Python Data Handling Workshop

T.M.Kinnear

1 Introduction and Overview

Near the Parkwood accommodation on the University of Kent Campus is the BEACON observatory. It is a 17 in telescope inside a 'clamshell' dome purchased with grant money from the 50th anniversary 'BEACON' awards. It is currently being used to do observational data acquisition for research, of publishable quality. We will not be working with the astronomical data, as it is subject to many data reduction and analysis types specific to astronomy. We will be examining the data from the accompanying weather station. It is important to keep good track of the weather at telescope locations. In addition to providing information relevant to whether it is imperative to close the dome to protect the telescope, it provides vital information regarding the quality of the observations, which contribute to an understanding of the uncertainties to apply to measurements. The weather station collects a wide variety of sensor data, and the gist of this workshop is that you will be provided with the entire backlog of data from the station (extending back to the 22nd of July 2015 at 10 AM). The intention is to undertake some processing and analysis of these data - building skills at accessing, presenting and evaluating arbitrary data. That the data happen to be about properties connected to the weather is not the object, this may be of specific interest to climate science, but of general interest for how interacting with data sets and being able to process and present those data using a variety of approaches can be applied to any data source in science.

2 Files

The github repository contains several files. This document, along with the relevant data files, and several example `.ipynb` notebook files containing example code to use as a basis. Some of the concepts for working with the data are explained here, but more elaborate examples of specific interaction with the data and operations which can be performed are in those notebook files. To read these you will need Jupyter Notebook, or alternatively `.html` exports of the notebooks are available to just browse the code itself.

3 The Data

The weather station data are recorded at five minute intervals (with occasional interruptions subject to power cuts) right up to today. Starting at 1000 on the 22nd of July 2015; up to 2024 this provides over 600 000 recorded data points (there have been outages of the system recently, meaning the data are not contiguous). Furthermore, there are at least 12 sensors. The 'at least' is because a new external temperature/humidity sensor was added in the first few years, which increases the original 12 to 14 sensors.

The original data are recorded on the source computer in the following format, where filenames have the format `MM_YYYY.mws`, where `MM` is the two-digit month number, and `YYYY` the four-digit year number. As a side-note, the following lines are the very first two lines of weather data recorded by the station when it first went into operation.

```
10:05:00,22.06.15,TE13.36,WU0.00,RT1.00,WK1.00,WR228.57,WT15.08,WG5.19,WS15.70,WD9.40,WV232.32,TI15.75,FI77.84,
10:10:00,22.06.15,TE13.42,WU0.00,RT0.00,WK1.00,WR213.05,WT15.48,WG9.14,WS16.93,WD10.34,WV228.70,TI15.83,FI77.62,
```

Col. header/ sensor code	Col. number	Description
`DateTime`	0	Date and Time in '`YY-MM-DD HH:MM:SS`' format
`TE`	1	External air temperature in degrees Celcius
`WU`	2	Cloud base in meters (unreliable)
`RT`	3	Rain detector, binary indicating rain/no rain with 1.00/0.00 (unreliable)
`WK`	4	Clouds, binary indicating clouds/no clouds with 1.00/0.00 (unreliable)
`WR`	5	Wind direction, in compass degrees
`WT`	6	WKS-Temp in degrees Celcius
`WG`	7	Wind speed in kilometers per hour
`WS`	8	Wind peak in kilometers per hour
`WD`	9	Wind average in kilometers per hour
`WV`	10	Main wind direction in compass degrees
`TI`	11	Internal air temperature in degrees Celcius
`FI`	12	Internal humidity in percentage
`TX`	13	Second external air temperature in degrees Celcius
`FX`	14	External humidity in percentage

Table 1: Sensor codes, column number and descriptions. Note that column number is in PYTHON notation, starting at zero inclusive.

