.

I can use a `.pivot` method to make the `source` column into separate columns. This format is very useful for plotting each location on the same graph as a separate line.

```
>>> common_cols = ['agency_cd', 'site_no', 'datetime',
...                 'cfs', 'gage_height', 'source']
>>> print(snake
...       .with_columns(source=pl.lit('snake'))
...       .select(common_cols)
...       .vstack(dd
...                 .with_columns(source=pl.lit('devil'))
...                 .select(common_cols)
...                 )
...       .sort('datetime')
...       .group_by_dynamic(index_column='datetime', every='1mo', by='source')
...       .agg(pl.col('gage_height').mean())
...       .pivot(columns='source', index='datetime', values='gage_height')
...     )
shape: (267, 3)
```

datetime	devil	snake
datetime[μs, America/Denver]	f64	f64
2001-05-01 00:00:00 MDT	null	null
2001-06-01 00:00:00 MDT	null	null
2001-07-01 00:00:00 MDT	null	null
.	.	.
2023-12-01 00:00:00 MST	null	2.95
2024-01-01 00:00:00 MST	null	2.89
2024-02-01 00:00:00 MST	null	2.65

I'm going to plot this with the `matplotlib` backend for `hvplot`. Here is the code to do that. This code has some rows where both `devil` and `snake` are missing gage height. I'm going to drop those rows using `pl.all_horizontal`.

```
>>> import hvplot
>>> hvplot.extension('matplotlib')
>>> common_cols = ['agency_cd', 'site_no', 'datetime',
...                 'cfs', 'gage_height', 'source']
>>> print(snake
...       .with_columns(source=pl.lit('snake'))
...       .select(common_cols)
```

```

... .vstack(dd
...     .with_columns(source=pl.lit('devil'))
...     .select(common_cols)
... )
... .sort('datetime')
... .group_by_dynamic(index_column='datetime', every='1mo', by='source')
... .agg(pl.col('gage_height').mean())
... .pivot(columns='source', index='datetime', values='gage_height')
... .filter(~pl.all_horizontal(pl.col('devil', 'snake').is_null()))
... .plot(x='datetime', y=['devil', 'snake'], rot=45, title='Gage Height',
...       width=1800, height=600)
... )

```

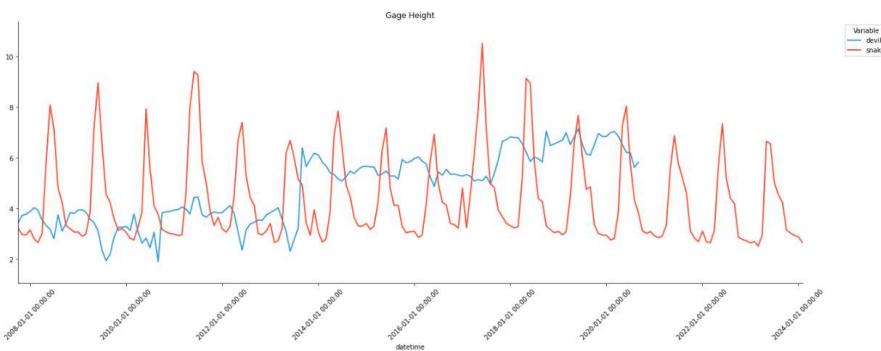


Figure 7.1: Plot of gage height for the Dirty Devil and Snake Rivers

7.9 Window Functions in Polars

Window functions are a powerful tool for performing complex data manipulations. You can think of a window function as a function that operates on a set of rows, where a window frame defines the set of rows. The result of a window function is a value for each row in the window frame.

An example of a window function is the *rolling average* function, which calculates the average of a set of rows related to the current row.

Table 7.2: Periods and offsets for the `period` parameter of the `rolling` method.

Period/Offset	Description
ns	nanoseconds
us	microseconds

7. Time Series Analysis

Period/Offset	Description
ms	milliseconds
s	seconds
m	minutes
h	hours
d	days
w	weeks
mo	months
q	quarters
y	years
i	index count

You can prefix the period with a number to specify the number of periods. For example, 3d is 3 days, 2w is 2 weeks, and 1y is 1 year.

Use the `index count` to do a rolling operation on an index column. To do every 10 rows, use `10i` on a monotonic index column.

Let's find the mean cfs over the months and add another column with the rolling average of the cfs over the last 3 months:

```
>>> print(dd
...     .set_sorted('datetime')
...     .group_by_dynamic(index_column='datetime', every='1mo')
...     .agg(pl.col('cfs').mean())
...     .with_columns(mean_cfs_3mo=pl.col('cfs').mean()
...                  .rolling('datetime', period='3mo'))
... )
shape: (226, 3)
datetime                                cfs      mean_cfs_3mo
datetime[μs, America/Denver]    f64        f64
+-----+-----+-----+
| 2001-05-01 00:00:00 MDT | 48.87 | 48.87 |
| 2001-06-01 00:00:00 MDT | 22.95 | 35.91 |
| 2001-07-01 00:00:00 MDT | 18.01 | 29.94 |
| .           .           .       |
| 2020-07-01 00:00:00 MDT | 25.04 | 19.31 |
| 2020-08-01 00:00:00 MDT | 4.12  | 13.13 |
| 2020-09-01 00:00:00 MDT | 4.90  | 11.36 |
```

Let's visualize the results. I'll use the Plotly backend this time.

```
>>> import hvplot
>>> hvplot.extension('plotly')
>>> (dd
...     .set_sorted('datetime')
...     .group_by_dynamic(index_column='datetime', every='1mo')
```

```
... .agg(pl.col('cfs').mean())
... .with_columns(mean_cfs_3mo=pl.col('cfs').mean()
...                 .rolling('datetime', period='3mo'))
... .plot(x='datetime', y=['cfs', 'mean_cfs_3mo'])
... )
```

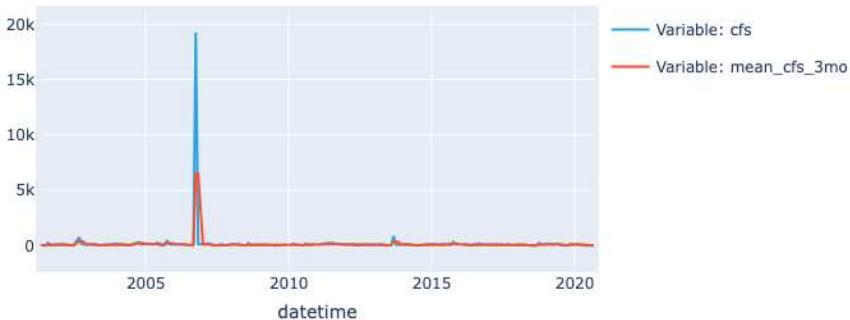


Figure 7.2: Rolling mean plotted on top of daily values.

It looks like there is a big outlier in 2006 or 2007. Let's clip the results to zoom in on the plot.

```
>>> cfs_max = 1_000
>>> (dd
...     .set_sorted('datetime')
...     .group_by_dynamic(index_column='datetime', every='1mo')
...     .agg(pl.col('cfs').mean())
...     .with_columns(pl.col('cfs').clip(upper_bound=cfs_max),
...                  mean_cfs_3mo=pl.col('cfs').mean()
...                               .rolling('datetime', period='3mo')
...                               .clip(upper_bound=cfs_max))
...     .plot(x='datetime', y=['cfs', 'mean_cfs_3mo'])
... )
```

7.10 Interpolation

A common operation in time series data is dealing with missing values. Let's find the missing values for the *cfs* column.

```
>>> print(dd
...     .filter(pl.col('cfs').is_null())
```

7. Time Series Analysis

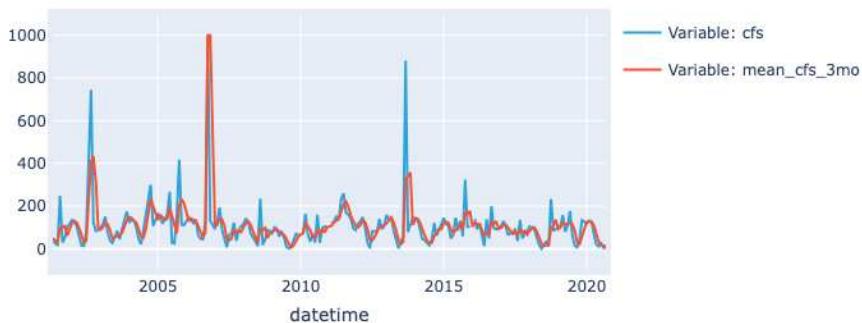


Figure 7.3: Clipped plot.

```
... )
shape: (46_181, 5)
   agency_cd    site_no      cfs      gage_height      datetime
   str          int64     float64     float64      datetime[μs,
   .           .           .           .           America/Denver]
USGS        9333500    null     3.80  2007-12-15 00:00:00 MST
USGS        9333500    null     3.78  2007-12-15 00:15:00 MST
USGS        9333500    null     3.77  2007-12-15 00:30:00 MST
.
USGS        9333500    null     5.48  2020-09-12 00:45:00 MDT
USGS        9333500    null     5.48  2020-09-12 01:00:00 MDT
USGS        9333500    null     5.48  2020-09-12 01:15:00 MDT
```

I'm going to zoom into July of 2018. There are some missing values in this month. I will use the `.is_between` method to filter the DataFrame to only include the dates in July of 2018. Because my `datetime` column has time zones, I need to ensure that the `datetime` objects I create are also in the same time zone. This code will fail if the `datetime` objects are naive (no time zone).

```
>>> import datetime
>>> import pytz
>>> denver = pytz.timezone('America/Denver')
>>> jul_7 = datetime.datetime(2018, 7, 7).replace(tzinfo=denver)
>>> jul_9 = datetime.datetime(2018, 7, 9).replace(tzinfo=denver)

>>> print(dd
... .filter(pl.col('datetime').is_between(jul_7, jul_9))
... .filter(pl.col('cfs').is_null())
```

```
... )
shape: (47, 5)
  agency_cd  site_no    cfs  gage_height   datetime
  str         i64      f64      f64           datetime[μs,
                                                America/Denver]
```

agency_cd	site_no	cfs	gage_height	datetime
str	i64	f64	f64	datetime[μs, America/Denver]
USGS	9333500	null	5.70	2018-07-07 13:15:00 MDT
USGS	9333500	null	5.70	2018-07-07 13:30:00 MDT
USGS	9333500	null	5.69	2018-07-07 13:45:00 MDT
.
USGS	9333500	null	5.71	2018-07-08 09:00:00 MDT
USGS	9333500	null	5.71	2018-07-08 09:15:00 MDT
USGS	9333500	null	5.71	2018-07-08 09:30:00 MDT

Here is a plot of some rows where the data is missing values.

```
>>> (dd
... .filter(pl.col('datetime').is_between(jul_7, jul_9))
... .plot(x='datetime', y=['cfs'])
... )
```

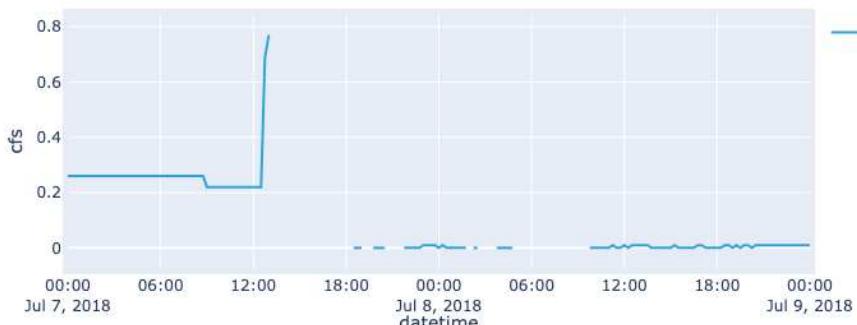


Figure 7.4: Visualization of missing *cfs* values.

The following code demonstrates various techniques for filling in missing values. Note that I'm plotting the results to help you get a visual understanding of what is happening, so I'm shifting each lineup by an offset amount.

```
>>> offset = .1
>>> (dd
```

7. Time Series Analysis

```
... .filter(pl.col('datetime').is_between(jul_7, jul_9))
... .with_columns(
...     'datetime', 'cfs',
...     fill0=pl.col('cfs').fill_null(0).add(offset),
...     interpolate=pl.col('cfs').interpolate().add(offset*2),
...     forward=pl.col('cfs').fill_null(strategy='forward').add(offset*3),
...     backward=pl.col('cfs').fill_null(strategy='backward').add(offset*4)
... )
... .plot(x='datetime', y=['cfs', 'fill0', 'interpolate', 'forward',
...                         'backward'])
... )
```

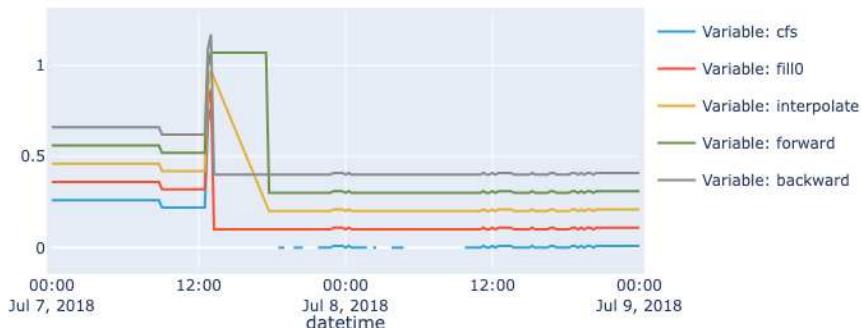


Figure 7.5: Visualization of filling in missing *cfs* values.

7.11 Upsampling and Downsampling

Sometimes, you need to change the frequency of the data. *Upsampling* is when you increase the frequency of the samples to a finer granularity. *Downsampling* is when you decrease the frequency of the samples to coarser granularity. Both are useful for merging data from different sources with different frequencies or smoothing data.

The river data is at a 15-minute frequency. Let's upsample to 5 minutes. We will use the `.upsample` method to upsample the data, and we will use the `.interpolate` method to interpolate the missing values.

```
>>> print(dd
... .set_sorted('datetime')
... .upsample('datetime', every='5m')
... .interpolate()
... )
```

7.11. Upsampling and Downsampling

datetime datetime[μ s, America/Denver]	agency_cd str	site_no f64	cfs f64	gage_height f64
2001-05-07 01:00:00 MDT	USGS	9333500.00	71.00	null
2001-05-07 01:05:00 MDT	null	9333500.00	71.00	null
2001-05-07 01:10:00 MDT	null	9333500.00	71.00	null
.
2020-09-28 09:20:00 MDT	null	9333500.00	9.20	6.15
2020-09-28 09:25:00 MDT	null	9333500.00	9.20	6.15
2020-09-28 09:30:00 MDT	USGS	9333500.00	9.20	6.15

Note that if we try to use a groupby with a finer granularity than the data, we will just get the original data back. For example, if we try to group by every 5 minutes, we will just get the original data back with a 15-minute frequency.

```
>>> print(dd
... .set_sorted('datetime')
... .group_by_dynamic(index_column='datetime', every='5m')
... .agg(pl.col('cfs').mean())
... )
```

datetime datetime[μ s, America/Denver]	cfs f64
2001-05-07 01:00:00 MDT	71.00
2001-05-07 01:15:00 MDT	71.00
2001-05-07 01:30:00 MDT	71.00
.	.
2020-09-28 09:00:00 MDT	9.20
2020-09-28 09:15:00 MDT	9.20
2020-09-28 09:30:00 MDT	9.20

Let's downsample to 30 minutes. This is just doing a dynamic group by with a 30 minute interval.

```
>>> print(dd
... .set_sorted('datetime')
```

7. Time Series Analysis

```
... .group_by_dynamic(index_column='datetime', every='30m')
... .agg(pl.col('cfs').mean())
...
shape: (270_620, 2)
+-----+-----+
| datetime | cfs |
| datetime[μs, America/Denver] | f64 |
+-----+-----+
| 2001-05-07 01:00:00 MDT | 71.00 |
| 2001-05-07 01:30:00 MDT | 70.50 |
| 2001-05-07 02:00:00 MDT | 69.50 |
| . | . |
| 2020-09-28 08:30:00 MDT | 9.36 |
| 2020-09-28 09:00:00 MDT | 9.20 |
| 2020-09-28 09:30:00 MDT | 9.20 |
```

7.12 Joining Time Series Data

I'm also going to load some meteorological data¹ 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

I have another dataset with meteorological data for the same time period. Let's load that data and join it to the river data.

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

>>> def tweak_temp(df_):
...     return (df_
...         .select(pl.col('DATE').str.to_datetime()
...             .dt.replace_time_zone('America/Denver'),
...             'PRCP', 'TMIN', 'TMAX', 'TOBS'))
...     )

>>> raw_temp = pl.read_csv(url)
>>> print(tweak_temp(raw_temp))
shape: (6_848, 5)
```

¹<https://www.ncdc.noaa.gov/cdo-web/>

DATE datetime[μs, America/Denver]	PRCP f64	TMIN i64	TMAX i64	TOBS i64
2000-01-01 00:00:00 MST	0.02	21	43	28
2000-01-02 00:00:00 MST	0.03	24	39	24
2000-01-03 00:00:00 MST	0.00	7	39	18
.
2020-09-22 00:00:00 MDT	0.00	54	84	77
2020-09-23 00:00:00 MDT	0.00	47	91	87
2020-09-24 00:00:00 MDT	0.00	43	94	88

It looks like this data is stored on a daily basis. If we want to join the flow data with the temperature data, we'll downsample the flow data to the daily level. Then, we can join the data based on the dates.

```
>>> dd_daily = (dd
...     .set_sorted('datetime')
...     .group_by_dynamic(index_column='datetime', every='1d',
...                       check_sorted=False)
...     .agg(pl.col('gage_height', 'cfs').mean())
... )
```

```
>>> print(dd_daily)
```

shape: (5_757, 3)

datetime datetime[μs, America/Denver]	gage_height f64	cfs f64
2001-05-07 00:00:00 MDT	null	71.08
2001-05-08 00:00:00 MDT	null	69.21
2001-05-09 00:00:00 MDT	null	61.20
.	.	.
2020-09-26 00:00:00 MDT	6.20	11.06
2020-09-27 00:00:00 MDT	6.18	10.20
2020-09-28 00:00:00 MDT	6.16	9.56

Note

If you are joining date columns, both date columns need the same time zone for Polars to join them.

Let's combine the two dataframes to have daily flow and temperature in a single DataFrame. We will use the `.join` method to join them.

```
>>> both = (tweak_temp(raw_temp)
...     .join(dd_daily, left_on='DATE', right_on='datetime',
...           validate='1:1')
```

7. Time Series Analysis

```
... )  
  
=> print(both)  
shape: (5_056, 7)  


| DATE<br>datetime[µs,<br>America/Denver] | PRCP<br>f64 | TMIN<br>i64 | . | TOBS<br>i64 | gage_height<br>f64 | cfs<br>f64 |
|-----------------------------------------|-------------|-------------|---|-------------|--------------------|------------|
| 2001-05-07 00:00:00<br>MDT              | 0.00        | 41          | . | 55          | null               | 71.08      |
| 2001-05-08 00:00:00<br>MDT              | 0.00        | 43          | . | 58          | null               | 69.21      |
| 2001-05-09 00:00:00<br>MDT              | 0.00        | 36          | . | 64          | null               | 61.20      |
| .                                       | .           | .           | . | .           | .                  | .          |
| 2020-09-22 00:00:00<br>MDT              | 0.00        | 54          | . | 77          | 6.09               | 7.28       |
| 2020-09-23 00:00:00<br>MDT              | 0.00        | 47          | . | 87          | 6.12               | 8.17       |
| 2020-09-24 00:00:00<br>MDT              | 0.00        | 43          | . | 88          | 6.16               | 9.41       |


```

7.13 Visualizing the Merged Data

Let's plot the temperature and cfs of the river data. I'm going to make a line plot where I lightly plot all of the past years levels, and then I will plot the median and most recent years in bold.

First, I will join the data and sort the results. Then I will make three dataframes. One is a pivot of the weekly rolling median values for daily temperature for each year. Another dataframe for the last year and the final for the median values.

I'll use Bokeh as the backend for the plots.

```
>>> import holoviews as hv  
>>> hvplot.extension('bokeh')  
>>> year_agg = (both  
... .with_columns(day_of_year=pl.col('DATE')  
... .dt.strftime('%j').cast(pl.Int16),  
... year=pl.col('DATE').dt.year())  
... .pivot(index='day_of_year', columns='year', values='TOBS')  
... .sort('day_of_year')  
... )  
  
>>> p1 = (year_agg  
... .with_columns(  
... pl.col(['2001', '2002', '2003', '2004', '2005', '2006',
```

```

...
    '2007', '2008', '2009', '2010', '2011', '2012',
...
    '2014', '2015', '2016', '2017', '2018', '2019', # '2020'
...
])
...
.rolling_median(7)
...
)
...
.plot(x='day_of_year', alpha=.5, line_width=1,
      color=hv.Palette('Greys'),
      title='Weekly Temperature (F)', width=1_000, height=500)
...
)

>>> # make 2020 thicker and blue
>>> p2 = p1 * (year_agg
... .select(pl.col('day_of_year', '2020'))
... .plot(x='day_of_year', y='2020', color='blue', line_width=2,
...       label='2020')
...
)

>>> # add median in red
>>> p2 * (both
...
    .with_columns(day_of_year=pl.col('DATE')
                  .dt.strftime('%j').cast(pl.Int16),
                  median=pl.lit('Median'))
...
    .pivot(index='day_of_year', columns='median', values='TOBS',
           aggregate_function='median')
...
    .sort('day_of_year')
...
    .plot(x='day_of_year', y='Median', c='r', label='Median')
...
)

```

Okay, I want to repeat this for the flow data. I'll refactor the code into a function.

```

>>> def plot_year_last_year_median(df, col, upper_limit=None,
...                                 lower_limit=None, width=1_000,
...                                 height=500):
...
...     if upper_limit is None:
...         upper_limit = df[col].max()
...
...
...     year_agg = (df
...     .with_columns(day_of_year=pl.col('DATE')
...                   .dt.strftime('%j').cast(pl.Int16),
...                   year=pl.col('DATE').dt.year())
...     .pivot(index='day_of_year', columns='year', values=col)
...     .sort('day_of_year')
...
...     )
...
...
...     # previous years in grey thin lines

```

7. Time Series Analysis

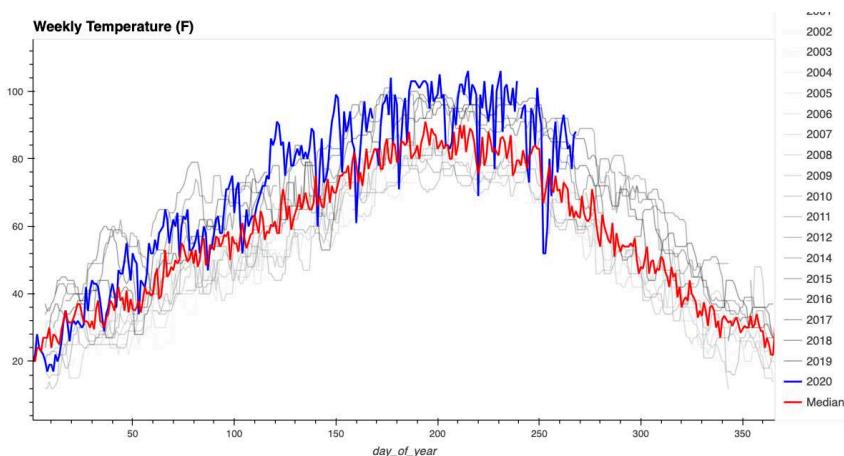


Figure 7.6: Plot of temperature for each year overlaid with the median and last year.

