battery_data_analytics_with_python

August 6, 2024

1 Getting started with Python

1.1 Setup Google Colab for workshop

We will use Google Colab for our workshop.

Steps to setup Google Colab:

1. Mount Google Drive to Google Colab and give permissions to all boxes;
2. Go to this git repos and download the code as ZIP: https://github.com/meichinpang/python_workshop/tree/master
3. Unpack the zipped folder and upload the newly downloaded folder into your `Colab Notebooks` workspace in Google Drive;
4. Change the directory to the place where you save your `python_worshop-master` folder.

[1]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

[2]:
```
# # Change the directory to the place where you save your
# # python_worshop-master folder.
# import os

# os.chdir('/content/drive/MyDrive/Colab Notebooks/python_workshop-master')
# !ls
```

1.2 Python ecosystem

Python has many open-sourced libraries created and managed by different people in the community for various applications. For example:

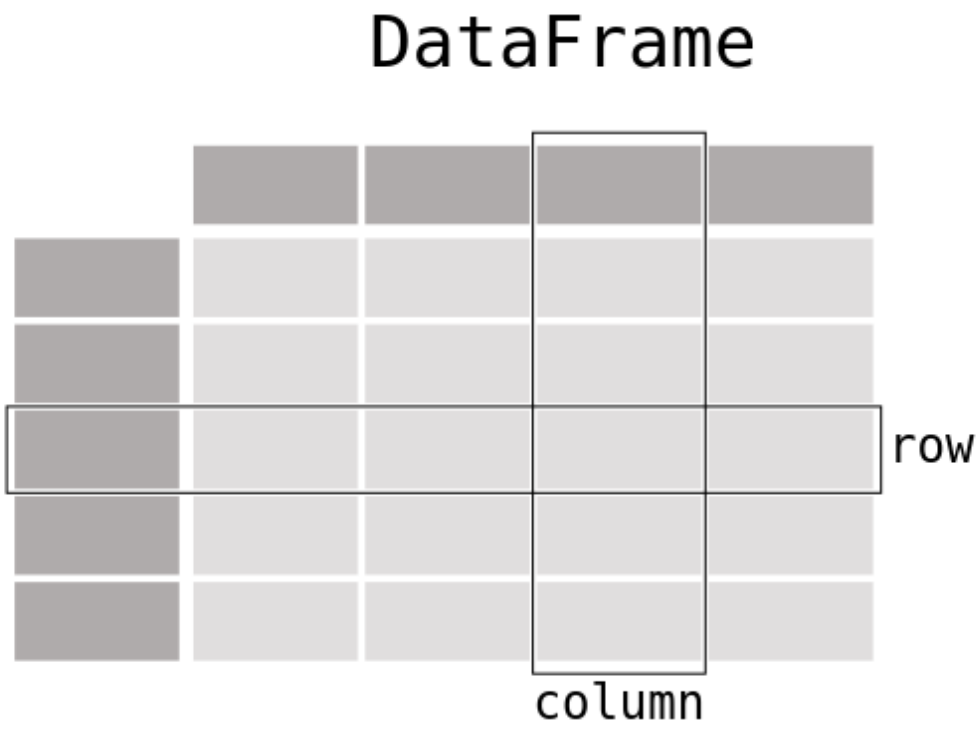
1. **Pandas** is developed for analysis of tabular data (*i.e.* data stored in spreadsheets or databases). Link: https://pandas.pydata.org/pandas-docs/stable/index.html
2. **Matplotlib** is developed for data visualization. Link: https://matplotlib.org/
3. **Numpy** is developed for numerical computing. Link: https://numpy.org/
4. **Scipy** is mainly developed for scientific computing & statistical calculations. Link: https://scipy.org/

As a starting point to learn how to use Python for data analysis and visualization, we will focus on pandas and matplotlib for this workshop.

2 Introduction to pandas

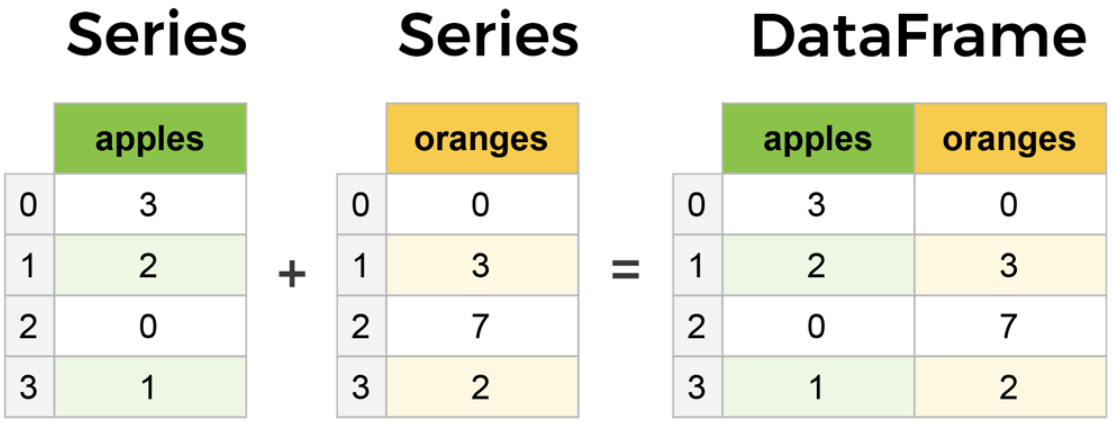
1. Tabular data with multiple rows and columns in pandas is known as `DataFrame`.
2. If the tabular data is one-dimensional, this data structure is denoted as `pandas.Series`.
3. `pandas.DataFrame` consists of multiple `pandas.Series`.
4. `pandas.Series` has different methods compared to `pandas.DataFrame`, it is important to distinguish whether a tabular data is one-dimensional or not.

[3]:
```
from IPython.display import display, Image

# image source:
# https://pandas.pydata.org/pandas-docs/stable/index.html
display(Image(
    filename="figures/dataframe.png",
    height=300,
    width=450))
```



[4]:
```
# image source:
# https://www.learndatasci.com/tutorials/
  ↪python-pandas-tutorial-complete-introduction-for-beginners/
display(Image(
    filename="figures/series_vs_df.png",
    height=300,
```

```
    width=600))
```



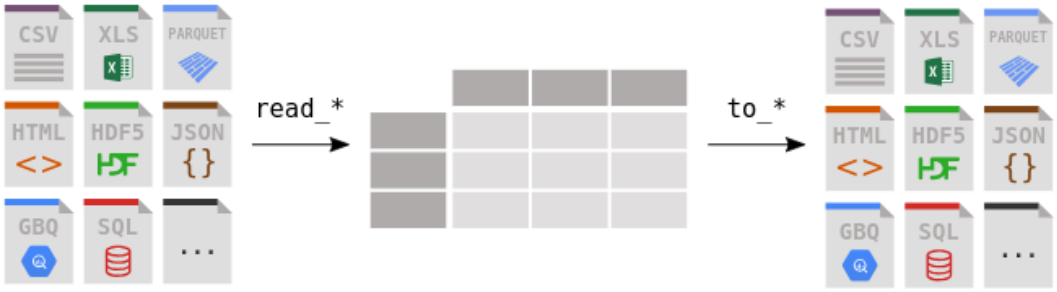
[5]:
```
import pandas as pd
```

Let's import the library pandas into our notebook: 1. The `import pandas` code tells Python to bring the pandas data analysis library into your notebook. 2. The `as pd` portion of the code then tells Python to give pandas the alias of `pd`, so that you can use pandas functions by simply typing `pd.function_name` instead of `pandas.function_name`.

2.1 How to import external dataset into our notebook?

1. Pandas can read data from different sources such as csv format, Excel format and databases.

[6]:
```
# image source:
# https://pandas.pydata.org/pandas-docs/stable/index.html

display(Image(
    filename="figures/pandas_read_and_write.png",
    height=250,
    width=800))
```



2.1.1 To import csv data

To import csv data, we can use the `pd.read_csv(filepath)` method:

If the csv file is stored inside a folder, first specify the `folder_name`, then the `csv_filename`.