Note that every line contains the time and date, followed by individually labelled values for each sensor. The two-character codes indicate which sensor, and the subsequent numerals its value at that time. For example, the sensor code `TE` is for the outside air temperature, and indicates this was 13.36 °C at 1000 on the 22nd of July 2015; then 13.42 °C at 1005. This format is used because the specific sensors, and their order, can change (as they did when the external temperature/humidity sensor pair was added). This makes the source data file (which are also stored month-by-month) very inconvenient to work with. To help with this, an alternative, pre-processed data file format has been created for you. The new format has fixed columns, rather than dynamic sequences; this means that column x will always be the value for sensor x, and will not move around. Any sensors which are not active throughout still have their own column, but placeholder 'null' values for dates/times at which they were not installed/operating. The columns have in-file headers, one line indicating the sensor code for each column, and preceded with a hash symbol. This makes the pre-processed data file appear as follows, showing the same initial two lines as the previous example.

```
#DateTime, TE, WU, RT, WK, WR, WT, WG, WS, WD, WV, TI, FI, TX, FX
2015-06-22 10:05:00,13.360,0.000,1.000,1.000,228.570,15.080,5.190,15.700,9.400,232.320,
↪  15.750,77.840,-99999.000,-99999.000
2015-06-22 10:10:00,13.420,0.000,0.000,1.000,213.050,15.480,9.140,16.930,10.340,228.700,
↪  15.830,77.620,-99999.000,-99999.000
```

Note that the lines above are wrapped at the locations indicated by the curved arrow, the actual files feature a single, continuous line for each date/time. There is one file `Weather_All.csv` which contains every data point throughout the recorded period. The sensor codes and columns in which they reside are as displayed in Table 1. Non-recorded values, or errors, are notated with `-99999.000`, evident in this example for the values associated with the two sensors which would not have been installed at this date (`TX` and `FX`). The two separate columns storing time and date have been merged into a single date-time column.

Some sensors have certain misbehaviours at various stages. One example is the wind direction. There is a prolonged period where the sensor was stuck in one orientation. Any investigation into wind direction should account for this.

4 Coding

4.1 Datetimes and NumPy

This section deals with the manner in which we can store the dates and times, as well as interacting with the data in Python using the Numerical Python 'NumPy' package.

The dataset you are presented with is slightly awkward due to the inclusion of date and time, which aren't easily stored in a useful fashion in native Python. Arguably, the most useful package for dealing with the dates and times, is the appropriately named `datetime` package. In essence, it provides several new datatypes storing dates and times in a way which 'appreciates' them in those contexts. The critical ones of interest are `datetime` which combines both separate `date` and `time` into a single type, and `timedelta`, which describes a difference in time, rather than an absolute time. The page for this package can be found at `docs.python.org/3/library/datetime.html`. However, below is presented an example of its use.

```
import datetime as dt
import numpy as np

def bytetodatetime(x):
    return dt.datetime.strptime(x.decode(), '%Y-%m-%d %H:%M:%S')

dDT = np.genfromtxt('filename.csv', delimiter=',', usecols=0,
↪   converters={0:bytetodatetime}, skip_header=1)
```

This code will extract *only* the first column from the file `filename.csv` (the `usecols=0` argument), and interpret the stored values as a `datetime` '`datetime`' datatype. To do this, it needs a converter function, the function `bytetodatetime` above the read-in line. Effectively, the converter is a function describing how to read the input bytes (`x`), convert it into a regular string (`x.decode()`), then parse it as a `datetime` object. The resulting array is the list of all combined dates and times in the correct sequence. An example of the type of information stored can be obtained with, for instance, the `print(dDT[20])` which would return the 21st entry from the dataset (Python is zero-indexed, the first item is 'offset' 0, second is 'offset' 1, etc.). This would appear as:

```
2015-06-22 11:45:00
```

The `datetime` data type is understood by NumPy and MatPlotLib so it can be used in certain calculations and in plotting.

A similar command as follows, can be used to import the first two columns, being the date/times and the '`TE`' external temperature data.

```
dDT, dTE = np.genfromtxt('filename.csv', delimiter=',', usecols=[0,1],
↪   converters={0:bytetodatetime}, skip_header=1, unpack=True)
```

Note the addition of `unpack=True`. This causes each column to be returned as a separate item which can be put in its own variable. Omitting this returns a single, 2D array containing every selected column and row.

Depending on what one wishes to do with which data, all columns could be read in with a line like the following.

```
dDT, dTE, dWU, dRT, dWK, dWR, dWT, dWG, dWS, dWD, dWV, dTI, dFI, dTX, dFX =
↪   np.genfromtxt(datapath+datafile, delimiter=',', converters={0:bytetodatetime},
↪   skip_header=1, unpack=True)
```

In this case, names have been given for all columns to the left of the assignment operator, and the `usecols=` argument has been omitted (it will assume all columns unless told otherwise).

