

Python for High Performance Data Analytics

Prerequisites

- Basic proficiency in Python (variables, flow control, functions)
- Basic grasp of descriptive statistics
- Basic knowledge of NumPy
- Basic knowledge of some plotting package (Matplotlib, Seaborn, Holoviz...)

20 min filename

Tabular data (aka Dataframes)

Questions

- What are series and dataframes?
- What do we mean by tidy and untidy data?
- What packages are available in Python to handle dataframes?

Objectives

- Learn how to manipulate dataframes in Pandas
- Lazy and eager dataframes in Polars
- Learn how to abstract the underlying libraries away with Narwhals

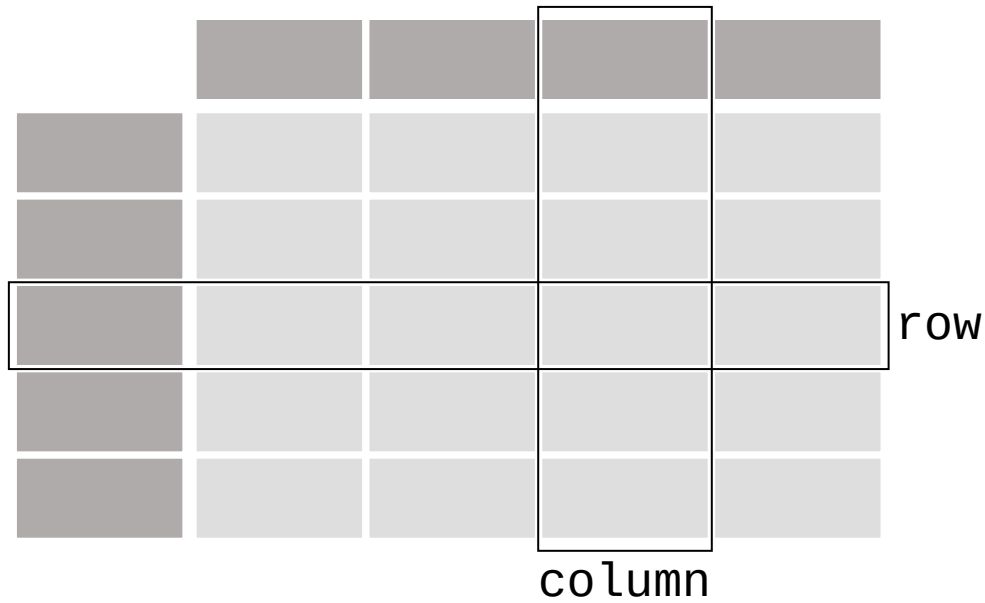
This episode will give an introduction to the concepts of *Series* and *DataFrame* and how they can be manipulated using different Python packages.

Series and dataframes

A collection of observations (e.g. a time series or simply a set of observations of a feature of a phenomenon) can be represented by a homogeneous vector, i.e. an array where all the elements are of the same type. This is known as a *Series* in many frameworks. Several series (of different types) can be used as columns of a tabular structure called a *Dataframe*, as depicted in the

[Skip to content](#)

DataFrame



Tidy vs untidy dataframes

Let us look at the following two dataframes:

Untidy format

Tidy format

	Runner	400	800	1200	1500
0	Runner 1	64	128	192	240
1	Runner 2	80	160	240	300
2	Runner 3	96	192	288	360

[Skip to content](#)

Most tabular data is either in a tidy format or a untidy format (some people refer them as the long format or the wide format). The main differences are summarised below:

- In untidy (wide) format, each row represents an observation consisting of multiple variables and each variable has its own column. This is intuitive and easy for us to understand and make comparisons across different variables, calculate statistics, etc.
- In tidy (long) format , i.e. column-oriented format, each row represents only one variable of the observation, and can be considered “computer readable”. When it comes to data analysis using Pandas, the tidy format is recommended:
- Each column can be stored as a vector and this not only saves memory but also allows for vectorized calculations which are much faster.
- It's easier to filter, group, join and aggregate the data.