```
...     p1 = (year_agg
...     .with_columns(pl.col(
...         ['2001', '2002', '2003', '2004', '2005', '2006',
...         '2007', '2008', '2009', '2010', '2011', '2012',
...         '2014', '2015', '2016', '2017', '2018', '2019',# '2020'
...         ])
...         .rolling_median(7).clip(upper_bound=upper_limit))
...     .plot(x='day_of_year', alpha=1, color=hv.Palette('Greys'),
...           line_width=.5, width=width, height=height)
... )
...
...
# make 2020 thicker and blue
p2 = p1*(year_agg
    .select('day_of_year', pl.col('2020')
        .rolling_median(7).clip(upper_bound=upper_limit))
    .plot(x='day_of_year', y='2020', color='blue', line_width=2,
          title=f'Weekly {col}', label='2020')
    )
...
#
# add median in red
p3 = p2*(df
    .with_columns(day_of_year=pl.col('DATE')
        .dt.strftime('%j').cast(pl.Int16),
        median=pl.lit('Median'))
    .pivot(index='day_of_year', columns='median', values=col,
          aggregate_function='median')
    .sort('day_of_year'))
```

```

...     .with_columns(pl.col('Median')
...                 .rolling_mean(7).clip(upper_bound=upper_limit))
...     .plot(x='day_of_year', y='Median', color='red', label='Median')
...
... )
... return p3

```

Let's try this out with the `cfs` column:

```
>>> plot_year_last_year_median(both, 'cfs', upper_limit=200)
```

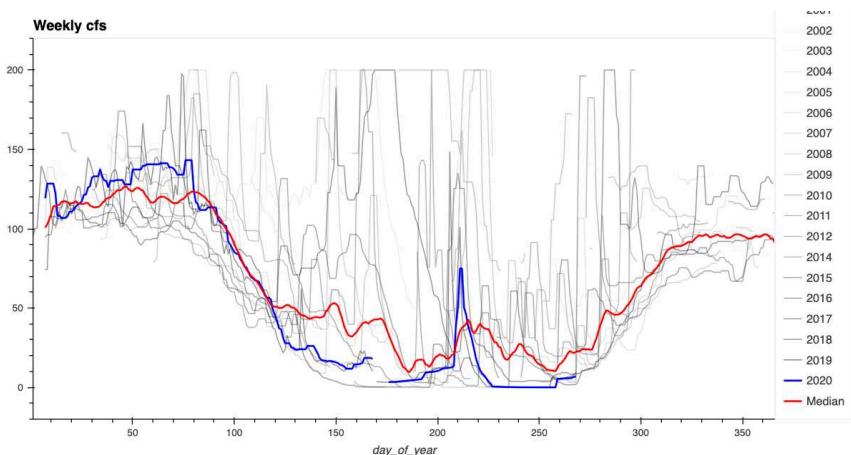


Figure 7.7: Plot of cfs for each year overlaid with the median and last year.

7.14 Chapter Methods and Functions

Table 7.3: Methods and functions for time series analysis.

Method / Function	Description
<code>col.str.to_date(format=None, strict=True, exact=True, cache=True)</code>	This function converts a string to a date, using a specified strftime format string. The format string from Rust's chrono ¹ library. If strict is True, it will raise an error if the conversion fails. If exact is True, then the string must match the format exactly. The cache parameter can improve performance if dates are being parsed repeatedly.

¹<https://docs.rs/chrono/latest/chrono/format/strftime/index.html>

7. Time Series Analysis

Method/Function	Description
<code>col.str.to_datetime(format=None, strict=True, exact=True, cache=True, ambiguous='raise')</code>	This function converts a string to <code>pl.Datetime</code> , using a specified <code>format</code> string. The <code>format</code> string is from Rust's chrono library. If <code>strict</code> is <code>True</code> , it will raise an error if the conversion fails. If <code>exact</code> is <code>True</code> , then the string must match the format exactly. The <code>cache</code> parameter can improve performance if dates are being parsed repeatedly. If <code>ambiguous</code> is <code>'raise'</code> , then ambiguous dates raise an error. If <code>ambiguous</code> is <code>'earliest'</code> , then ambiguous dates are interpreted as the earliest possible date. If <code>ambiguous</code> is <code>'latest'</code> , then ambiguous dates are interpreted as the latest possible date.
<code>col.str.strptime(dtype, format=None, strict=True, exact=True, cache=True, ambiguous='raise')</code>	This function is the same as <code>to_datetime</code> , except that you can specify the output type. The <code>dtype</code> parameter can be <code>pl.Date</code> , <code>pl.Datetime</code> , or <code>pl.Time</code> .
<code>col.str.replace(pat, val, case=True, flags=0)</code>	This function replaces a pattern with a value. The <code>pat</code> parameter is the pattern to replace. The <code>val</code> parameter is the value to replace it with. The <code>case</code> parameter is whether to ignore case. The <code>flags</code> parameter is flags to pass to the regex engine.
<code>col.dt.replace_time_zone(time_zone, ambiguous='raise')</code>	This function replaces the time zone of a naive date or datetime column. The <code>time_zone</code> parameter can be a string or a <code>datetime.tzinfo</code> object. If <code>ambiguous</code> is <code>'raise'</code> , then ambiguous dates raise an error. If <code>ambiguous</code> is <code>'earliest'</code> , then ambiguous dates are interpreted as the earliest possible date. If <code>ambiguous</code> is <code>'latest'</code> , then ambiguous dates are interpreted as the latest possible date.
<code>col.dt.convert_time_zone(time_zone)</code>	This function converts a date or datetime column to a different time zone. The <code>time_zone</code> parameter can be a string or a <code>datetime.tzinfo</code> object.
<code>col.dt.truncate(every, offset=None)</code>	This function truncates a date or datetime column to a specific time interval. The <code>every</code> parameter is the size of the window. The <code>offset</code> parameter is the offset of the window.

Method / Function	Description
<code>df.group_by(by, *more_by, maintain_order=False)</code>	Group this DataFrame by the specified columns.
<code>df.group_by_dynamic(index_column, every, period='every', offset=None, include_boundaries=False, closed='left', label='left', by=None, start_by='window', check_sorted=True)</code>	This function groups a DataFrame by a column that is specifically for grouping based on time. The <code>index_column</code> parameter is the column to group by. The <code>every</code> parameter is the size of the window. The <code>period</code> parameter is the size of the period. This allows you to overlap windows. By default, it is set to 'every'. The <code>offset</code> parameter is the offset of the window. It only is applied if <code>start_by='window'</code> . The <code>include_boundaries</code> parameter is whether to include the boundaries of the window. The <code>closed</code> parameter is whether the window is closed on the 'left', 'right', 'both', or 'none'. The <code>label</code> parameter is whether the label is on the 'left', 'right', or 'datapoint'. The <code>by</code> parameter is additional columns to group by. The <code>start_by</code> parameter is the strategy for starting the window. It can be 'window', 'datapoint', 'monday' - 'sunday'. The <code>check_sorted</code> parameter is whether to check if the DataFrame is sorted. If you know the data is sorted by the <code>index_column</code> , then you can set this to False to improve performance.
<code>col.rolling(index_column, period, offset=None, closed='right', check_sorted=True)</code>	This method applies a previous aggregation step to a period and offset. The period and offset are specified as strings from the table below. The <code>closed</code> parameter specifies whether the window interval is closed on 'left', 'right', 'both', or 'none'. If data in groups is sorted, you can set <code>check_sorted</code> to False to improve performance.
<code>df.set_sorted(column, *more_columns, descending=False)</code>	This method sets the sorted column(s) for the query planner. This is useful when you know that the data is sorted by a column. It is often required when using the <code>.group_by_dynamic</code> method.
<code>col.name.suffix(suffix)</code>	This method adds a suffix to the column name. It is useful when you have multiple aggregations and want to distinguish between the columns.

7. Time Series Analysis

Method/Function	Description
<code>col.name.prefix(prefix)</code>	This method adds a prefix to the column name. It is useful when you have multiple aggregations and want to distinguish between the columns.
<code>df.upsample(time_column, every, offset=None, by=None, maintain_order=False)</code>	This method upsamples the DataFrame to a finer granularity. The <code>time_column</code> parameter is the column to upsample. The <code>every</code> parameter is the frequency to upsample to. The <code>offset</code> parameter is the offset to start the upsample. The <code>by</code> parameter is additional columns to group by. The <code>maintain_order</code> parameter is whether to maintain the order of the rows. If you know the data is sorted by the <code>time_column</code> , then you can set this to <code>False</code> to improve performance.
<code>df.interpolate()</code>	This method interpolates missing values with linear interpolation.
<code>col.interpolate(method='linear')</code>	This method interpolates missing values with linear interpolation. <code>method</code> can be ' <code>linear</code> ' or ' <code>nearest</code> '.
<code>col.over(expr, *more_exprs, mapping_strategy='group_to_rows')</code>	This method takes <code>expr</code> (expression) to group by. It should be preceded by an aggregation method. You can provide multiple expressions to group by. The <code>mapping_strategy</code> parameter can be ' <code>group_to_rows</code> ', ' <code>join</code> ', or ' <code>explode</code> '.
<code>pl.date(year, month, day)</code>	Create a date expression

7.15 Summary

In this chapter, we loaded some time series data. We showed how to change time zones, resample, and interpolate missing values. We also showed how to join two time series datasets and visualize the results.

7.16 Exercises

1. What is required for data to be “time series” data?
2. What is the difference between upsampling and downsampling?
3. What is the difference between a window function and a group by?
4. What is the difference between a rolling window and a dynamic group by?
5. Calculate the sum and the average of the `cfs` column by year.
6. Calculate the sum and the average of the `cfs` column by quarter.

7. Calculate the sum and the average of the *cfs* column by week.
8. Calculate the weekly rolling average of the *cfs* column.
9. Calculate the weekly rolling average of the *cfs* column, but only include weekdays.
10. Calculate the monthly rolling average of the *cfs* column.
11. Calculate the monthly range of the *cfs* column.
12. Calculate the five-month rolling average of the *cfs* column starting in November.

Chapter 8

Data Import and Export

8.1 Introduction

Without importing data, we would get nowhere in our data analysis journey. We also want to be able to export our results to share with others or use them in other applications. This chapter will teach how to import and export data in Polars.

8.2 Importing Data

We will start by importing the Fuel Economy dataset, which we will use throughout this chapter. We will load it from a CSV file. Much of the data in the real world is stored in CSV files. However, CSV files are not the most efficient way to store data. We will learn how to write back to CSV and how to write in more efficient formats.

I'll include the `tweak_auto` code here so you can see the steps I performed after loading the data from a CSV file.

```
>>> import polars as pl
>>> path = 'data/vehicles.csv'
>>> raw = pl.read_csv(path, null_values=['NA'])

>>> def tweak_auto(df):
...     cols = ['year', 'make', 'model', 'displ', 'cylinders', 'trany',
...             'drive', 'VClass', 'fuelType', 'barrels08', 'city08',
...             'highway08', 'createdOn']
...     return (df
...             .select(pl.col(cols))
...             .with_columns(pl.col('year').cast(pl.Int16),
...                         pl.col(['cylinders', 'highway08', 'city08']).cast(pl.UInt8),
...                         pl.col(['displ', 'barrels08']).cast(pl.Float32),
...                         pl.col(['make', 'model', 'VClass', 'drive', 'fuelType'])
...                         .cast(pl.Categorical),
```

8. Data Import and Export

```
...     pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...     is_automatic=pl.col('trany').str.contains('Auto'),
...     num_gears=pl.col('trany').str.extract(r'(\d+)').cast(pl.Int8)
...   )
...
>>> autos = tweak_auto(raw)
```

The `read_csv` and its lazy counterpart, `scan_csv`, are the workhorses for ingesting CSV data. They include options for parsing dates. In general, I prefer to explicitly handle dates later in my tweak pipeline because the CSV parser can silently fail to convert dates.

8.3 Exporting to CSV

We can export a DataFrame to a CSV file using the `write_csv` method.

This function allows you to adjust the character encoding, separator, quote mechanism, line endings, and more. Generally, you can use the default values for all of the arguments. However, you may want to adjust precision or date formatting. Let's export the `df` dataframe to a CSV format. If we don't provide a path, the method will return a string of the CSV data. This will let us quickly see the output.

```
>>> print(autos
... .head(3)
... .write_csv()
... )

year,make,model,displ,cylinders,trany,drive,VClass,fuelType,barrels08,cit
y08,highway08,createdOn,is_automatic,num_gears
1985,Alfa Romeo,Spider Veloce 2000,2.0,4,Manual 5-spd,Rear-Wheel
Drive,Two
Seaters,Regular,15.695714,19,25,2013-01-01T00:00:00.000000,false,5
1985,Ferrari,Testarossa,4.9,12,Manual 5-spd,Rear-Wheel Drive,Two
Seaters,Regular,29.964546,9,14,2013-01-01T00:00:00.000000,false,5
1985,Dodge,Charger,2.2,4,Manual 5-spd,Front-Wheel Drive,Subcompact
Cars,Regular,12.207778,23,33,2013-01-01T00:00:00.000000,false,5
```

Here, we adjust the precision to 2 decimal places. You can see that the values of `displ` and `barrels08` are now rounded to 2 decimal places.

```
>>> print(autos
... .head(3)
... .write_csv(float_precision=2)
... )
```

```
year,make,model,displ,cylinders,trany,drive,VClass,fuelType,barrels08,cit
y08,highway08,createdOn,isAutomatic,num_gears
1985,Alfa Romeo,Spider Veloce 2000,2.00,4,Manual 5-spd,Rear-Wheel
Drive,Two Seaters,Regular,15.70,19,25,2013-01-01T00:00:00.000000,false,5
1985,Ferrari,Testarossa,4.90,12,Manual 5-spd,Rear-Wheel Drive,Two
Seaters,Regular,29.96,9,14,2013-01-01T00:00:00.000000,false,5
1985,Dodge,Charger,2.20,4,Manual 5-spd,Front-Wheel Drive,Subcompact
Cars,Regular,12.21,23,33,2013-01-01T00:00:00.000000,false,5
```

8.4 Exporting to JSON

Another common data format is JavaScript Object Notation (JSON). We can export a DataFrame to JSON using the `.write_json` method. Exporting tabular data to JSON format offers the advantage of being highly readable and easily parsed by web applications, making it ideal for data exchange in web contexts. However, one drawback is that JSON, a text-based format, can lead to larger file sizes than binary formats, potentially impacting performance and storage, especially with large datasets.

Note that Polars embeds the type information, making it possible to read the JSON file back into a DataFrame with the correct data types. However, other applications may not read the data types from the JSON file. In that case, you may want to use a more efficient format such as Parquet or Arrow.

Let's export the first three rows of the `autos` DataFrame to a JSON file. I will pretty print it to make it easier to see the structure.

```
>>> import pprint
>>> import json
>>> pprint.pprint(json.loads(
...     autos.head(2)
...     .write_json()))
{'columns': [{}{'bit_settings': '',
    'datatype': 'Int16',
    'name': 'year',
    'values': [1985, 1985]},
 {}{'bit_settings': '',
    'datatype': {'Categorical': [None, 'Physical']},
    'name': 'make',
    'values': ['Alfa Romeo', 'Ferrari']},
 {}{'bit_settings': '',
    'datatype': {'Categorical': [None, 'Physical']},
    'name': 'model',
    'values': ['Spider Veloce 2000', 'Testarossa']},
 {}{'bit_settings': '',
    'datatype': 'Float32',
    'name': 'displ',
    'values': [2.0, 4.9]}],
```

8. Data Import and Export

```
{'bit_settings': '',
 'datatype': 'UInt8',
 'name': 'cylinders',
 'values': [4, 12]},
{'bit_settings': '',
 'datatype': 'String',
 'name': 'trany',
 'values': ['Manual 5-spd', 'Manual 5-spd']},
{'bit_settings': '',
 'datatype': {'Categorical': [None, 'Physical']},
 'name': 'drive',
 'values': ['Rear-Wheel Drive', 'Rear-Wheel Drive']},
{'bit_settings': '',
 'datatype': {'Categorical': [None, 'Physical']},
 'name': 'VClass',
 'values': ['Two Seaters', 'Two Seaters']},
{'bit_settings': '',
 'datatype': {'Categorical': [None, 'Physical']},
 'name': 'fuelType',
 'values': ['Regular', 'Regular']},
{'bit_settings': '',
 'datatype': 'Float32',
 'name': 'barrels08',
 'values': [15.695714, 29.964546]},
{'bit_settings': '',
 'datatype': 'UInt8',
 'name': 'city08',
 'values': [19, 9]},
{'bit_settings': '',
 'datatype': 'UInt8',
 'name': 'highway08',
 'values': [25, 14]},
{'bit_settings': '',
 'datatype': {'Datetime': ['Microseconds', None]},
 'name': 'createdOn',
 'values': [1356998400000000, 1356998400000000]},
{'bit_settings': '',
 'datatype': 'Boolean',
 'name': 'is_automatic',
 'values': [False, False]},
{'bit_settings': '',
 'datatype': 'Int8',
 'name': 'num_gears',
 'values': [5, 5]}}
```

8.5 Reading JSON

Reading JSON generated from Polars is not too difficult. The `read_json` method can read the JSON file back into a DataFrame.

```
>>> from io import StringIO
>>> print(pl.read_json(StringIO(autos.write_json())))
shape: (41_144, 15)
```

year	make	model	.	createdOn	is_automat	num_gears
i16	cat	cat	.	datetime[tic	i8
				μs]	bool	
1985	Alfa Romeo	Spider	.	2013-01-0	false	5
		Veloce		1		
		2000		00:00:00		
1985	Ferrari	Testarossa	.	2013-01-0	false	5
		a		1		
				00:00:00		
1985	Dodge	Charger	.	2013-01-0	false	5
				1		
				00:00:00		
.
1993	Subaru	Legacy	.	2013-01-0	true	4
		AWD		1		
				00:00:00		
1993	Subaru	Legacy	.	2013-01-0	false	5
		AWD		1		
				00:00:00		
1993	Subaru	Legacy	.	2013-01-0	true	4
		AWD Turbo		1		
				00:00:00		

The difficulties arise when we need to read JSON from other sources.

Let's export the first three rows using a Pandas JSON export and see if we can read it back into Polars. If you export it with the Pandas 'records' orientation, you can read it back into Polars. However, if you use the default ('columns') orientation, you will not be able to cleanly read it back into Polars.

Let's export the first three rows of the `autos` DataFrame to a JSON file using the 'records' orientation.

```
autos.head(3).to_pandas().to_json('/tmp/pd.json', orient='records')
```

Here's what the export looks like. We can see that the data is in a list of dictionaries. Each dictionary represents a row in the DataFrame. The keys are the column names, and the values are the data.

8. Data Import and Export

```
>>> import pprint
>>> import json
>>> pprint.pprint(json.loads(autos.head(2)
...     .to_pandas().to_json(orient='records')))
[{'VClass': 'Two Seaters',
 'barrels08': 15.6957139969,
 'city08': 19,
 'createdOn': 1356998400,
 'cylinders': 4,
 'displ': 2.0,
 'drive': 'Rear-Wheel Drive',
 'fuelType': 'Regular',
 'highway08': 25,
 'is_automatic': False,
 'make': 'Alfa Romeo',
 'model': 'Spider Veloce 2000',
 'num_gears': 5,
 'trany': 'Manual 5-spd',
 'year': 1985},
 {'VClass': 'Two Seaters',
 'barrels08': 29.9645462036,
 'city08': 9,
 'createdOn': 1356998400,
 'cylinders': 12,
 'displ': 4.9000000954,
 'drive': 'Rear-Wheel Drive',
 'fuelType': 'Regular',
 'highway08': 14,
 'is_automatic': False,
 'make': 'Ferrari',
 'model': 'Testarossa',
 'num_gears': 5,
 'trany': 'Manual 5-spd',
 'year': 1985}]
```

Let's load the JSON file back into Polars with the `read_json` method.

```
>>> from_pd = pl.read_json('/tmp/pd.json')
>>> print(from_pd)
shape: (3, 15)
+-----+
| year      make          model      .  createdOn  is_automa num_gears |
| i64       str           str        .  i64        bool       i64    |
+-----+
| 1985     Alfa Romeo   Spider     .  135699840  false      5    |
|                 Veloce   Veloce    .  0          true      5    |
+-----+
```

			2000			
1985	Ferrari	Testaross	.	135699840	false	5
		a		0		
1985	Dodge	Charger	.	135699840	false	5
				0		


```
>>> print(from_pd
... .with_columns(createdOn=pl.from_epoch('createdOn', time_unit='s'))
... )
shape: (3, 15)
   year      make      model      .  createdOn      is_automat      num_gears
    i64        str        str      .  datetime[      tic          i64
                                µs]      bool

1985  Alfa Romeo  Spider      .  2013-01-0  false      5
      Veloce      1  00:00:00
      2000      1  00:00:00
1985  Ferrari   Testaross      .  2013-01-0  false      5
      a      1  00:00:00
1985  Dodge     Charger      .  2013-01-0  false      5
      1  00:00:00
```

8.6 Custom JSON Handling

I'll now discuss loading JSON formatted differently than the Polars expects. It will be an excellent excuse to learn more about nested structs and lists in Polars.

I'll export the *autos* data as JSON using the default Pandas orientation, which is called '`columns`'. This orientation is a dictionary of column names mapped to a dictionary, mapping the index to the value for that row. (Note that the index can't be duplicated for this export, although, in general, Pandas DataFrames can have duplicated indices.)

The data for the first two rows looks like this:

```
>>> import pprint
>>> import json
>>> pprint.pprint(json.loads(autos.head(2))
...       .to_pandas().to_json()))
{'VClass': {'0': 'Two Seaters', '1': 'Two Seaters'},
 'barrels08': {'0': 15.6957139969, '1': 29.9645462036},
```

8. Data Import and Export

```
'city08': {'0': 19, '1': 9},
'createdOn': {'0': 1356998400, '1': 1356998400},
'cylinders': {'0': 4, '1': 12},
'displ': {'0': 2.0, '1': 4.9000000954},
'drive': {'0': 'Rear-Wheel Drive', '1': 'Rear-Wheel Drive'},
'fuelType': {'0': 'Regular', '1': 'Regular'},
'highway08': {'0': 25, '1': 14},
'isAutomatic': {'0': False, '1': False},
'make': {'0': 'Alfa Romeo', '1': 'Ferrari'},
'model': {'0': 'Spider Veloce 2000', '1': 'Testarossa'},
'num_gears': {'0': 5, '1': 5},
'trany': {'0': 'Manual 5-spd', '1': 'Manual 5-spd'},
'year': {'0': 1985, '1': 1985}}
```

When we try to read this data into Polars, it does not error. It even gets the column names correct. However, there is only one row with *struct* type columns. You can think of a struct type in Polars as one of the nested types. I would compare it to a tuple in Python. It is a single value that contains multiple values. In this case, it contains all the values for a column.

```
>>> import io
>>> print(pl.read_json(io.StringIO(autos.head(2).to_pandas().to_json())))
shape: (1, 15)
+-----+-----+-----+-----+-----+-----+-----+
| year | make | model | . | createdOn | is_autom | num_gear |
+-----+-----+-----+-----+-----+-----+-----+
| 1985,1985 | {"Alfa Romeo", "Ferrari"} | {"Spider Veloce 2000", "Testarossa"} | . | {1356998400, 1356998400} | {false, false} | {5,5} |
+-----+-----+-----+-----+-----+-----+-----+
```

Polars has the ability to *explode* a sequence into multiple rows. But that only works with *list* types, not struct types. Here is a basic demo. It is a dataframe with a numeric column, a struct column, and a list column.

```
>>> print(pl.DataFrame({
...     'num': [1, 2, 3],
...     'listy': [[1, 2], [3, 4], [5, 6]],
...     'structy': [{a: 1, b: 2}, {a: 3, b: 4},
...                 {a: 5, b: 6}],
...     })
... )
```

	shape: (3, 3)			
	num	listy	structy	
	i64	list[i64]	struct[2]	
1		[1, 2]	{1,2}	
2		[3, 4]	{3,4}	
3		[5, 6]	{5,6}	

Notice that the *structy* column is the values from the dictionary in the *struct* column. The keys are lost. If you remember, the JSON export we loaded had the index as the key in the dictionary. But we don't have the index entries when we load the data into Polars. We only have the values.

Okay, I would like to *explode* the nested data into multiple rows. It turns out that Polars is more than happy to do this with *list* types.

```
>>> print(pl.DataFrame({
...     'num': [1, 2, 3],
...     'listy': [[1, 2], [3, 4], [5, 6]],
...     'structy': [{('a'): 1, 'b': 2}, {'a': 3, 'b': 4}, {'a': 5, 'b': 6}],
...     }
... )
... .explode('listy')
... )
```

shape: (6, 3)

	num	listy	structy	
	i64	i64	struct[2]	
1	1	1	{1,2}	
1	2	2	{1,2}	
2	3	3	{3,4}	
2	4	4	{3,4}	
3	5	5	{5,6}	
3	6	6	{5,6}	

However, if we try to explode the *struct* column, we get an error. It isn't a supported operation.

```
>>> print(pl.DataFrame({
...     'num': [1, 2, 3],
...     'listy': [[1, 2], [3, 4], [5, 6]],
...     'structy': [{('a'): 1, 'b': 2}, {'a': 3, 'b': 4}, {'a': 5, 'b': 6}],
...     }
... )
... .explode('structy')
... )
```

Traceback (most recent call last)

8. Data Import and Export

```
...
InvalidOperationError: `explode` operation not supported for dtype
`struct[2]`
```

I'll jump through a hoop to convert the *struct* column to a *list* column. Polars won't let me simply cast it into a `pl.List` type. Rather, I'll use the (slow) `.map_elements` method to convert each individual struct into a list.

```
>>> print(pl.DataFrame({
...     'num': [1, 2, 3],
...     'listy': [[1, 2], [3, 4], [5, 6]],
...     'structy': [{('a': 1, 'b': 2), ('a': 3, 'b': 4), ('a': 5, 'b': 6)}]
...     .with_columns(structy=pl.col('structy').map_elements(
...         lambda d: list(d.values())))
...     )
... })
shape: (3, 3)
+-----+
| num  listy      structy |
| i64  list[i64]  list[i64] |
+-----+
| 1    [1, 2]      [1, 2]  |
| 2    [3, 4]      [3, 4]  |
| 3    [5, 6]      [5, 6]  |
+-----+
```

Armed with the ability to convert a *struct* column to a *list* column, I'll use it to convert the *autos* JSON export into a usable format. I'll also convert the *createdOn* column to a date.

```
>>> import io
>>> print(pl.read_json(io.StringIO(autos.to_pandas().to_json())))
... .with_columns(pl.all().map_elements(lambda d: list(d.values())))
... .explode(pl.all())
... .with_columns(createdOn=pl.from_epoch('createdOn', time_unit='s'))
... )
shape: (41_144, 15)
+-----+
| year   make      model      .   createdOn      is_automotive  num_gears |
| i64    str       str        .   datetime[      bool          f64        |
|                   .           μs]          |                         |
+-----+
| 2018   BMW        440i Conv...ertible   .   2017-04-05 00:00:00  true      8.00 |
|                   .           .           .           .           .           .           |
| 2019   Lexus      GS 350     .   2018-10-17 00:00:00  true      8.00 |
|                   .           .           .           .           .           .           |
+-----+
```

2011	Jeep	Compass 2WD	.	2013-01-0 1 00:00:00	false	5.00
.
2003	Mercedes-B enz	SL55 AMG	.	2013-01-0 1 00:00:00	true	5.00
1984	BMW	5 Series	.	2013-01-0 1 00:00:00	true	4.00
2004	Volvo	S80/S80 Premier	.	2013-01-0 1 00:00:00	true	4.00

Hopefully, you will deal with JSON data that is easier to load into Polars. But if you aren't, you might be able to use the techniques I've shown here to convert it into a format that Polars can handle. The `.explode` method is a powerful tool for working with nested data.

8.7 Munging JSON

Okay, let's do one more example with non-friendly JSON. If you export Pandas data with the 'split' orientation, you will get a JSON file that Polars will read but will not be useable. The technique I show here will be useful if you can't get Polars-friendly JSON from the source. We will write a custom Python generator to read the JSON data and convert it into a format that Polars can read.

Here's what the JSON looks like when we export the first two rows of the `autos` DataFrame with the 'split' orientation.

```
>>> pprint.pprint(json.loads(autos.head(2).to_pandas()
... .to_json(orient='split')))
{'columns': ['year',
             'make',
             'model',
             'displ',
             'cylinders',
             'trany',
             'drive',
             'VClass',
             'fuelType',
             'barrels08',
             'city08',
             'highway08',
             'createdOn',
             'is_automatic',
```

8. Data Import and Export

```
'num_gears'],
'data': [[1985,
    'Alfa Romeo',
    'Spider Veloce 2000',
    2.0,
    4,
    'Manual 5-spd',
    'Rear-Wheel Drive',
    'Two Seaters',
    'Regular',
    15.6957139969,
    19,
    25,
    1356998400,
    False,
    5],
[1985,
    'Ferrari',
    'Testarossa',
    4.9000000954,
    12,
    'Manual 5-spd',
    'Rear-Wheel Drive',
    'Two Seaters',
    'Regular',
    29.9645462036,
    9,
    14,
    1356998400,
    False,
    5]],
'index': [0, 1]}
```

Let's read this into Polars and see what we get.

```
>>> print(pl.read_json(io.StringIO(autos.to_pandas()
...                               .to_json(orient='split'))))
shape: (1, 3)
+-----+-----+-----+
| columns      | index | data   |
| list[str]    | list[i64]| list[list[str]] |
+-----+-----+-----+
| ["year", "make", . . . [0, 1, . . . 41143] | [{"1985", "Alfa Romeo", . . .
| "num_gears"] | "5"], . . . |
```

There are columns and a row, but this is completely unusable. The *columns* column has a list of the column names, the *index* column has a list of the index

values, and the `data` column has a list of the data values. This is not what we want.

It turns out that the `pl.DataFrame` constructor can take a Python generator as an argument. We can use this to read the JSON file and convert it into a format that Polars can read. Our generator will accept a JSON string and yield a dictionary for each row in the DataFrame. This should be pretty straightforward. The JSON has the columns for us, and we can look over the data, zip the columns and the data together into a dictionary, and yield the result. We will ignore the index column.

Here's the generator.

Note

If you aren't familiar with generators, you can think of them as a function that can pause and resume. When you call the generator, it will run until it hits a `yield` statement. It will then pause and return the value. When you call the generator again, it will resume from where it left off. This is useful for working with large datasets because you don't need to load the entire dataset into memory at once. You can load it one row at a time.

See my book *Intermediate Python* for more information on generators.