4.2 Pandas

The prior example made direct use of the NumPy package. Another popular data handling package is Pandas. This section has an example of reading the data in using Pandas instead. There are two example code notebooks, one for each approach.

Pandas is designed from the ground up for data handling, and has an arguably more streamlined approach to reading in and working with datasets. It *is* possible to use structured arrays in NumPy in a similar way to the Pandas 'dataframe' we will see here, however. This workshop is not designed to be an exhausted coverage of all aspects of both packages.

A dataframe is (arguably) the main storage structure in Pandas. It stores all of the values themselves, and has an appreciation of the nature of each column against its index (a 'series') as well as operations one may wish to perform on the data. Some of the key points are fairly similar to the initial read-in in NumPy, as shown in the following:

```python
import pandas as pd
import numpy as np
import datetime as dt

def strtodatetime(x):
    return dt.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')

data = pd.read_csv('filename.csv',header=0,converters={0:strtodatetime},index_col=0)
data.columns = data.columns.str.replace(' ', '') #strip spaces from the column headings
data.columns = data.columns.str.replace('#', '') #strip hash symbol from the column
↪  headings
data[data < -99998] = np.nan #replace all missing data (with it's default -99999) with
↪  numpy nans, as this is understood as missing data by pandas
```

The last lines of this code are to tidy up some of the default presentation of the column names automatically drawn from the header, and fix the placeholder values in the data.

In this example I have used the comparison `data < -99998` to look for the invalid data. This could be written as `data == -99999`, which might seem more logical. However, the data are actually floating point values, and testing exact equalities on floating point values can be considered fragile for data representation and precision issues.

Note that this approach reads in *all* of the data, for every sensor. You are also able to select a subset of columns here, or modify the NumPy approach to read in all columns. It depends what you would like to do with the data.

4.3 Plotting the Data

Having read in at least two columns of data (the example in Section 4.1 was date-time and the main external air temperature sensor, we are ready to produce a simple plot. using the same read in commands as Section 4.1, we could plot temperature against date-time with the following commands.

```python
import matplotlib.pyplot as plt #this would generally be placed at the top of any overall
↪  script, along with the other imports

plt.plot(dDT,dTE)
plt.show()
```

The result of these commands is shown in Figure 1. It is evident that there is something wrong, as it is unlikely that the temperature outside of the dome at times descended to orders of magnitude below absolute zero...

The equivalent basic plot command for Pandas would be as follows.

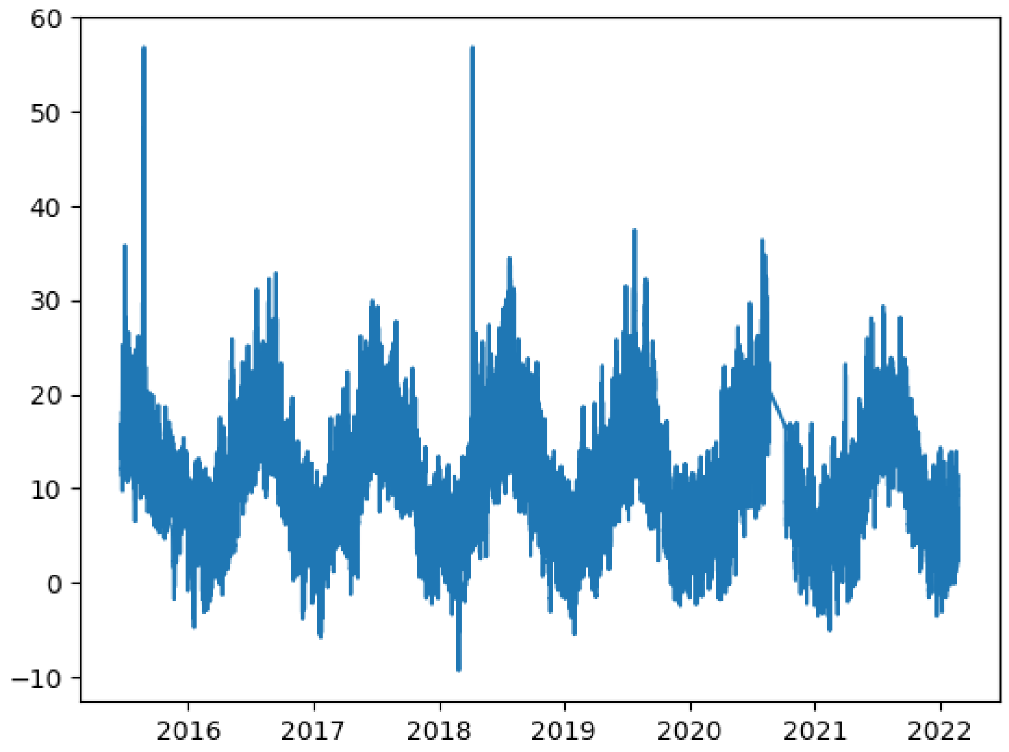

Figure 1: Plot of the TE sensor outside air temperature (OAT) in degrees Celcius against date/time for all data.

```
ax = data.plot(y="TE")
```