Pandas & Polars

Historically, [Pandas](#) has been the go-to package to handle dataframes in Python. It is based on NumPy (each column is a Numpy vector) and has been the traditional workhorse for tabular data, with a stable API and a large ecosystem built around it, including the [Seaborn](#) statistical plotting framework. More recently, [Polars](#) was introduced as a more modern and faster alternative to handle dataframes. It is written in Rust and supports out of the box out of core evaluation (i.e. does not need loading the whole dataset in memory), lazy evaluation of queries and automatically uses multiple threads. Moreover, experimental GPU support is available through [cuDF](#). In the remainder of this episode, the [NYC taxi](#) will be used to showcase how datasets can be accessed, summarised and manipulated in both Pandas and Polars. The dataset can be download in [Parquet](#) format from the link above (the file for the month of January was used in this case). The dataset contains information about taxi trips performed in New York, such as the ID of the vendor, the total fare, pickup and drop-off time and location (expressed as an ID), type of payment, whether additional fees were charged and more.

Opening a dataset

Assuming the file is called `yellow_tripdata_2025-01.parquet`, the dataset can be opened as:

Pandas

Polars

```
import pandas as pd
df = pd.read_parquet("yellow_tripdata_2025-01.parquet")
```

Skip to content

Description and summarisation

We can get a first understanding of the contents of a dataframe by printing the first few lines, the “schema” (i.e. the number and type of each column) and summary statistics as follows:

Pandas

Polars

df.head()

Output

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	...	congestion_surcharge	Airport_fee	cbd
0	1	2025-01-01 00:18:38	2025-01-01 00:26:59	...	2.5	0.0	
1	1	2025-01-01 00:32:40	2025-01-01 00:35:13	...	2.5	0.0	
2	1	2025-01-01 00:44:04	2025-01-01 00:46:01	...	2.5	0.0	
3	2	2025-01-01 00:14:27	2025-01-01 00:20:01	...	0.0	0.0	
4	2	2025-01-01 00:21:34	2025-01-01 00:25:06	...	0.0	0.0	

df.info()

Output

RangeIndex: 3475226 entries, 0 to 3475225

Data columns (total 20 columns):

#	Column	Dtype
0	VendorID	int32
1	tpep_pickup_datetime	datetime64[us]
2	tpep_dropoff_datetime	datetime64[us]
3	passenger_count	float64
4	trip_distance	float64
5	RatecodeID	float64
6	store_and_fwd_flag	object
7	PULocationID	int32
8	DOLocationID	int32
9	payment_type	int64
10	fare_amount	float64
11	extra	float64
12	mta_tax	float64
13	tip_amount	float64
14	tolls_amount	float64
15	improvement_surcharge	float64
16	total_amount	float64
17	congestion_surcharge	float64
	fee	float64
	estion_fee	float64
	ime64[us](2), float64(13), int32(3), int64(1), object(1)	

memory usage: 490.5+ MB

Skip to content

```
df.describe()
```

Output

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	...	congestion_surcha
count	3.475226e+06	3475226	3475226	...	2.935077e
mean	1.785428e+00	2025-01-17 11:02:55.910964	2025-01-17 11:17:56.997901	...	2.225237e
min	1.000000e+00	2024-12-31 20:47:55	2024-12-18 07:52:40	...	-2.500000e
25%	2.000000e+00	2025-01-10 07:59:01	2025-01-10 08:15:29.500000	...	2.500000e
50%	2.000000e+00	2025-01-17 15:41:33	2025-01-17 15:59:34	...	2.500000e
75%	2.000000e+00	2025-01-24 19:34:06	2025-01-24 19:48:31	...	2.500000e
max	7.000000e+00	2025-02-01 00:00:44	2025-02-01 23:44:11	...	2.500000e
std	4.263282e-01	NaN	NaN	...	9.039932e

[8 rows x 19 columns]

Indexing

We can index data in the dataframe as follows:

Pandas

Polars

```
# With this we can select a column  
df['VendorID'] # Could also be df.VendorID
```

Output

0	1
1	1
2	1
3	2
4	2
...	...
3475221	2
3475222	2
3475223	2
3475224	2
3475225	2

```
# Get a row  
df.iloc[1000,:]
```

[Skip to content](#)