```
import json

def split_json_to_dict(json_str):
    """ Convert pandas "split" json to a sequence of dictionaries
    representing the rows of the dataframe.
    """
    data = json.loads(json_str)
    columns = data['columns']
    for row in data['data']:
        yield dict(zip(columns, row))
```

Let's try reading the JSON file with our generator.

```
>>> print(pl.DataFrame(
...     split_json_to_dict(autos.to_pandas().to_json(orient='split'))))
shape: (41_144, 15)
   year      make      model      .  createdOn  is_automat
   i64        str        str        .      i64       tic
                                bool
1985  Alfa Romeo  Spider  .  135699840  false    5.00
                               Veloce  0
                               2000
1985  Ferrari    Testaross  .  135699840  false    5.00
                               a  0
1985  Dodge      Charger  .  135699840  false    5.00
```

8. Data Import and Export

					θ	
1993	Subaru	Legacy	AWD	135699840	true	4.00
1993	Subaru	Legacy	AWD	135699840	false	5.00
1993	Subaru	Legacy	AWD Turbo	135699840	true	4.00

Success! If you come across JSON that Polars can't read, you can use a generator to convert it into a format that Polars can read. This powerful technique can be used to work with data that is not in a format that Polars can read.

8.8 Exporting to Excel

We can export a DataFrame to an Excel file using the `.write_excel` method.

Let's export the first three rows of the `autos` DataFrame to an Excel file.

```
>>> (autos
... .head(3)
... .write_excel('/tmp/autos.xlsx')
... )
```

Here's a screenshot of the resulting Excel file.

year	make	model	disp	cylinders	trans	drive	VClass	fuelType	barrels0	city08	highway08	createdOn	is_automobile	num_germs	
1.985	Alfa Romeo	Spider V6	2.000	4	Manual	5	Rear-Wheel Drive	Two Seats	Regular	15.696	19	25	2013-01-01 00:00:00	FALSE	5
1.985	Ferrari	Testarossa	4.900	12	Manual	5	Rear-Wheel Drive	Two Seats	Regular	29.965	9	14	2013-01-01 00:00:00	FALSE	5
1.985	Dodge	Charger	2.200	4	Manual	5	Front-Wheel Drive	Subcompact	Regular	12.208	23	33	2013-01-01 00:00:00	FALSE	5

Figure 8.1: Export of dataframe to Excel

Let's roundtrip the Excel file into a DataFrame using the `pl.read_excel` function. Note that you must ensure you have the `xlsx2csv` Python package installed for this to work.

```
>>> a3 = (pl.read_excel('/tmp/autos.xlsx'))
>>> print(a3)
shape: (3, 15)
   year      make       model      .  createdOn  is_automati num_gears
   i64        str        str        .    str          bool        i64
1985  Alfa Romeo  Spider     .  2013-01-0  false         5
           Veloce     1  00:00:00
           2000     00:00:00
1985  Ferrari   Testaross     .  2013-01-0  false         5
           a         1  00:00:00
1985  Dodge     Charger     .  2013-01-0  false         5
           1  00:00:00
```

Again, note that the *createdOn* column is created as a string column.

The Polars `read_excel` function fronts four different libraries for reading Excel files. The default is `xlsx2csv`. I would probably recommend using the `calamine` Rust library. Unless you are familiar with Rust, the `calamine` docs are a little difficult to understand. Luckily, there is a Python wrapper to the library, called `fastexcel` that cracks open the features a little more. I've had Excel files where I needed to skip the first few rows, and the documentation for neither `pl.read_excel` nor `calamine` mentions how to do this. However, the `fastexcel` documentation¹ does.

The `read_options` argument to `pl.read_excel` is a dictionary that is passed to the underlying library. Some of the relevant options to pass when using `engine='calamine'` (`fastexcel`) are:

- `idx_or_name`: The sheetname or index to load.
- `header_row=0`: 0-based index of the row containing the column labels. If `None`, the sheet does not have any column labels.
- `column_names`: Names to use for the columns instead of the first row.
- `skip_rows=0`: Number of rows to skip from the start of the sheet.
- `n_rows=None`: Number of rows to load. Default is all rows.
- `schema_sample_rows=1000`: Number of rows to use to determine the `dtype` of a column. Set to `None` to use all rows.
- `use_columns=None`: Columns to read. `None` to read all columns. A list of column names, indices (0-based), or a string of Excel column letters and ranges (e.g. "B:F" or "D,G,I:M").
- `dtypes=None`: Dictionary mapping columns (or indices) to `dtypes`.

¹https://fastexcel.toucantoco.dev/fastexcel.html#ExcelReader.load_sheet_by_name

8. Data Import and Export

8.9 Exporting to Parquet

Parquet is a columnar storage format widely used in “big data” ecosystems. It is a binary format that is highly efficient for both reading and writing data. It is also a self-describing format that embeds the schema information in the file. This makes it possible to read the data back into a DataFrame with the correct data types. One downside of Parquet is that it is not human-readable. That tends not to be a problem since we can use a library to read the data back into a DataFrame to inspect it.

Let’s round trip a few rows of the *autos* DataFrame to a Parquet file, read and export using Pandas, and then back into Polars.

```
>>> (autos
... .head(3)
... .write_parquet('/tmp/a3.parquet')
... )

>>> import pandas as pd
>>> pd.read_parquet('/tmp/a3.parquet').to_parquet('/tmp/a4.parquet')

>>> a4 = pl.read_parquet('/tmp/a4.parquet')
>>> print(a4)
shape: (3, 15)
+-----+-----+-----+-----+-----+-----+
| year | make | model | . | createdOn | is_automotic | num_gears |
| i16  | cat   | cat    | . | datetime[ | bool          | i8          |
|      |       |        |   | μs]        |              |             |
+-----+-----+-----+-----+-----+-----+
| 1985 | Alfa Romeo | Spider | . | 2013-01-0 | false         | 5           |
|       |             | Veloce |   | 1           |              |             |
|       |             | 2000   |   | 00:00:00   |              |             |
| 1985 | Ferrari     | Testaross | . | 2013-01-0 | false         | 5           |
|       |             | a       |   | 1           |              |             |
|       |             |         |   | 00:00:00   |              |             |
| 1985 | Dodge       | Charger | . | 2013-01-0 | false         | 5           |
|       |             |         |   | 1           |              |             |
|       |             |         |   | 00:00:00   |              |             |
+-----+-----+-----+-----+-----+-----+
```

Let’s check and see that the data is the same. We will use the `.frame_equal` method to compare the two dataframes.

```
>>> autos.head(3).equals(a4)
True
```

Note

You can use the `.shrink_dtype` column method to have Polars automatically reduce numeric columns to the minimum size required.

```
>>> (autos
... .head(3)
... .select(pl.all().shrink_dtype())
... .write_parquet('/tmp/a3-shrink.parquet')
... )
```

In this case, I've already manually performed the shrinkage when we went through the columns in the first chapter.

8.10 Exporting to SQL

If we have the SQLAlchemy library installed, we can export a DataFrame to a SQL database using the `.write_database` method. SQLAlchemy uses database *URI* strings to connect to databases. The URI string typically contains the database type, username, password, host, port, and database name. The format of the URI string depends on the database type. For example, here is a URI string for a PostgreSQL database.

```
postgresql://scott:tiger@localhost/mydatabase
```

I'm going to create a SQLite database table from the *autos* data.

```
import sqlite3
with sqlite3.connect('/tmp/vehicles.db') as conn:
    uri = 'sqlite:///tmp/vehicles.db'
    autos.head(3).write_database(table_name='autos', connection=uri,
                                 if_table_exists='replace')
```

Polars has two functions for reading from a database, `pl.read_database` and `pl.read_database_uri`. If you have SQLAlchemy installed, you can use the `pl.read_database` function to read data from a SQL database using a connection. If you have a URI, you can use the `pl.read_database_uri` function to read data from a SQL database using a URI string. Both functions return a DataFrame.

Let's connect to the SQLite database and read the *autos* table into a DataFrame.

```
>>> from sqlalchemy import create_engine
>>> uri = 'sqlite:///tmp/vehicles.db'
>>> with create_engine(uri).connect() as conn:
...     query = 'SELECT * FROM autos'
...     a4 = pl.read_database(query=query, connection=conn)
```

8. Data Import and Export

```
>>> print(a4)
shape: (3, 15)
   year      make      model      .  createdOn  is_automati    num_gears
   i64        str        str        .        str          tic        i64
1985  Alfa Romeo    Spider     .  2013-01-0    0            5
           Veloce
           2000
1985  Ferrari      Testarossa
           a
1985  Dodge         Charger    .  2013-01-0    0            5
           1 00:00:0
           0.000000
```

Note

I'm using a *context manager*, (the `with` keyword) with the database connection. The connection is open when you are inside the indented portion (the context). After the dedent, Python will automatically close the connection. You can check the `conn.closed` attribute inside and outside of the context to verify this.

See my book, *Intermediate Python*, for more details on context managers.

Note that type information might be partially lost when writing and reading from a database. For example, the `createdOn` column was converted into a string type, and `is_automatic` was changed from a boolean to an integer.

8.11 Using Arrow to Convert DataFrames

Because Polars leverages the Arrow data format, we should be able to do a zero-copy conversion to other Arrow-compatible libraries. Let's try converting a Polars DataFrame to DuckDB, doing an aggregation, and then converting it to Pandas for plotting.

DuckDB can query Polars DataFrames directly. Let's try it with a SQL query to aggregate the `city08` column by `year`.

```
>>> import duckdb
>>> sql = '''SELECT mean(city08) AS mean_city08,
...           mean(highway08) AS mean_highway08,
...           year
... FROM autos
... GROUP BY year'''
```

8.11. Using Arrow to Convert DataFrames

```
>>> agg = duckdb.sql(sql)
```

```
>>> print(agg)
```

mean_city08 double	mean_highway08 double	year int16
17.135170603674542	23.451443569553806	1997
17.275521405049396	23.32821075740944	2001
17.665289256198346	22.699173553719007	1986
16.78065134099617	22.836206896551722	2003
17.33445945945946	24.017736486486488	2009
22.5919175911252	29.606973058637085	2016
16.998170173833486	22.7804208600183	1993
16.740641711229948	23.06417112299465	2004
16.605683836589698	23.08348134991119	2007
17.310344827586206	22.445068163592623	1987
.	.	.
.	.	.
.	.	.
16.851629502572898	23.297598627787306	2005
17.333628318584072	22.702654867256637	1988
20.66131756756757	27.504222972972972	2013
22.761020881670532	29.554524361948957	2017
23.31814730447988	29.66438876233865	2019
16.918533604887983	22.725050916496944	1994
17.272300469483568	23.552816901408452	1999
16.848939929328623	22.253533568904594	1991
19.36284722222222	26.10503472222222	2012
22.679425837320576	29.617224880382775	2020

37 rows (20 shown)

3 columns

Cool. Now, let's get that result into a Pandas DataFrame and plot it. DuckDB has a .df method that will convert the result to a Pandas DataFrame. However, since this is a Polars book, we will convert it back to Polars and then to Pandas. We will use the DuckDB .pl method to convert the result to a Polars DataFrame. Then, we will use the .to_pandas method to convert the result to a Pandas DataFrame.

```
>>> pd_agg = agg.pl().to_pandas()
```

```
>>> print(pd_agg)
```

	mean_city08	mean_highway08	year
0	17.135171	23.451444	1997
1	17.275521	23.328211	2001
2	17.665289	22.699174	1986
3	16.780651	22.836207	2003
4	17.334459	24.017736	2009

8. Data Import and Export

5	22.591918	29.606973	2016
6	16.918534	22.725051	1994
7	17.272300	23.552817	1999
8	16.848940	22.253534	1991
9	19.362847	26.105035	2012
10	22.679426	29.617225	2020
11	16.998170	22.780421	1993
12	16.740642	23.064171	2004
13	16.605684	23.083481	2007
14	17.310345	22.445068	1987
15	18.669027	25.169912	2011
16	16.626812	23.048913	2006
17	16.900590	23.455771	2008
18	17.878307	23.042328	1985
19	16.569804	22.671148	1995
20	17.113300	23.546798	1998
21	17.982688	23.075356	1984
22	17.221429	23.414286	2000
23	18.105500	24.947701	2010
24	17.143972	22.465742	1989
25	17.033395	22.337662	1990
26	21.033469	27.978776	2014
27	21.445830	28.586906	2015
28	22.564732	29.273065	2018
29	17.289780	23.569211	1996
30	16.893333	23.030769	2002
31	16.805531	22.439786	1992
32	16.851630	23.297599	2005
33	17.333628	22.702655	1988
34	20.661318	27.504223	2013
35	22.761021	29.554524	2017
36	23.318147	29.664389	2019

I'll use the `.plot` method of the Pandas DataFrame to plot the result.

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(figsize=(10, 5))
>>> (agg
...     .pl()
...     .to_pandas()
...     .set_index('year')
...     .plot(ax=ax, title='Average MPG by Year'))
>>> fig.savefig('img/agg1.png', bbox_inches='tight')
<Figure size 1000x500 with 1 Axes>
```

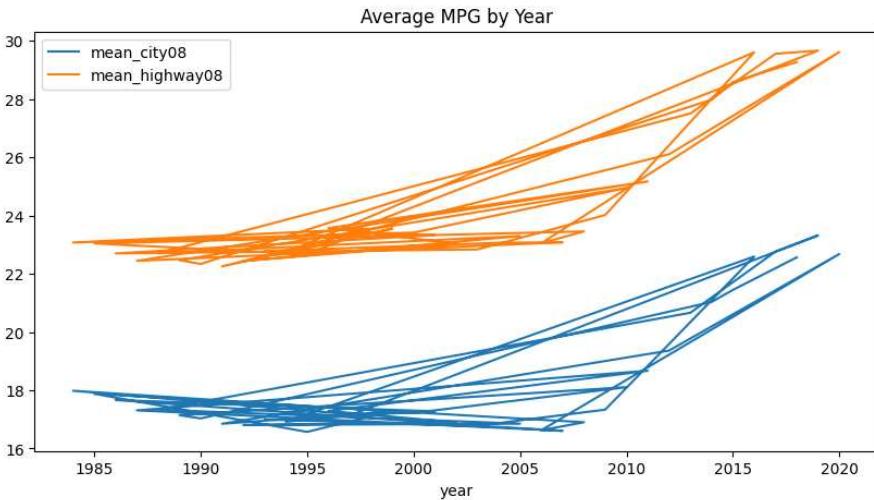


Figure 8.2: Bad plot of average mileage.

Note

If the plot looks weird, it might be because the *year* column is not sorted. We can fix this by adding an ORDER BY clause to the SQL query. Remember that DuckDB and Polars run the groupby operation in parallel, and the result is not sorted by the groupby column.

We have a plot, but it looks terrible. Upon closer inspection of the DuckDB results, we see that the *year* column is not sorted. DuckDB, like Polars, runs the groupby operation in parallel. The result is not sorted by the groupby column. We can fix this by adding an ORDER BY clause to the SQL query.

Here's the complete round trip from Polars to DuckDB to Pandas.

```
>>> import duckdb
>>> import matplotlib.pyplot as plt

>>> autos = tweak_auto(raw)
>>> sql = '''SELECT mean(city08) AS mean_city08,
...           mean(highway08) AS mean_highway08,
...           year
...      FROM autos
...     GROUP BY year
...    ORDER BY year'''
>>> agg = duckdb.sql(sql)
>>> fig, ax = plt.subplots(figsize=(10, 5))
```

8. Data Import and Export

```
>>> (agg
...     .pl()
...     .to_pandas()
...     .set_index('year')
...     .plot(ax=ax, title='Average MPG by Year'))
```

<Figure size 1000x500 with 1 Axes>

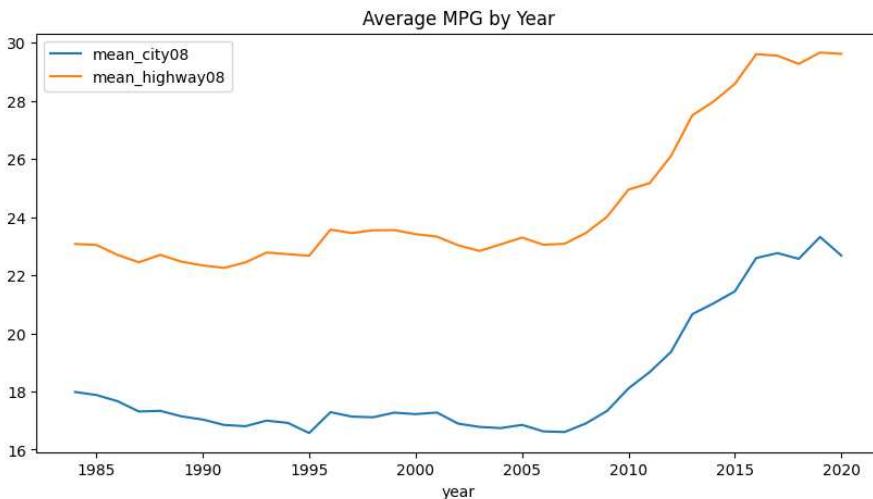


Figure 8.3: Sorted plot of average mileage by year.

That looks better.

8.12 Converting to Pandas

Polars has a `.to_pandas` method that will convert a DataFrame to a Pandas DataFrame. This should be efficient if you are using Pandas 2 with PyArrow and pass in the `use_pyarrow_extension_array=True` argument. Otherwise, it will be a slower copy operation.

To read from a Pandas DataFrame, use the `pl.from_pandas` function. This will create a Polars DataFrame from a Pandas DataFrame.

Let's round trip the `autos` data from Polars to Pandas and back to Polars.

```
>>> autos2 = (autos.to_pandas(use_pyarrow_extension_array=True)
...     .pipe(pl.from_pandas)
... )
```

Awesome, let's make sure that the round trip was successful by comparing the two DataFrames.

```
>>> autos2.equals(autos)
```

True

8.13 Working with Other Libraries

One advantage of Pandas is that it has a large ecosystem of libraries that work with it. Polars is a new library and doesn't have the same level of support. However, this is changing thanks to the efforts of the Python community. There is a consortium for having a consistent API for data manipulation libraries in Python.¹

Because Polars adopts this API, it should be possible to use Polars with libraries that support the API.

8.14 Using XGBoost to Predict Mileage

Let's use the XGBoost library to predict the mileage of a car based on its features. We'll use the *autos* data to train the model and then use the model to predict the mileage of a car.

An XGBoost regression model takes a matrix of features and a vector of labels. The features are the columns that we will use to predict the label. The label is the column that we want to predict. In this case, we want to predict the *city08* column based on the other columns.

I will make a variable called *X* that contains the features and a variable called *y* that includes the labels.

```
>>> import polars.selectors as cs
>>> X = (autos
...     .select(cs.numeric() - cs.matches('city08|highway08'))
... )
>>> y = (autos.select(pl.col('city08')))
```

Here's what the *X* variable looks like. It is a DataFrame with all the columns except for *city08*.

```
>>> print(X)
shape: (41_144, 5)
| year    displ   cylinders   barrels08   num_gears |
| i16      f32       u8          f32           i8        |
|-----|
| 1985    2.00      4          15.70        5        |
| 1985    4.90     12          29.96        5        |
| 1985    2.20      4          12.21        5        |
| .         .         .          .            .        |
| 1993    2.20      4          15.70        4        |
```

¹<https://data-apis.org/>

8. Data Import and Export

1993	2.20	4	15.70	5
1993	2.20	4	18.31	4

Here's what the `y` variable looks like. It is a Series with the `city08` column. Generally, when using scikit-learn and similar libraries (like XGBoost), the features are a matrix (DataFrame), and the label is a vector (Series). That's why the `X` variable is capitalized. In linear algebra, a matrix is often represented by a capital letter by convention, and a vector is often represented by a lowercase letter.

XGBoost supports Polars having `y` as a dataframe with a single column.

```
>>> print(y)
shape: (41_144, 1)
| city08 |
u8
|
19
9
23
.
18
18
16
```

Now, I need to split the data into training and test sets. This is a required step for many machine-learning tasks. The training set is used to train the model, and the test set is used to evaluate the model on data that it has not seen before. This is important to ensure the model does not overfit the training data. Overfitting occurs when the model memorizes the training data and does not work well on new data.

I'll use the sci-kit learn library to split the data into training and test sets. Then, I'll make an `XGBRegressor` model and train it on the training data using the `.fit` method. I can evaluate the model's R-squared score on the test data using the `.score` method. I do this with the testing data to understand how well the model will work on new data.

```
>>> import xgboost as xgb
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> xg = xgb.XGBRegressor()
>>> xg.fit(X_train, y_train)
>>> xg.score(X_test, y_test)
0.994349611956841
```

It looks like my R-squared score is 0.99. That value ranges from 0 to 1. A value of 1 means that the model perfectly predicts the label. A value of 0 means that the model is no better than guessing. .99 is pretty good. It means that 99% of the variance in the label can be explained by the features.

8.15 Plotting Residuals with Matplotlib

Another library that you will want to use with Polars is a plotting library. Polars recently added a plotting library building on top of hvplot. However, it is easy to get Polars to work with Matplotlib as well. Matplotlib is a popular Python plotting library. Pandas and many other libraries use it.

I will use Matplotlib to plot the model's *residuals*. The residuals are the difference between the actual and the predicted mileage. We can plot the residuals to see how well the model is doing. If the residuals are randomly distributed around zero and relatively small, then the model is doing a good job. If the residuals are large and have a pattern, then the model is not doing a good job. Quantifying how good the residuals are without plotting them can be hard.

First, I will calculate the residuals by subtracting the predicted mileage from the actual mileage. Then, I will plot the residuals using Matplotlib. To do the math, we need to convert *y* to a series so it works with NumPy.

```
>>> residuals = y_test.to_series() - xg.predict(X_test)
>>> print(residuals)
shape: (10_286,)
Series: 'city08' [f32]
[
    -0.26
    -0.13
    0.64
    .
    0.78
    -0.45
    -0.21
]
```

Now, let's create a scatter plot of the residuals against the actual mileage. We will use the Matplotlib library to do this.

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(figsize=(10, 5))
>>> ax.scatter(y_test, residuals)

<Figure size 1000x500 with 1 Axes>
```

That is a decent basic residual plot. I won't get into the details of interpreting residual plots here. But generally, you want the residuals to be randomly distributed around zero and relatively small. If the residuals have a pattern or are large, then the model is not doing well.

Let's fix up the plot by adding a title and labels and adjusting the alpha value of the points. The alpha value is the transparency of the points. I'll set it to 0.1 so we can see the density of the points.

8. Data Import and Export

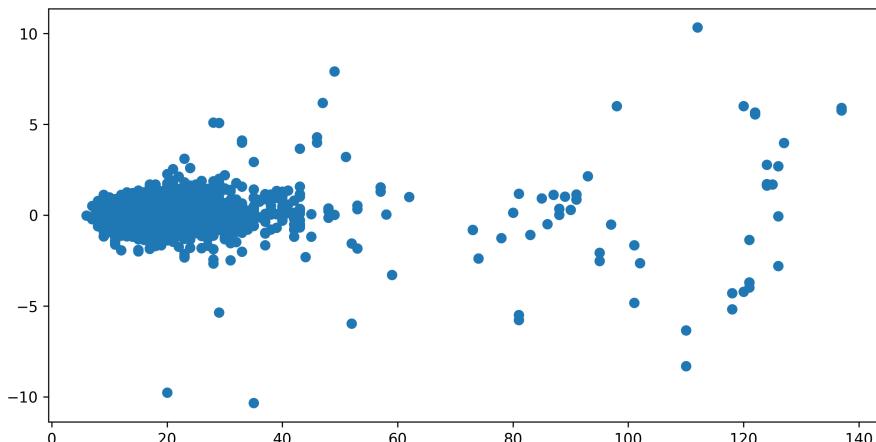


Figure 8.4: Basic residual plot.

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(figsize=(10, 5))
>>> ax.scatter(pl.DataFrame(xg.predict(X_test)) , residuals, alpha=0.1)
>>> ax.set_title('City MPG Residuals vs. Predicted City MPG')
>>> ax.set_xlabel('Predicted City MPG')
>>> ax.set_ylabel('Residuals')
```

<Figure size 1000x500 with 1 Axes>

We can see that many operations that support NumPy or Pandas will also work with Polars. However, there are some cases where the result is not what you expect. This is because Polars is a new library and doesn't have the same level of support as NumPy and Pandas. Tread carefully, and check your results as you go.

8.16 PCA of the Autos Data

Let's try using the scikit-learn library for Principal Component Analysis (PCA) on the *autos* data. PCA is a technique for dimensionality reduction. It finds the most important features in the data and reduces the number of features to a smaller set. This can be useful for machine learning algorithms that don't perform well with many features.

We need to prepare the data by standardizing it. This will give each feature a mean of zero and a standard deviation of one. This is generally a good idea for PCA because the PCA algorithm is sensitive to the scale of the data.

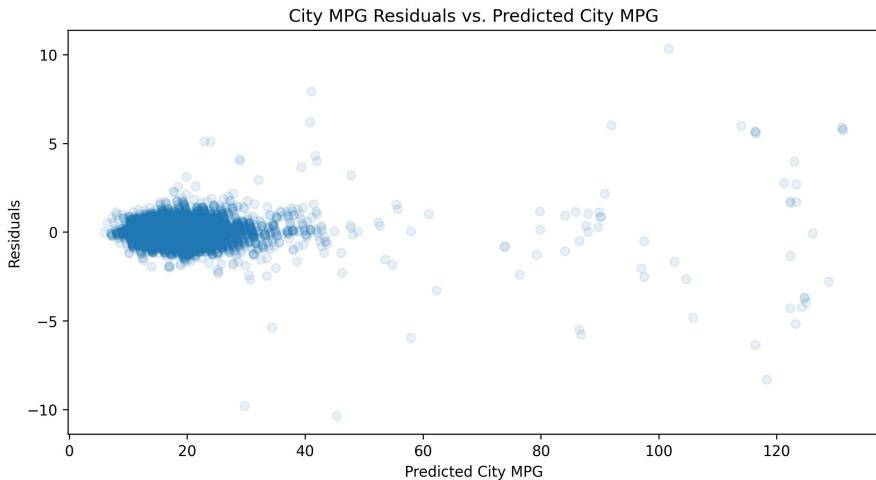


Figure 8.5: A residual plot of our XGBoost regression model. This plot visualizes the residuals (the difference between the predicted mileage and the actual mileage) against the predicted mileage. In a good model, the residuals should be randomly distributed around zero.

