Unit 7: Data input and output

Richard Foltyn
University of Glasgow

May 26, 2023

Contents

7 Data input and output

In this unit we discuss input and output, or I/O for short. We focus exclusively on I/O routines used to load and store data from files that are relevant for numerical computation and data analysis.

7.1 Input/output with NumPy

7.1.1 Loading text data

We have already encountered the most basic, and probably most frequently used NumPy I/O routine, `np.loadtxt()`. We often use files that store data as text files containing character-separated values (CSV) since virtually any application supports this data format. The most important I/O functions to process text data are:

- `np.loadtxt()`: load data from a text file.
- `np.genfromtxt()`: load data from a text file and handle missing data.
- `np.savetxt()`: save a NumPy array to a text file.

There are a few other I/O functions in NumPy, for example to write arrays as raw binary data. We won't cover them here, but you can find them in the official documentation.

Example: Load character-separated text data

Imagine we have the following tabular data from FRED, where the first two rows look as follows:

Year	GDP	CPI	UNRATE
1948	2118.5	24.0	3.8
1949	2106.6	23.8	6.0

These data are stored as character-separated values (CSV). To load this CSV file as a NumPy array, we use `loadtxt()`. As in the previous unit, it is advantageous to globally set the path to the `data/` directory that can point either to the local directory or to the `data/` directory on GitHub.

```
[1]: # Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/python-intro-PGR/main/data'
```

```
[2]: import numpy as np

# Path to CSV file
file = f'{DATA_PATH}/FRED.csv'

# load CSV
data = np.loadtxt(file, skiprows=1, delimiter=',')

data[:2]         # Display first two rows
```

```
[2]: array([[1948. , 2118.5,   24. ,    3.8],
       [1949. , 2106.6,   23.8,    6. ]])
```

The default settings will in many cases be appropriate to load whatever CSV file we might have. However, we'll occasionally want to specify the following arguments to override the defaults:

- `delimiter`: Character used to separate individual fields (default: space).
- `skiprows=n`: Skip the first `n` rows. For example, if the CSV file contains a header with variable names, `skiprows=1` needs to be specified as NumPy by default cannot process these names.
- `dtype`: Enforce a particular data type for the resulting array.
- `encoding`: Set the character encoding of the input data. This is usually not needed, but can be required to import data with non-latin characters that are not encoded using Unicode.

While `loadtxt()` is simple to use, it quickly reaches its limits with more complex data sets. For example, when we try to load our sample of universities with `loadtxt()`, we get the following error:

```
[3]: import numpy as np

file = f'{DATA_PATH}/universities.csv'

# Try to load CSV data that contains strings
# This will result in an error!
data = np.loadtxt(file, delimiter=';', skiprows=1)
```

```
ValueError: could not convert string to float: '"University of Glasgow"'


The above exception was the direct cause of the following exception:

ValueError: could not convert string '"University of Glasgow"' to float64 at row 0, column 1.
 ↪
```

This code fails for two reasons:

1. The file contains strings and floats, and `loadtxt()` by default cannot load mixed data (e.g., strings and numerical data).
2. There are missing values (empty fields), which `loadtxt()` cannot handle either.

The simplest way to address these issues is to use pandas to load the data which we turn to in the next section.

7.1.2 Saving data to text files

To save a NumPy array to a CSV file, there is a logical counterpart to `np.loadtxt()` which is called `np.savetxt()`.

[4]:
```
import numpy as np
import os.path
import tempfile

# Generate three columns of 5 observations each
data = np.linspace(0.0, 1.0, 15).reshape((3, 5))

# create temporary directory
d = tempfile.TemporaryDirectory()

# path to CSV file
file = os.path.join(d.name, 'data.csv')

# Print destination file - this will be different each time
print(f'Saving CSV file to {file}')

# Write NumPy array to CSV file. The fmt argument specifies
# that data should be saved as floating-point using a
# field width of 8 characters and 5 decimal digits.
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