Link: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

[7]:
```
df_csv_data = pd.read_csv("cycling_data/example_dataset_clean.csv")
df_csv_data
```

[7]:

	time	current	power	capacity	specific_capacity	voltage	\
0	0.01	-0.000079	0.000004	0.000000	0.000000	-0.050613	
1	60.01	-0.000001	0.000000	0.000000	0.000000	-0.052044	
2	120.01	-0.000003	0.000000	0.000000	0.000000	-0.053148	
3	180.01	-0.000003	0.000000	0.000000	0.000000	-0.054713	
4	240.01	-0.000002	0.000000	0.000000	0.000000	-0.055935	
…	…	…	…	…	…	…	
71404	350925.25	-1.026419	-4.310664	0.068311	30.495922	4.199712	
71405	350930.28	-0.974455	-4.092579	0.069701	31.116516	4.199866	
71406	350935.25	-0.929606	-3.904286	0.071023	31.706661	4.199935	
71407	350940.25	-0.885487	-3.719071	0.072281	32.268368	4.200029	
71408	350945.28	-0.845378	-3.550408	0.073481	32.803875	4.199786	

	cycle_time	step_time	cycle_index	step	mode	pattern_name
0	0.01	0.01	1.0	1.0	Rest	1.0
1	60.01	60.01	1.0	1.0	Rest	1.0
2	120.01	120.01	1.0	1.0	Rest	1.0
3	180.01	180.01	1.0	1.0	Rest	1.0
4	240.01	240.01	1.0	1.0	Rest	1.0
…	…	…	…	…	…	…
71404	220.01	220.01	312.0	1.0	Charge	5.0
71405	225.01	225.01	312.0	1.0	Charge	5.0
71406	230.01	230.01	312.0	1.0	Charge	5.0
71407	235.01	235.01	312.0	1.0	Charge	5.0
71408	240.01	240.01	312.0	1.0	Charge	5.0

[71409 rows x 12 columns]

2.1.2 To import excel data

We can achieve the same result using `pd.read_excel("filepath")`, however, when using import on your local machine (outside of Google Colab), you will also need install `openpyxl`:

- `conda install -c conda-forge openpyxl` (for conda environment)
- `pip install openpyxl` (for pip environment)

[8]:
```
df_excel_data = pd.read_excel('cycling_data/example_dataset_clean.xlsx')
df_excel_data
```

[8]:

	time	current	power	capacity	specific_capacity	voltage	\
0	0.01	-0.000079	0.000004	0.000000	0.000000	-0.050613	
1	60.01	-0.000001	0.000000	0.000000	0.000000	-0.052044	
2	120.01	-0.000003	0.000000	0.000000	0.000000	-0.053148	
3	180.01	-0.000003	0.000000	0.000000	0.000000	-0.054713	
4	240.01	-0.000002	0.000000	0.000000	0.000000	-0.055935	
…	…	…	…	…	…	…	
71404	350925.25	-1.026419	-4.310664	0.068311	30.495922	4.199712	
71405	350930.28	-0.974455	-4.092579	0.069701	31.116516	4.199866	
71406	350935.25	-0.929606	-3.904286	0.071023	31.706661	4.199935	
71407	350940.25	-0.885487	-3.719071	0.072281	32.268368	4.200029	
71408	350945.28	-0.845378	-3.550408	0.073481	32.803875	4.199786	

	cycle_time	step_time	cycle_index	step	mode	pattern_name
0	0.01	0.01	1.0	1.0	Rest	1.0
1	60.01	60.01	1.0	1.0	Rest	1.0
2	120.01	120.01	1.0	1.0	Rest	1.0
3	180.01	180.01	1.0	1.0	Rest	1.0
4	240.01	240.01	1.0	1.0	Rest	1.0
…	…	…	…	…	…	…	
71404	220.01	220.01	312.0	1.0	Charge	5.0
71405	225.01	225.01	312.0	1.0	Charge	5.0
71406	230.01	230.01	312.0	1.0	Charge	5.0
71407	235.01	235.01	312.0	1.0	Charge	5.0
71408	240.01	240.01	312.0	1.0	Charge	5.0

[71409 rows x 12 columns]

2.1.3 Exercise: Import external dataset

- Can you try to import the dataset `charge_discharge_data` from the folder `exercise_import_data`?

2.2 How to select only relevant columns?

We do not need all the columns for further analysis. We can use two methods to select only a subset of data: * We specify all the relevant columns we need or * We drop all the unwanted columns

Useful links for further reading: * https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html

2.2.1 Method 1: Select relevant columns

[9]:
```
# Method 1: select only relevant columns
# syntax:
# (1) df_name[["col_name1", "col_name2", "col_name3"]]
# (2) df_name.loc[:, ["col_name1", "col_name2", "col_name3"]]
df_selected_data = df_csv_data[["time", "current", "power",
```

```
    "capacity", "specific_capacity",
    "voltage", "cycle_time", "cycle_index"]]

df_selected_data
```

[9]:
```
             time   current     power  capacity  specific_capacity   voltage  \
0            0.01 -0.000079  0.000004  0.000000           0.000000 -0.050613
1           60.01 -0.000001  0.000000  0.000000           0.000000 -0.052044
2          120.01 -0.000003  0.000000  0.000000           0.000000 -0.053148
3          180.01 -0.000003  0.000000  0.000000           0.000000 -0.054713
4          240.01 -0.000002  0.000000  0.000000           0.000000 -0.055935
…             …       …       …       …                  …        …
71404  350925.25 -1.026419 -4.310664  0.068311          30.495922  4.199712
71405  350930.28 -0.974455 -4.092579  0.069701          31.116516  4.199866
71406  350935.25 -0.929606 -3.904286  0.071023          31.706661  4.199935
71407  350940.25 -0.885487 -3.719071  0.072281          32.268368  4.200029
71408  350945.28 -0.845378 -3.550408  0.073481          32.803875  4.199786

       cycle_time  cycle_index
0            0.01          1.0
1           60.01          1.0
2          120.01          1.0
3          180.01          1.0
4          240.01          1.0
…             …          …
71404      220.01        312.0
71405      225.01        312.0
71406      230.01        312.0
71407      235.01        312.0
71408      240.01        312.0

[71409 rows x 8 columns]
```

[10]:
```
# select all rows with the columns "time", "current", "power"
# dataframe.loc[row, column]
df_selected_data_using_loc = (df_csv_data
    .loc[:, ["time", "current", "power"]])
df_selected_data_using_loc
```

[10]:
```
             time   current     power
0            0.01 -0.000079  0.000004
1           60.01 -0.000001  0.000000
2          120.01 -0.000003  0.000000
3          180.01 -0.000003  0.000000
4          240.01 -0.000002  0.000000
…             …       …       …
71404  350925.25 -1.026419 -4.310664
```

```
71405  350930.28 -0.974455 -4.092579
71406  350935.25 -0.929606 -3.904286
71407  350940.25 -0.885487 -3.719071
71408  350945.28 -0.845378 -3.550408

[71409 rows x 3 columns]
```

2.2.2 Method 2: Drop unwanted columns

[11]:
```
# Method 2: we drop unwanted columns
# Let's say we want to drop the column "mode" and "pattern_name"
df_csv_data.drop(
    ['mode', 'pattern_name'], axis=1)
```

[11]:
```
            time   current     power  capacity  specific_capacity   voltage  \
0           0.01 -0.000079  0.000004  0.000000           0.000000 -0.050613
1          60.01 -0.000001  0.000000  0.000000           0.000000 -0.052044
2         120.01 -0.000003  0.000000  0.000000           0.000000 -0.053148
3         180.01 -0.000003  0.000000  0.000000           0.000000 -0.054713
4         240.01 -0.000002  0.000000  0.000000           0.000000 -0.055935
…            …       …         …         …                  …        …
71404  350925.25 -1.026419 -4.310664  0.068311          30.495922  4.199712
71405  350930.28 -0.974455 -4.092579  0.069701          31.116516  4.199866
71406  350935.25 -0.929606 -3.904286  0.071023          31.706661  4.199935
71407  350940.25 -0.885487 -3.719071  0.072281          32.268368  4.200029
71408  350945.28 -0.845378 -3.550408  0.073481          32.803875  4.199786

       cycle_time  step_time  cycle_index  step
0            0.01       0.01          1.0   1.0
1           60.01      60.01          1.0   1.0
2          120.01     120.01          1.0   1.0
3          180.01     180.01          1.0   1.0
4          240.01     240.01          1.0   1.0
…             …          …          …    …
71404      220.01     220.01        312.0   1.0
71405      225.01     225.01        312.0   1.0
71406      230.01     230.01        312.0   1.0
71407      235.01     235.01        312.0   1.0
71408      240.01     240.01        312.0   1.0

[71409 rows x 10 columns]
```

2.2.3 Exercise: Select a subset of data

Can you select only the columns `current`, `specific_capacity` and `voltage` data from the dataset?

2.3 Data cleaning

Depending on the quality of the raw dataset, data cleaning is often a complex yet important step. Simple data cleaning can be just about removing some empty rows present in the dataset.

Let's look at the `cycle_index` column. We can find out the min, max and unique cycle, if we do not know in advance how many cycles are there in the dataset.