Since Scikit-learn version 1.4, you can use `set_config` to have the library output Polars datastructures instead of NumPy arrays.

```
>>> from sklearn import preprocessing, decomposition
>>> import sklearn
>>> sklearn.set_config(transform_output='polars')
>>> std = preprocessing.StandardScaler()
>>> X_std = std.fit_transform(
...     autos.select(pl.col(['displ', 'cylinders', 'barrels08', 'city08',
...                         'highway08']))
...     .fill_null(0)))
>>> pca = decomposition.PCA(n_components=2)
>>> res = pca.fit_transform(X_std)
```

Let's look at the first two principal components. (Sadly, I wish these were labelled as PC_1 and PC_2 , but I digress.)

```
>>> print(res)
shape: (41_144, 2)
| pca0    pca1 |
| f64     f64  |
| 1.05   -0.86 |
| -4.47   1.11 |
```

8. Data Import and Export

2.03	-0.05
.	.
0.87	-0.92
0.87	-0.92
0.32	-1.23

I will create a scatter plot of the first two principal components. This will give us a sense of how the data is distributed in the first two dimensions. I will color this by the number of cylinders in the car. Note that I'm updating the `res` dataframe with the `cylinders` column and using that to color the result.

```
>>> def fix_naming(name):
...     return f'{name.upper()[:-2]}{int(name[-1])+1}'
>>> (res
... .rename(fix_naming)
... .with_columns(color=autos['cylinders'])
... .plot.scatter('PC1', 'PC2', color='color',
...              title='PCA of Autos', cmap='viridis')
... )
```

:Scatter [PC1] (PC2,color)

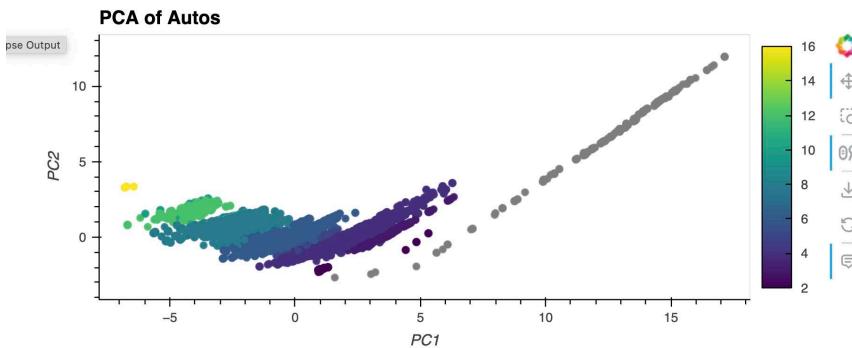


Figure 8.6: Scatter plot of the first two principal components of the autos data. The points are colored by the number of cylinders in the car.

The number of cylinders impacts both the first and second principal components. I'm not going to discuss PCA much more here. We can see that scikit-learn works with Polars as well. You can even have scikit-learn output Polars natively.

8.17 Configuration

Polars has a configuration system that allows you to set options for the library. You can set options globally or for a specific DataFrame.

I don't generally mess around with these too much. For example, in this book, I set the table width for output, the float precision, and the number of columns to display.

If you use this, I would recommend putting it at the top of your notebook along with your imports so that you can see the settings you are using and that you use them consistently.

```
import polars as pl
pl.Config.set_tbl_width_chars(70)
pl.Config.set_float_precision(2)
pl.Config.set_tbl_cols(6)
```

You can also temporarily set options using the `pl.Config` object as a context manager. This is useful to set options for a specific code block.

To temporarily set the table width to 70 characters, you can use the `pl.Config` object as a context manager like this:

```
with pl.Config(set_tbl_width_chars=70):
    print(df)
```

Here are the configuration options.

load(cfg) Loads the Polars configuration (output of `Config.save`) from an in-memory representation.

load_from_file(file) Loads the Polars configuration from a file.

restore_defaults() Restores the default Polars configuration settings.

save() Saves the current Polars configuration to JSON string.

save_to_file(file) Saves the current Polars configuration to a file.

set_ascii_tables(active=True) Sets the use of ASCII characters for printing tables. Otherwise, UTF-8 characters are used.

set_auto_structify(active=False) Enables or disables automatic structification in data processing.

set_float_precision(precision) Sets the precision level for floating point numbers.

set_fmt_float(fmt='mixed') Configures the formatting for floating-point types. 'mixed' will use limit the number of decimal places and use scientific notation for very large or small numbers. 'full' will use the full precision of the floating-point.

set_fmt_str_lengths(n=None) Sets the maximum length for formatting string data.

set_fmt_table_cell_list_len(n=None) Sets the length of lists displayed in table cells.

8. Data Import and Export

set_streaming_chunk_size(size=None) Configures the chunk size for streaming datasets. Number of rows per chunk.

set_tbl_cell_alignment(format) Sets the alignment for table cells. Can be 'LEFT', 'RIGHT', or 'CENTER'.

set_tbl_cell_numeric_alignment(format) Sets the numeric alignment for table cells. Can be 'LEFT', 'RIGHT', or 'CENTER'.

set_tbl_cols(n) Configures the number of columns to display in tables.

set_tbl_column_data_type_inline(active=True) Determines whether to display column data types inline with column names in tables.

set_tbl_dataframe_shape_below(active=True) Display of DataFrame shapes below the table.

set_tbl_formatting(format=None, rounded_corners=False) Sets the table formatting style. format can be "ASCII_FULL", "ASCII_FULL_CONDENSED", "ASCII_NO_BORDERS", "ASCII_BORDERS_ONLY", "ASCII_BORDERS_ONLY_CONDENSED", "ASCII_HORIZONTAL_ONLY", "ASCII_MARKDOWN", "NOTHING". Can use Utf-8 by replacing "ASCII" with "UTF8". rounded_corners (Utf-8 only) can be True or False.

set_tbl_hide_column_data_types(active=True) Hides the data types of columns in table displays.

set_tbl_hide_column_names(active=True) Hides the column names in table displays.

set_tbl_hide_dataframe_shape(active=True) Hides the DataFrame shape in table displays.

set_tbl_hide_dtype_separator(active=True) Hides the data type separator in tables.

set_tbl_rows(n) Sets the number of rows to display in tables.

set_tbl_width_chars(width) Sets the width of tables in characters.

set_verbose(active=True) Enables or disables verbose output in operations.

state(*, if_set=False, env_only=False) Retrieves the current state of the Polars configuration as a dict.

8.18 Chapter Methods and Functions

Table 8.1: Methods and functions from chapter

Method	Description
<code>df.write_csv(file, *, include_bom=False, include_header=True, separator=',', line_terminator='\n', quote_char="'", batch_size=1024, datetime_format=None, date_format=None, time_format=None, float_precision=None, null_value='', quote_style='necessary')</code>	Write DataFrame to a CSV file. If <code>file</code> set to <code>None</code> , the output is returned as a string instead. <code>datetime_format</code> and <code>date_format</code> are defined in the Rust chrono crate. <code>quote_style</code> can be <code>'necessary'</code> , <code>'always'</code> , <code>'non_numeric'</code> , or <code>'never'</code> . <code>float_precision</code> is the number of decimal places to write, applied to both <code>Float32</code> and <code>Float64</code> datatypes.
<code>pl.read_csv(source, *, has_header=True, columns=None, new_columns=None, separgator=',', comment_prefix=None, quote_char="'", skip_rows=0, dtypes=None, schema=None, null_values=None, missing_utf8_is_empty_string=False, ignore_errors=False, try_parse_dates=False, n_threads=None, infer_schema_length=100, batch_size=8192, n_rows=None, encoding='utf8', low_memory=False, rechunk=False, use_pyarrow=False, storage_options=None, skip_rows_after_header=0, row_count_name=None, row_count_offset=0, sample_size=1024, eol_char='\n', raise_if_empty=True, truncate_ragged_lines=False)</code>	Read a CSV file into a DataFrame. <code>null_values</code> can be a string, a list of strings, or a dictionary mapping column names to strings. <code>try_parse_dates</code> will try to parse ISO8601-ish formats. If it fails, it will fall back to the string type.

8. Data Import and Export

Method	Description
<code>df.write_json(file=None, *, pretty=False, row_oriented=False)</code>	Write DataFrame to a JSON file. If <code>file</code> set to <code>None</code> , the output is returned as a string instead. <code>pretty</code> will pretty serialize the JSON. <code>row_oriented</code> will write to row-oriented JSON. This is slower but is a more common JSON format. Each row of the DataFrame will be written as a separate JSON object.
<code>pl.read_json(source, *, schema=None, infer_schema_length=100, schema_overrides=None, batch_size=8192, n_rows=None, encoding='utf8', low_memory=False, rechunk=False, use_pyarrow=False, storage_options=None, eol_char='\n', raise_if_empty=True, truncate_ragged_lines=False)</code>	Read a JSON file into a DataFrame. <code>schema</code> can be a string, a list of strings, or a dictionary mapping column names to strings. <code>schema_overrides</code> can be a dictionary mapping column names to strings.

Method	Description
<code>df.write_excel(workbook=None, worksheet=None, position='A1', table_style=None, table_name=None, column_formats=None, dtype_formats=None, conditional_formats=None, header_format=None, column_totals=None, column_widths=None, row_totals=None, row_heights=None, sparklines=None, float_precision=3, has_header=True, autofilter=True, autofit=False, hidden_columns=None, hide_gridlines=False, sheet_zoom=None, freeze_panes=None)</code>	Write DataFrame to an Excel file. If <code>workbook</code> set to <code>None</code> , a new workbook named <code>dataframe.xlsx</code> is created in the working directory. <code>worksheet</code> defaults to "Sheet1" in a new workbook. <code>position</code> is the starting cell location for writing the table. <code>table_style</code> can be a named Excel table style or a dictionary of style options. <code>column_formats</code> is a dictionary mapping column names or selectors to Excel format strings. <code>dtype_formats</code> is a dictionary mapping data types to default Excel formats. <code>conditional_formats</code> is a dictionary specifying conditional formatting for columns. <code>header_format</code> is a dictionary of <code>xlsxwriter</code> format options for the table header row. <code>column_totals</code> defines the total row for the table. <code>column_widths</code> sets table column widths. <code>row_totals</code> adds a row-total column to the table. <code>row_heights</code> sets the height of rows intersecting with the table body. <code>sparklines</code> is a dictionary defining sparklines to be added to the table.

8. Data Import and Export

Method	Description
<code>pl.read_excel(source, *, sheet_id=None, sheet_name=None, engine=None, xlsx2csv_options=None, read_options=None, schema_overrides=None, raise_if_empty=True)</code>	<p>Read an Excel file into a DataFrame. <code>sheet_id</code> specifies the sheet number(s) to read from the Excel file. If set to 0, all sheets are read into a dictionary keyed by sheet names. <code>sheet_name</code> is the name(s) of the sheet(s) to be read. <code>engine</code> is the library used to parse the Excel file. Options include '<code>xlsx2csv</code>' (default), '<code>openpyxl</code>' (slower, with additional type inference), '<code>pyxlsb</code>' (for Excel Binary Workbooks), and '<code>calamine</code>'. <code>xlsx2csv_options</code> are additional options for the <code>xlsx2csv.Xlsx2csv()</code> function.</p> <p><code>read_csv_options</code> are extra options for parsing the CSV file generated by <code>xlsx2csv.Xlsx2csv().convert()</code>.</p> <p><code>schema_overrides</code> allows for the specification or override of data types for one or more columns. Use <code>engine='calamine'</code> for fast loading. <code>read_options</code> for calamine include <code>name</code> (sheet name), <code>header_row</code> (0-based index with header), <code>column_names</code> (override column names), <code>n_rows</code> (number of rows to load), <code>skip_rows</code> (how many rows to skip after header), and <code>schema_sample_rows</code> (how many rows used to determine schema).</p>

Method	Description
<code>df.write_parquet(file, *, compression='zstd', compression_level=None, statistics=False, row_group_size=None, use_pyarrow=False, pyarrow_options=None)</code>	<p>Write DataFrame to a Parquet file. <code>compression</code> specifies the compression technique for the Parquet file. Options include 'lz4', 'uncompressed', 'snappy', 'gzip', 'lzo', 'brotli', and 'zstd'. <code>compression_level</code> is the degree of compression applied, with higher levels resulting in smaller file sizes. Each compression type has its range, like 0-10 for 'gzip', 0-11 for 'brotli', and 1-22 for 'zstd'. <code>statistics</code> appends statistical data to the Parquet file headers, though it increases computational load.</p> <p><code>row_group_size</code> determines the number of rows in each row group, defaulting to the square of 512. <code>use_pyarrow</code> toggles the use of the PyArrow C++ implementation for Parquet instead of the Rust version, offering broader feature support.</p> <p><code>pyarrow_options</code> are parameters for <code>pyarrow.parquet.write_table</code>. Using <code>partition_cols</code> here allows dataset partitioning similar to Spark's approach, writing to a directory structure.</p>

8. Data Import and Export

Method	Description
<pre>pl.read_parquet(source, *, columns=None, n_rows=None, use_pyarrow=False, memory_map=True, storage_options=None, parallel='auto', row_count_name=None, row_count_offset=0, low_memory=False, pyarrow_options=None, use_statistics=True, rechunk=True)</pre>	Read a Parquet file into a DataFrame. <code>columns</code> specifies the columns to select. <code>n_rows</code> stops reading from the Parquet file after reading <code>n_rows</code> . <code>use_pyarrow</code> toggles the use of PyArrow instead of the Rust native Parquet reader. <code>memory_map</code> maps the underlying file to memory, increasing performance. <code>storage_options</code> are extra options for <code>fsspec.open()</code> or a particular storage connection. <code>parallel</code> determines the direction of parallelism, with options including <code>'auto'</code> , <code>'columns'</code> , <code>'row_groups'</code> , and <code>'none'</code> . <code>row_count_name</code> inserts a row count column with the given name into the DataFrame. <code>row_count_offset</code> sets the offset to start the row count column. <code>low_memory</code> reduces memory pressure at the expense of performance. <code>pyarrow_options</code> are keyword arguments for <code>pyarrow.parquet.read_table</code> . <code>use_statistics</code> uses statistics in the Parquet to determine if pages can be skipped from reading. <code>rechunk</code> aggregates the chunks into a single array, ensuring that all columns are contiguous in memory.
<pre>df.write_database(table_name, Write DataFrame to a SQL database. connection, *, if_exists='fail', engine='sqlalchemy')</pre>	<code>table_name</code> specifies the name of the table within the SQL database where the data will be written. <code>connection</code> is the connection URI string that defines the database's location and access credentials. <code>if_exists</code> defines the action to take if the specified table already exists. Options include <code>'append'</code> (to add data to the existing table), <code>'replace'</code> (to overwrite the existing table), and <code>'fail'</code> (to raise an error if the table already exists). <code>engine</code> selects the backend engine for database operations. Options include <code>'sqlalchemy'</code> and <code>'adbc'</code> , which determine how the DataFrame is written to the database.

Method	Description
<code>pl.read_database(query, connection, *, iter_batches=False, batch_size=None, schema_overrides=None, execute_options=None)</code>	Read data from a SQL database into a DataFrame. <code>query</code> is the SQL query to be executed. <code>connection</code> is an active connection or cursor object to execute the query against. <code>iter_batches</code> returns an iterator of DataFrames with each DataFrame representing a batch of data. <code>batch_size</code> specifies the size of each batch when <code>iter_batches</code> is <code>True</code> . <code>schema_overrides</code> is a dictionary for overriding column data types in the resulting DataFrame. <code>execute_options</code> are passed to the query execution method as keyword arguments.
<code>pl.read_database_uri(query, uri, *, partition_on=None, partition_range=None, partition_num=None, protocol=None, engine=None, schema_overrides=None)</code>	Read data from a SQL database into a DataFrame using a connection URI string. <code>query</code> is the SQL query to be executed. <code>uri</code> is the connection URI string, beginning with the backend's driver name. <code>partition_on</code> identifies the column to use for partitioning the result set. <code>partition_range</code> defines the value range for the partition column. <code>partition_num</code> determines the number of partitions to generate for the query result. <code>protocol</code> is a backend-specific directive for the data transfer protocol. <code>engine</code> selects the engine for database read operations. Options include ' <code>connectorx</code> ' and ' <code>adbc</code> '. <code>schema_overrides</code> is a dictionary mapping column names to data types (<code>dtypes</code>) to override the schema inferred from the data returned by the query.
<code>df.to_pandas(*, use_pyarrow_extension_array= False)</code>	Cast to a Pandas DataFrame. <code>use_pyarrow_extension_array</code> uses PyArrow-backed extension arrays instead of NumPy arrays for each column of the Pandas DataFrame, allowing zero-copy operations and preservation of null values. Subsequent operations on the resulting Pandas DataFrame may trigger conversion to NumPy arrays if that operation is not supported by PyArrow compute functions.

8. Data Import and Export

Method	Description
<code>pl.from_pandas(data, *, schema_overrides=None, rechunk=True, nan_to_null=True, include_index=False)</code>	Construct a Polars DataFrame or Series from a Pandas DataFrame, Series, or Index. <code>schemas_overrides</code> supports the override of inferred types for one or more columns. <code>rechunk</code> ensures that all data is in contiguous memory. <code>nan_to_null</code> converts NaN values to None if True. <code>include_index</code> loads any non-default Pandas indexes as columns.
<code>df.equals(other, *, null_equal=True)</code>	Return whether two dataframes are equal
<code>col.cast(data_type)</code>	Cast the column to a new data type. <code>data_type</code> can be a string or a <code>pl.DataType</code> object.
<code>col.str.to_datetime(format=None, *, time_unit=None, time_zone=None, strict=True, exact=True, cache=True, use_earliest=None, ambiguous='raise')</code>	Convert a Utf8 column into a Datetime column. <code>format</code> is the format to use for conversion. <code>time_unit</code> is the unit of time for the resulting Datetime column. <code>time_zone</code> is the time zone for the resulting Datetime column. <code>strict</code> raises an error if any conversion fails. <code>exact</code> requires an exact format match. <code>cache</code> uses a cache of unique, converted datetimes to apply the conversion. <code>use_earliest</code> determines how to deal with ambiguous datetimes. <code>ambiguous</code> determines how to deal with ambiguous datetimes.
<code>cs.expand_selector(target, selector)</code>	Expand a selector to column names with respect to a specific frame or schema target. <code>target</code> is a Polars DataFrame, LazyFrame, or schema. <code>selector</code> is an arbitrary Polars selector (or compound selector).
<code>cs.numeric()</code>	Selects numeric columns.
<code>cs.matches(pattern)</code>	Selects columns that match a regular expression pattern.
<code>df.explode(columns, *more_columns)</code>	Explode the DataFrame to long format by exploding the given columns. <code>columns</code> is a column name, expression, or a selector defining the underlying columns to be exploded. The underlying columns being exploded must be of List or Utf8 datatype. <code>more_columns</code> are additional names of columns to explode, specified as positional arguments.

8.19 Summary

This chapter taught us how to read and write data with Polars. We learned to read and write CSV, JSON, Excel, Parquet, and SQL data. We also saw how to interact with other libraries in the Python ecosystem, like XGBoost and scikit-learn. We learned how to use Polars with Matplotlib for plotting and how to use Polars with the scikit-learn library for machine learning. We also learned about the Polars configuration system and how to set options for the library.

8.20 Exercises

1. Export the *autos* DataFrame to a CSV file and read it back into Polars. Check if the data is the same.
2. Export the *autos* DataFrame to a JSON file using the 'records' orientation and then read it back into Polars. Check if the data is the same.
3. Create an Excel file with the *autos* DataFrame and then read it back into Polars. Check if the data is the same.
4. Export the *autos* DataFrame to a Parquet file and then read it back into Polars. Check if the data is the same.
5. Export the *autos* DataFrame to a SQLite database and read it back into Polars. Check if the data is the same.
6. Use the XGBoost library to predict the barrels of gas a car uses based on its features. Evaluate the model using the R-squared score.

Chapter 9

Being Lazy and Streaming

Because Polars is built on top of a query engine, it can be smart about how it executes your code. For example, it can figure out that due to filtering of both rows and columns, it doesn't need to load the entire dataset into memory. It can also rearrange the order of operations to be more efficient. This is called query planning and is enabled because Polars can lazily evaluate your code.

Polars can also handle data that is larger than your memory. It does this by streaming data from disk and processing it in chunks. This chapter will explore how Polars does this and how you can take advantage of it.

9.1 Loading Data

We are going to be using the Fuel Economy dataset from the EPA.

I will show a chunk of code summarizing the yearly city mileage by country of origin. This chain of operations exercises a lot of Polars' features. First, I read the data from a CSV file. Then I will create a new column called *origin* that is the country of origin. Then, I will filter out unknown origins and years after 2019. I will select a few of the 83 columns to keep. Then, I will group the results by origin and year and compute the mean city mileage. Next, I will pivot the data so that each country is a column and the years are the rows. Finally, I will sort the data by year.

Hopefully, this chain of code at the bottom is starting to look familiar. It's a lot like the code we've been writing in the previous chapters. The more you practice and work with Polars, the easier it will become to read and write code like this.

The `make_to_origin_expr` function is a little more complicated. It uses the dictionary of mappings to build up a big if/else expression. Instead of having to write (note the ellipsis indicating that many lines are omitted) this:

```
pl.when(pl.col('make') == 'Chevrolet')
    .then('USA')
    .when(pl.col('make') == 'Ford')
    .then('USA')
```

9. Being Lazy and Streaming

```
...  
.when(pl.col('make') == 'Tesla')  
.then('USA')  
.otherwise('Unknown')
```

I let the function generate the expression for me.

```
>>> import polars as pl  
  
>>> def make_to_origin_expr(make_col):  
...     # Dictionary mapping car makes to countries of origin  
...     origin_dict = {  
...         'Chevrolet': 'USA',  
...         'Ford': 'USA',  
...         'Dodge': 'USA',  
...         'GMC': 'USA',  
...         'Toyota': 'Japan',  
...         'BMW': 'Germany',  
...         'Mercedes-Benz': 'Germany',  
...         'Nissan': 'Japan',  
...         'Volkswagen': 'Germany',  
...         'Mitsubishi': 'Japan',  
...         'Porsche': 'Germany',  
...         'Mazda': 'Japan',  
...         'Audi': 'Germany',  
...         'Honda': 'Japan',  
...         'Jeep': 'USA',  
...         'Pontiac': 'USA',  
...         'Subaru': 'Japan',  
...         'Volvo': 'Sweden',  
...         'Hyundai': 'South Korea',  
...         'Chrysler': 'USA',  
...         'Tesla': 'USA'  
...     }  
...     expr = None  
...     col = pl.col(make_col)  
...     for k, v in origin_dict.items():  
...         if expr is None:  
...             expr = pl.when(col == k).then(pl.lit(v))  
...         else:  
...             expr = expr.when(col == k).then(pl.lit(v))  
...     expr = expr.otherwise(pl.lit('Unknown'))  
...     return expr  
  
>>> df_pl = pl.read_csv('data/vehicles.csv', null_values='NA')
```

```

>>> result = (df_pl
...     .with_columns(
...         pl.col('createdOn')
...             .str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_expr('make'))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08',
...         'origin', 'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )
... )

>>> print(result)
shape: (36, 6)
  year | South Korea | Japan | USA | Germany | Sweden
  --- | --- | --- | --- | --- | ---
  i64 | f64 | f64 | f64 | f64 | f64
  1984 | null | 21.71 | 16.46 | 20.06 | 18.54
  1985 | null | 21.53 | 16.58 | 18.18 | 17.53
  1986 | 24.00 | 20.61 | 16.42 | 18.07 | 19.00
  1987 | 24.00 | 20.09 | 16.04 | 17.72 | 17.56
  1988 | 23.33 | 19.92 | 16.09 | 16.50 | 17.71
  .
  .
  2015 | 22.14 | 23.89 | 21.47 | 20.77 | 20.67
  2016 | 23.14 | 26.10 | 24.28 | 20.80 | 21.11
  2017 | 28.84 | 25.96 | 23.41 | 21.09 | 22.27
  2018 | 29.67 | 25.71 | 22.99 | 20.77 | 22.74
  2019 | 32.55 | 27.05 | 23.63 | 21.06 | 22.29

```

Let's time this code and see how long it takes to run. Note that this code loads and processes the data.

```

>>> %%timeit
>>> df_pl = pl.read_csv('data/vehicles.csv', null_values='NA')
>>> result = (df_pl
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_expr('make'))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...         'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean()))

```

9. Being Lazy and Streaming

```
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
...
31.8 ms ± 1.67 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

On my machine, it takes around 30 milliseconds to run. In this case, this time is so tiny that it doesn't really matter if I can speed it up. With larger datasets, it can be important to understand how to optimize your code. I will show you how to do that in the next few sections.

9.2 The replace Method

It turns out that there is a dedicated method for replacing values in a column. It is called `.replace`. It is more efficient than our previous solution and easier to use.

```
>>> def make_to_origin_replace(make_col):
...     origin_dict = {
...         'Chevrolet': 'USA',
...         'Ford': 'USA',
...         'Dodge': 'USA',
...         'GMC': 'USA',
...         'Toyota': 'Japan',
...         'BMW': 'Germany',
...         'Mercedes-Benz': 'Germany',
...         'Nissan': 'Japan',
...         'Volkswagen': 'Germany',
...         'Mitsubishi': 'Japan',
...         'Porsche': 'Germany',
...         'Mazda': 'Japan',
...         'Audi': 'Germany',
...         'Honda': 'Japan',
...         'Jeep': 'USA',
...         'Pontiac': 'USA',
...         'Subaru': 'Japan',
...         'Volvo': 'Sweden',
...         'Hyundai': 'South Korea',
...         'Chrysler': 'USA',
...         'Tesla': 'USA'
...     }
...     return make_col.replace(origin_dict, default='Unknown')

>>> result = (df_pl
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make'))))
```

```

...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

```

It looks like it works. Let's time it.

```

>>> %%timeit
>>> df_pl = pl.read_csv('data/vehicles.csv', null_values='NA')
>>> result = (df_pl
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make'))))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

```

18.9 ms ± 494 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

That's a little bit faster. I'll take it!

9.3 Lazy Version Take One

Again, our example uses really small data, but I want to illustrate how lazy evaluation works. Up until now, most of our examples have been eager. That is, when I execute a method, it runs immediately. The code is not executed with lazy evaluation until I call `collect()`. Polars will try to determine the most efficient way to execute the code.

To get started with lazy mode, I will switch the `pl.read_csv` to `pl.scan_csv`. This will return a lazy DataFrame. The rest of the code is the same. Let's run it and see what happens.

```

>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')
>>> result = (df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make'))))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))

```

9. Being Lazy and Streaming

```
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

>>> print(result)
Traceback (most recent call last)

...
AttributeError: 'LazyFrame' object has no attribute 'pivot'
```

I got an error! It says the `.pivot` method is not supported on a lazy DataFrame. This is because Polars doesn't know how to execute the pivot operation lazily. It needs to know the values of a column that will become the new column headers. And it can't know that until it has read all the data. But I can run this code lazily up to the pivot operation. Let's try that. I will add a `.collect` right before the pivot.