In this case, as we have set the datetime column as the indexing column, it is assumed to be the x axis. We could also specify it using an argument `x=` in the call to the `.plot()` method.

For the Pandas data, we have already filtered out all of the invalid (-99999) values; for NumPy, and for generally applying filters, there are a variety of approaches. Figure 2 looks at the data for the TX sensor, which was not installed until mid-2017. You should be able to see that the invalid data obscures all of the useful data in that first two years.

We can apply the same process as with Pandas to NumPy data in the following manner.

```
dTX[dTX < -99998] = np.nan
```

This indexes (the square brackets) the `dTX` data series, using a boolean selection of all points matching the criterion `dTX < 99998`; all data selected by this are being set to the NumPy Not A Number value. Whilst this doesn't fix it for all circumstances - some operations will still have an issue with the nan's being present - matplotlib understands `np.nan` as data not to present (in most contexts, there are some where this remains an issue).

Following this, the same plotting command as previously would produce the graph shown in Figure 3. Note that it has automatically scaled the horizontal axis to include only valid data.

There are a very wide variety of plot types available which might be of use. Consider looking at the MatPlotLib documentation generally, and particularly some of the examples - pages (two separate hyperlinks).

4.4 Plotting Features

The prior plots are excessively rudimentary and lack some critical plot features - decent axis labels, formatting and the like.

Features can be added to plots in a number of ways, a sample NumPy example is as follows.

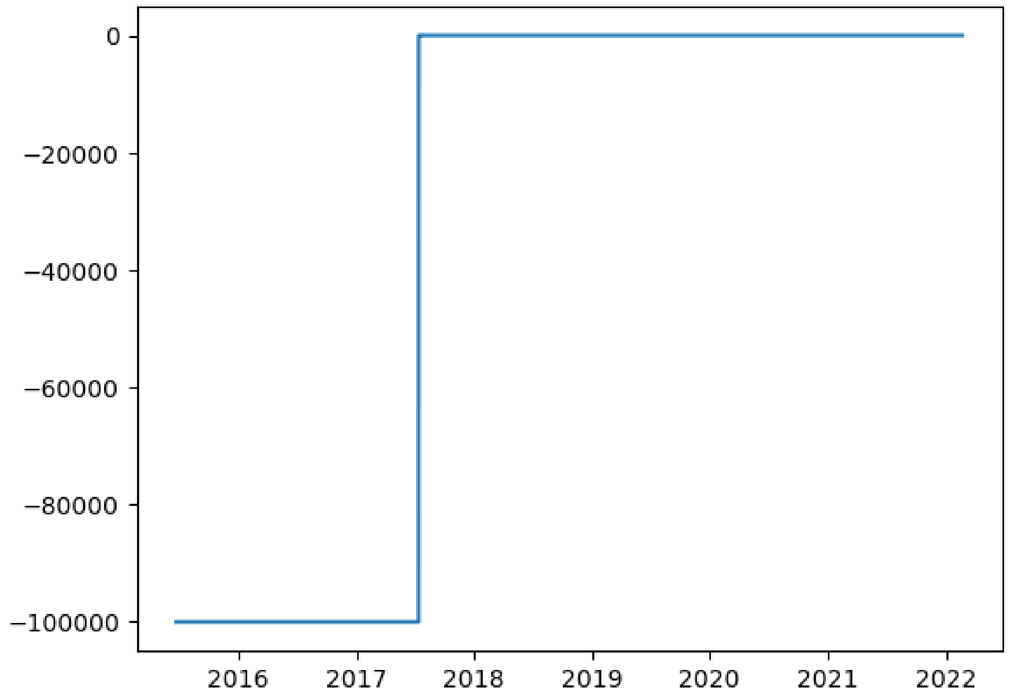

Figure 2: Plot of the TX sensor outside air temperature (OAT) in degrees Celcius against date/time for all data, featuring the data read in using NumPy with no invalid value filtering.

```python
plt.plot(dDT,dTE,'r,',label='TE Sensor')

plt.legend() #adds a legend (superfluous here, with only one data series)
plt.xlabel('Date')
plt.ylabel('OAT (°C)')
plt.gcf().autofmt_xdate() #attempts to automatically tidy up datetime axes
plt.show()
```