Output

```
VendorID                2
tpep_pickup_datetime    2025-01-01 00:08:06
tpep_dropoff_datetime   2025-01-01 00:16:20
passenger_count         4.0
trip_distance           1.53
RatecodeID              1.0
store_and_fwd_flag      N
PULocationID            114
DOLocationID            90
payment_type            1
fare_amount             10.0
extra                   1.0
mta_tax                 0.5
tip_amount              2.25
tolls_amount            0.0
improvement_surcharge   1.0
total_amount            17.25
congestion_surcharge    2.5
Airport_fee             0.0
cbd_congestion_fee      0.0
```

```
Name: 1000, dtype: object
```

```
>>> df.iloc[1000,:]
```

```
VendorID                2
tpep_pickup_datetime    2025-01-01 00:08:06
tpep_dropoff_datetime   2025-01-01 00:16:20
passenger_count         4.0
trip_distance           1.53
RatecodeID              1.0
store_and_fwd_flag      N
PULocationID            114
DOLocationID            90
payment_type            1
fare_amount             10.0
extra                   1.0
mta_tax                 0.5
tip_amount              2.25
tolls_amount            0.0
improvement_surcharge   1.0
total_amount            17.25
congestion_surcharge    2.5
Airport_fee             0.0
cbd_congestion_fee      0.0
```

In both cases, a similar syntax can be used to do in-place modification (e.g. `df[row][column]=...`). Please note that this kind of replacement carries a big performance penalty, which is designed to do column-wide operations with minimal overhead. This is commonly achieved through the [expression API](#), as detailed in the next section.

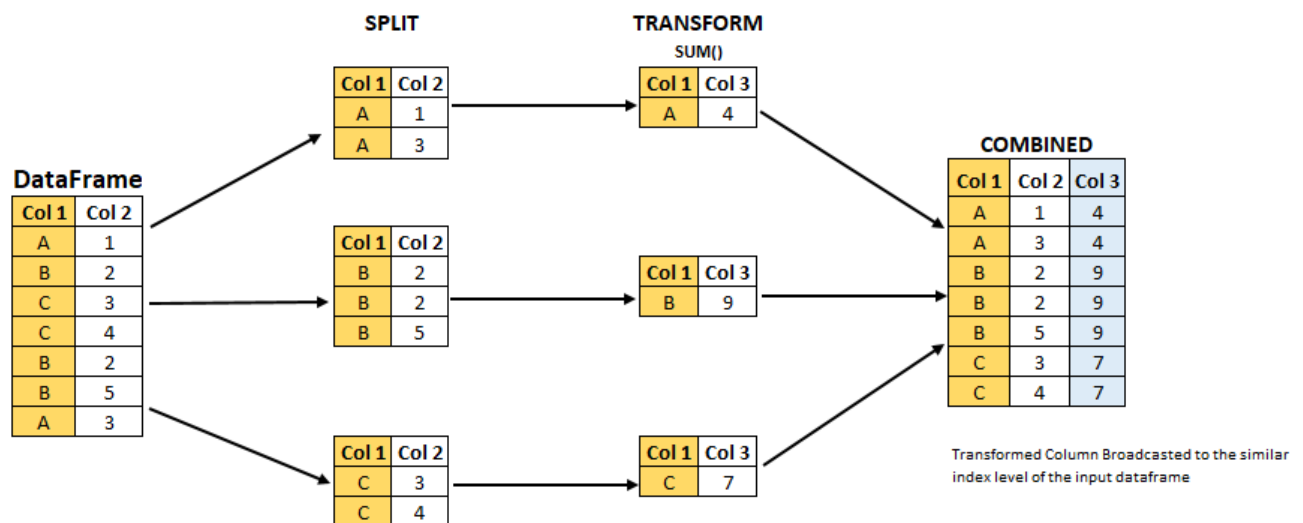
[Skip to content](#)

Common workflows

It is quite common to compute stratified statistics of different groups to produce descriptive statistics. This is commonly achieved through a `group-by` workflow, where the following happens:

- Splitting: data is partitioned into different groups based on some criterion
- Applying: applying a function/performing a calculation to each group
- Combining: assembling a dataframe (of potentially any size) with the results.

This type of workflow is represented below.



As an example, let us try to compute the total fare for each hour, split by payment type.