```
>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')

>>> result = (df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make')))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .collect()
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

>>> print(result)
```

shape: (36, 6)

year	Japan	South Korea	Germany	USA	Sweden
---	---	---	---	---	---
i64	f64	f64	f64	f64	f64
1984	21.71	null	20.06	16.46	18.54
1985	21.53	null	18.18	16.58	17.53
1986	20.61	24.00	18.07	16.42	19.00
1987	20.09	24.00	17.72	16.04	17.56

1988	19.92	23.33	16.50	16.09	17.71
.
2015	23.89	22.14	20.77	21.47	20.67
2016	26.10	23.14	20.80	24.28	21.11
2017	25.96	28.84	21.09	23.41	22.27
2018	25.71	29.67	20.77	22.99	22.74
2019	27.05	32.55	21.06	23.63	22.29

That worked! Let's time it and see how long it takes.

```
>>> %%timeit
>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')

>>> result = (df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make')))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .collect()
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

12.8 ms ± 512 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

It looks like it took around 10 milliseconds (on my machine). That's a little faster than the eager version. It's pretty cool that we can improve the performance of our code by doing very little. Later on, we will see what it did to run faster.

Note

For more benchmarks, see the benchmarks on the Polars website^a that are representative of the performance you can expect from Polars with larger datasets.

^a<https://pola.rs/posts/benchmarks/>

9.4 Fetching Data

The `.fetch` method can get a chunk of data from a lazy DataFrame. Unlike `.collect`, which gets all the data, `.fetch` gets a specified number of rows. It allows you to run a *tracer bullet* through your code without loading the entire

9. Being Lazy and Streaming

dataset into memory. This is useful for debugging and understanding how Polars is executing your code. I

I will replace .collect with .fetch to use this method. I will fetch 5 rows of data. Let's see what happens.

```
>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')

>>> result = (df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make')))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     #.collect()
...     .fetch(5)
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )

>>> print(result)
shape: (2, 3)
year | USA | Japan |
--- | --- | --- |
i64 | f64 | f64 |
1985 | 16.50 | null |
1993 | null | 17.00 |
```

Because the first five rows only have cars that originate from Japan and the US, the pivot operation will only have two columns.

Note

Remember, our chain has a filter in it, so it is possible that the result might be empty. If you have groupby or filter operations, you might need to bump up the number of rows you fetch to see a real result.

9.5 Pandas Comparison

This isn't a Pandas book, but many reading it are familiar with Pandas. So, let's compare the performance of Polars to Pandas. I will port the above code to Pandas and see how long it takes to run. Note that Pandas doesn't have lazy evaluation, so I will run the eager code.

```
%%timeit
import pandas as pd
def make_to_origin(make):
    """
    Convert car make to country of origin.

    Args:
        make (str): Car make.

    Returns:
        str: Country of origin.
    """
    # Dictionary mapping car makes to countries of origin
    origin_dict = {
        'Chevrolet': 'USA',
        'Ford': 'USA',
        'Dodge': 'USA',
        'GMC': 'USA',
        'Toyota': 'Japan',
        'BMW': 'Germany',
        'Mercedes-Benz': 'Germany',
        'Nissan': 'Japan',
        'Volkswagen': 'Germany',
        'Mitsubishi': 'Japan',
        'Porsche': 'Germany',
        'Mazda': 'Japan',
        'Audi': 'Germany',
        'Honda': 'Japan',
        'Jeep': 'USA',
        'Pontiac': 'USA',
        'Subaru': 'Japan',
        'Volvo': 'Sweden',
        'Hyundai': 'South Korea',
        'Chrysler': 'USA',
        'Tesla': 'USA'
    }
    return origin_dict.get(make, "Unknown")

df_pd = pd.read_csv('data/vehicles.csv',
                    engine='pyarrow', dtype_backend='pyarrow')

(df_pd
 .assign(origin=lambda df: df['make'].apply(make_to_origin),
        # replace EST and EDT with offset in createdOn
        createdOn=lambda df: df['createdOn']
```

9. Being Lazy and Streaming

```
.str.replace('EDT', '-04:00').str.replace('EST', '-05:00')
)
.assign(
    createdOn=lambda df: pd.to_datetime(df['createdOn'],
        format='%a %b %d %H:%M:%S %z %Y', utc=True),
)
.query('origin != "Unknown" and year < 2020')
.loc[:, ['make', 'model', 'year', 'city08', 'highway08', 'origin',
    'createdOn']]
.groupby(['origin', 'year'])
.city08
.mean()
.unstack('origin')
)
```

On my machine, it took 99 milliseconds to run. That's about 10 times slower than Polars. And this is with a small dataset. If I had a larger dataset, the difference might be even greater.

9.6 Viewing Plans

Let's see if we can figure out what Polars did to make the code run faster. We can do this by looking at the query plan. The query plan is a tree of operations that Polars will execute. We can view the query plan by inspecting the lazy object.

```
>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')

>>> print(df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make')))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     #.collect()
... )
naive plan: (run LazyFrame.explain(optimized=True) to see the optimized
plan)
```

AGGREGATE

```
[col("city08").mean().alias("avg_city08")] BY [col("origin"),
    col("year")]
SELECT [col("make"), col("model"), col("year"), col("city08"),
```

```

col("highway08"), col("origin"), col("createdOn")] FROM
FILTER [([(col("origin")) != (String(Unknown))]) & ([(col("year")) <
(2020)])] FROM

WITH_COLUMNS:
[col("createdOn").str.strptime([String(raise)]),
col("make").replace([Series, Series, String(Unknown)]).alias("origin")]

Csv SCAN data/vehicles.csv
PROJECT */83 COLUMNS

```

This says it is a *naive* plan. It is the eager operations that would be executed if I had run the code eagerly. You read this from the bottom up.

The first operation is to *PROJECT* all 83 columns, which means it will read all 83 columns from the CSV file.

Then, it will make a new *origin* column (*WITH_COLUMNS*).

Then, it will filter out rows, select a few columns, and do the group by and mean.

If we use the `.explain` method, Polars will show us the optimized query plan for the entire chain of operations.

```

>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')

>>> print(df_pl_lazy
...     .with_columns(
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),
...         origin=make_to_origin_replace(pl.col('make')))
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',
...             'createdOn'])
...     .group_by(['origin', 'year'])
...     .agg(avg_city08=pl.col("city08").mean())
...     .explain()
...     #.collect()
... )
AGGREGATE
[col("city08").mean().alias("avg_city08")] BY [col("origin"),
col("year")] FROM
FAST_PROJECT: [year, city08, origin]
FILTER [([(col("origin")) != (String(Unknown))]) & ([(col("year")) <
(2020)])] FROM

WITH_COLUMNS:
[col("make").replace([Series, Series,
String(Unknown)]).alias("origin")]

```

9. Being Lazy and Streaming

```
Csv SCAN data/vehicles.csv  
PROJECT 3/83 COLUMNS
```

You'll notice that the *PROJECT* now says "3/83" columns. Polars is only going to read 3 columns from the CSV file. It also says *FAST_PROJECT* later on instead of *SELECT*. Even though my chain indicates:

```
.select(['make', 'model', 'year', 'city08', 'highway08', 'origin',  
        'createdOn'])
```

I'm only using three of those columns later, so it can basically ignore the rest. This is called *projection pushdown*.

9.7 Streaming

Let's look at the support for *streaming*. Streaming is loading portions of data at a time so the data doesn't overwhelm the memory. I like to describe Pandas as a *small data* tool. By small data, I mean that the data fits in memory. Polars is a small and *medium data* tool. It can handle data that is larger than memory but that still fits onto a single machine. It currently is not a *big data* tool. It can't handle data that is larger than a single machine.

Polars supports medium data by streaming data from disk. It does this by reading a chunk of data from disk, processing it, and then reading the next chunk. This is called *streaming*. To use streaming, you need to work with lazy dataframes and add the `stream=True` parameter to the `.collect` method. That's it!

Let's try it out. I don't have a bunch of large datasets sitting around on my computer. However, the `pl.scan_csv` method let's you pass in a list of files to load. And you can pass the same file multiple times. So, I will simulate medium data by loading the same file factor times.

```
>>> factor = 400  
>>> df_pl_lazy = pl.scan_csv(['data/vehicles.csv']*factor, null_values='NA')  
  
>>> df_pl_lazy = pl.scan_csv('data/vehicles.csv', null_values='NA')  
  
>>> print(df_pl_lazy  
...     .with_columns(  
...         pl.col('createdOn').str.to_datetime('%a %b %d %H:%M:%S %Z %Y'),  
...         origin=make_to_origin_replace(pl.col('make')))  
...     .filter((pl.col("origin") != "Unknown") & (pl.col("year") < 2020))  
...     .select(['make', 'model', 'year', 'city08', 'highway08', 'origin',  
...             'createdOn'])  
...     .group_by(['origin', 'year'])  
...     .agg(avg_city08=pl.col("city08").mean())  
...     .collect(streaming=True)
```

```
...     .pivot(index='year', columns='origin', values='avg_city08')
...     .sort('year')
... )
```

Note that we can use the Pandas `make_to_origin` function. However, because this is using a Python dictionary to do the lookup, we aren't using a Polars expression. So, Polars can't optimize it. And on my machine, if I swap the line:

```
origin = make_to_origin_expr('make')
```

for:

```
origin = pl.col('make').map_elements(make_to_origin)
```

It runs three times slower. When possible, use Polars expressions. Avoid using `.map_elements` when possible. This is a very powerful method, but it is also very slow.

When I bumped up this factor to 1,000, using the `.map_elements` code, it took ten minutes to run and used 13 GB of memory on my machine. However, using the `make_to_origin_replace` code took less than a minute. That's a huge difference!

Be aware that not all operations can be streamed. For example, the `pivot` operation can't be streamed. You will get an error if you try to stream a DataFrame containing a pivot.

9.8 Chapter Methods and Functions

9. Being Lazy and Streaming

Method/Function	Description
<pre>pl.scan_csv(source, *, has_header=True, separator=',', comment_prefix=None, quote_char='''', skip_rows=0, dtypes=None, schema=None, null_values=None, missing_utf8_is_empty_string=False, ignore_errors=False, cache=True, with_column_names=None, infer_schema_length=100, n_rows=None, encoding='utf8', low_memory=False, rechunk=True, skip_rows_after_header=0, row_count_name=None, row_count_offset=0, try_parse_dates=False, eol_char='\n', new_columns=None, raise_if_empty=True, truncate_ragged_lines=False) lazy.collect(*, type_coercion=True, predicate_pushdown=True, projection_pushdown= True, simplify_expression= True, slice_pushdown= True, comm_subplan_elim= True, comm_subexpr_elim= True, no_optimization= False, streaming= False, _eager= False)</pre>	Lazily read from a CSV file or multiple files via glob patterns. Materialize this LazyFrame into a DataFrame. Various optimizations can be turned on or off. Use streaming=True to process the query in batches to handle larger-than-memory data.

Method / Function	Description
<code>lazy.fetch(n_rows=500, *, type_coercion=True, predicate_pushdown=True, projection_pushdown=True, simplify_expression=True, no_optimization=False, slice_pushdown=True, comm_subplan_elim=True, comm_subexpr_elim=True, streaming=False)</code>	Collect a small number of rows for debugging purposes.
<code>lazy.explain(*, optimized=True, type_coercion=True, predicate_pushdown=True, projection_pushdown=True, simplify_expression=True, slice_pushdown=True, comm_subplan_elim=True, comm_subexpr_elim=True, streaming=False)</code>	Create a string representation of the query plan. Different optimizations can be turned on or off.
<code>df.lazy()</code>	Convert a DataFrame to a LazyFrame.

9.9 Summary

This chapter was about how Polars can be smart about how it executes your code. Because of its smarts, it can handle data larger than your memory. It does this by streaming data from disk and processing it in chunks. We saw how to use the `.scan_csv` method to read data lazily. We also saw how to use the `.collect` method to materialize the data. We saw how to view the query plan using the `.explain` method. Remember that not all methods are available when using streaming. For example, the `pivot` method can't be streamed.

9.10 Exercises

1. Determine which methods can be streamed and which can't. (Hint: use the built-in `dir` function to inspect the methods of a DataFrame.)
2. When would lazy evaluation be useful? When would it not be useful?
3. What is the difference between the `pl.scan_csv` and `pl.read_csv` methods?
4. What does the `.lazy` method do? When would you use it?
5. What does the `.collect` method do? When would you use it?

Chapter 10

Porting from Pandas

Many people are coming to Polars from Pandas. This chapter will show you how to port your Pandas code to Polars.

10.1 The 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 that we find in the wild.

I recommend this book if you're interested in algorithmic short-selling. Here's the first part of the 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],
```

¹<https://github.com/PacktPublishing/Algorithmic-Short-Selling-with-Python-Published-by-Packt/blob/main/Chapter%2013/Chapter%2013.ipynb>

10. Porting from Pandas

```
'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)  
  
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,:]
```

The existing code structure starts with library imports. This is followed by several 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 dataframe. To make this code easy to reproduce, I want to limit the number of times I hit a web service. I will just read the data from a CSV instead.

The resulting `port` dataframe looks like this:

```
>>> print(port)  
   Beta Shares    rSL  Side  rCost  rPrice    Cost    Price  
QCOM  1.34 -1900  42.75   -1  28.58  26.63  109.25  116.36  
TSLA  2.00  -100  231.00   -1  32.79  59.85  125.35  261.47  
NFLX  0.75  -400  156.00   -1  77.90  96.28  297.75  420.61  
DIS   1.20  -800   54.20   -1  22.67  20.23  86.67   88.39  
PG    0.41 -5500   37.50   -1  39.32  34.29  150.30  149.79  
MMM   0.95  1600   42.75    1  30.92  22.95  118.20  100.27  
IBM   1.23  1800   29.97    1  35.92  29.97  137.30  130.91  
BRK-B  0.90  2800   59.97    1  79.19  77.12  302.69  336.91  
UPS   1.05  1100   39.97    1  45.11  39.46  172.42  172.37  
F     1.15  20800   2.10    1   2.75   3.27  10.51   14.28
```

Then there is more code to calculate returns on this data.

```
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

```

The end result is a dataframe called `price_df` that looks like this:

```

>>> print(price_df)
              BRK-B      DIS       F      IBM   ...
Date
2022-12-22 00:00:00-05:00  302.69  86.67  10.51  137.30   ...
2022-12-23 00:00:00-05:00  306.49  88.01  10.55  138.05   ...
2022-12-27 00:00:00-05:00  305.55  86.37  10.41  138.80   ...
2022-12-28 00:00:00-05:00  303.43  84.17  10.17  136.46   ...
2022-12-29 00:00:00-05:00  309.06  87.18  10.72  137.47   ...

```

10. Porting from Pandas

```
...          ...          ...          ...          ...          ...
2023-06-15 00:00:00-04:00 339.82  92.94  14.45  138.40  ...
2023-06-16 00:00:00-04:00 338.31  91.32  14.42  137.48  ...
2023-06-20 00:00:00-04:00 338.67  89.75  14.22  135.96  ...
2023-06-21 00:00:00-04:00 338.61  88.64  14.02  133.69  ...
2023-06-22 00:00:00-04:00 336.91  88.39  14.28  130.91  ...

          Returns_Long Returns_Short Returns \
Date
2022-12-22 00:00:00-05:00      NaN        NaN        NaN
2022-12-23 00:00:00-05:00     0.007     -0.002     0.005
2022-12-27 00:00:00-05:00     0.005      0.000     0.005
2022-12-28 00:00:00-05:00    -0.008      0.016     0.008
2022-12-29 00:00:00-05:00     0.012      0.002     0.014
...
          ...          ...          ...
2023-06-15 00:00:00-04:00     0.102     -0.076     0.026
2023-06-16 00:00:00-04:00     0.098     -0.075     0.023
2023-06-20 00:00:00-04:00     0.092     -0.065     0.027
2023-06-21 00:00:00-04:00     0.084     -0.059     0.025
2023-06-22 00:00:00-04:00     0.079     -0.061     0.018

          Net_Beta
Date
2022-12-22 00:00:00-05:00  0.343349
2022-12-23 00:00:00-05:00  0.343220
2022-12-27 00:00:00-05:00  0.348606
2022-12-28 00:00:00-05:00  0.349037
2022-12-29 00:00:00-05:00  0.344450
...
          ...
2023-06-15 00:00:00-04:00  0.310061
2023-06-16 00:00:00-04:00  0.312408
2023-06-20 00:00:00-04:00  0.311862
2023-06-21 00:00:00-04:00  0.318336
2023-06-22 00:00:00-04:00  0.318226
```

[124 rows x 23 columns]

This code looks a little innocuous, but it's doing a lot of work. It uses NumPy and does multiplication that Pandas makes easy—multiplying a Series by a DataFrame where the index of the Series matches the column name of the DataFrame. This multiplication operation is more complicated in Polars.

10.2 Multiplication Diversion

Let's look at the multiplication of a row and a dataframe. Here's a simple example.

Let's make a dataframe of folks' daily consumption of various food groups. We'll also create a series with the number of days for each person. Consider calculating the total consumption of each food group for each person. We can do this by multiplying the number of days by the consumption of each food group.

```
>>> import pandas as pd
>>> days = pd.Series([10, 11, 9], index=['Suzy', 'Bob', 'Alice'])
>>> daily_consumption = pd.DataFrame({'Suzy': [.2, .9, .1],
...                                         'Bob': [.3, .2, .8],
...                                         'Alice': [.5, .1, .1],
...                                         'Joe': [.1, .8, .0]},
...                                         index=['meat', 'plants', 'dairy'])

>>> print(days)
Suzy    10
Bob     11
Alice    9
dtype: int64
```

Here's the *daily_consumption* dataframe:

```
>>> print(daily_consumption)
      Suzy  Bob  Alice  Joe
meat    0.2  0.3   0.5  0.1
plants   0.9  0.2   0.1  0.8
dairy    0.1  0.8   0.1  0.0
```

Let's multiply the *days* series by the *daily_consumption* DataFrame.

```
>>> print(daily_consumption * days)
      Alice  Bob  Joe  Suzy
meat    4.5  3.3  NaN  2.0
plants   0.9  2.2  NaN  9.0
dairy    0.9  8.8  NaN  1.0
```

In Pandas, this math works because Pandas *aligns* the index names with the column names. Polars doesn't have an index, so we need to do a little more work.

Let's try to convert this code to Polars.

```
>>> import polars as pl
>>> days_pl = pl.DataFrame(days.reset_index())
```

10. Porting from Pandas

```
>>> print(days_pl)
shape: (3, 2)
| index | 0 |
| --- | --- |
| str | i64 |
| Suzy | 10 |
| Bob | 11 |
| Alice | 9 |

>>> daily_consumption_pl = pl.DataFrame(daily_consumption.reset_index())
>>> print(daily_consumption_pl)
shape: (3, 5)
| index | Suzy | Bob | Alice | Joe |
| --- | --- | --- | --- | --- |
| str | f64 | f64 | f64 | f64 |
| meat | 0.20 | 0.30 | 0.50 | 0.10 |
| plants | 0.90 | 0.20 | 0.10 | 0.80 |
| dairy | 0.10 | 0.80 | 0.10 | 0.00 |
```

Let's try the multiplication in Polars.

```
>>> print(daily_consumption_pl * days_pl)
Traceback (most recent call last)
...
PanicException: data types don't match: InvalidOperation(ErrString("mul
operation not supported for dtypes `str` and `str`"))
```

There is no easy way to do the multiplication in Polars. I will loop over the entries in *days_pl* and create a list with the expressions to do the appropriate multiplication for each column.

```
>>> [(name,i, val) for i, (name, val) in enumerate(days_pl.iter_rows())]
[('Suzy', 0, 10), ('Bob', 1, 11), ('Alice', 2, 9)]
```

Now, I will build up a list of expressions using the comprehension.

```
>>> print(daily_consumption_pl
... .with_columns([pl.col(name) * val
...                 for i, (name, val) in enumerate(days_pl.iter_rows())])
... )
shape: (3, 5)
| index | Suzy | Bob | Alice | Joe |
| --- | --- | --- | --- | --- |
| str | f64 | f64 | f64 | f64 |
```

meat	2.00	3.30	4.50	0.10
plants	9.00	2.20	0.90	0.80
dairy	1.00	8.80	0.90	0.00

Alternatively, we can convert the row data into a dictionary and do the math with the keys and values.

```
>>> days_dict = days.to_dict()
>>> days_dict
{'Suzy': 10, 'Bob': 11, 'Alice': 9}

>>> print(daily_consumption_pl
...     .with_columns(pl.col(name) * val
...                 for name, val in days_dict.items())
... )
shape: (3, 5)
index      Suzy    Bob    Alice    Joe
---        ---    ---    ---    ---
str       f64     f64     f64     f64
meat      2.00   3.30   4.50   0.10
plants     9.00   2.20   0.90   0.80
dairy      1.00   8.80   0.90   0.00
```

We will use this technique later to convert the Pandas code to Polars.

10.3 Leveraging AI

When you come across code you are not familiar with, consider using AI to help you understand it. AI tools can do a decent job with libraries with which they are familiar. Pandas is a popular library, and AI tools understand it well. I ran this code through ChatGPT 4 with this prompt:

Describe this code. Tell me what it is doing and what the end goal is. Explain what the variables and the columns it creates are [followed by the code].

Here is Chat's response (until the end of the section). Please don't skip over this, I encourage you to read it. You will probably be using similar techniques to understand code in the future.

This code is designed for financial analysis and portfolio management using Python libraries such as `matplotlib`, `numpy`, `pandas`, and `yfinance`. It calculates various metrics for a portfolio of stocks and compares them against a benchmark index. Here's a breakdown of its components:

1. Library Imports:

10. Porting from Pandas

- `matplotlib.pyplot` for plotting graphs.
- `numpy` for numerical operations.
- `pandas` for data manipulation and analysis.
- `yfinance` to fetch historical market data from Yahoo Finance.

2. Variable Initialization:

- `K`: A scaling factor, presumably for financial values, set to 1,000,000.
- `lot`: Another scaling factor, set to 100.
- `port_tickers`: A list of stock tickers representing the portfolio.
- `bm_ticker`: The ticker symbol for the benchmark index, set to `^GSPC` (S&P 500).
- `ticker_list`: A combined list of the benchmark ticker and portfolio tickers.

3. Portfolio DataFrame Creation (`port`):

- `df_data`: A dictionary containing 'Beta', 'Shares', and 'rSL' for each stock in the portfolio.
- `Beta`: A measure of the stock's volatility compared to the market.
- `Shares`: The number of shares held for each stock (negative values indicate short positions).
- `rSL`: Some specific ratio or value related to each stock, not standard terminology.
- `port`: A DataFrame created from `df_data`, indexed by `port_tickers`.
- `port['Side']`: A new column indicating the side of the trade (long or short) based on the sign of 'Shares'.

4. Fetching Market Data:

- Uses `yf.download` to download historical data for the tickers over the last 6 months.
- `price_df`: A DataFrame storing closing prices for each ticker, rounded to two decimal places.

5. Portfolio Metrics Calculation:

- `bm_cost` and `bm_price`: The initial and final price of the benchmark index.
- `port` DataFrame is expanded with new columns: 'rCost', 'rPrice', 'Cost', and 'Price', representing adjusted and raw cost/prices of the stocks.

6. Portfolio Management System (Chapter 13):

- `price_df` is expanded with columns calculating various returns and metrics like 'bm returns', 'rNet_Beta', 'rNet', 'rReturns_Long', 'rReturns_Short', and 'rReturns'.

- These columns involve complex financial calculations, including handling long and short positions, calculating net beta, and computing overall returns.

7. Market Value (MV) Calculations:

- Long and short market values are computed both in relative and absolute terms, adjusted for beta.
- ‘Gross’, ‘Net’, ‘Returns_Long’, ‘Returns_Short’, and ‘Returns’ columns are added to `price_df`, reflecting different aspects of the portfolio’s performance.

In summary, this script is used for detailed analysis and management of a stock portfolio, comparing its performance against a benchmark index using various financial metrics. It handles both long and short positions and calculates returns adjusted for market movements and individual stock volatilities.

Ok, that ends the output from the chat. I think it did a decent job. Current AI tools tend to be pretty good with Pandas code (Polars, as of this writing, is a different story). It also does a good job of inferring what the variables are for. Ideally, in the real world, I would work with the code’s creator or a subject matter expert. But, if that’s impossible, AI can be a good substitute.

10.4 Code Review

This section focuses on the code we just listed. It’s important to clarify that this code isn’t inherently bad. 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.

Another concern is the code’s rigidity, which stems from its heavy reliance on hard coding. A lack of precise inputs and outputs makes modifications

10. Porting from Pandas

challenging. 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 discuss documentation in detail, 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 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.

10.5 Enhancing Your Coding Process

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 `price_df`. However, several intermediate variables were also created along the way. These intermediate objects are sourced from both the `port` and `price_df` objects.

I ended up sketching a dependency graph to visualize the code.

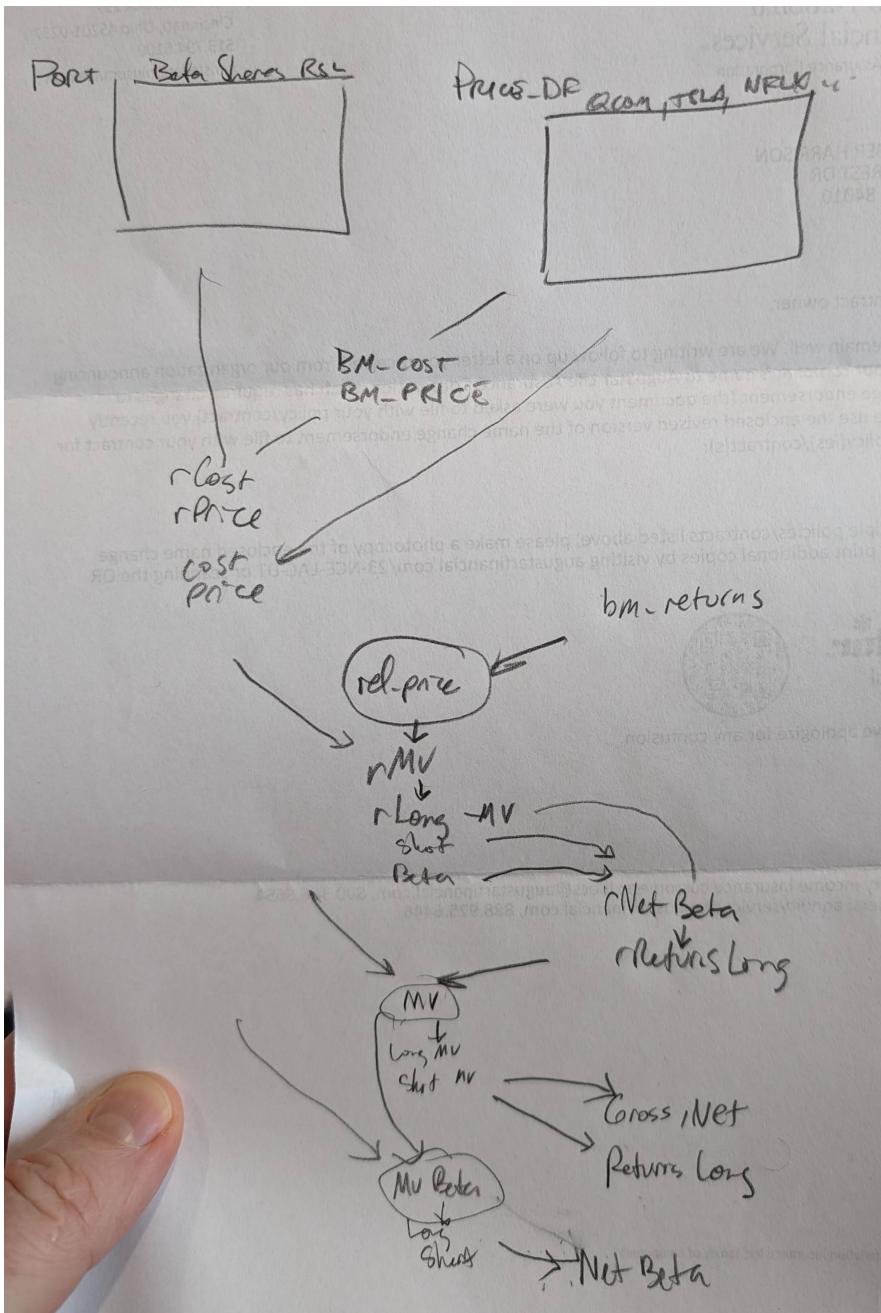


Figure 10.1: Sketching out the code can be a useful exercise to understand the dependencies between objects.

10. Porting from Pandas

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.

10.6 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. Here's the code:

```
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)
```

10.7 Creating a get_price Function

To improve the structure and readability of our code, we will refactor the process of pulling data into a new function called `get_price`. This will create a dataframe similar to `price_df` (prices) in the above code. I've added comments to the code to explain the steps. This process is complicated because the `yahoo` finance library returns Pandas dataframes with multi-index columns. Polars doesn't support multi-index columns, so I must jump through a few hoops to post-process the data. (My life could have been easier if I had made a simpler export, but I wanted to show you some of the pains you might encounter when working with data from other sources.)