[12]:
```
# We use the ``pandas.DataFrame.min()`` method
min_cycle_count = df_selected_data["cycle_index"].min()
print(f"Min cycle count: {min_cycle_count}")

# We use the ``pandas.DataFrame.max()`` method
max_cycle_count = df_selected_data["cycle_index"].max()
print(f"Max cycle count: {max_cycle_count}")

# We use the ``pandas.DataFrame.unique()`` method
unique_cycle_count = df_selected_data["cycle_index"].unique()
unique_cycle_count
```

```
Min cycle count: 1.0
Max cycle count: 312.0
```

[12]:
```
array([  1.,  nan,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,
        33.,  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,
        44.,  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,
        55.,  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,
        66.,  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,
        77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,
        88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,
        99., 100., 101., 102., 103., 104., 105., 106., 107., 108., 109.,
       110., 111., 112., 113., 114., 115., 116., 117., 118., 119., 120.,
       121., 122., 123., 124., 125., 126., 127., 128., 129., 130., 131.,
       132., 133., 134., 135., 136., 137., 138., 139., 140., 141., 142.,
       143., 144., 145., 146., 147., 148., 149., 150., 151., 152., 153.,
       154., 155., 156., 157., 158., 159., 160., 161., 162., 163., 164.,
       165., 166., 167., 168., 169., 170., 171., 172., 173., 174., 175.,
       176., 177., 178., 179., 180., 181., 182., 183., 184., 185., 186.,
       187., 188., 189., 190., 191., 192., 193., 194., 195., 196., 197.,
       198., 199., 200., 201., 202., 203., 204., 205., 206., 207., 208.,
       209., 210., 211., 212., 213., 214., 215., 216., 217., 218., 219.,
       220., 221., 222., 223., 224., 225., 226., 227., 228., 229., 230.,
       231., 232., 233., 234., 235., 236., 237., 238., 239., 240., 241.,
       242., 243., 244., 245., 246., 247., 248., 249., 250., 251., 252.,
       253., 254., 255., 256., 257., 258., 259., 260., 261., 262., 263.,
       264., 265., 266., 267., 268., 269., 270., 271., 272., 273., 274.,
       275., 276., 277., 278., 279., 280., 281., 282., 283., 284., 285.,
```

```
       286., 287., 288., 289., 290., 291., 292., 293., 294., 295., 296.,
       297., 298., 299., 300., 301., 302., 303., 304., 305., 306., 307.,
       308., 309., 310., 311., 312.])
```

You would notice `nan` in the `unique_cycle_count`, if you open the csv file, the `nan` refer to those empty rows in the csv file. We can drop all the nans (empty rows) in our dataset to improve our data quality.

[13]:
```
# Drop all empty rows
df_selected_data_clean = df_selected_data.dropna(how='all')
df_selected_data_clean
```

[13]:

	time	current	power	capacity	specific_capacity	voltage	\
0	0.01	-0.000079	0.000004	0.000000	0.000000	-0.050613	
1	60.01	-0.000001	0.000000	0.000000	0.000000	-0.052044	
2	120.01	-0.000003	0.000000	0.000000	0.000000	-0.053148	
3	180.01	-0.000003	0.000000	0.000000	0.000000	-0.054713	
4	240.01	-0.000002	0.000000	0.000000	0.000000	-0.055935	
…	…	…	…	…	…	…	
71404	350925.25	-1.026419	-4.310664	0.068311	30.495922	4.199712	
71405	350930.28	-0.974455	-4.092579	0.069701	31.116516	4.199866	
71406	350935.25	-0.929606	-3.904286	0.071023	31.706661	4.199935	
71407	350940.25	-0.885487	-3.719071	0.072281	32.268368	4.200029	
71408	350945.28	-0.845378	-3.550408	0.073481	32.803875	4.199786	

	cycle_time	cycle_index
0	0.01	1.0
1	60.01	1.0
2	120.01	1.0
3	180.01	1.0
4	240.01	1.0
…	…	…
71404	220.01	312.0
71405	225.01	312.0
71406	230.01	312.0
71407	235.01	312.0
71408	240.01	312.0

[70787 rows x 8 columns]

By using just one line of code, we have removed all empty rows at once, instead of manual search and delete individual empty row in the csv file.

[14]:
```
total_empty_rows = len(df_selected_data) - len(df_selected_data_clean)
print(f"There are {total_empty_rows} empty rows in our dataset.")
```

```
There are 622 empty rows in our dataset.
```

[15]:
```
# There are no more nans in the dataset
# All empty rows have been removed
unique_cycle_count = df_selected_data_clean["cycle_index"].unique()
unique_cycle_count
```

[15]:
```
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,  22.,
        23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,  33.,
        34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.,
        45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,  55.,
        56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,  66.,
        67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,
        78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.,
        89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,  99.,
       100., 101., 102., 103., 104., 105., 106., 107., 108., 109., 110.,
       111., 112., 113., 114., 115., 116., 117., 118., 119., 120., 121.,
       122., 123., 124., 125., 126., 127., 128., 129., 130., 131., 132.,
       133., 134., 135., 136., 137., 138., 139., 140., 141., 142., 143.,
       144., 145., 146., 147., 148., 149., 150., 151., 152., 153., 154.,
       155., 156., 157., 158., 159., 160., 161., 162., 163., 164., 165.,
       166., 167., 168., 169., 170., 171., 172., 173., 174., 175., 176.,
       177., 178., 179., 180., 181., 182., 183., 184., 185., 186., 187.,
       188., 189., 190., 191., 192., 193., 194., 195., 196., 197., 198.,
       199., 200., 201., 202., 203., 204., 205., 206., 207., 208., 209.,
       210., 211., 212., 213., 214., 215., 216., 217., 218., 219., 220.,
       221., 222., 223., 224., 225., 226., 227., 228., 229., 230., 231.,
       232., 233., 234., 235., 236., 237., 238., 239., 240., 241., 242.,
       243., 244., 245., 246., 247., 248., 249., 250., 251., 252., 253.,
       254., 255., 256., 257., 258., 259., 260., 261., 262., 263., 264.,
       265., 266., 267., 268., 269., 270., 271., 272., 273., 274., 275.,
       276., 277., 278., 279., 280., 281., 282., 283., 284., 285., 286.,
       287., 288., 289., 290., 291., 292., 293., 294., 295., 296., 297.,
       298., 299., 300., 301., 302., 303., 304., 305., 306., 307., 308.,
       309., 310., 311., 312.])
```

3 Introduction to data visualization

There are many libraries that can be used for data visualizations. One popular python library for plotting data is `matplotlib`. Let's import matplotlib into our notebook environment:

[16]:
```
import matplotlib.pyplot as plt
import matplotlib
```

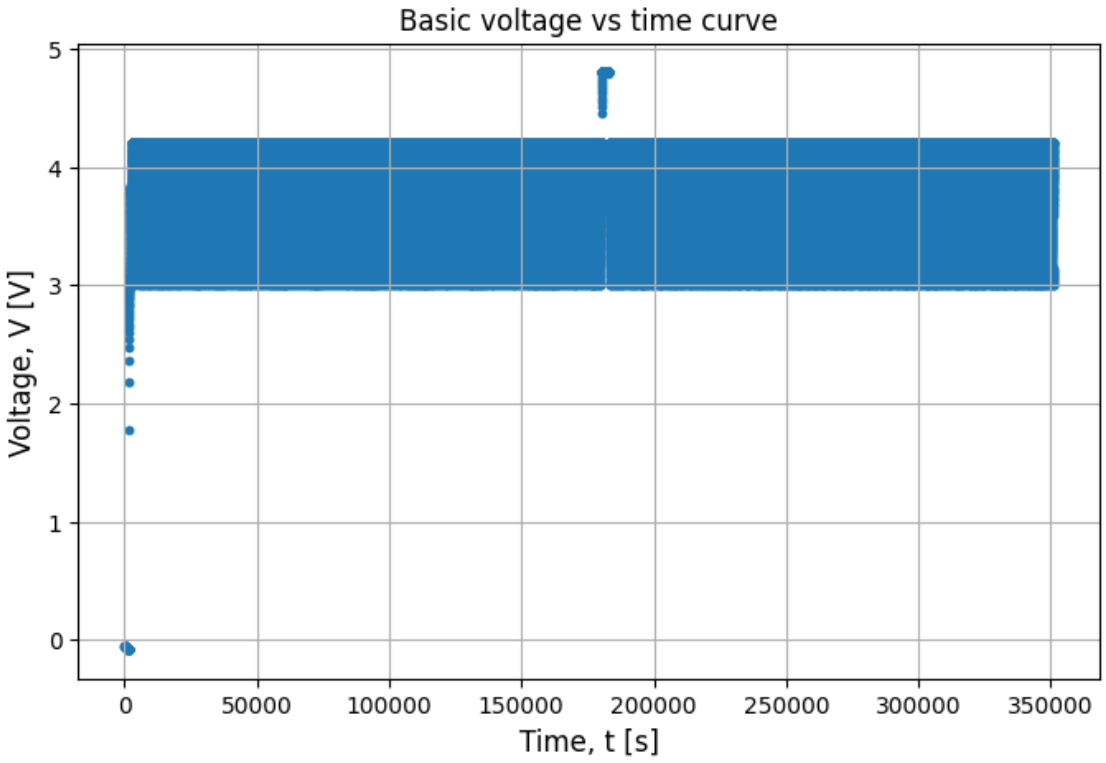
3.1 Basic plot of voltage vs time

Let's plot a simple voltage vs time curve.