Pandas

Polars

```
#First let us extract the hour from the tpep_pickup_datetime column
df["hour"] = df['tpep_pickup_datetime'].dt.hour

hourly_fare = (
    df.groupby(['hour', 'payment_type'], observed=False)['fare_amount']
      .sum()
      .reset_index()
      .sort_values(['hour', 'payment_type'])
)
```

[Skip to content](#)

Output

	hour	payment_type	fare_amount
0	0	0	352227.86
1	0	1	1088201.12
2	0	2	156546.07
3	0	3	3537.91
4	0	4	3941.24
...
116	23	0	534063.76
117	23	1	1618143.37
118	23	2	219991.22
119	23	3	4765.54
120	23	4	3293.61

The `groupby` statement is used to stratify the `fare_amount` column by hour and payment type. Then the amounts per hour and type get summed and sorted according to time and payment type.

Idiomatic Polars

Polars introduces a few variations to dataset operations compared to the traditional Pandas approach. In particular, a domain-specific language (DSL) was developed, where *expressions* are written to represent dataset operations and *contexts* provide the environment where they produce a result.

Expressions

Let's say that we created a `trip_duration_sec` column in our NYC cab database and, given the `trip_distance` column, we want to compute the average speed. In Polars, this can be achieved with:

```
pl.col('trip_distance') / pl.col('trip_duration_sec')
```

This is a lazy representation of an operation we want to perform, which can be further manipulated or just printed. For it to actually produce data, a *context* is needed.

Contexts

The same Polars expression can produce different results depending on the context where it is used. Four common contexts include:

- `select`

Skip to content

- `filter`
- `group_by`

Both `select` and `with_columns` can produce new columns, which may be aggregations, combinations of other columns, or literals. The difference between the two is that `select` only includes the columns contained in its input expression, whereas `with_columns` returns a new dataframe which contains all the columns from the original dataframe and the new ones created by the expression. To exemplify, using our earlier example of computing the average speed during a trip, using `select` would yield a single column, whereas `with_columns` would return the original dataframe with an additional column called `trip distance`:

```
df.select(pl.col('trip_distance')/pl.col('trip_duration_sec')*3600)
shape: (3_475_226, 1)
```

trip_distance

f64
11.497006
11.764706
18.461538
5.60479
11.207547
...
13.68899
19.42398
9.879418
9.339901
12.781395

[Skip to content](#)

```
df.with_columns((pl.col('trip_distance')/pl.col('trip_duration_sec')*3600).alias("avg_speed_mph"))
shape: (3_475_226, 22)
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	...	Airport_fee	cbd_congestion_fee	trip_duration_sec	avg_speed_mph
i32	datetime[μs]	datetime[μs]	i64		f64	f64	i64	f64
1	2025-01-01 00:18:38	2025-01-01 00:26:59	1	...	0.0	0.0	501	11.6
1	2025-01-01 00:32:40	2025-01-01 00:35:13	1	...	0.0	0.0	153	11.6
1	2025-01-01 00:44:04	2025-01-01 00:46:01	1	...	0.0	0.0	117	18.8
2	2025-01-01 00:14:27	2025-01-01 00:20:01	3	...	0.0	0.0	334	5.6
2	2025-01-01 00:21:34	2025-01-01 00:25:06	3	...	0.0	0.0	212	11.7
...
2	2025-01-31 23:01:48	2025-01-31 23:16:29	null	...	null	0.75	881	13.6
2	2025-01-31 23:50:29	2025-02-01 00:17:27	null	...	null	0.75	1618	19.4
2	2025-01-31 23:26:59	2025-01-31 23:43:01	null	...	null	0.75	962	9.8
2	2025-01-31 23:14:34	2025-01-31 23:34:52	null	...	null	0.75	1218	9.3
2	2025-01-31 23:56:42	2025-02-01 00:07:27	null	...	null	0.0	645	12.5

Skip to content

ext filters the rows of a dataframe based on one (or more) expressions which lean, e.g.