This code produces the graph seen in Figure 4. The same results can be achieved in Pandas natively as follows.

```python
ax = data.plot(y="TE",xlabel='Date',ylabel='OAT
↪  (°C)',linestyle='none',marker=',',color='red')
```

Though similar MatPlotLib commands to that with NumPy can also be used separately.

4.5 Filtering - NumPy

It is possible to apply multiple filters simultaneously. This is especially useful for the date and time data, where you might wish to select data only between a certain pair of dates/times. This can be accomplished with the NumPy functions `np.logical_and()` and `np.logical_or()`, for ‘and’ing and ‘or’ing values between arrays in one go. So, for the ‘and’ operation, to select a certain date range, we might use:

```python
date_selection = np.logical_and(dDT >= dt.datetime(2018,1,1), dDT < dt.datetime(2019,1,1))
```

it takes multiple arrays of `True`s and `False`s (such as those produced by existing styles of comparison operations), and in this case returns `True` at a position in the array is the values in both other arrays at that position are also `True`. In other words, the positions in the array of all dates in the above statement are selected if they are *both* greater than or equal to the first of January 2018, *and less* than

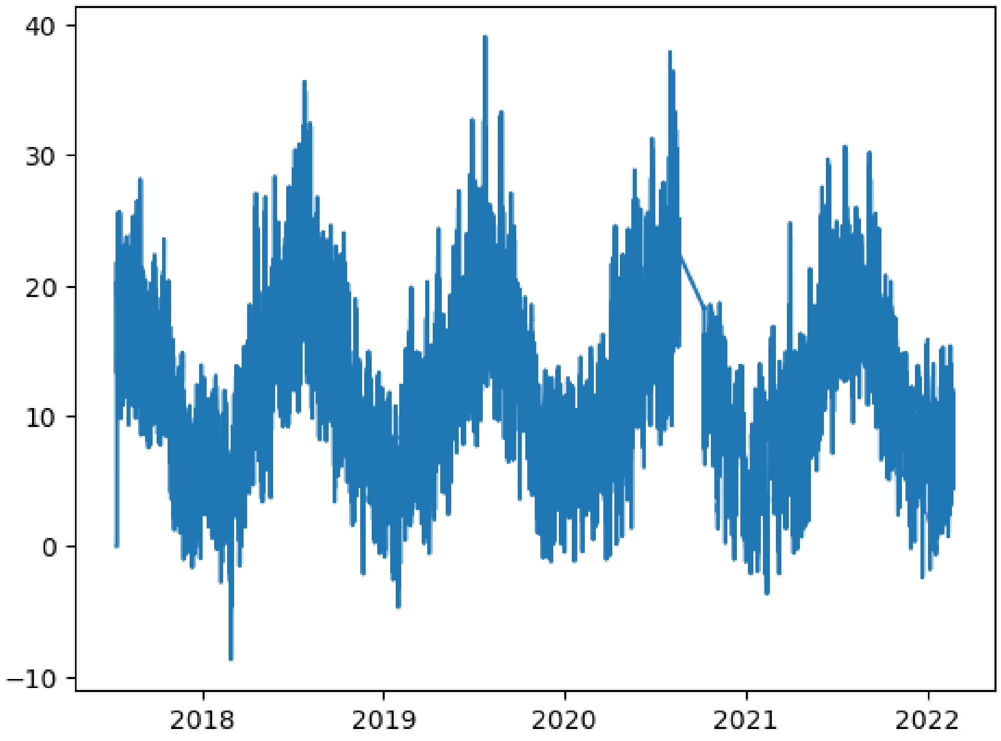

Figure 3: Plot of the TX sensor outside air temperature (OAT) in degrees Celcius against date/time for all data, featuring the data read in using NumPy *with* invalid value filtering.

the first of January 2019. In other words, the above filter references only dates and times which are in 2018 (is it is inclusive-exclusive at the bounds). This selection could be applied using a plot command with the data arrays indexed with that filter. So,

```
plt.plot(dDT[date_selection],dTE[date_selection],'r,',label='TE Sensor')
```

This would plot temperature against date, for only 2018. This plot is shown in Figure 5. Of note in this plot are some temperatures jumping up to almost 60 °C. These are most likely anomalies related to the sensor rather than representation of a physical phenomenon. This leads to the application of an additional filter to deselect these.