What do you see?

- All the data points are cluttered together, there could be some outliers in the dataset, but no trend is clearly visible.
- This is because the dataset has recorded continuous cycle, and the data of all cycles are tightly packed into a single figure.

[17]:
```
# Create a placeholder for the figure and axes
# figsize=(width, height)
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for all cycles
ax.scatter(
    df_selected_data_clean["time"],    # x-axis data
    df_selected_data_clean["voltage"], # y-axis data
    s=10,                              # marker size
    marker="o")                        # marker shape

# create a grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Voltage, V [V]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Basic voltage vs time curve")

plt.show()
```



3.2 Plot one cycle

3.2.1 Scatterplot

Let's improve our cluttered plot to visualize only one cycle instead of multiple cycles.

We can filter the dataset to select only specific cycle. We can find out the cycle number from our `unique_cycle_count` variable.

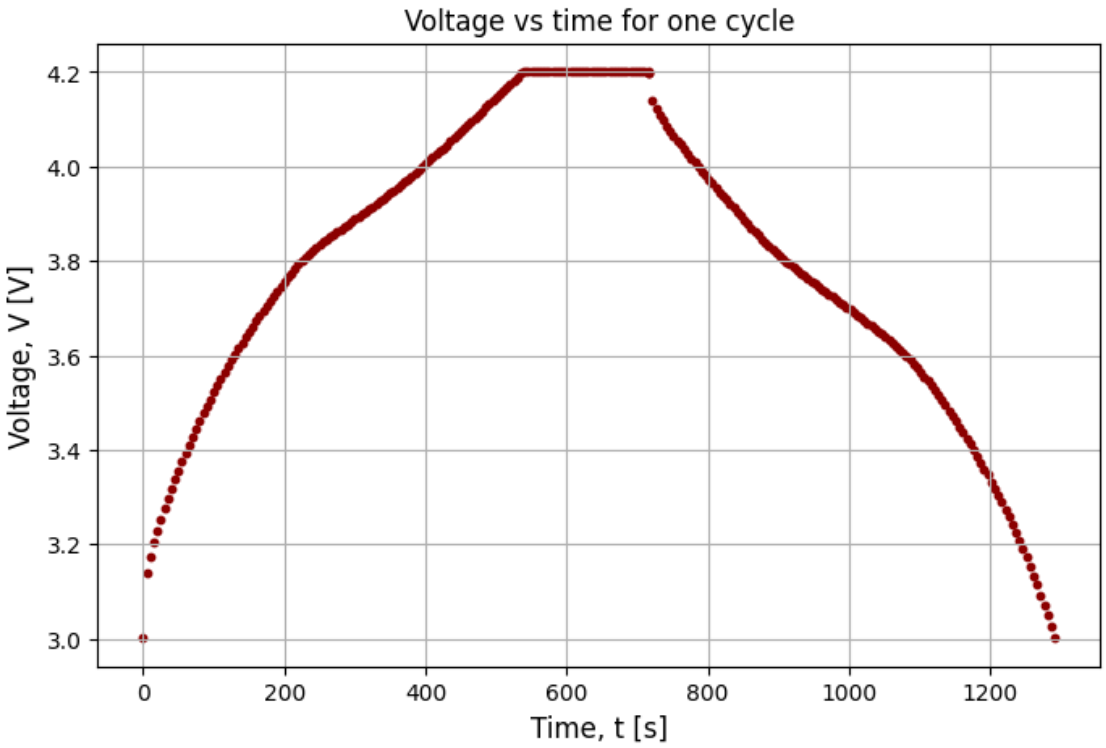
[18]:
```
df_one_cycle = df_selected_data_clean[
    df_selected_data_clean["cycle_index"] == 5]
df_one_cycle
```

[18]:

	time	current	power	capacity	specific_capacity	voltage	\
970	6411.47	1.075927	3.230154	0.000003	0.001388	3.002205	
971	6416.47	-1.119609	-3.515918	0.001558	0.695543	3.140309	
972	6421.47	-1.119492	-3.553971	0.003113	1.389689	3.174627	
973	6426.47	-1.119498	-3.585493	0.004668	2.083841	3.202767	
974	6431.47	-1.119583	-3.614670	0.006223	2.777995	3.228587	
…	…	…	…	…	…	…	
1227	7682.86	1.120956	3.467044	0.172817	77.150520	3.092935	
1228	7687.86	1.120705	3.442137	0.174374	77.845551	3.071404	
1229	7692.86	1.121044	3.418191	0.175931	78.540581	3.049113	

```
1230  7697.86  1.121005  3.391758  0.177488          79.235603  3.025641
1231  7702.77  1.120901  3.364389  0.179017          79.918137  3.001504

      cycle_time  cycle_index
970         0.01          5.0
971         5.01          5.0
972        10.01          5.0
973        15.01          5.0
974        20.01          5.0
…            …           …
1227     1271.40          5.0
1228     1276.40          5.0
1229     1281.40          5.0
1230     1286.40          5.0
1231     1291.31          5.0

[261 rows x 8 columns]
```

[19]:
```python
# figsize=(width, height)
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for one cycle
ax.scatter(
    df_one_cycle["cycle_time"],  # x-axis data
    df_one_cycle["voltage"],     # y-axis data
    s=10,                        # markersize
    c="darkred",                 # color
    marker="o")                  # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Voltage, V [V]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Voltage vs time for one cycle")

plt.show()
```

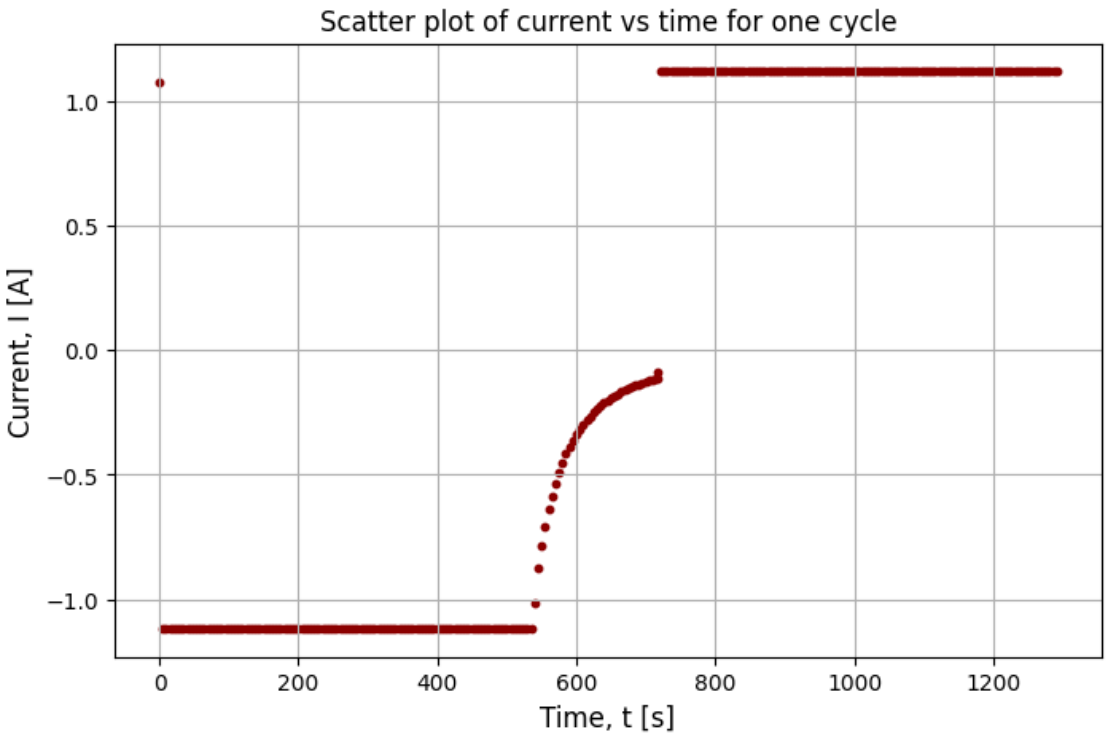


3.2.2 Lineplot

- `Scatterplot` only plot data in the region with data: useful to highlight sparse and dense data region.
- But what if we want to create lines to join different data region? We can use `lineplot` for this purpose.