```
df.filter(pl.col('avg_speed_mph') < 1)
shape: (104_410, 22)
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	...	Airport_fee	cbd_congestion_fee	trip_duration_sec	avg_speed_mph
i32	datetime[μs]	datetime[μs]	i64		f64	f64	i64	f64
2	2025-01-01 00:37:43	2025-01-01 00:37:53	1	...	0.0	0.0	10	0.0
2	2025-01-01 00:57:08	2025-01-01 00:57:16	3	...	0.0	0.0	8	0.0
1	2025-01-01 00:27:40	2025-01-01 00:59:30	1	...	0.0	0.0	1910	0.0
2	2025-01-01 00:56:49	2025-01-01 00:56:54	4	...	0.0	0.0	5	0.0
1	2025-01-01 00:42:42	2025-01-01 00:42:44	0	...	0.0	0.0	2	0.0
...
1	2025-01-31 23:59:17	2025-02-01 00:03:43	null	...	null	0.75	266	0.0
1	2025-01-31 23:17:38	2025-01-31 23:35:58	null	...	null	0.75	1100	0.0
2	2025-01-31 23:39:25	2025-01-31 23:42:06	null	...	null	0.0	161	0.0
1	2025-01-31 23:30:42	2025-01-31 23:31:06	null	...	null	0.75	24	0.0
1	2025-01-31 23:10:25	2025-01-31 23:36:21	null	...	null	0.75	1556	0.0

The `group_by` context behaves like its Pandas counterpart.

[Skip to content](#)

Transformations

A `join` operation combines columns from one or more dataframes into a new dataframe. There are different joining strategies, which influence how columns are combined and what rows are included in the final set. A common type is the *equi* join, where rows are matched by a key expression. Let us clarify this with an example. The `df` dataframe does not include specific coordinates for each pickup and drop-off, rather only a `PULocationID` and a `DOLocationID`. There is a `taxi_zones_xy.csv` file that contains, for each `LocationID`, the latitude (X) and longitude (Y) of each location, as well as the name of zone and borough:

```
lookup_df = pl.read_csv('taxi_zones_xy.csv', has_header=True)
lookup_df.head()
```

LocationID	X	Y	zone	borough
---	---	---	---	---
i64	f64	f64	str	str
1	-74.176786	40.689516	Newark Airport	EWB
2	-73.826126	40.625724	Jamaica Bay	Queens
3	-73.849479	40.865888	Allerton/Pelham Gardens	Bronx
4	-73.977023	40.724152	Alphabet City	Manhattan
5	-74.18993	40.55034	Arden Heights	Staten Island

This can be used to append these columns to the original `df` to have some form of geographical data as follows (e.g. for the `PULocationID`):

```
df = df.join(lookup_df, left_on='PULocationID', right_on='LocationID', how='left',
, suffix='_pickup')
```

In the line above, `left_on` is used to indicate the *key* in the original dataframe, `right_on` is used to specify the *key* in the `lookup_df` dataframe, `how=left` means that the columns from the second dataframe will be added to the first (and not the other way around) and `suffix` is what will be added to the names of the joined columns (i.e., `df` will contain columns called `X_pickup`, `Y_pickup`, `zone_pickup` and `borough_pickup`). More information on join operations can be found [here](#).

[Skip to content](#)

Exercises

Joining geographical data

We have already seen how to add actual latitude and longitude for the pickups. Now do the same for the drop-offs!

Solution

```
df = df.join(lookup_df, left_on='DOLocationID', right_on='LocationID', how='left',
            suffix='_dropoff')
```

[Skip to content](#)

🔥 Feature engineering: enriching the dataset

We want to understand a bit more of the traffic in the city by creating new features (i.e. columns), in particular:

- Split the pickup datetime into hour, minute, day of the week and month to indentify daily, weekly and monthly trends
- Compute the average speed as an indicator of congestion (low speed -> traffic jam)
- Stratify the trip distance and fare by zone to identify how expensive different zones are. Below is a skeleton of the code, where some lines have been blanked out for you to fill (marked with `TODO: ...`)

```
import polars as pl
raw_df = pl.read_parquet('yellow_tripdata_2025-01.parquet')
df = raw_df.with_columns([
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),
    #TODO: do this for the minute
    pl.col("tpep_pickup_datetime").dt.day_of_week().alias("pickup_dow"),    # Mon=0 ... Sun=6
    pl.col("tpep_pickup_datetime").dt.month().alias("pickup_month"),
    # Trip duration in seconds
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
        .dt.total_seconds()
        .alias("trip_duration_sec"),
])

df = df.with_column(
    #TODO: add expression for average velocity here
    .replace_nan(None)                # protect against div-by-zero
    .alias("avg_speed_mph")
)

# Compute per-pickup-zone statistics once
zone_stats = (
    df.groupby("PULocationID")
        .agg([
            pl.mean("fare_amount").alias("zone_avg_fare"),
            #TODO: do the same for the trip distance here
            pl.count().alias("zone_trip_cnt"),
        ])
        .rename({"PULocationID": "pickup_zone_id"})    # avoid name clash later
)

# Join those stats back onto the original rows
df = df.join(zone_stats, left_on="PULocationID", right_on="pickup_zone_id", how="left")
```