```
Saving CSV file to /tmp/tmp8hnv2edo/data.csv
```

The above code creates a 5×3 matrix of floats and stores these in the file `data.csv` using 5 significant digits.

We store the destination file in a temporary directory which we create as follows:

- Because we cannot know in advance on which system this code is run (e.g., the operating system and directory layout), we cannot hard-code a file path.
- Moreover, we do not know whether the code is run with write permissions in any particular folder.
- We work around this issue by asking the Python runtime to create a writeable temporary directory *for the system where the code is being run*.
- We use the routines in the `tempfile` module to create this temporary directory.

Of course, on your own computer you do not need to use a temporary directory, but can instead use any directory where your user has write permissions. For example, on Windows you could use something along the lines of

```
file = 'C:/Users/Path/to/file.txt'
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

You can even use relative paths. To store a file in the current working directory it is sufficient to just pass the file name:

```
file = 'file.txt'
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

7.2 Input/output with pandas

Pandas's I/O routines are more powerful than those implemented in NumPy:

- They support reading and writing numerous file formats.
- They support heterogeneous data without having to specify the data type in advance.
- They gracefully handle missing values.

For these reasons, it is often preferable to directly use pandas to process data instead of NumPy.

The most important routines are:

- `read_csv()`, `to_csv()`: Read or write CSV text files
- `read_fwf()`: Read data with fixed field widths, i.e., text data that does not use delimiters to separate fields.
- `read_excel()`, `to_excel()`: Read or write Excel spreadsheets
- `read_stata()`, `to_stata()`: Read or write Stata's `.dta` files.
- `read_pickle()`, `to_pickle()`: Read or write Python's binary pickle format, optionally using compression (see here for details). This should only be used to create temporary files, not to store data permanently.

For a complete list of I/O routines, see the official documentation.

To illustrate, we repeat the above examples using pandas's `read_csv()`. Since the FRED data contains only floating-point data, the result is very similar to reading in a NumPy array.

[5]:
```
import pandas as pd

# relative path to CSV file
file = f'{DATA_PATH}/FRED.csv'

df = pd.read_csv(file, sep=',')
df.head(2)          # Display the first 2 rows of data
```

[5]:
```
   Year     GDP   CPI  UNRATE
0  1948  2118.5  24.0     3.8
1  1949  2106.6  23.8     6.0
```

The difference between NumPy and pandas becomes obvious when we try to load our university data: this works out of the box:

[6]:
```
import pandas as pd

# relative path to CSV file
file = f'{DATA_PATH}/universities.csv'

df = pd.read_csv(file, sep=';')
df.tail(3)      # show last 3 rows
```

[6]:
```
                   Institution           Country  Founded  Students   Staff  \
20      University of Stirling          Scotland     1967      9548     NaN
21  Queen's University Belfast  Northern Ireland     1810     18438  2414.0
22           Swansea University            Wales     1920     20620     NaN

     Admin  Budget  Russell
20  1872.0   113.3        0
21  1489.0   369.2        1
22  3290.0     NaN        0
```

Note that missing values are correctly converted to `np.nan`.

Unlike NumPy, pandas can also process other popular data formats such as MS Excel files (or OpenDocument spreadsheets):

[7]:
```
import pandas as pd

# Excel file containing university data
file = f'{DATA_PATH}/universities.xlsx'

df = pd.read_excel(file, sheet_name='universities')
df.head(3)
```

```
[7]:                  Institution   Country  Founded  Students   Staff   Admin  \
     0     University of Glasgow  Scotland     1451     30805  2942.0  4003.0
     1   University of Edinburgh  Scotland     1583     34275  4589.0  6107.0
     2  University of St Andrews  Scotland     1413      8984  1137.0  1576.0

        Budget  Russell
     0   626.5        1
     1  1102.0        1
     2   251.2        0
```

The routine `read_excel()` takes the argument `sheet_name` to specify the sheet that should be read.

- Note that the Python package `openpyxl` needs to be installed in order to read files from Excel 2003 and above.
- To read older Excel files (`.xls`), you need the package `xlrd`.

Finally, we often encounter text files with fixed field widths, since this is a commonly used format in older applications (for example, fixed-width files are easy to create in Fortran). To illustrate, the fixed-width variant of our FRED data looks like this:

```
Year GDP    CPI  UNRATE
1948 2118.5   24     3.8
1949 2106.6 23.8       6
1950 2289.5 24.1     5.2
1951 2473.8   26     3.3
1952 2574.9 26.6       3
```

You see that the column `Year` occupies the first 5 characters, the `GDP` column the next 7 characters, and so on. To read such files, the width (i.e., the number of characters) has to be explicitly specified:

```
[8]: import pandas as pd

     # File name of FRED data, stored as fixed-width text
     file = f'{DATA_PATH}/FRED-fixed.csv'

     # field widths are passed as list to read_fwf()
     df = pd.read_fwf(file, widths=[5, 7, 5, 8])
     df.head(3)
```

```
[8]:    Year     GDP   CPI  UNRATE
     0  1948  2118.5  24.0     3.8
     1  1949  2106.6  23.8     6.0
     2  1950  2289.5  24.1     5.2
```

Here the `widths` argument accepts a list that contains the number of characters to be used for each field.

7.3 Retrieving macroeconomic / financial data from the web

7.3.1 Yahoo! Finance data

`yfinance` is a user-written library to access data from Yahoo! Finance using the public API (see the project's GitHub repository for detailed examples). This project is not affiliated with Yahoo! Finance and is intended for personal use only. Before using the library, it needs to be installed from PyPi as follows:

```
pip install yfinance
```

[9]:
```
# When running via Google Colab, uncomment and execute the following line
#! pip install yfinance
```

`yfinance` allows us to retrieve information for a single symbol via properties of the `Ticker` object, or for multiple ticker symbols at once.

Example: Retrieving data for a single symbol

We first use the API to retrieve data for a single symbol, in this case the S&P 500 index which has the (somewhat unusual) ticker symbol `^GSPS`. One can easily find the desired ticker symbol by searching for some stock, index, currency or other asset on Yahoo! Finance.

[10]:
```
import yfinance as yf

# Symbol for S&P 500 index
symbol = '^GSPC'

# Create ticker object
ticker = yf.Ticker(symbol)
```

We can now use the attributes of the `ticker` object to get all sorts of information. For example, we can get some meta data from the `info` attribute as follows:

[11]:
```
# Descriptive name and asset class
shortname = ticker.info['shortName']
quoteType = ticker.info['quoteType']

# 52-week low and high
low = ticker.info['fiftyTwoWeekLow']
high = ticker.info['fiftyTwoWeekHigh']

print(f'{shortname} is an {quoteType}')
print(f'{shortname} 52-week range: {low} - {high}')

# To see which keys are available, use the keys() method
# ticker.info.keys()
```

```
S&P 500 is an INDEX
S&P 500 52-week range: 3491.58 - 4325.28
```

We use the `history` attribute to get detailed price data. Unless we want all available data, we should select the relevant period using the `start=...` and `end=...` arguments.

[12]:
```
# Retrieve daily index values data for first quarter of this year
daily = ticker.history(start='2023-01-01', end='2023-03-31')