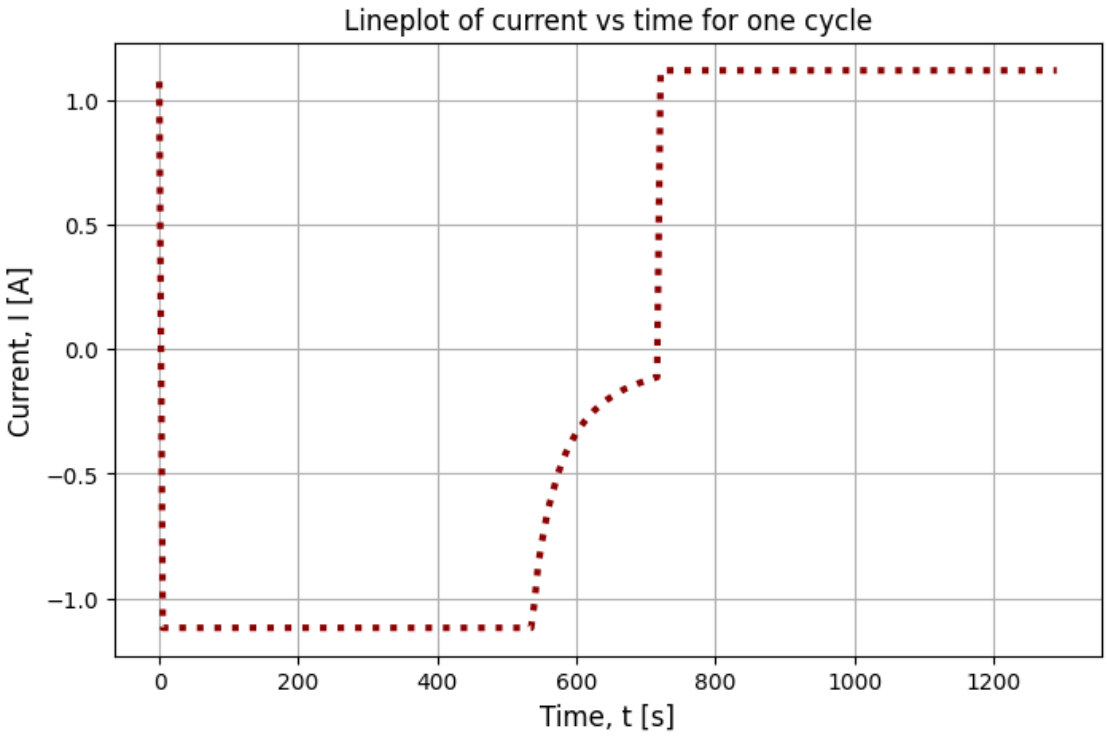
[20]:
```
# figsize=(width, height)
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for one cycle
ax.scatter(
    df_one_cycle["cycle_time"],  # x-axis data
    df_one_cycle["current"],     # y-axis data
    s=10,                        # markersize
    c="darkred",                 # color
    marker="o")                  # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
```

```
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Current, I [A]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Scatter plot of current vs time for one cycle")

plt.show()
```



[21]:
```
# figsize=(width, height)
fig, ax = plt.subplots(figsize=(8,5))

# lineplot
ax.plot(
    df_one_cycle["cycle_time"],  # x-axis data
    df_one_cycle["current"],     # y-axis data
    linewidth=3,                 # How thick should the line be?
    linestyle=":",               # Which linestyle should I use?
    c="darkred",                 # color
```

```
)

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Current, I [A]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Lineplot of current vs time for one cycle")

plt.show()
```



3.2.3 Exercise: Customize scatterplot and lineplot for one cycle

Can you change

- the markershape (`v`), markersize (`s`) and color (`c`) of the scatterplot?
- the linestyle `linestyle` of the lineplot?

Link: https://matplotlib.org/stable/gallery/color/named_colors.html

[22]:
```
supported_marker_shape = ('.', 'o', 'v', '^', '<', '>', '8',
                          's', 'p', '*', 'h', 'H', 'D', 'd',
                          'P', 'X')

supported_line_style = ('-', '--', '-.', ':',
                        'None', ' ', '',
                        'solid', 'dashed', 'dashdot', 'dotted')
```

3.3 Plot multiple cycles

To plot multiple cycles of data, we can use the logical comparison operators:

- `eq` (equivalent to `==`) — equals to
- `ne` (equivalent to `!=`) — not equals to
- `le` (equivalent to `<=`) — less than or equals to
- `lt` (equivalent to `<`) — less than
- `ge` (equivalent to `>=`) — greater than or equals to
- `gt` (equivalent to `>`) — greater than

[23]:
```
# Method 1: Use characters-based operators
df_multiple_cycles = df_selected_data_clean[
    (df_selected_data_clean["cycle_index"].ge(5)) &
    (df_selected_data_clean["cycle_index"].lt(10))]
df_multiple_cycles
```

[23]:

	time	current	power	capacity	specific_capacity	voltage	\
970	6411.47	1.075927	3.230154	0.000003	0.001388	3.002205	
971	6416.47	-1.119609	-3.515918	0.001558	0.695543	3.140309	
972	6421.47	-1.119492	-3.553971	0.003113	1.389689	3.174627	
973	6426.47	-1.119498	-3.585493	0.004668	2.083841	3.202767	
974	6431.47	-1.119583	-3.614670	0.006223	2.777995	3.228587	
…	…	…	…	…	…	…	
2260	12750.20	1.120915	3.454049	0.169714	75.765259	3.081456	
2261	12755.20	1.120921	3.429208	0.171271	76.460312	3.059276	
2262	12760.20	1.121032	3.403732	0.172828	77.155342	3.036250	
2263	12765.20	1.121047	3.376398	0.174385	77.850365	3.011826	
2264	12767.35	1.120964	3.364145	0.175054	78.149239	3.001118	

	cycle_time	cycle_index
970	0.01	5.0
971	5.01	5.0
972	10.01	5.0
973	15.01	5.0
974	20.01	5.0

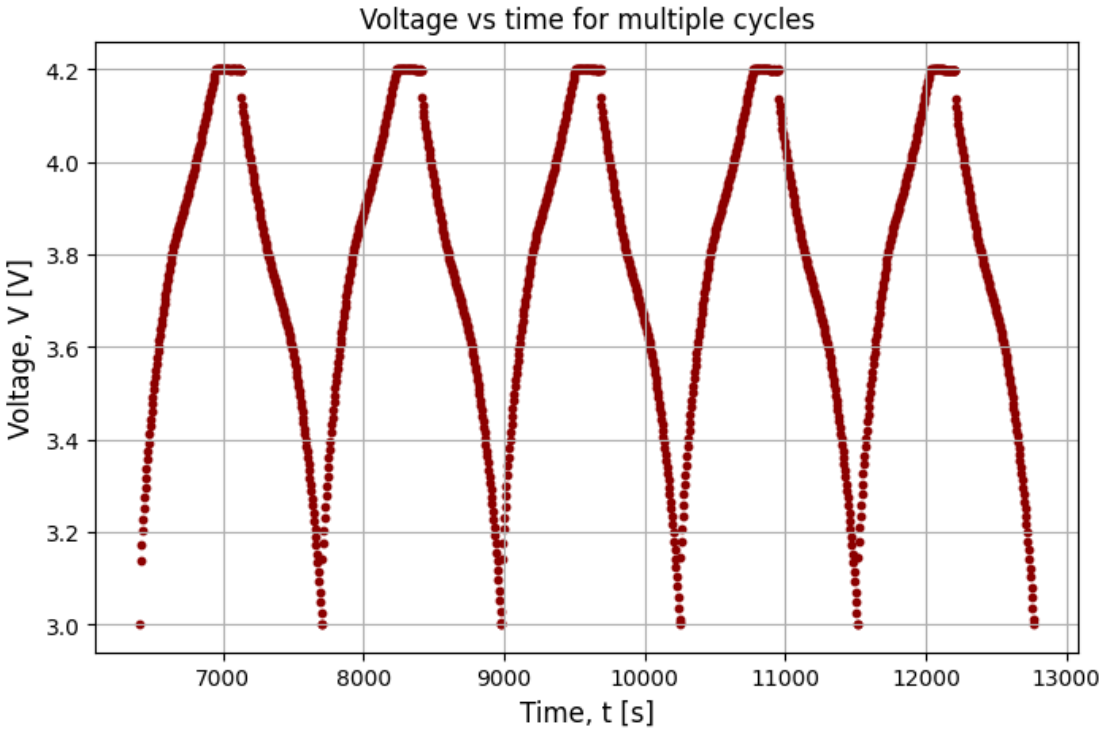
```
…          …          …
2260     1237.16          9.0
2261     1242.16          9.0
2262     1247.16          9.0
2263     1252.16          9.0
2264     1254.31          9.0

[1286 rows x 8 columns]
```