There are several ways to do this, and probably most ideally would be specific labeling of the individual points. However, a quick way is to deselect above a particular threshold temperature, which closely coincides with excising all of the problem data. The following is an example of this applied. In this case, `.reduce()` is essentially collapsing the `.logical_and()` operation across the arrays. In principle, this can also be done on a single line collapsing all of the sub-selections in one tuple (the inner brackets inside the `.reduce` call).

```
date_selection = np.logical_and(dDT >= dt.datetime(2018,1,1), dDT < dt.datetime(2019,1,1))
tem_selection = np.logical_and(dTE >= -20, dTE < 35)
all_selection = np.logical_and.reduce((date_selection,tem_selection))
```

4.6 Filtering - Pandas

There are some alternative approaches to achieving the results of the prior section. The main time selection depicted can be implemented very convenient as follows.

```
ax = data['2018-1-1':'2019-1-1'].plot(y="TE",xlabel='Date',ylabel='OAT
↪  (°C)',linestyle='none',marker=',',color='red')
```

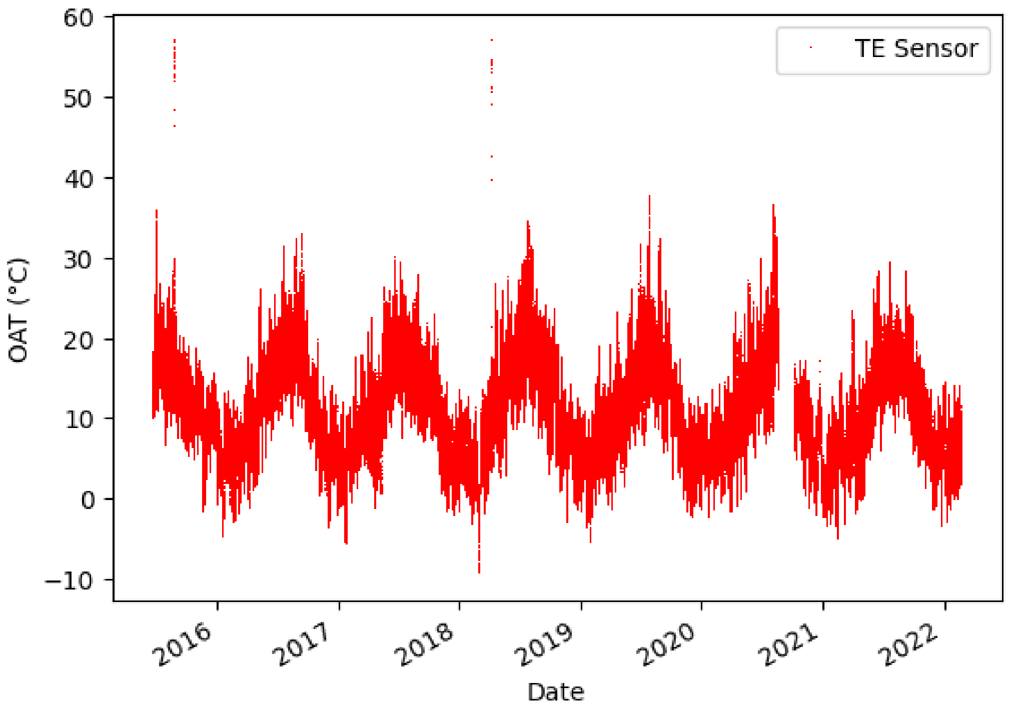

Figure 4: Plot of the TE sensor outside air temperature (OAT) in degrees Celcius against date/time for all data, with some additional formatting features. Compare with Figure 1.

Note that this leads directly to the plot in addition to filtering. The filter is applied based on the date/time column being the indexed column. As it 'understands' dates and times in this context, the strings representing the start and end times (separated by a colon) are suitable for that selection.