```
>>> import polars as pl
>>> import polars.selectors as cs
>>> def get_price():
...     # read first two rows to get headers
...     head_df = pl.read_csv('data/raw-yfinance.csv', n_rows=2)
...     # get tickers from first row
...     cols = ['_'.join([a, str(b)])
...             for a, b in zip(head_df.columns, head_df.row(0))]
...     return (pl.read_csv('data/raw-yfinance.csv', has_header=False,
...                         skip_rows=3, new_columns=cols)
...             # convert first column to datetime
...             .with_columns(pl.col('_None')
...                         .str.to_datetime('%Y-%m-%d %H:%M:%S%z'))
...             .rename({'_None': 'Date'})
...             # columns starting with Close
...             .select(cs.matches(r'^(Date|Close.*)'))
...             # rename columns, take value after last _
...             .pipe(lambda df_: df_.rename(dict(zip(df_.columns,
...                                         [c.split('_')[-1] for c in df_.columns])))))
... )
>>> print(get_price())
shape: (124, 12)
  Date      BRK-B      DIS      .      TSLA      UPS    ^GSPC
  ---      ---      ---      .      ---      ---      ---
  datetime[μs,   f64      f64      .      f64      f64      f64
  UTC]
  2022-12-22  302.69  86.67  .  125.35  172.42  3822.39
  05:00:00 UTC
  2022-12-23  306.49  88.01  .  123.15  173.79  3844.82
  05:00:00 UTC
  2022-12-27  305.55  86.37  .  109.10  173.72  3829.25
  05:00:00 UTC
  2022-12-28  303.43  84.17  .  112.71  170.46  3783.22
```

10. Porting from Pandas

05:00:00 UTC						
2022-12-29	309.06	87.18	.	121.82	172.56	3849.28
05:00:00 UTC						
.
2023-06-15	339.82	92.94	.	255.90	179.00	4425.84
04:00:00 UTC						
2023-06-16	338.31	91.32	.	260.54	178.58	4409.59
04:00:00 UTC						
2023-06-20	338.67	89.75	.	274.45	177.27	4388.71
04:00:00 UTC						
2023-06-21	338.61	88.64	.	259.46	173.63	4365.69
04:00:00 UTC						
2023-06-22	336.91	88.39	.	261.47	172.37	4368.72
04:00:00 UTC						

Let's compare the results with the Pandas version. I'm only going to compare the initial columns of the Polars version. The Pandas version has additional columns we have not yet created in Polars.

We will use the `.equals` method to compare the two dataframes. This method returns `True` if the two dataframes are equal and `False` otherwise.

```
>>> price_pl = get_price()
>>> cols = price_pl.columns
>>> price_pl.equals(pl.DataFrame(price_df
...     .reset_index()
...     .astype({'Date': 'datetime64[ns, UTC]'})
...     .loc[:, cols])
... )
False
```

That didn't work. Let's try the `polars.testing.assert_frame_equal` function. You will want to know about this very convenient function when porting code from Pandas to Polars, or even refactoring Polars code. It will compare two dataframes and raise an error if they are not equal. The error should contain information in its message to give you a good idea of what is different between the two dataframes.

```
>>> import polars.testing as pt
>>> pt.assert_frame_equal(price_pl,
...     pl.DataFrame(price_df
...     .reset_index()
...     .astype({'Date': 'datetime64[ns, UTC]'})
...     .loc[:, cols])
... )
Traceback (most recent call last)
...
```

```
AssertionError: DataFrames are different (dtypes do not match)
[left]: {'Date': Datetime(time_unit='us', time_zone='UTC'), 'BRK-B':
    Float64, 'DIS': Float64, 'F': Float64, 'IBM': Float64, 'MMM':
    Float64, ...
[right]: {'Date': Datetime(time_unit='ns', time_zone='UTC'), 'BRK-B':
    Float64, 'DIS': Float64, 'F': Float64, 'IBM': Float64, 'MMM':
    Float64, ...}
```

That is more helpful. It looks like the time unit on the date is different; one is in microseconds, and the other is nanoseconds. Let's try to fix those. I'll update the `get_tickers_pl` function to fix the date and the column types. Note that last cast in the chain.

```
>>> import polars as pl
>>> import polars.selectors as cs
>>> def get_price():
...     # read first two rows to get headers
...     head_df = pl.read_csv('data/raw-yfinance.csv', n_rows=2)
...     # get tickers from first row
...     cols = ['_'.join([a, str(b)])
...             for a, b in zip(head_df.columns, head_df.row(0))]
...     return (pl.read_csv('data/raw-yfinance.csv', has_header=False,
...                         skip_rows=3, new_columns=cols)
...             # convert first column to datetime
...             .with_columns(pl.col('_None').str.to_datetime('%Y-%m-%d %H:%M:%S%z'))
...             .rename({'_None': 'Date'})
...             # columns starting with Close
...             .select(cs.matches(r'^(Date|Close.*)'))
...             # rename columns, take value after last
...             .pipe(lambda df_: df_.rename(dict(zip(df_.columns,
...                                         [c.split('_')[-1] for c in df_.columns]))))
...             .with_columns(pl.col('Date')
...                         .cast(pl.Datetime(time_unit='ns', time_zone='UTC'))))
...     )
>>> tick_mine = get_price()
>>> tick_from_pd = pl.DataFrame(price_df.reset_index().astype(
...     {'Date': 'datetime64[ns, UTC]'}))
...     .loc[:, tick_mine.columns])
>>> pt.assert_frame_equal(tick_mine, tick_from_pd)
Traceback (most recent call last)

...
AssertionError: DataFrames are different (value mismatch for column
'DIS')
[left]: [86.66999816894531, 88.01000213623047, 86.37000274658203,
84.16999816894531, 87.18000030517578, 86.87999725341797,
88.9700012207031...]
```

10. Porting from Pandas

```
[right]: [86.67, 88.01, 86.37, 84.17, 87.18, 86.88, 88.97, 91.98, 91.92,  
93.92, 94.77, 95.56, 96.33, 99.81, 99.4, 99.91, 99.04, 99.08, 103.4...]
```

Now, it looks like we have rounding issues. I'll round my data to two decimal places and try again. Notice the `cs.float().round(2)` in the `get_price` function. This column selector will select all the float columns and round them to two decimal places.

```
>>> import polars as pl  
>>> import polars.selectors as cs  
>>> def get_price():  
...     # read first two rows to get headers  
...     head_df = pl.read_csv('data/raw-yfinance.csv', n_rows=2)  
...     # get tickers from first row  
...     cols = ['_'.join([a, str(b)])  
...             for a, b in zip(head_df.columns, head_df.row(0))]  
...     return (pl.read_csv('data/raw-yfinance.csv', has_header=False,  
...                         skip_rows=3, new_columns=cols)  
...             # convert first column to datetime  
...             .with_columns(pl.col('_None').str.to_datetime('%Y-%m-%d %H:%M:%S%z'))  
...             .rename({'_None': 'Date'})  
...             # columns starting with Close  
...             .select(cs.matches(r'^^(Date|Close.*)'))  
...             # rename columns, take value after last _  
...             .pipe(lambda df_: df_.rename(dict(zip(df_.columns,  
...                                         [c.split('_')[-1] for c in df_.columns]))))  
...             .with_columns(pl.col('Date')  
...                         .cast(pl.Datetime(time_unit='ns', time_zone='UTC')))  
...             .with_columns(cs.float().round(2))  
... )  
>>> tick_mine = get_price()  
>>> tick_from_pd = pl.DataFrame(price_df.reset_index().astype(  
...     {'Date': 'datetime64[ns, UTC]'})  
...     .loc[:, tick_mine.columns])  
>>> pt.assert_frame_equal(tick_mine, tick_from_pd)
```

Booyah! We have a match.

And that is typically the process I use when porting from one dataframe library to another. Make a new column and then check the values to ensure it is correct. Luckily, the Polars testing library makes this process easy.

10.8 Creating a `get_port` Function

Let's now try to create a function to create the portfolio dataframe. This function will be called `get_port_pl`. I'll use the same steps as the Pandas

version, but I'll use Polars syntax. I'll also add some comments to explain the steps.

Here's the relevant code from the Pandas version. Note that there is a dependency on the `price_df` dataframe to calculate the `rCost`, `rPrice`, `Cost`, and `Price` columns.

```
port_tickers = ['QCOM', 'TSLA', 'NFLX', 'DIS', 'PG', 'MMM', 'IBM', 'BRK-B',
                'UPS', 'F']
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'])
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,:]
```

And here is my initial refactoring into Polars. This is more complicated in Polars because converting rows from the `price_df` dataframe and making them into columns is harder. Pandas will happily align column and index names, but Polars doesn't have a notion of index names. I ended up pulling off the rows, transposing them, and using `.hstack` to add them to the dataframe. Once I had the columns in the dataframe, I used the `.with_column` method to update the values by the cost and price entries. Finally, I used the `.select` method to select the columns I wanted to keep. This is a great practice to add to the end of chains of operations. It makes it easier to see what columns are being kept.

```
>>> def get_port(price_df):
...     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_tickers = ['QCOM', 'TSLA', 'NFLX', 'DIS', 'PG', 'MMM', 'IBM', 'BRK-B',
...                     'UPS', 'F']
...     cost = (price_df
...             .select(port_tickers)
...             .head(1)
...             .transpose()
...             .rename({'column_0': 'Cost'})
...     )
...     price = (price_df
```

10. Porting from Pandas

```
>>> port pl = get_port(get_price())
```

```
>>> print(port_pl)
```

shape: (10, 9)

tickers	Beta	Shares	.	rPrice	Cost	Price
---	---	---		---	---	---
str	f64	i64		f64	f64	f64
QCOM	1.34	-1900	.	26.63	109.25	116.36
TSLA	2.00	-100	.	59.85	125.35	261.47
NFLX	0.75	-400	.	96.28	297.75	420.61
DIS	1.20	-800	.	20.23	86.67	88.39
PG	0.41	-5500	.	34.29	150.30	149.79
MMM	0.95	1600	.	22.95	118.20	100.27
IBM	1.23	1800	.	29.97	137.30	130.91
BRK-B	0.90	2800	.	77.12	302.69	336.91
UPS	1.05	1100	.	39.46	172.42	172.37
F	1.15	20800	.	3.27	10.51	14.28

I will use our friend `pl.testing.assert_frame_equal` to compare the results to the Pandas version (jumping through a small hoop to remove the index and rename the columns).

```
>>> pt.assert_frame_equal(port_pl,
...     pl.DataFrame(port.reset_index()).rename({'index': 'tickers'}))
```

Awesome!

10.9 More Refactoring

That's a decent start. Now, we want to add many new columns to the `price_df` dataframe. Along the way, we will create a few intermediate dataframes.

```

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'll just proceed step by step, checking the results as I go. Let's create a `bm_returns` column. I'll make a function called `update_price_df` and test the

10. Porting from Pandas

results. Note that because *bm returns* has a space in it, I need to use the `.alias` method to rename the column or an unpacked dictionary as I do below.

Let's try to calculate the *bm returns* column in Polars.

```
>>> #round(np.exp(np.log(price_df[bm_ticker])/
>>> #    price_df[bm_ticker].shift()).cumsum()) - 1, 3)
>>> def update_price_df(price_df, bm_ticker):
...     return (price_df
...         .with_columns(bm_shift=pl.col(bm_ticker).shift(),
...             **{'bm returns': (pl.col(bm_ticker)/
...                 (pl.col(bm_ticker).shift())).log().cum_sum().exp().sub(1))})
...     )

>>> pt.assert_frame_equal(update_price_df(get_price(), bm_ticker)
...     .select(['bm returns']),
...     pl.DataFrame(price_df[['bm returns']])
... )
Traceback (most recent call last)
...
AssertionError: DataFrames are different (value mismatch for column 'bm
    returns')
[left]: [None, 0.005868056373107056, 0.001794688663375732,
    -0.010247515298020149, 0.007034865620724418, 0.004476257001509776,
    0.00045782874...
[right]: [None, 0.006, 0.002, -0.01, 0.007, 0.004, 0.0, 0.008, -0.004,
    0.019, 0.018, 0.025, 0.039, 0.042, 0.046, 0.044, 0.028, 0.02, 0.039, ...]
```

It looks like we are on the right track. We have some rounding errors. I will create a `returns` function to calculate the returns. I can see that the code I'm refactoring calculates the returns a few times, so I'll make a function to do it. I'll also add rounding in the function. Note that I can also use the `atol` parameter to the `pl.testing.assert_frame_equal` function to allow for some rounding errors. The `atol` parameter is the absolute tolerance. It's the maximum difference between the two dataframes. If the difference is more significant than this, the function will raise an error.

```
# function and address rounding
def returns(col_name, result_name=None):
    if result_name is None:
        result_name = f'{col_name} returns'
    return ((pl.col(col_name)
        / pl.col(col_name).shift())
        .log()
        .cum_sum()
        .exp()
        .sub(1))
```

```

        .round(3)
        .alias(result_name)
    )

def update_price_df(price_df, bm_ticker):
    return (price_df
            .with_columns(returns(bm_ticker, 'bm returns'))
    )

```

Let's see if this works now!

```

>>> pt.assert_frame_equal(
...     (update_price_df(get_price(), bm_ticker).select(['bm returns'])),
...     pl.DataFrame(price_df[['bm returns']])
)

```

Success! Now, let's create the intermediate `rel_price` dataframe. I could use the `cs.matches` column selector to select everything but the `Date` column, but I already have the `ticker_list` variable, so I'll use that with `pl.col`.

```

#rel_price = round(price_df.div(price_df['^GSPC'],axis=0 )*1000,2)
rel_price_pl = (get_price()
    .with_columns((pl.col(ticker_list) / (pl.col(bm_ticker)) * 1000).round(2))
)

```

Next step is to create the `rMV` dataframe.

Now we will do the multiplication that was easy in Pandas but a bit more complicated in Polars to create the `rMV` dataframe.

```

# rMV = rel_price.mul(port['Shares'])
rMV_pl = (rel_price_pl
    .with_columns([pl.col(name)*shares for i, (name, shares) in
                  enumerate(port_pl.select(['tickers', 'Shares']).iter_rows())])
)

```

Now we will sum up the positive values.

```

#rLong_MV = rMV[rMV >0].sum(axis=1)
rLong_MV_pl = (rMV_pl
    .select(pl.when(pl.col(pl.Float64).gt(0))
            .then(pl.col(pl.Float64).otherwise(0)))
    .select(pl.sum_horizontal(pl.all()))
)

```

Note that I'm using a `when/then/otherwise` expression about to clip the values below zero to zero. This is a common pattern in Polars. In fact, Polars has the `.clip` method that will do this for us. I'll use that instead.

Now we will keep doing this. I'll show the Pandas code in comments and then the Polars code.

10. Porting from Pandas

```
#rLong_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
rLong_MV_pl = (rMV_pl
#.select(pl.when(pl.col(pl.Float64).gt(0))
#           .then(pl.col(pl.Float64)).otherwise(0))
.select(pl.col(pl.Float64).clip(lower_bound=0))
.select(pl.sum_horizontal(pl.all()).alias('rLong_MV'))
)

rMV_Beta_pl = (rMV_pl
.with_columns([pl.col(name)*shares for i, (name, shares) in
               enumerate(port_pl.select(['tickers', 'Beta']).iter_rows())])
)
rLong_MV_Beta_pl = (rMV_Beta_pl
.with_columns(pl.col(pl.Float64).clip(lower_bound=0))
.select('Date',
       rLong_MV_Beta=(pl.sum_horizontal(pl.col(pl.Float64)) /
                      rLong_MV_pl['rLong_MV']))
)
)

#rShort_MV = rMV[rMV <0].sum(axis=1)
#rShort_MV_Beta = rMV_Beta[rMV_Beta <0].sum(axis=1)/ rShort_MV
rShort_MV_pl = (rMV_pl
.select(pl.col(pl.Float64).clip(upper_bound=0))
.select(pl.sum_horizontal(pl.all()).alias('rShort_MV'))
)

rShort_MV_Beta_pl = (rMV_Beta_pl
.with_columns(pl.col(pl.Float64).clip(upper_bound=0))
.select('Date',
       rShort_MV_Beta=(pl.sum_horizontal(pl.col(pl.Float64)) /
                      rShort_MV_pl['rShort_MV']))
)
```

Once I have the appropriate intermediate dataframes; I can start putting columns back into the *price_df* dataframe. I'll start with the *rNet_Beta* column.

In this case, I need to use *.to_series* to convert the result to a series to add the values back into the dataframe.

```
>>> # need .to_series()
>>> print((update_price_df(get_price(), bm_ticker))
... .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta')
...                   - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series())
... )
```

shape: (124, 14)

Date --- datetime[ns, UTC]	BRK-B ---	DIS f64	.	^GSPC ---	bm returns ---	rNet_Beta ---
2022-12-2 2 05:00:00 UTC	302.69	86.67	.	3822.39	null	0.34
2022-12-2 3 05:00:00 UTC	306.49	88.01	.	3844.82	0.01	0.34
2022-12-2 7 05:00:00 UTC	305.55	86.37	.	3829.25	0.00	0.35
2022-12-2 8 05:00:00 UTC	303.43	84.17	.	3783.22	-0.01	0.35
2022-12-2 9 05:00:00 UTC	309.06	87.18	.	3849.28	0.01	0.34
.
2023-06-1 5 04:00:00 UTC	339.82	92.94	.	4425.84	0.16	0.31
2023-06-1 6 04:00:00 UTC	338.31	91.32	.	4409.59	0.15	0.31
2023-06-2 0 04:00:00 UTC	338.67	89.75	.	4388.71	0.15	0.31
2023-06-2 1 04:00:00 UTC	338.61	88.64	.	4365.69	0.14	0.32
2023-06-2 2 04:00:00	336.91	88.39	.	4368.72	0.14	0.32

10. Porting from Pandas

UTC							
-----	--	--	--	--	--	--	--

Again, I will be pedantic and check the results.

```
## added atol=1e-4 to get rid of rounding error
pt.assert_frame_equal((update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta'
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series())
    .select(['rNet_Beta'])),
    pl.DataFrame(price_df[['rNet_Beta']]), atol=1e-4)
```

Proceeding in this fashion, I will update my chain to the price_df dataframe.

```
# 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']

(update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta'
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
    rLong_MV=rLong_MV_pl['rLong_MV'],
    rShort_MV=rShort_MV_pl['rShort_MV']
    )
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'))
    )
```

Let's test it again.

```
pt.assert_frame_equal(update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta'
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
    rLong_MV=rLong_MV_pl['rLong_MV'],
    rShort_MV=rShort_MV_pl['rShort_MV']
    )
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'))
    .select(['rReturns']),
    pl.DataFrame(price_df[['rReturns']]), atol=1e-3)
```

More code to port over... Note that I have to use multiple `.with_columns` because some of the columns depend on other columns.

```
# 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)

MV_pl = (price_pl
    .with_columns([pl.col(ticker)*port_pl.select('Shares').row(i)[0]
        for i, ticker in enumerate(port_pl['tickers'])])
    .with_columns(pl.lit(0).alias(bm_ticker)) # clear ^GSPC column
)
Long_MV_pl = (MV_pl
    .select(pl.col(pl.Float64).clip(lower_bound=0))
    .select(Long_MV=pl.sum_horizontal(pl.all())))
Short_MV_pl = (MV_pl
    .select(pl.col(pl.Float64).clip(upper_bound=0))
    .select(Short_MV=pl.sum_horizontal(pl.all())))
net_denom = (MV_pl
    .select(pl.col(pl.Float64))
    .select(pl.sum_horizontal(pl.all().abs())))
(update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta')
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
        rLong_MV=rLong_MV_pl['rLong_MV'],
        rShort_MV=rShort_MV_pl['rShort_MV'],
        Long_MV=Long_MV_pl['Long_MV'],
        Short_MV=Short_MV_pl['Short_MV'])
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
        Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
        Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
            net_denom.to_series()))
)
)
```

10. Porting from Pandas

More testing. This probably seems pedantic and a waste of space in a book, but I wanted to show you my process. Working in a notebook environment makes it easy to test the results after each step, which is a good practice to get into.

Whenever I think, “I’ll just wait until the end to test everything,” I regret it and tend to take more time than I would have if I had tested after each step.

```
pt.assert_frame_equal((update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta')
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
    rLong_MV=rLong_MV_pl['rLong_MV'],
    rShort_MV=rShort_MV_pl['rShort_MV'],
    Long_MV=Long_MV_pl['Long_MV'],
    Short_MV=Short_MV_pl['Short_MV'])
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
        Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
        Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
            net_denom.to_series())
    )
    .select(['Gross', 'Net'])),
    pl.DataFrame(price_df[['Gross', 'Net']]), atol=1e-3)
```

And we move on with the next columns.

```
# 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']
```

```
(update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta')
        - rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
    rLong_MV=rLong_MV_pl['rLong_MV'],
    rShort_MV=rShort_MV_pl['rShort_MV'],
    Long_MV=Long_MV_pl['Long_MV'],
    Short_MV=Short_MV_pl['Short_MV'])
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        Returns_Long=returns('Long_MV'),
        Returns_Short=-returns('Short_MV'),
        )
```

```

.with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
              Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
              Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
                    net_denom.to_series()),
              Returns=((pl.col('Returns_Long') + pl.col('Returns_Short'))))
)
.select(['Returns', 'Returns_Long', 'Returns_Short'])
)

```

Let's compare again to see how the calculations perform

```

pt.assert_frame_equal((update_price_df(get_price(), bm_ticker)
.with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta')
- rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
rLong_MV=rLong_MV_pl['rLong_MV'],
rShort_MV=rShort_MV_pl['rShort_MV'],
Long_MV=Long_MV_pl['Long_MV'],
Short_MV=Short_MV_pl['Short_MV'])
.with_columns(rReturns_Long=returns('rLong_MV'),
             rReturns_Short=-returns('rShort_MV'),
             Returns_Long=returns('Long_MV'),
             Returns_Short=-returns('Short_MV'),
             )
.with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
              Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
              Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
                    net_denom.to_series()),
              Returns=((pl.col('Returns_Long') + pl.col('Returns_Short'))))
)
.select(['Returns'])
),
pl.DataFrame(price_df[['Returns']], atol=1e-3)
)

```

Finally, let's add the last column, *rNet_Beta*!

```

# 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

```

```

MV_Beta_pl = (MV_pl
    .with_columns([pl.col(ticker)*port_pl.select('Beta').row(i)[0]
                  for i, ticker in enumerate(port_pl['tickers'])]))
)
Long_MV_Beta_pl = (MV_Beta_pl
    # use clip

```

10. Porting from Pandas

```
.select(pl.col(pl.Float64).clip(lower_bound=0))
.select(Long_MV_Beta=pl.sum_horizontal(pl.all()) /
    Long_MV_pl['Long_MV'])
)

Short_MV_Beta_pl = (MV_Beta_pl
    .select(pl.col(pl.Float64).clip(upper_bound=0))
    .select(Short_MV_Beta=pl.sum_horizontal(pl.all()) /
        Short_MV_pl['Short_MV'])
)
(update_price_df(get_price(), bm_ticker)
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('rLong_MV_Beta') -
        rShort_MV_Beta_pl.select('rShort_MV_Beta')).to_series(),
        rLong_MV=rLong_MV_pl['rLong_MV'],
        rShort_MV=rShort_MV_pl['rShort_MV'],
        Long_MV=Long_MV_pl['Long_MV'],
        Short_MV=Short_MV_pl['Short_MV'])
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        Returns_Long=returns('Long_MV'),
        Returns_Short=-returns('Short_MV'),
        )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
        Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
        Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
            net_denom.to_series()),
        Returns=((pl.col('Returns_Long') + pl.col('Returns_Short'))),
        Net_Beta=(Long_MV_Beta_pl['Long_MV_Beta'] -
            Short_MV_Beta_pl['Short_MV_Beta']))
)
)
```

10.10 Final Refactoring

Here's my final code. I aim to create a single notebook cell (or maybe two or three) that I can stick at the top of my notebook. This allows me (and collaborators) to easily come back to the analysis and rerun the notebook without futzing running 50 different cells strewn throughout the notebook in some arbitrary order. I will repeatedly restart the kernel and ensure this code works independently of the other cells in the notebook.

You'll also notice that I've moved everything into functions. This makes it easier to test and debug. It eliminates random global intermediate variables that can cause problems. It also makes it easier to reuse the code in other notebooks or move the code into a library.