While we haven't covered the `join` instruction earlier, its main role is to "spread" the `zone_stats` over all the rides in the original dataframe (i.e. write the `zone_avg_fare` on each ride in `df`). `join` has its roots in relational databases, where different tables can be merged based on a common column.

[Skip to content](#)

Solution

```
import polars as pl
raw_df = pl.read_parquet('yellow_tripdata_2025-01.parquet')
df = raw_df.with_columns([
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),
    pl.col("tpep_pickup_datetime").dt.minute().alias("pickup_minute"),
    pl.col("tpep_pickup_datetime").dt.day_of_week().alias("pickup_dow"), # Mon=0 ... Sun=6
    pl.col("tpep_pickup_datetime").dt.month().alias("pickup_month"),
    # Trip duration in seconds
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
    .dt.seconds()
    .alias("trip_duration_sec"),
])

df = df.with_column(
    (
        pl.col("trip_distance") /
        (pl.col("trip_duration_sec") / 3600) # seconds → hours
    )
    .replace_nan(None) # protect against div-by-zero
    .alias("avg_speed_mph")
)

# Compute per-pickup-zone statistics once
zone_stats = (
    df.groupby("PULocationID")
    .agg([
        pl.mean("fare_amount").alias("zone_avg_fare"),
        pl.mean("trip_distance").alias("zone_avg_dist"),
        pl.count().alias("zone_trip_cnt"),
    ])
    .rename({"PULocationID": "pickup_zone_id"}) # avoid name clash later
)

# Join those stats back onto the original rows
df = df.join(zone_stats, left_on="PULocationID", right_on="pickup_zone_id", how="left")
```

[Skip to content](#)

More feature engineering!

Similarly to the exercise above, define the following features in the data:

- `pickup_hour` extracted from `tpep_pickup_time`
- `is_weekend`, a Boolean value for each trip
- `avg_speed_mph`, exactly as before
- `tip_to_fare_ratio`, dividing the tip amount by the total fare. Be careful with division by 0
- `fare_per_mile`, dividing the total fare by the distance
- `dist_per_passenger`, the average distance travelled by each passenger (sum of all trip distances divided by number of trips)
- `speed_per_pickup_area`, the average velocity stratified by pickup location
- `dropoff_trip_count`, count of trips stratified per dropoff location

[Skip to content](#)

Solution

```
import polars as pl
raw_df = pl.read_parquet("yellow_tripdata_2025-01.parquet")
df = raw_df.with_columns([
    # 1. pickup_hour
    pl.col("tpep_pickup_datetime").dt.hour().alias("pickup_hour"),

    # 2. is_weekend (Sat=5, Sun=6)
    pl.col("tpep_pickup_datetime")
        .dt.day_of_week()
        .is_in([5, 6])
        .alias("is_weekend"),

    # 3. trip_duration_sec
    (pl.col("tpep_dropoff_datetime") - pl.col("tpep_pickup_datetime"))
        .dt.seconds()
        .alias("trip_duration_sec"),

    # 4. avg_speed_mph
    (
        pl.col("trip_distance") /
        (pl.col("trip_duration_sec") / 3600)
    )
    .replace_nan(None) # protect against div-by-zero
    .alias("avg_speed_mph"),

    # 5. tip_to_fare_ratio
    (pl.col("tip_amount") / pl.col("fare_amount"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("tip_to_fare_ratio"),

    # 6. fare_per_mile
    (pl.col("fare_amount") / pl.col("trip_distance"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("fare_per_mile"),

    # 7. dist_per_passenger
    (pl.col("trip_distance") / pl.col("passenger_count"))
        .replace_inf(None)
        .replace_nan(None)
        .alias("dist_per_passenger"),
])

dropoff_stats = (
    df.groupby("DOLocationID")
        .agg([
            pl.mean("avg_speed_mph").alias("dropoff_avg_speed"),
            pl.count().alias("dropoff_trip_cnt"),
        ])
        .rename({"DOLocationID": "dropoff_zone_id"}) # avoid name clash later
)

# zone stats back onto every row
df = df.with_columns(dropoff_stats, left_on="DOLocationID", right_on="dropoff_zone_id", how="left")
```