Extending to include the same filter for within a temperature range, we could have the following.

```
ax = data['2018-1-1':'2019-1-1'].query('TE < 35 and TE >
↪  -20').plot(y="TE",xlabel='Date',ylabel='OAT
↪  (°C)',linestyle='none',marker=',',color='red')
```

Note the addition of the `.query()` call after the data has been indexed (the square brackets to select the date) and before the dataframe is operated on by `.plot()`.

Ideas

The intention is to attempt to work with the data provided and see how it can be used, presented and interpreted to find interesting features. The following are some ideas of what to look for if you would like some guidance on how to proceed.

Some simple ideas for topics to investigate:

- The most common wind direction, and the most common wind direction weighted by wind speed/gust speed. Does this vary over times of day/time of year? Consider the use of histograms along with lines/scatter plots for variation over time.
- Profile of temperature over the day. How does this profile change with time of year? Line scatter plots could be used, or histograms across temperature to show common temperatures. When are the hottest/coldest temperatures, either across the day or across the year.
- Whether any correlation exists between wind speed and temperature. Whether the coldest/hottest temperatures are associated with fast/slow winds or particular wind directions.
- The relationship between temperature and humidity. Is it linear? Can it be fit to/characterised?

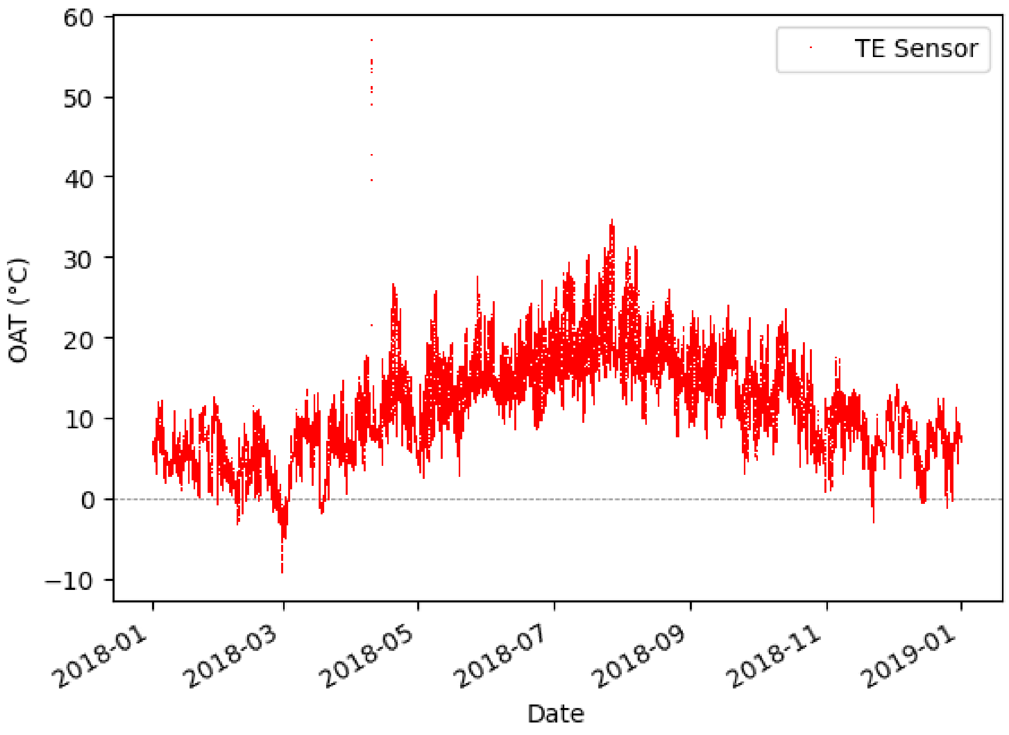

Figure 5: Plot of the TE sensor outside air temperature (OAT) in degrees Celcius against date/time for all *only* 2018, and with some additional formatting features.

More complex ideas:

- There are times are which temperatures change rapidly, which can be associated with similarly rapid changes in windspeed. Can these be identified? can you determine a possible reason why? Do any other properties change/correlate with these shifts?
- Produce an array of vector arrows indicating average wind speed and direction, on a grid representing month along one axis, and time of day along the other. Note that it will be awkward to display the vector arrows correctly. Are there any patterns? What applications might such a plot have?
- As mentioned when discussing the data earlier in the document, the wind direction sensor stops operating correctly for a period. Can this be characterised? Were there indications of approaching a failure state leading up to this?