Once I had all the code in a single cell, some other patterns started appearing. I noticed that the long and short calculations were running multiple times. So, I refactored that logic into a function. Also, the multiplication of columns by a row with the same height as the columns (we saw this at the start of the chapter) was repeated. I refactored that into a function as well.

```
import polars as pl
import polars.selectors as cs

def returns(col_name, result_name=None):
    if result_name is None:
        result_name = f'{col_name} returns'
    return ((pl.col(col_name)
             / pl.col(col_name).shift())
            .log()
            .cum_sum()
            .exp()
            .sub(1)
            .round(3)
            .alias(result_name)
    )

def calc_long(df_, divisor=1, alias='long'):
    return (df_
            .select(pl.col(pl.Float64).clip(lower_bound=0))
            .select((pl.sum_horizontal(pl.all()) / divisor).alias(alias)))
            )

def calc_short(df_, divisor=1, alias='short'):
    return (df_
            .select(pl.col(pl.Float64).clip(upper_bound=0))
            .select((pl.sum_horizontal(pl.all()) / divisor).alias(alias)))
            )

def multiply_tickers_rows_by_column(df_, col_name, port_pl):
    return (df_
            .with_columns([pl.col(ticker)*port_pl.select(col_name).row(i)[0]
                           for i, ticker in enumerate(port_pl['tickers'])]))
            )

def get_price():
    # read first two rows to get headers
    head_df = pl.read_csv('data/raw-yfinance.csv', n_rows=2)
    # get tickers from first row
```

10. Porting from Pandas

```
cols = ['_'.join([a, str(b)]) for a, b in
        zip(head_df.columns, head_df.row(0))]
return (pl.read_csv('data/raw-yfinance.csv', has_header=False,
                    skip_rows=3, new_columns=cols, try_parse_dates=True)
       # convert first column to datetime
       .with_columns(pl.col('None').str.to_datetime('%Y-%m-%d %H:%M:%S%z'))
       .rename({'None': 'Date'})
       # columns starting with Close
       .select(cs.matches(r'^(Date|Close.*)'))
       # rename columns, take value after last _
       .pipe(lambda df_: df_.rename(dict(zip(df_.columns,
                                              [c.split('_')[-1] for c in df_.columns]))))
       .with_columns(
           pl.col('Date').cast(pl.Datetime(time_unit='ns', time_zone='UTC')),
           pl.col('NFLX').cast(pl.Float64))
       .with_columns(cs.float().round(2))
)
def get_port(price_df, bm_ticker):
    df_data = {
        'Beta': [1.34, 2.0, 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_tickers = ['QCOM', 'TSLA', 'NFLX', 'DIS', 'PG', 'MMM', 'IBM', 'BRK-B',
                    'UPS', 'F']
    cost = (price_df
            .select(port_tickers)
            .head(1)
            .transpose()
            .rename({'column_0': 'Cost'}))
    price = (price_df
            .select(port_tickers)
            .tail(1)
            .transpose()
            .rename({'column_0': 'Price'}))
    bm_cost = price_df[bm_ticker][0]
    bm_price = price_df[bm_ticker][-1]
    return (pl.DataFrame(df_data)
            .with_columns(
                tickers=pl.lit(pl.Series(port_tickers)),
                Side=pl.when(pl.col('Shares') > 0).then(1).otherwise(-1)
                      .cast(pl.Int64)
            )
)
```

```

    .hstack(cost)
    .hstack(price)
    .with_columns(rCost=(pl.col('Cost') / bm_cost * 1000).round(2),
                  rPrice=(pl.col('Price') / bm_price * 1000).round(2)
    )
    .select(['tickers', 'Beta', 'Shares', 'rSL', 'Side', 'rCost',
            'rPrice', 'Cost', 'Price'])
)

```

```

def get_price_and_port():
    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
    price_pl = get_price()
    port_pl = get_port(price_pl, bm_ticker)

    # inline rel_price
    rMV_pl = (price_pl
               .with_columns((pl.col(ticker_list) /
                             (pl.col(bm_ticker)) * 1000).round(2))
               .pipe(multiply_tickers_rows_by_column, 'Shares', port_pl)
    )
    rMV_Beta_pl = (rMV_pl
                    .pipe(multiply_tickers_rows_by_column, 'Beta', port_pl)
    )
    rShort_MV_pl = rMV_pl.pipe(calc_short)
    rShort_MV_Beta_pl = rMV_Beta_pl.pipe(calc_short,
                                           divisor=rShort_MV_pl['short'])

    rLong_MV_pl = rMV_pl.pipe(calc_long)
    rLong_MV_Beta_pl = rMV_Beta_pl.pipe(calc_long, divisor=rLong_MV_pl['long'])

    MV_pl = (price_pl
              .pipe(multiply_tickers_rows_by_column, 'Shares', port_pl)
              .with_columns(pl.lit(0).alias(bm_ticker))  # clear ^GSPC column
    )
    Long_MV_pl = MV_pl.pipe(calc_long)
    Short_MV_pl = MV_pl.pipe(calc_short)

    net_denom = (MV_pl
                  .select(pl.col(pl.Float64))
                  .select(pl.sum_horizontal(pl.all().abs())))
)

```

10. Porting from Pandas

```
MV_Beta_pl = (MV_pl
    .pipe(multiply_tickers_rows_by_column, 'Beta', port_pl)
)
Long_MV_Beta_pl = MV_Beta_pl.pipe(calc_long,
    divisor=Long_MV_pl['long'])
Short_MV_Beta_pl = MV_Beta_pl.pipe(calc_short,
    divisor=Short_MV_pl['short'])

final_price = (price_pl
    .with_columns(returns(bm_ticker))
    .with_columns(
        rNet_Beta=(rLong_MV_Beta_pl.select('long')
            - rShort_MV_Beta_pl.select('short')).to_series(),
        rLong_MV=rLong_MV_pl['long'],
        rShort_MV=rShort_MV_pl['short'],
        Long_MV=Long_MV_pl['long'],
        Short_MV=Short_MV_pl['short']
    )
    .with_columns(
        rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        Returns_Long=returns('Long_MV'),
        Returns_Short=-returns('Short_MV'),
    )
    .with_columns(
        rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
        Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
        Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
            net_denom.to_series()),
        Returns=((pl.col('Returns_Long') + pl.col('Returns_Short'))),
        Net_Beta=(Long_MV_Beta_pl['long'] - Short_MV_Beta_pl['short'])
    )
)
return final_price, port_pl
```

You will want to make sure that you test the results as you go through this process. Below, I'm testing that the main columns that were calculated are the same in both the Pandas and Polars code.

```
import polars.testing as pt
price_pl, port_pl = get_price_and_port()
pt.assert_frame_equal(
    price_pl.select(['Gross', 'Net', 'Returns', 'Net_Beta']),
    pl.DataFrame(price_df[['Gross', 'Net', 'Returns', 'Net_Beta']]),
    atol=1e-3)
```

10.11 To Port or Not to Port

I had another thought at the end of this chapter (maybe I should have put this at the beginning). Consider whether the juice from porting to Polars is worth the squeeze. In this case, I'm not sure that it is. I would probably start by refactoring the Pandas code to function. Then I would just use Polars ability to suck in a Pandas dataframe. If this existing code was already working fine and debugged, it might be more beneficial to just use the Pandas code and invest time into new functionality.

Let's time both and compare.

Here's the timing for the calculations in the Pandas code. I'm not including the timing for loading the data. If loading the data from CSV is something you do every time you run the notebook, then you should include that in the timing.

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

bm_cost = price_df[bm_ticker].iloc[0]
bm_price = price_df[bm_ticker].iloc[-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,:]

# 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)
```

10. Porting from Pandas

```
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
price_df
```

And here's the Polars code timing:

```
%%timeit

# inline rel_price
rMV_pl = (price_pl
    .with_columns((pl.col(ticker_list) /
        (pl.col(bm_ticker)) * 1000).round(2))
    .pipe(multiply_tickers_rows_by_column, 'Shares', port_pl)
)
rMV_Beta_pl = (rMV_pl
    .pipe(multiply_tickers_rows_by_column, 'Beta', port_pl)
)
rShort_MV_pl = (rMV_pl.pipe(calc_short))
rShort_MV_Beta_pl = (rMV_Beta_pl.pipe(calc_short,
    divisor=rShort_MV_pl['short']))
rLong_MV_pl = (rMV_pl.pipe(calc_long))
rLong_MV_Beta_pl = (rMV_Beta_pl.pipe(calc_long,
    divisor=rLong_MV_pl['long']))

MV_pl = (price_pl
    .pipe(multiply_tickers_rows_by_column, 'Shares', port_pl)
```

```

    .with_columns(pl.lit(0).alias(bm_ticker)) # clear ^GSPC column
)
Long_MV_pl = (MV_pl.pipe(calc_long))
Short_MV_pl = (MV_pl.pipe(calc_short))

net_denom = (MV_pl
    .select(pl.col(pl.Float64))
    .select(pl.sum_horizontal(pl.all().abs()))
)

MV_Beta_pl = (MV_pl
    .pipe(multiply_tickers_rows_by_column, 'Beta', port_pl)
)
Long_MV_Beta_pl = (MV_Beta_pl.pipe(calc_long, divisor=Long_MV_pl['long']))
Short_MV_Beta_pl = (MV_Beta_pl.pipe(calc_short,
    divisor=Short_MV_pl['short']))

final_price = (price_pl
    .with_columns(returns(bm_ticker))
    .with_columns(rNet_Beta=(rLong_MV_Beta_pl.select('long') -
        rShort_MV_Beta_pl.select('short')).to_series(),
        rLong_MV=rLong_MV_pl['long'],
        rShort_MV=rShort_MV_pl['short'],
        Long_MV=Long_MV_pl['long'],
        Short_MV=Short_MV_pl['short']
    )
    .with_columns(rReturns_Long=returns('rLong_MV'),
        rReturns_Short=-returns('rShort_MV'),
        Returns_Long=returns('Long_MV'),
        Returns_Short=-returns('Short_MV'),
    )
    .with_columns(rReturns=pl.col('rReturns_Long') + pl.col('rReturns_Short'),
        Gross=((pl.col('Long_MV') - pl.col('Short_MV')) / K),
        Net=((pl.col('Long_MV') + pl.col('Short_MV')) /
            net_denom.to_series()),
        Returns=((pl.col('Returns_Long') + pl.col('Returns_Short'))),
        Net_Beta=(Long_MV_Beta_pl['long'] - Short_MV_Beta_pl['short'])
    )
)

```

The Polars code runs about 3x faster than the Pandas code. Your mileage may vary. I also tried this in lazy mode, throwing in a `.lazy()` at the start of the chain and a `.collect()` at the end. This didn't appear to help in this case. This isn't surprising because there isn't a lot of filtering, joining, or grouping going on. The code mostly does math on columns.

10. Porting from Pandas

Porting can be valuable beyond just performance. Having a standard codebase with a single technology can help future velocity. Porting can be a great way to learn a new library or language. I learned quite a bit myself writing this chapter.

10.12 Chapter Methods

Method	Description
<code>df.iter_rows(named=False, buffer_size=500)</code>	Return an iterator over the rows of the DataFrame. The <code>buffer_size</code> parameter controls how many rows are returned at a time.
<code>df.equals(other, * null_equal=True)</code>	Returns True if the DataFrames are equal. <code>null_equal</code> controls whether null values are considered equal.
<code>pl.testing.assert_frame_equal(left, right, check_row_order=True, check_column_order=True, check_dtype=True, check_exact=False, rtol=1e-05, atol=1e-08, categorical_as_str=False, nans_compare_equal=None)</code>	Raises an <code>AssertionError</code> if two DataFrames are not equal. You can control absolute and relative tolerances with the <code>rtol</code> and <code>atol</code> parameters. Relative tolerance is computed on the fractional difference between the two values. Absolute tolerance is the maximum difference between the two values.
<code>col.round(decimals=0)</code>	Round the column to the specified number of decimal places.

10.13 Summary

In this chapter, we ported complicated Pandas code to Polars. We used the `pl.testing.assert_frame_equal` function to compare the Pandas and Polars code results. I showed the process that I used to change the code. I first made a drawing that gave me an idea of the inputs and outputs. You can also leverage AI to help you understand code you did not write. As I created the new columns, I checked the results along the way. I also refactored the code into functions and made sure that the code worked in a single cell. Code that was repeated frequently was refactored into functions.

10.14 Exercises

- Given some Pandas code (or another tabular manipulation library) of your choosing, come up with a plan to port it to Polars. Analyze the code and develop a roadmap. What are the inputs and outputs? What

are the intermediate steps? What are the dependencies? What are the repeated patterns?

2. Port the code from the previous exercise to Polars. Test the results.

Chapter 11

Extending Polars

While Polars is a feature-rich library, sometimes you need to extend Polars with custom functionality. This chapter will show how to use Numba to write custom logic. You can also use Python to extend Polars and add your own API to the dataframe class.

11.1 Loans with Polars

Let's imagine an example that is hard to represent with expressions. I'm going to create a schedule for a mortgage payment. The schedule will contain the payment amount, the interest paid, the principal paid, and the remaining balance. Writing Polars expressions to create columns for the schedule is possible, but most folks just use a for loop and track the variables as they iterate over the rows. Generally, when working with dataframes, we want to avoid for loops because they are slow. Let's compare and contrast the options for this problem.

Here is the formula to calculate the payment amount:

$$P = L \frac{c(1 + c)^n}{(1 + c)^n - 1}$$

Where - P is the payment amount - L is the loan amount - c is the monthly interest rate - n is the number of payments.

Let's calculate the payment amount for a \$600,000 loan with a monthly interest rate of 5.5% and a term of 30 years.

```
>>> L = 600_000
>>> c = 0.055 / 12
>>> n = 360
>>> P = (L *
...     (c * (1 + c) ** n) / \
...     ((1 + c) ** n - 1))
>>> P # monthly payment 3_406.734
3406.734008082003
```

11. Extending Polars

I'm going to write the pure Python code to solve this. This uses Python generators to create a sequence of dictionaries. Each dictionary represents a row in the schedule. I will use the `DataFrame` constructor to create a dataframe from the sequence of dictionaries.

```
def payment_schedule_gen(principal, number_of_payments,
                         monthly_interest_rate, monthly_payment):
    remaining_balance = principal
    done = False
    for month in range(number_of_payments):
        if remaining_balance < monthly_payment:
            interest_payment = remaining_balance * monthly_interest_rate
            monthly_payment = remaining_balance + interest_payment
            principal_payment = remaining_balance
            remaining_balance = 0
            done = True
        else:
            interest_payment = remaining_balance * monthly_interest_rate
            principal_payment = monthly_payment - interest_payment
            remaining_balance -= principal_payment
        yield {'month': month,
               'Principal': principal_payment,
               'Interest': interest_payment,
               'Remaining Balance': remaining_balance,
               'Monthly Payment': monthly_payment}
    if done:
        break
```

Now we want to make sure that it works.

```
>>> print(pl.DataFrame(payment_schedule_gen(L, n, c, P)))
shape: (360, 5)
```

month	Principal	Interest	Remaining Balance	Monthly Payment
---	---	---	---	---
i64	f64	f64	---	f64
0	656.73	2750.00	599343.27	3406.73
1	659.74	2746.99	598683.52	3406.73
2	662.77	2743.97	598020.75	3406.73
3	665.81	2740.93	597354.95	3406.73
4	668.86	2737.88	596686.09	3406.73
.
355	3329.73	77.01	13472.21	3406.73
356	3344.99	61.75	10127.23	3406.73
357	3360.32	46.42	6766.91	3406.73

358	3375.72	31.02	3391.19	3406.73	
359	3391.19	15.54	0.00	3406.73	

Let's see how long this takes to run.

```
>>> %%timeit
>>> pl.DataFrame(payment_schedule_gen(L, n, c, P))
254 µs ± 2.53 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

11.2 Use Numba

An option that I like to use with Pandas is the Numba library. Numba is a just-in-time compiler for Python. It can compile Python code to machine code. This can speed up the code by a factor of 100x or more. It's relatively easy to use. Often, we can take pure Python code and add a decorator to the function, and it will run much faster.

However, in this case, I cannot use my previous function, `payment_schedule_gen`, because it uses generators. Numba doesn't support generators, so I will adapt my code to use a for loop with a NumPy array. Then I will decorate the function with the `numba.njit` decorator. The `njit` decorator compiles the code to machine code.

```
from numba import njit
import numpy as np

@njit
def payment_schedule_numba(principal, number_of_payments,
                           monthly_interest_rate, monthly_payment):
    remaining_balance = principal
    done = False
    results = np.zeros((number_of_payments, 5), dtype=np.float64)
    for month in range(number_of_payments):
        if remaining_balance < monthly_payment:
            interest_payment = remaining_balance * monthly_interest_rate
            monthly_payment = remaining_balance + interest_payment
            principal_payment = remaining_balance
            remaining_balance = 0
            done = True
        else:
            interest_payment = remaining_balance * monthly_interest_rate
            principal_payment = monthly_payment - interest_payment
            remaining_balance -= principal_payment
        results[month, 0] = month
        results[month, 1] = principal_payment
        results[month, 2] = interest_payment
        results[month, 3] = remaining_balance
```

11. Extending Polars

```
results[month, 4] = monthly_payment
if done:
    break
return results
```

Let's try it out.

```
>>> print(pl.DataFrame(payment_schedule_numba(L, n, c, P))
...     .rename({'column_0': 'month', 'column_1': 'Principal',
...             'column_2': 'Interest', 'column_3': 'Remaining Balance',
...             'column_4': 'Monthly Payment'}))
shape: (360, 5)
  month Principal Interest Remaining Balance Monthly Payment
  ---      ---       ---      ---           ---
  f64       f64       f64      ---           f64
  0.00   656.73  2750.00  599343.27  3406.73
  1.00   659.74  2746.99  598683.52  3406.73
  2.00   662.77  2743.97  598020.75  3406.73
  3.00   665.81  2740.93  597354.95  3406.73
  4.00   668.86  2737.88  596686.09  3406.73
  .
  355.00 3329.73  77.01   13472.21  3406.73
  356.00 3344.99  61.75   10127.23  3406.73
  357.00 3360.32  46.42   6766.91   3406.73
  358.00 3375.72  31.02   3391.19   3406.73
  359.00 3391.19  15.54    0.00    3406.73
```

Let's time this code.

```
>>> %%timeit
>>> (pl.DataFrame(payment_schedule_numba(L, n, c, P))
...     .rename({'column_0': 'month', 'column_1': 'Principal',
...             'column_2': 'Interest', 'column_3': 'Remaining Balance',
...             'column_4': 'Monthly Payment'}))
24.7 µs ± 146 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Wow! This is 6x faster than the pure Python code. This is a good option if you want to use Python and you want to write code that is fast. Let's see if we can get the code to run in Polars without a for loop or Numba.

11.3 Closed Form Solution

Another way to calculate the mortgage schedule is to use the closed-form solution. This solution is faster than the iterative solution but is not as flexible (if we needed to have some custom logic embedded in a for loop).

The formula for the remaining balance is:

$$B = P \cdot \frac{(1 + c)^n - (1 + c)^p}{(1 + c)^n - 1}$$

Where

- B is the remaining balance
- P is the principal
- c is the monthly interest rate
- n is the term (number of months)
- p is the number of payments made.

The nice thing about this is that if we have a formula, we can use Polars to calculate the schedule. I'll need to create a dataframe with the number of payments and then create a column for the remaining balance. Once I have calculated the remaining balance, I can calculate the interest paid. From the interest paid, I can calculate the principal payment amount. I need to use multiple `.with_columns` calls to create the schedule because the columns depend on each other. If they were all in the same `.with_columns` call, then Polars would try to calculate the columns in parallel, which would fail.

```
sched = pl.DataFrame({'month': np.arange(360)})

def remaining_balance_pl(principal, number_of_payments,
                         monthly_interest_rate, num_month):
    return principal * ((
        (1 + monthly_interest_rate)**number_of_payments - \
        (1 + monthly_interest_rate)**num_month) / \
        ((1 + monthly_interest_rate)**number_of_payments - 1))
```

Let's use the function.

```
>>> print(sched
... .with_columns(
...     remaining_balance_pl(L, n, c,
...         pl.col('month')).alias('Remaining Balance'),
...         pl.lit(P).alias('Monthly Payment'))
... .with_columns(Interest=(pl.col('Remaining Balance') * c))
... .with_columns(Principal=(pl.col('Monthly Payment') - pl.col('Interest')))
... )
shape: (360, 5)
| month | Remaining Balance | Monthly Payment | Interest | Principal |
| --- | --- | --- | --- | --- |
| i64 | f64 | f64 | f64 | f64 |
```

11. Extending Polars

0	600000.00	3406.73	2750.00	656.73
1	599343.27	3406.73	2746.99	659.74
2	598683.52	3406.73	2743.97	662.77
3	598020.75	3406.73	2740.93	665.81
4	597354.95	3406.73	2737.88	668.86
.
355	16801.94	3406.73	77.01	3329.73
356	13472.21	3406.73	61.75	3344.99
357	10127.23	3406.73	46.42	3360.32
358	6766.91	3406.73	31.02	3375.72
359	3391.19	3406.73	15.54	3391.19

Not too bad. Let's time this code.

```
>>> %%timeit
>>> (sched
...     .with_columns(
...         remaining_balance_pl(L, n, c,
...             pl.col('month')).alias('Remaining Balance'),
...             pl.lit(P).alias('Monthly Payment'))
...     .with_columns(Interest=(pl.col('Remaining Balance') * c))
...     .with_columns(Principal=
...         (pl.col('Monthly Payment') - pl.col('Interest'))))
... )
116 µs ± 2.67 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Hmmm, this is faster than the pure Python code but slower than Numba. I'll also include my standard benchmark caveat: this is a small dataframe. Different code might be faster with a larger dataframe.

11.4 Adding a PCA API to Polars

Polars provides an API for extending the API. We can create attributes like `.str` and `.dt` that provide additional functionality. Let's create a PCA API for Polars. We will create a `DataFrame` attribute called `.pca` with methods for working with principal component analysis (PCA).

If you are unfamiliar with PCA, it is a technique for reducing the dimensionality of a dataset. It does this by finding the directions in which the data varies the most. These directions are called principal components. The first principal component is the direction in which the data varies the most. The second principal component is the direction in which the data varies the second most, and so on.

I'm going to use the Fuel Economy dataset to create run PCA. Let's load the data.

```
>>> autos = tweak_auto(raw)
```

Calculating the principal components involves a few steps. First, we need to center the data, which means subtracting the mean from each column. Then, we calculate the covariance matrix. Finally, we calculate the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors are the weights for the principal components. We get the principal components when we take the dot product of the centered data and the eigenvectors.

I want a method that will calculate the principal components. I also want a method that will calculate the explained variance. The explained variance is the proportion of the variance in the data explained by each principal component. In addition, I want a method for the weights of the principal components. I also want to create plots of the weights and a scatter plot of the principal components.

11.5 Calculating the Principal Components

Here's the enumerated steps to calculate the principal components:

1. Standardize the data (optional but generally recommended)
2. Center the data (required if you don't standardize the data)
3. Calculate the covariance matrix
4. Calculate the eigenvalues and eigenvectors of the covariance matrix
5. Calculate explained variance from sorted eigenvalues
6. Calculate weights of the principal components from the eigenvectors
7. Calculate the principal components from the dot product of the centered data and the eigenvectors

I'm going to standardize the data before I calculate the principal components. This means subtracting the mean from each column and dividing by the standard deviation. This is a common step when working with PCA. Because PCA is sensitive to the scale of the data, standardizing the data can make the principal components more interpretable.

Here's the code to calculate the principal components. I'll need to use the NumPy library to calculate the covariance matrix, eigenvalues, and eigenvectors. I'll test this to make sure that NumPy is working correctly with Polars.

```
import numpy as np
import polars.selectors as cs
X = (autos
      .select(cs.numeric().fill_null(0))
      .select((pl.all() - pl.all().mean()) / pl.all().std()) # 1
)
num_df = X
centered = (num_df # 2
            .select((pl.all() - pl.all().mean())))
            )
```

11. Extending Polars

```
cov = np.cov(centered.transpose()) # 3
vals, vecs = np.linalg.eig(cov) # 4

exp_var = pl.DataFrame( # 5
    {'PC': [f'PC{i+1}' for i in range(len(num_df.columns))],
     'var':sorted(vals, reverse=True)})

idxs = np.argsort(vals)[::-1]
comps = (pl.DataFrame(vecs[:, idxs]) # 6
         .rename(mapping={f'column_{i}': f'PC{i+1}' for i in range(len(num_df.columns))}))
)

pcas = (pl.DataFrame(np.dot(centered, comps)) # 7
        .rename(mapping={f'column_{i}': f'PC{i+1}' for i in range(len(num_df.columns))}))
)
```

Let's look at the results:

```
>>> print(pcas)
shape: (41_144, 7)
  PC1   PC2   PC3   .   PC5   PC6   PC7
---  ---  ---  ---  ---  ---  ---
  f64   f64   f64   .   f64   f64   f64

  0.89  -1.25  1.05  .  0.06  -0.03  0.05
  -4.58  -0.56  -0.85  .  0.08  1.73  -0.05
   1.87  -1.05  0.41  .  -0.40  -0.23  -0.24
  -3.75  -1.86  -0.57  .  0.87  0.24  -0.08
   0.50  -0.80  1.06  .  0.33  -0.05  -0.01
   .
   1.05  -1.15  0.55  .  -0.08  -0.10  -0.06
   1.27  -0.67  0.68  .  -0.07  -0.15  -0.10
   0.80  -1.19  0.69  .  0.01  -0.08  0.02
   0.78  -0.75  0.98  .  0.09  -0.12  0.06
   0.25  -1.29  0.92  .  0.38  0.01  0.02
```

That looks pretty good. I will perform a sanity check to ensure the results are correct. I will compare the results to the results from the scikit-learn library.

```
>>> from sklearn.decomposition import PCA
>>> import sklearn
>>> sklearn.set_config(transform_output='polars')
>>> pca = PCA()
```

```
>>> print(pca.fit_transform(X))
shape: (41_144, 7)

```

pca0	pca1	pca2	.	pca4	pca5	pca6
---	---	---	.	---	---	---
f64	f64	f64	.	f64	f64	f64
0.89	-1.25	-1.05	.	-0.06	-0.03	0.05
-4.58	-0.56	0.85	.	-0.08	1.73	-0.05
1.87	-1.05	-0.41	.	0.40	-0.23	-0.24
-3.75	-1.86	0.57	.	-0.87	0.24	-0.08
0.50	-0.80	-1.06	.	-0.33	-0.05	-0.01
.
1.05	-1.15	-0.55	.	0.08	-0.10	-0.06
1.27	-0.67	-0.68	.	0.07	-0.15	-0.10
0.80	-1.19	-0.69	.	-0.01	-0.08	0.02
0.78	-0.75	-0.98	.	-0.09	-0.12	0.06
0.25	-1.29	-0.92	.	-0.38	0.01	0.02

Awesome. It looks like the calculations are correct.

11.6 Scatter Plot of the Principal Components

I want to create a 3D scatter plot of the principal components. I know that the hvplot library is going to be problematic for this, so I'm going to use the plotly library.