[Skip to content](#)

Summary

We have seen how to deal with common workflows in both Pandas and Polars, starting from basic tasks like opening a dataset and inspecting it to performing split-apply-combine pipelines. We have seen how to use Polars to manipulate datasets and perform some basic feature engineering.

⌵ Keypoints

- Dataframes are combinations of series
- Both Pandas and Polars can be used to manipulate them
- The expression API in Polars allows to perform advanced operations with a simple DSL.

See also

There is a lot more to Polars than what we covered in this short introduction. For example, queries like the ones we introduced can be performed lazily, i.e. just declared and then run all together, giving the backend a chance to optimise them. This can dramatically improve performance in the case of complex queries. For this and a lot more, we refer you to the official [documentation](#).

Storage & serialisation backends

❓ Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

🔍 Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

[Skip to content](#)

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")  
# This uses the default highlighting language
```

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

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

[Skip to content](#)

Visualisations and dashboards

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

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

Exercises: description

Skip to content L: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Benchmarking

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

[Skip to content](#)

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")  
# This uses the default highlighting language
```

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

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

[Skip to content](#)

Multithreading

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")
# This uses the default highlighting language
```

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

Exercises: description

Skip to content

L: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Dask

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- ...

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

Skip to [content](#)

Discussion

Discuss the following.

- A discussion section
- Another discussion topic

Section

```
print("hello world")  
# This uses the default highlighting language
```

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

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- ...

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Skip to content **reference**

Instructor's guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

What to expect from this course

Discussion

How large are the datasets you are working with?

Both for classical machine/deep learning and (generative) AI, the amount of data needed to train ever-growing models is becoming bigger and bigger. Moreover, great strides in both hardware and software development for high performance computing (HPC) applications allow for large scale computations that were not possible before. This course focuses on high performance data analytics (HPDA). The data can come from simulations or experiments (or just generally available datasets), and the goal is to pre-process, analyse and visualise it. The lesson introduces some of the modern Python stack for data analytics, dealing with packages such as Pandas, Polars, multithreading and Dask, as well as Streamlit for large-scale data visualisations.

Learning outcomes

This lesson provides a broad overview of methods to work with large datasets using tools and libraries from the Python ecosystem. Since this field is fairly extensive, we will try to expose just enough details on each topic for you to get a good idea of the picture and an understanding of of tools and libraries will work well for your particular use case.

[Skip to content](#)

Specifically, this lesson covers:

- Tools for efficiently storing data and writing/reading it to/from disk
- Interfacing with databases and object storage solutions
- Main libraries to work with arrays and tabular data
- Performance monitoring and benchmarking
- Workload parallelisation: threads and Dask

See also

 Credit

Don't forget to check out additional course materials from the [Data carpentry](#).

[Skip to content](#)

License

CC BY-SA for media and pedagogical material

Copyright © 2025 XXX. This material is released by XXX under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0).

Canonical URL: <https://creativecommons.org/licenses/by-sa/4.0/>

[See the legal code](#)

You are free to

1. **Share** — copy and redistribute the material in any medium or format for any purpose, even commercially.
2. **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.
3. The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

1. **Attribution** — You must give [appropriate credit](#) , provide a link to the license, and [indicate if changes were made](#) . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
2. **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
3. **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable [exception or limitation](#) .

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as [publicity, privacy, or moral rights](#) may limit how you use the material.

This deed highlights only some of the key features and terms of the actual license. It is not a license and has no legal value. You should carefully review all of the terms and conditions of the actual license before using the licensed material.

[Skip to content](#)

⚠ MIT for source code and code snippets

MIT License

Copyright (c) 2025, ENCCS project, The ENCCS TEAM

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



Copyright © 2025, ENCCS, The ENCCS TEAM
Made with [Sphinx](#) and [@pradyunsg's Furo](#)