[24]:
```
# Method 2: Use symbols-based operators
df_multiple_cycles = df_selected_data_clean[
    (df_selected_data_clean["cycle_index"] >= (5)) &
    (df_selected_data_clean["cycle_index"] < 10)]
df_multiple_cycles
```

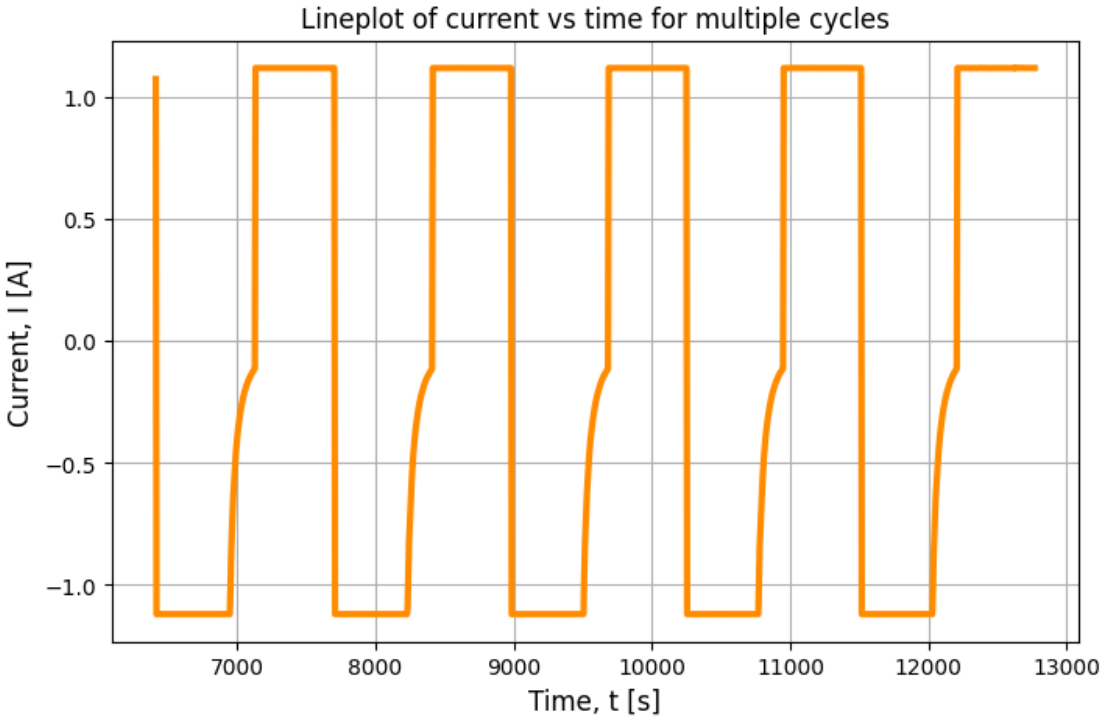
[24]:
```
          time   current     power  capacity  specific_capacity   voltage  \
970    6411.47  1.075927  3.230154  0.000003           0.001388  3.002205
971    6416.47 -1.119609 -3.515918  0.001558           0.695543  3.140309
972    6421.47 -1.119492 -3.553971  0.003113           1.389689  3.174627
973    6426.47 -1.119498 -3.585493  0.004668           2.083841  3.202767
974    6431.47 -1.119583 -3.614670  0.006223           2.777995  3.228587
…          …       …       …       …                  …         …
2260  12750.20  1.120915  3.454049  0.169714          75.765259  3.081456
2261  12755.20  1.120921  3.429208  0.171271          76.460312  3.059276
2262  12760.20  1.121032  3.403732  0.172828          77.155342  3.036250
2263  12765.20  1.121047  3.376398  0.174385          77.850365  3.011826
2264  12767.35  1.120964  3.364145  0.175054          78.149239  3.001118

      cycle_time  cycle_index
970         0.01          5.0
971         5.01          5.0
972        10.01          5.0
973        15.01          5.0
974        20.01          5.0
…          …          …
2260     1237.16          9.0
2261     1242.16          9.0
2262     1247.16          9.0
2263     1252.16          9.0
2264     1254.31          9.0

[1286 rows x 8 columns]
```

[25]:
```
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for multiple cycles
ax.scatter(
```

```
    df_multiple_cycles["time"],        # x-axis data
    df_multiple_cycles["voltage"],     # y-axis data
    s=10,                              # markersize
    c="darkred",                       # color
    marker="o")                        # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Voltage, V [V]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Voltage vs time for multiple cycles")

plt.show()
```



[26]:
```
fig, ax = plt.subplots(figsize=(8,5))

# lineplot for multiple cycles
ax.plot(
    df_multiple_cycles["time"],      # x-axis data
    df_multiple_cycles["current"],  # y-axis data
    linewidth=3,                    # How thick should the line be?
    linestyle="-",                  # Which linestyle should I use?
    c="darkorange",                 # color
)

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Time, t [s]",
    fontsize=12)
ax.set_ylabel(
    "Current, I [A]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Lineplot of current vs time for multiple cycles")

plt.show()
```



3.4 Plot voltage vs discharge capacity curve

- We have plotted one cycle and multiple cycles of voltage and current data so far, let's proceed with plotting voltage vs capacity curve like those curves we read on publications.
- **Note:**
  - Most battery dataset has a mix of charge, discharge and rest periods, if we are interested in the voltage vs discharge capacity curve, then we will need to filter out the charge and rest periods when plotting our voltage vs discharge capacity curve.
  - It is also important to check the current convention for each dataset: some people define charge current as positive and discharge current as negative, whereas some others define the current convention in the opposite direction.

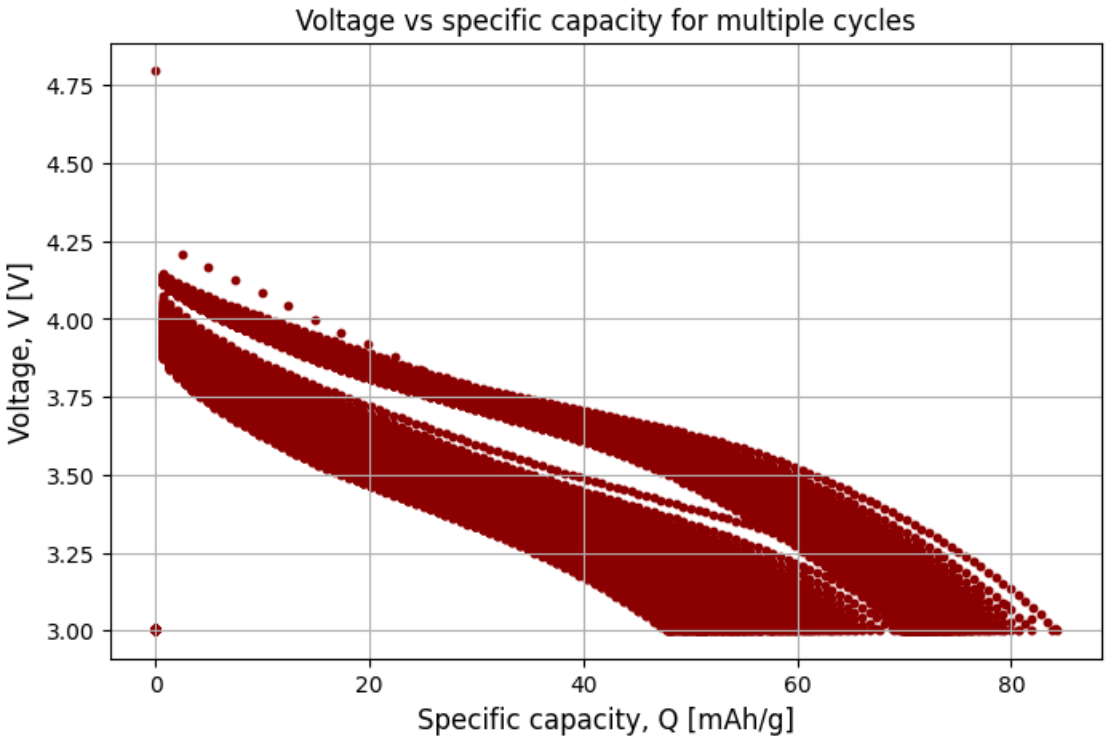
[27]:
```
df_discharge = df_selected_data_clean[
    (df_selected_data_clean["current"] > 0)]
df_discharge
```