Here's the code to create the scatter plot.

```
import plotly.express as px
def plot_pca_3d(df, x='PC1', y='PC2', color_col=None, size_col=None,
                 symbol_col=None, cmap='viridis', components=None,
                 biplot=True, biplot_scale=20, biplot_limit=.2,
                 alpha=1, width=600, height=600):

    if color_col is not None:
        data = (df
                .with_columns(color_col)
                )
    else:
        data = df

    fig = px.scatter_3d(data, x='PC1', y='PC2', z='PC3', color=color_col,
                        color_continuous_scale=cmap,
                        hover_data=df.columns[:10],
                        symbol=symbol_col,
                        size=size_col,
                        opacity=alpha,
```

11. Extending Polars

```
width=width, height=height,
#log_y=True,
)

if size_col is None:
    fig.update_traces(marker_size=3)

if biplot:
    scale = biplot_scale
    annots = []
    for column in components.columns:
        loadings = {f'PC{i}':val*scale
                    for i, val in enumerate(components[column], 1)}
        loadings['name'] = [column]
        new_fig = px.line_3d(x=[0, loadings['PC1']],
                              y=[0, loadings['PC2']],
                              z=[0, loadings['PC3']], width=20)
        for trace in new_fig['data']:
            fig.append_trace(trace, row=1, col=1)

    annots.append(
        dict(
            showarrow=False,
            x=loadings['PC1'],
            y=loadings['PC2'],
            z=loadings['PC3'],
            text=column,
            xanchor="left",
            xshift=1,
            opacity=0.7)
    )
    fig.update_layout(scene={'annotations':annots})
return fig
```

Let's try this out and see the plot. I'm going to color the points by the year of the car.

```
plot_pca_3d(pcals, color_col=autos['year'], biplot=False, width=800)
```

One of the reasons that I like to use Plotly for 3D plots is that you can rotate the plot. This is useful for exploring the data. You can also zoom in and hover over the points to see the values.

Let's add the API to the DataFrame class.

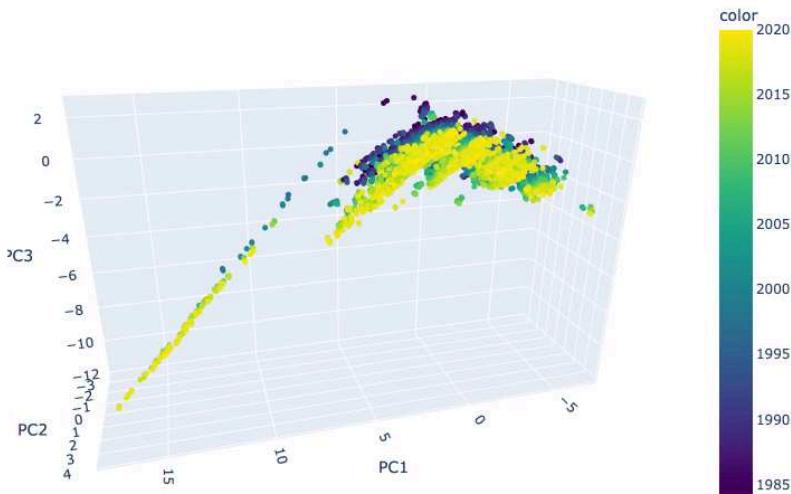


Figure 11.1: 3-D scatterplot of principal components.

11.7 Adding the API

Here's the code to create a .pca API. We create a class with the methods that we want for the dataframe. Then, we decorate the class with the `pl.api.register_dataframe_namespace` decorator. Our PCA methods will be available off of that attribute now. Pretty cool!

```
import polars as pl
import numpy as np

@pl.api.register_dataframe_namespace('pca')
class PCA:
    def __init__(self, df):
        self.df = df

    def fit(self):
        centered = (self.df
                    .select(pl.all() - pl.all().mean())
                    )
        cov = np.cov(centered.transpose())
```

11. Extending Polars

```
vals, vecs = np.linalg.eig(cov)
self._explained_variance = pl.DataFrame(
    {'PC': [f'PC{i+1}' for i in range(len(num_df.columns))],
     'var':sorted(vals, reverse=True)})
)

idxs = np.argsort(vals)[::-1]
comps = (pl.DataFrame(vecs[:, idxs])
    .rename(mapping={f'column_{i}': f'PC{i+1}' for i in range(len(num_df.columns))}))
)

self.pcs = (pl.DataFrame(np.dot(centered, comps))
    .rename(mapping={f'column_{i}': f'PC{i+1}' for i in range(len(num_df.columns))}))
)

self._components = comps
return self.df

def transform(self):
    return self.pcs

def explained_variance(self):
    return self._explained_variance

def components(self):
    return (self._components
        .with_columns(Feature=pl.Series(self.df.columns)))
)

def filter_components(self, limit_components, mag_threshold):
    comps = self.components()
    columns = comps.columns[:limit_components]
    res = (comps
        .select(*columns, pl.col('Feature'))
        .filter(pl.any_horizontal(cs.numeric().abs() > mag_threshold)))
    )
    return res

def component_plot(self, limit_components=3, mag_threshold=.1):
    comps = self.filter_components(limit_components, mag_threshold)
    return (comps
        .select(cs.numeric())
        .transpose()
        .rename(mapping={f'column_{i}': col for i, col in enumerate(comps['Feature'])}))
```

```

        .select(pl.Series([f'PC{i+1}' for i in range(limit_components)]).alias('PC'),
                *comps['Feature']))
    .plot.bar(x='PC',
               rot=90)
)

def scatter3d_plot(self, x='PC1', y='PC2', z='PC3', color_col=None,
                   size_col=None, symbol_col=None, cmap='viridis',
                   biplot=True, biplot_scale=20, biplot_limit=.2,
                   alpha=1, width=600, height=600):
    return plot_pca_3d(self.pcs, x, y, color_col, size_col,
                       symbol_col, cmap, self._components, biplot, biplot_scale,
                       biplot_limit, alpha, width, height)

```

The API is loosely based on the scikit-learn PCA API.

- `fit` calculates the principal components
- `transform` returns the principal components
- `components` returns the weights of the principal components
- `explained_variance` returns the explained variance
- `component_plot` creates a plot of the weights
- `scatter3d_plot` creates a 3D scatter plot of the principal components

Let's try it out. We need to call `.fit` to calculate the principal components. Then, we can call the other methods to get the results.

```
fit = (X
       .pca.fit())
```

Let's look at the weights of the principal components. This uses `hvplot` to create a bar plot of the weights. We can see that the mileage impacts the first principal component in the positive direction, while the *barrels08*, *cylinders*, and *displ* impact the first principal component in the negative direction. Intuitively, this makes sense. The first principal component represents efficiency. Moving to the right on the first principal component means the car is more efficient.

In the second principal component, the number of gears and the year of the car have the most impact. These two seem to be positively correlated. Presumably, vehicles with more gears are newer.

The explained variance ratio shows that the first principal component explains around 57% of the variance in the data. The second principal component explains around 23% of the variance.

```
>>> print(fit.pca.explained_variance())
...   .select(pl.all() / pl.all().sum()))
shape: (7, 2)
```

11. Extending Polars

PC	var
---	---
f64	f64
null	0.57
null	0.23
null	0.11
null	0.05
null	0.02
null	0.01
null	0.00

Here's the code to create a bar plot of the weights.

```
fit.pca.component_plot()
```

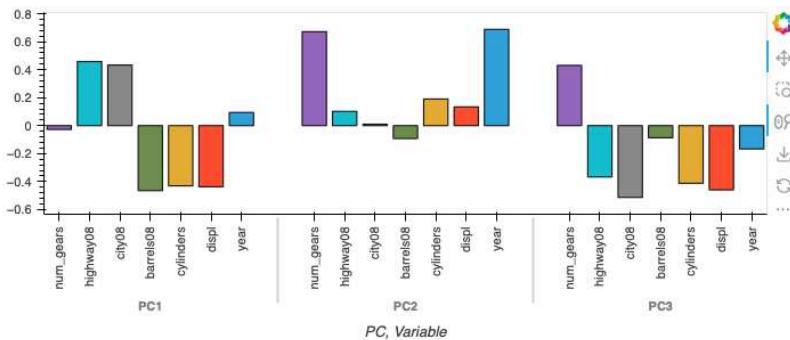


Figure 11.2: Bar plot of the weights of the principal components. The magnitude of the weights is shown on the y-axis. The features are shown on the x-axis. Features with larger weights have a larger impact on the principal component.

I often use PCA to explore data. It is a great way to see what linear combinations exist in the data. By viewing the weights and the scatter plot of the principal components, I gain insights into the data that I wouldn't have otherwise.

11.8 Summary

In this chapter, we learned how to use Numba to speed up Python code. We created a mortgage schedule and compared the iterative and closed-form solutions' performances. We also created a plotting API for Polars.

11.9 Exercises

1. Write a Python function to perform rolling window calculations. The function should allow the user to specify the window size and the function to apply to it. For example, the user should be able to calculate the rolling mean of a column with a window size of 3. The function should return a new column with the rolling mean.
2. Use Numba to create a rolling window function. Compare the performance of the Numba function to the pure Python function.

Chapter 12

Conclusion

Congratulations on getting this far. We have examined many real-world examples of Polars code. You might be asking yourself, “What’s next?”

Here are my suggestions:

- Reading is great, but practice is better. Try to solve the problems on your own. This is even better if you can find datasets relevant to your work or personal interests.
- Typing out the code examples is a great way to help you learn. You can also experiment with the code examples to see what happens. If you are new to Python, just typing out the examples in the book is a great way to practice. Then, you can try to modify the code to see what happens. Finally, you can adapt the code to solve your own problems.
- Get involved in the Polars community. You can ask questions and get help from the community. You can also help others in the community. This is a great way to learn.

I provide useful examples, clear explanations, and best practices as an educator. However, I am human, and I make mistakes. I would appreciate your help in improving this book. If you find any errors, typos, or confusing explanations, please report them. Your feedback will help me to make this book better. The GitHub repository for this book is the best place to do that.

12.1 One Last Thing

This book is a self-published labor of love. I am an independent author, and I don’t have the large marketing budget of a big publisher. If you found this book helpful, please spread the word. Tell your friends, write a review on Amazon, or share this book on social media. The best way you can thank me for writing the book is to leave a review and tell others about this book.

If your team at work is using (or considering using) Polars, I’d love to help you. My company, MetaSnake, provides corporate training and consulting for Polars and other data science tools. We have helped some

12. Conclusion

of the world's largest entertainment, technology, energy, hardware, and financial companies. We'd love to help your team, too. You can contact me at matt@metasnake.com.

About the Author

Matt Harrison, a seasoned Python expert since 2000, runs MetaSnake, a Python and Data Science consultancy and corporate training enterprise. With a diverse professional background, Matt has made significant contributions in areas including search, build management, testing, business intelligence, and storage.

He is a recognized voice in the Python community, having imparted knowledge at esteemed conferences like ODSC, SciPy, SCALE, PyCON, and PyData, as well as various local user events. Drawing from his extensive experience in teaching Python, this book encapsulates the insights and strategies he has found most effective in his instructional journey.

He blogs at hairsun.com and occasionally tweets useful Python related information at `@__mharrison__`.

Technical Editors

Anique Khawar, a Senior Data Scientist at Blue Cross Blue Shield of Minnesota, applies his deep enthusiasm for data wrangling to set up data pipelines and reshape the healthcare data field. He finds joy in Python programming and constantly seeks out new technologies to optimize efficiency.

Thomas M. Ahern is a Data Scientist with 18 years of experience in discovery analytics and extracting patterns and insights from unstructured data. A Navy veteran, ultra-endurance athlete, and photographer who's survived the 7 continents, Thomas finds challenging problem sets wonderfully interesting, which lends itself to his unwitting tendency to break systems and push things to the extreme.

Index

.contains, 22
.str.contains, 22
.agg, 73
.alias, 22, 48, 53
.cast, 16
.cat.set_ordering, 130
.collect, 40
.collect (streaming), 288
.columns, 51
.contains, 114
.corr, 37
.cov, 337
.describe, 15, 36, 163
.dot, 337
.drop_nulls, 165
.drop, 51
.dtypes, 12
.equals, 252
.estimated_size, 36
.eval, 111
.exclude, 52
.explain, 11, 287
.explode, 245
.extract, 26, 107
.fetch, 283
.fill_nan, 173
.fill_null, 84, 170
.filter, 74, 145
.group_by_dynamic, 209
.group_by_dynamic additional columns,
 214
.group_by, 73
.insert_column, 37
.interpolate, 171
.interpolate example, 224
.is_between, 150
.is_duplicated, 80
.is_duplicate, 79
.is_finite, 83
.is_nan, 82, 83, 164, 173
.is_null, 82, 83, 161
.is_null example, 218
.join, 54, 104, 227
.len, 102, 109, 133, 154
.list.eval, 111
.list.join, 104
.list.len, 102, 109
.list.sort example, 126
.list.sum, 112
.list.to_struct, 102
.map_batches, 91
.map_elements (example), 104
.map_elements example, 246
.mean, 337
.melt, 77
.melt example, 66
.null_count, 162
.otherwise, 155
.over, 137, 213
.over filter, 152
.pipe, 37, 164
.pivot, 74, 133
.plot, 220
.read_excel, 65
.rename, 53
.replace example, 67
.sample, 19
.schema, 12
.select, 13, 48
.select order, 215

Index

.set_sorted, 210
.shrink_dtype, 18
.sort, 73, 180
.split, 109
.starts_with, 100, 116
.std, 337
.str.contains, 114
.str.extract, 26, 107
.str.split, 109
.str.split example, 104
.str.starts_with, 100, 116
.str.strptime, 204
.str.to_datetime, 32, 116, 204
.str.to_date, 204
.strptime, 204
.struct indexing, 155
.sum, 112
.then, 155
.time_zone, 205
.to_datetime, 32, 116, 204
.to_date, 204
.to_pandas, 38, 258
.to_struct, 102
.truncate, 217
.upsample, 224
.value_counts, 24
.vstack, 72, 215
.when, 155
.when example, 67
.with_columns, 16, 49
.with_columns order, 215
.with_row_index, 50
.write_csv, 238
.write_database, 253
.write_excel, 250
.write_json, 239
.write_parquet, 252
1:m, 63
Categorical, 29
Enum, 190
Series, 92
StringCache, 70, 187
UInt8, 16
Utf8, 28
8, 146
~, 146
|, 146
aggregate_function strings, 75
agg, 73
alias, 22, 48
all_horizontal, 169
api.register_dataframe_namespace, 341
assert_frame_equal, 306
atol, 312
by sort control, 180
cast, 16
collect, 40
columns, 51
concat_list, 126
conda, 4
contains, 114
corr, 37
cs.matches example, 126
cs.numeric, 88, 164
cs, 88
datetime, 150
date, 204
describe, 15, 36, 163
drop_nulls, 165
drop, 51
dt.truncate, 217
duckdb, 254
eig, 337
equals, 252
estimated_size, 36
eval, 111
exclude, 52
expand_selector, 89
explain, 11, 287
explode, 245
extract, 26, 107
fetch, 283
fill_nan, 173
fill_null, 84, 170
filter, 74, 145
from_epoch, 243
from_pandas, 258, 280
group_by_dynamic, 209
group_by_dynamic additional columns,
 214
group_by, 73
hInt8, 16

header_row, 65
insert_column, 37
interpolate, 171
interpolate example, 224
is_between, 150
is_duplicated, 80
is_duplicate, 79
is_finite, 83
is_nan, 82, 83, 164, 173
is_null, 82, 83, 161
join, 54, 104, 227
key sorting, 180
len, 102, 109, 133, 154
lexical, 133
list.eval, 111
list.join, 104
list.len, 102, 109
list.sum, 112
list.to_struct, 102
lit example, 157
m:m, 63
maintain_order, 130, 195
map_batches, 91
matches examples, 126
melt, 77
njit, 333
null_count, 162
numeric, 88
ordering, 186
ordering category, 130
otherwise, 155
over, 137, 213
pipe, 37, 164
pip, 4
pivot, 74, 133
pivot aggregate function strings,
 75
pl.Enum, 190
pl.Series, 92
pl.StringCache, 70, 187
pl.String, 28
pl.Utf8, 28
pl.all(), 161
pl.all_horizontal, 169
pl.all_horizontal example, 218
pl.all example, 139
pl.api.register_dataframe_namespace,
 341
pl.col('*'), 161
pl.concat_list, 126
pl.date, 204
pl.element, 111
pl.from_epoch, 243
pl.from_pandas, 258, 280
pl.lit, 157
pl.read_csv, 9, 237
pl.read_database_uri, 253
pl.read_database, 253
pl.read_excel, 250
pl.read_json, 241, 248
pl.read_parquet, 252
pl.scan_csv, 10, 282
pl.struct, 80
plot, 220
polars.testing.assert_frame_equal,
 306
pytz, 206
read_csv, 9, 237
read_database_uri, 253
read_database, 253
read_excel, 65, 250
read_json, 241, 248
read_parquet, 252
register_dataframe_namespace, 341
sample, 19
scan_csv, 10, 282
schema, 12
selectors, 88, 164
select, 13, 48
select order, 215
set_sorted, 210
sheet_name, 65
shrink_dtype, 18
sort, 73, 180
split, 109
split example, 104
sql, 254
starts_with, 100, 116
str.contains, 114
str.extract, 26, 107
str.split, 109
str.starts_with, 100, 116

Index

`str.to_datetime`, 32, 116
`stream`, 288
`strptime`, 204
`struct`, 244
`struct indexing`, 155
`sum`, 112
`testing.assert_frame_equal`, 306
`then`, 155
`time_zone`, 205
`to_datetime`, 32, 116, 204
`to_date`, 204
`to_pandas`, 38, 258
`to_struct`, 102
`truncate`, 217
`try_parse_dates`, 203
`tweak_auto` function, 47
`upsample`, 224
`value_counts`, 24
`vstack`, 72
`when`, 155
`when example`, 67
`with_columns`, 16, 49
`with_columns order`, 215
`with_row_index`, 50
`write_csv`, 238
`write_database`, 253
`write_excel`, 250
`write_json`, 239
`write_parquet`, 252
`xgboost`, 259
`zipfile`, 9
`zip`, 12
`'float_precision)`, 238
3D scatter plot, 339
8 bit integer, 16

add methods to dataframe, 341
Adding a PCA API to Polars, 336
adding columns, 48
adding rows, 72
aggregate to a list, 139
aggregation, 123
alias, 132
alias rant, 23
and, 146
anti-join, 62

API extension, 336
`api register_dataframe_namespace`, 341
apply function, 91
ascending sort, 185
automatic type shrinking, 18

back fill, 171
background style, 38
between values, 150
boolean operations, 146

calalime read options, 251
calamine, 65
cartesian product, 61
cast string to float, 66
cast to ordered category, 186
casting to dates, 204
category limiting, 159
category order, 130
category ordering examples, 133
chaining, 8, 38
change timezone, 205
changing column types, 16
close database connection, 254
closed form solution, 334
code review, 301
column filtering, 164
column names from keyword arguments, 23
column names unique, 132
column regex selection, 126
column selection, 51
column selection set operation, 127
column selectors, 88, 164
column sorting, 179
column types, 12
combine column selectors, 88
combine columns into list, 126
combine columns to date, 204
combining columns, 80
comparing dataframes, 306
comparing two dataframes, 252
contains, 22
context, 7

-
- context manager, 187, 254
 convert epoch to date, 243
 convert list to struct, 102
 convert string to float, 66
 convert struct to list, 104, 246
 Convert to Pandas, 258
 convert to pandas, 38
 correlation, 37
 count of missing values, 162
 counting occurrences, 154
 create database, 253
 create date from columns, 204
 create list, 126
 cross join, 61
 CSV files, 9
 CSV float precision, 238
 CSV read, 237
 CSV type inference, 12
 CSV writing, 238
 custom JSON parsing, 248
 custom ordering, 187
- data type, 12
 database creation, 253
 dataframe, 5
 dataframe test comparison, 306
 datatypes of columns, 12
 date (month) sorting, 190
 date conversion, 204
 date conversion in CSV, 203
 date filtering, 150
 date from columns, 204
 date from UNIX epoch, 243
 date grouping with additional columns, 214
 date parsing, 32
 dealing with nan, 173
 debugging lazy code, 283
 decorator, 187
 decorators, 333
 descending sort, 185
 dictionary replacement, 280
 direction of sort, 185
 downsample, 224
 drop rows with missing values, 218
- DuckDB, 254
 duplicate column names, 132
 duplicates, 79
 eager, 5
 element evaluation, 111
 emoji extraction, 114
 Enum, 190
 enum example, 192
 epoch conversion, 243
 equality (dataframes), 306
 equality filtering, 149
 evaluating element expressions in a list, 111
 excel read options, 251
 Excel reading, 250
 Excel writing, 250
 exclude columns from dataframe, 52
 explain query plan, 287
 exploding data, 245
 exporting to JSON, 239
 expression, 5, 7
 extract digit from string, 26
- fake index, 50
 fastexcel, 65
 fastexcel read options, 251
 fetch a small number of rows, 283
 fill missing values, 170
 filter columns, 13
 filter with z-score, 151
 filtering columns, 164
 filtering row, 145
 filtering rows with semi-join, 60
 filtering with dates, 150
 filtering with window function, 152
 find nan values, 164
 find outliers, 151
 finding duplicates, 79
 float conversion, 19
 float precision, 238
 forward fill, 171
 function application, 91
 generalize categories, 159

Index

- generator, 248
- global variables, 301
- group by time window, 209
- groupby maintain order, 130
- groupby ordering, 195
- grouping by multiple columns and dates, 214
- grouping by time, 209
- grouping to a list, 139
- grouping with multiple columns, 133
- horizontal operations, 169
- hvPlot, 117
- if else example, 67
- if-else statements, 155
- imputation, 170
- index creating, 50
- inner join, 58
- install plotting, 117
- install polars, 4
- integers, 16
- interpolate, 171
- interpolation, 221
- intervals (time), 211
- jitter, 114
- join example, 67
- join list into string, 104
- join validation, 63
- joining dataframes, 54
- joining time series, 226
- JSON generator parsing, 248
- JSON read, 241
- JSON writing, 239
- keyword argument column names, 23
- keyword filtering, 149
- lazy, 5
- lazy evaluation, 282
- lazy mode, 38
- lazy reading, 10
- left join, 56
- length of list, 102, 109
- lexical ordering, 130, 186
- limit categories, 159
- limit lazy rows, 283
- line plot, 220
- list comprehension, 66
- list evaluation, 111
- list from columns, 126
- list from struct, 246
- list length, 102, 109
- list sort, 126
- list sum, 112
- list timezones, 206
- list to struct, 102
- literal, 157
- Load from Pandas, 258
- Loan Schedule, 331
- maintain order, 195
- many to many join, 63
- many to one join, 63
- materialize data, 40
- melting data, 77
- memory usage, 36
- merging data, 54
- merging time series, 226
- method chaining, 8
- method registration, 341
- missing data, 161
- missing number removal, 165
- missing value count, 162
- missing value replacement, 170
- missing values, 82
- missing values in time series, 171, 221
- month order, 192
- month sorting, 190
- mortgage payment, 331
- multiple aggregations, 131
- multiple column sort, 184
- multiple groupings and dates, 214
- multiplication with a row, 297
- named parameter column names, 23
- NaN, 82
- nan, 173

-
- nested values, 80
 - new methods, 341
 - normal distribution, 151
 - not, 146
 - NULL, 82
 - null count, 162
 - null removal, 165
 - null values, 161
 - Numba, 333
 - number select, 88
 - numeric type shrinking, 18
 - NumPy, 86
 - NumPy PCA code, 337
 - offset alias, 209, 211
 - one to one join, 63
 - Optimize slow code, 333
 - or, 146
 - order categoricals in custom way, 187
 - order for months, 192
 - order of groupby, 195
 - ordered category, 186
 - ordered category example, 133
 - ordering in category, 130
 - outer joins, 59
 - outlier filtering, 151
 - output polars with scikit learn, 262
 - over aggregations, 137
 - Pandas .transform, 137
 - Pandas conversion, 38, 258
 - pandas plotting, 135
 - Pandas transform, 213
 - parentheses, 8
 - Parquet reading, 252
 - Parquet writing, 252
 - parsing dates, 32, 203
 - PCA, 165, 262, 336
 - Pearson correlation, 37
 - physical ordering, 130, 186
 - pivot (lazy), 282
 - pivoting, 73
 - pivoting data, 74
 - pivoting with multiple aggregations, 133
 - plan query, 287
 - plotly, 220
 - Plotly scatter plot, 339
 - plotting in polars, 136
 - plotting installation, 117
 - plotting with pandas, 135
 - polars output from scikit learn, 262
 - polars plotting, 136
 - positive column selection, 51
 - precision, 19
 - precision (float), 238
 - prefix, 132
 - Principal Component Analysis, 336
 - Principal component analysis, 165, 262
 - query optimization, 11
 - query optimizer, 38
 - query plan, 287
 - RAM usage, 36
 - range filtering, 150
 - rant about alias, 23
 - read CSV, 237
 - read Excel, 250
 - reading JSON, 241
 - reduction, 123
 - refactoring, 304
 - regex column selection, 126
 - register dataframe namespace, 341
 - regular expression, 26
 - regular expression extract, 107
 - regular expression to extract digit, 26
 - remove columns, 51
 - remove missing numbers, 165
 - renaming columns, 53
 - reorder category, 186
 - replace empty string, 66
 - replace values, 280
 - right join, 57
 - rolling functions, 219
 - rounding errors, 312
 - row filtering, 145
 - row multiplication, 297

Index

- row-wise and operation, 169
scatterplot with jitter, 114
scikit learn output polars, 262
select all columns, 161
select columns, 88
select columns with regex, 126
select numeric columns, 88
selecting string columns, 28
semi-join, 60
series, 5
set categorical ordering, 186
set column name, 22
set difference, 127
set operation, 127
set operations on column selec-
tors, 88
shrink numeric types, 18
simulating an index, 50
skip rows in Excel, 251
slow code, 333
sort aggregated data, 133
sort direction, 185
sorting and groupby, 130
sorting by a column, 179
sorting months, 190
sorting my multiple columns, 184
sorting stable, 196
sorting with arbitrary logic, 180
split string, 104
SQLite, 253
stable sorting, 196
standardize, 337
str.contains, 22
streaming, 288
strftime, 32
string cache, 70, 187
string columns, 28
string extract, 107
strings for aggregate functions, 75
struct, 80, 244
struct indexing, 155
struct to list, 104, 246
suffix, 132
sum a list of booleans in a column,
 112
summary statistics, 36
testing equality, 312
time aggregations, 209
time conversion, 204
time intervals, 211
time series merge, 226
time series missing values, 171
time window grouping, 209
timezone change, 205
tolerance, 312
top n, 73
transform (Pandas), 213
transform (pandas), 137
truncate date, 217
tweak function, 47
type inference, 11, 12
type shrinking, 18
types of columns, 12
undo pivot, 77
unique column names, 132
UNIX epoch conversion, 243
unnest struct, 102
unsigned integer, 16
upsample, 224
validate joins, 63
value counts, 154
value counts (example), 133
value_counts, 24
white space, 8
window function example, 137
window function filter, 152
window functions, 219
words in columns, 109
write CSV, 238
write Excel, 250
writing JSON, 239
XGBoost, 86, 259
z-score filter, 151
ZIP file, 9