[27]:

	time	current	power	capacity	specific_capacity	voltage	\
306	3156.99	1.121012	4.645783	0.001560	0.696459	4.144277	
307	3161.99	1.120874	4.628559	0.003117	1.391509	4.129419	
308	3166.99	1.121071	4.615182	0.004674	2.086572	4.116763	
309	3171.99	1.120907	4.601511	0.006231	2.781631	4.105168	
310	3176.99	1.120939	4.589427	0.007788	3.476704	4.094270	
…	…	…	…	…	…	…	
71355	350691.88	1.120914	3.414298	0.102758	45.874241	3.045994	

```
71356  350696.88  1.120873  3.395375  0.104315          46.569290  3.029223
71357  350701.88  1.120911  3.376383  0.105872          47.264351  3.012177
71358  350705.25  1.121098  3.364594  0.106925          47.734200  3.001158
71360  350705.28  1.076056  3.232744  0.000003           0.001388  3.004252

       cycle_time  cycle_index
306       1356.99          2.0
307       1361.99          2.0
308       1366.99          2.0
309       1371.99          2.0
310       1376.99          2.0
…            …           …
71355      965.94        311.0
71356      970.94        311.0
71357      975.94        311.0
71358      979.32        311.0
71360        0.01        312.0

[28798 rows x 8 columns]
```

[28]:
```python
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for multiple cycles
ax.scatter(
    df_discharge["specific_capacity"],  # x-axis data
    df_discharge["voltage"],            # y-axis data
    s=10,                               # markersize
    c="darkred",                        # color
    marker="o")                         # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Specific capacity, Q [mAh/g]",
    fontsize=12)
ax.set_ylabel(
    "Voltage, V [V]",
    fontsize=12)

# Define the title for the plot
ax.set_title("Voltage vs specific capacity for multiple cycles")

plt.show()
```



3.4.1 Creating color map and colorbar

- The above plot does look a bit ugly and unreadable. We can improve the plot quality by including colormap and colorbar.

Supported colormap:

https://matplotlib.org/stable/users/explain/colors/colormaps.html

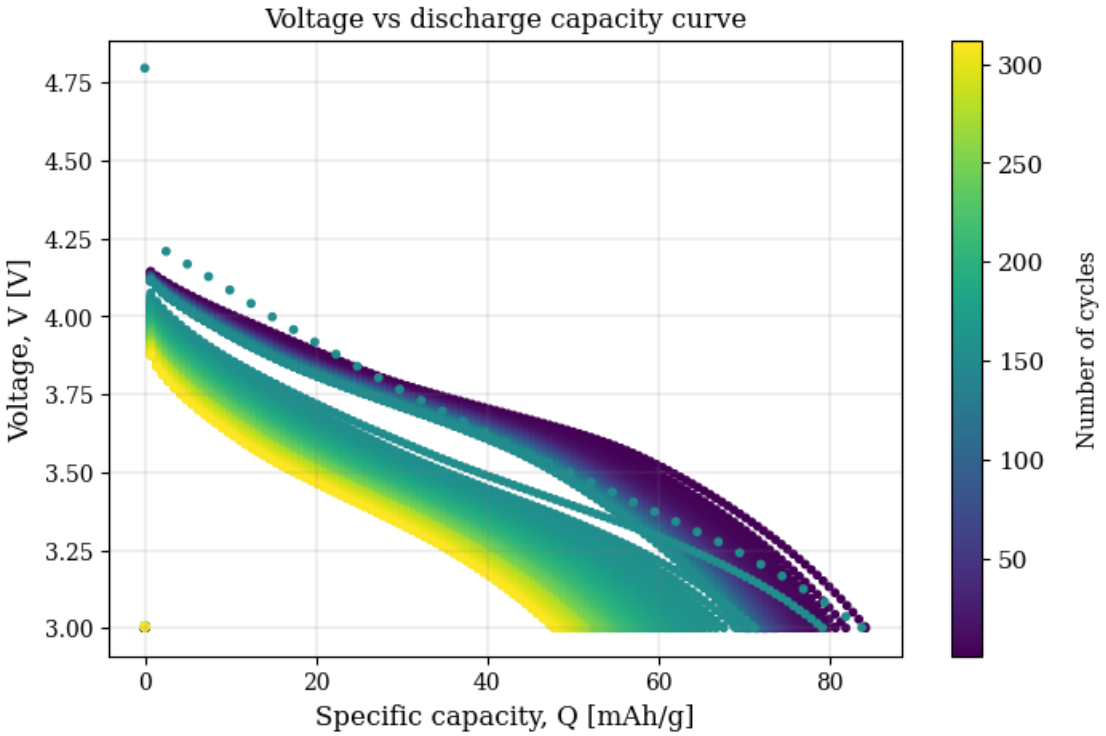
[29]:
```
import matplotlib

# define sequential colormap
color_map = matplotlib.colormaps.get_cmap("viridis")

# reverse the sequential colormap
reverse_color_map = matplotlib.colormaps.get_cmap("viridis_r")
```

[30]:
```
plt.rcParams["font.family"] = "Serif"
```

[31]:
```
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot with colormap ---------------------------------
ax.scatter(
    df_discharge["specific_capacity"],
```

```
    df_discharge["voltage"],
    s=10,
    marker="o",
    # map the color to the cycle_index
    c=df_discharge["cycle_index"],
    # map the min color value to min_cycle_count
    vmin=min_cycle_count,
    # map the max color value to max_cycle_count
    vmax=max_cycle_count,
    # Define the color_map here
    cmap=color_map)

# Create the colorbar ----------------------------------------
smap = plt.cm.ScalarMappable(
    cmap=color_map)

# Define the min and max values
# for mapping the colorbar
smap.set_clim(
    vmin=min_cycle_count,
    vmax=max_cycle_count)

cbar = fig.colorbar(
    smap,
    ax=ax)

cbar.ax.tick_params(labelsize=11)
cbar.ax.set_ylabel(
    'Number of cycles',
    rotation=90,
    labelpad=15,
    fontdict = {"size":10})

# Define x and y-axis labels ---------------------------------------
ax.set_xlabel(
    "Specific capacity, Q [mAh/g]",
    fontsize=12)
ax.set_ylabel(
    "Voltage, V [V]",
    fontsize=12)

ax.set_title("Voltage vs discharge capacity curve")

# Customize the grid style
ax.grid(color="grey", linestyle="-", linewidth=0.25, alpha=0.7)

plt.savefig("figures/my_first_colormap_discharge_curve.png")
```

```
plt.show()
```



3.4.2 Extras: Plotting with LaTeX

Note: Only if you care deeply about maths and have LaTeX installed in your system

LaTeX is a programming language used for professional representation of mathematical expressions and equations, but we can also use LaTeX to label our plots. Everything else remains the same compared to the previous plot, but we have made slight changes to the axis label:

```
ax.set_xlabel(
    r"Specific capacity, $Q$ [mAh/g]",
    fontsize=12)
ax.set_ylabel(
    r"Voltage, $V$ [V]",
    fontsize=12)

ax.set_title(r"Voltage vs discharge capacity curve")
```

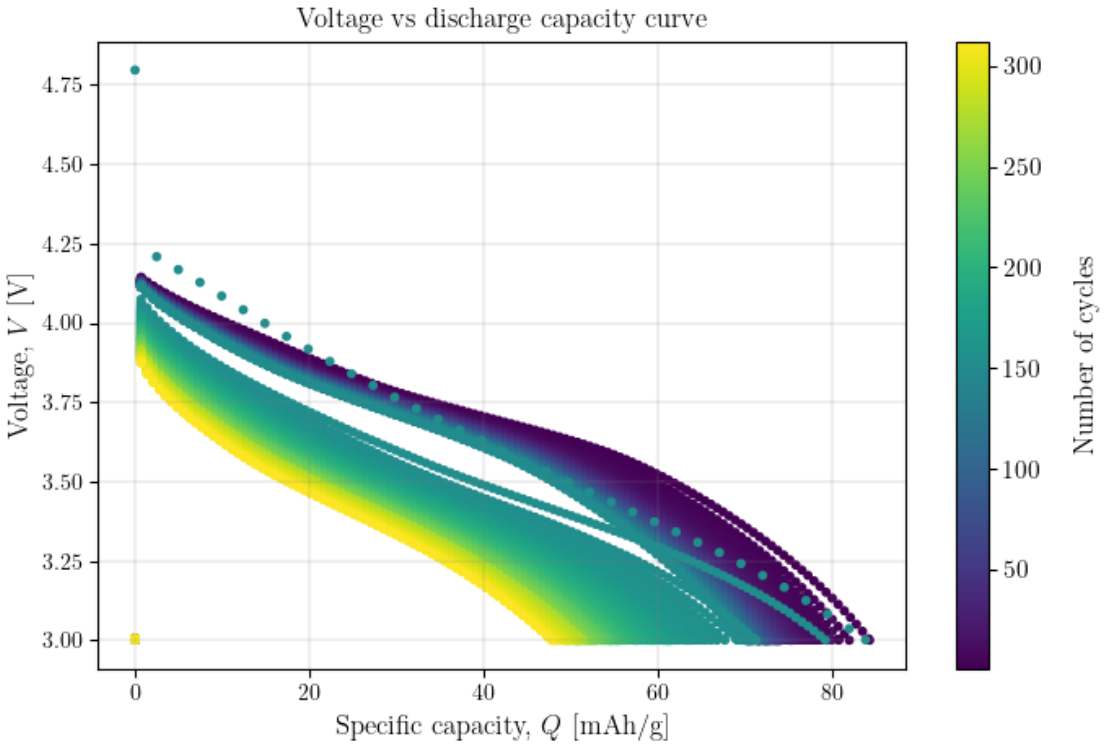
Notice that I have added `r` in front of my axis-label string to tell Python that this string should be interpreted according to LaTeX-formatting. I have also added `$` in front and at the end of my symbol to tell Python that this is a mathematical LaTeX symbol.

[32]:
```
# Tell Python to use LaTeX before plotting
matplotlib.rcParams["text.usetex"] = True
```

[33]:
```
# Install the necessary dependencies to run LaTeX in Google Colab
# ! sudo apt-get update
# ! sudo apt-get install texlive-latex-recommended
# ! sudo apt-get install dvipng texlive-latex-extra texlive-fonts-recommended
# ! sudo apt-get install cm-super
```

[34]:
```
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot with colormap ---------------------------------
ax.scatter(
    df_discharge["specific_capacity"],
    df_discharge["voltage"],
    s=10,
    marker="o",
    # map the color to the cycle_index
    c=df_discharge["cycle_index"],
    # map the min color value to min_cycle_count
    vmin=min_cycle_count,
    # map the max color value to max_cycle_count
    vmax=max_cycle_count,
    # Define the color_map here
    cmap=color_map)

# Create the colorbar ---------------------------------------
smap = plt.cm.ScalarMappable(
    cmap=color_map)

# Define the min and max values
# for mapping the colorbar
smap.set_clim(
    vmin=min_cycle_count,
    vmax=max_cycle_count)

cbar = fig.colorbar(
    smap,
    ax=ax)

cbar.ax.tick_params(labelsize=11)
cbar.ax.set_ylabel(
    'Number of cycles',
    rotation=90,
    labelpad=15,
    fontdict = {"size":12})

# Define x and y-axis labels --------------------------------------
ax.set_xlabel(
    r"Specific capacity, $Q$ [mAh/g]",
```

```
    fontsize=12)
ax.set_ylabel(
    r"Voltage, $V$ [V]",
    fontsize=12)

ax.set_title(r"Voltage vs discharge capacity curve")

# Customize the grid style
ax.grid(color="grey", linestyle="-", linewidth=0.25, alpha=0.7)

plt.savefig("figures/my_first_colormap_discharge_curve_with_latex.png")
plt.show()
```



3.4.3 Exercise: Colormap for charging profile

Can you create a `voltage` vs `specific_capacity` curve, but for charging? You can choose different colormap options from the referenced link.

3.5 Plot capacity retention curve

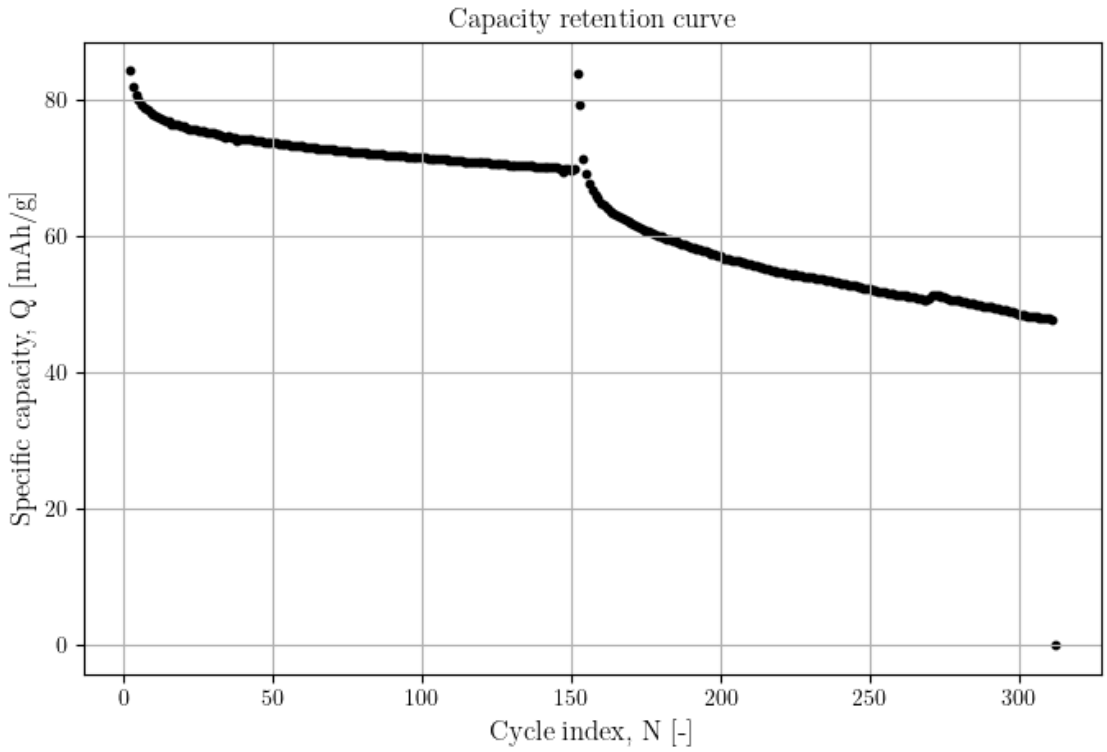
We have plotted the discharge capacity curve for every data point in our cell discharge dataset. But what if we want to plot the discharge capacity as a function of cycle index to evaluate how the capacity changes over time?

$$q_{\text{pct}} = \frac{q_{\text{dis}}}{q_{\text{dis,init}}} \times 100\%$$

[35]:
```
# Create a placeholder list to store our calculation
specific_capacity_list = []

# Find how many unique cycle we have in our discharge dataset
unique_discharge_cycle = df_discharge["cycle_index"].unique()

# Loop through every unique cycle
for cycle in unique_discharge_cycle:

    # Filter for each individual cycle
    df_one_cycle = df_discharge[df_discharge["cycle_index"] == cycle]

    # Find the max capacity per cycle
    max_capacity_per_cycle = df_one_cycle["specific_capacity"].max()

    # Store our max capacity value to the placeholder list we created
    specific_capacity_list.append(max_capacity_per_cycle)

# Calculate the capacity percentage according to the above equation
capacity_percentage = ((
    specific_capacity_list/specific_capacity_list[0])*100)
```

[36]:
```
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for specific capacity vs cycle number
ax.scatter(
    unique_discharge_cycle,     # x-axis data
    specific_capacity_list,     # y-axis data
    s=10,                       # markersize
    c="black",                  # color
    marker="o")                 # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
ax.set_xlabel(
    "Cycle index, N [-]",
    fontsize=12)
ax.set_ylabel(
    "Specific capacity, Q [mAh/g]",
```

```
    fontsize=12)

# Define the title for the plot
ax.set_title("Capacity retention curve")

plt.show()
```



[37]:
```
fig, ax = plt.subplots(figsize=(8,5))

# scatterplot for capacity retention percentage
ax.scatter(
    unique_discharge_cycle,  # x-axis data
    capacity_percentage,     # y-axis data
    s=10,                    # markersize
    c="black",               # color
    marker="o")              # markershape

# create grid for the plot
ax.grid()

# Define labels for x-axis and y-axis
# Recommended convention:
# Name, symbol [unit]
```

```
ax.set_xlabel(
    "Cycle index, N [-]",
    fontsize=12)
ax.set_ylabel(
    "Capacity retention ratio, Q [%]",
    fontsize=12)

# Create a straight line at 80% to mark the threshold
ax.axhline(
    y=80,
    color="red",
    linestyle="--")

# Define the title for the plot
ax.set_title("Capacity retention curve")

plt.savefig("figures/my_first_capacity_retention_curve.png")
plt.show()
```