# Print first 5 rows
daily.head()
```

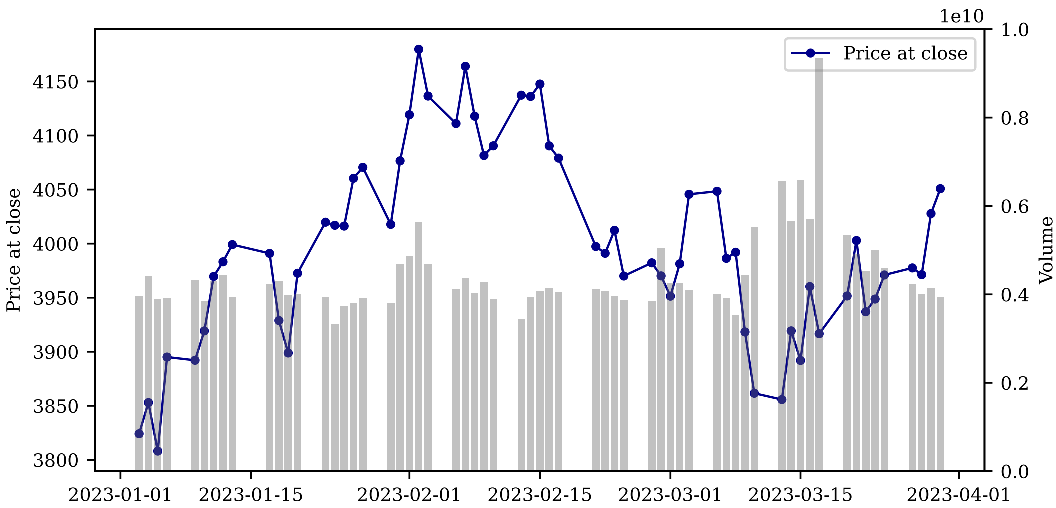
[12]:
```
                                  Open         High          Low        Close  \
Date
2023-01-03 00:00:00-05:00  3853.290039  3878.459961  3794.330078  3824.139893
2023-01-04 00:00:00-05:00  3840.360107  3873.159912  3815.770020  3852.969971
2023-01-05 00:00:00-05:00  3839.739990  3839.739990  3802.419922  3808.100098
2023-01-06 00:00:00-05:00  3823.370117  3906.189941  3809.560059  3895.080078
2023-01-09 00:00:00-05:00  3910.820068  3950.570068  3890.419922  3892.090088

                               Volume  Dividends  Stock Splits
Date
2023-01-03 00:00:00-05:00  3959140000        0.0           0.0
2023-01-04 00:00:00-05:00  4414080000        0.0           0.0
2023-01-05 00:00:00-05:00  3893450000        0.0           0.0
```

```
2023-01-06 00:00:00-05:00  3923560000        0.0           0.0
2023-01-09 00:00:00-05:00  4311770000        0.0           0.0
```

We can then use this data to plot the daily closing price and trading volume.

[13]:
```python
import matplotlib.pyplot as plt

fix, ax = plt.subplots(1, 1, figsize=(7,3.5))

# Plot closing price
ax.plot(daily.index, daily['Close'], color='darkblue', marker='o', ms=3, lw=1)
ax.set_ylabel('Price at close')
ax.legend(['Price at close'], loc='upper right')

# Create secondary y-axis for trading volume
ax2 = ax.twinx()

# Plot trading volume as bar chart
ax2.bar(daily.index, daily['Volume'], color='#666666', alpha=0.4, zorder=-1, lw=0)
ax2.set_ylim((0.0, 1.0e10))
ax2.set_ylabel('Volume')
```

[13]:
```
Text(0, 0.5, 'Volume')
```



The above code uses `twinx()` to create a second (invisible) x-axis with an independent y-axis which allows us to plot the trading volume on a different scale.

Example: Retrieving data for multiple symbols

We can download trading data for multiple symbols at once using the `download()` function. Unlike the `Ticker` class, this immediately returns a `DataFrame` containing data similar to the `history` method we called previously, but now the column index contains an additional level for each ticker symbol. For example, to get the trading data for Amazon and Microsoft for the first 3 months of 2023, we proceed as follows:

[14]:
```python
import yfinance as yf

# Get data for Amazon (AMZN) and Microsoft (MSFT) for first quarter of 2023
data = yf.download(('AMZN', 'MSFT'), start='2023-01-01', end='2023-03-31')
data.head()
```

```
[*********************100%***********************]  2 of 2 completed
```

[14]:
```
              Adj Close               Close                  High  \
                   AMZN        MSFT    AMZN        MSFT      AMZN
Date
2023-01-03   85.820000  238.460144  85.820000  239.580002  86.959999
2023-01-04   85.139999  228.029129  85.139999  229.100006  86.980003
2023-01-05   83.120003  221.270844  83.120003  222.309998  85.419998
2023-01-06   86.080002  223.878601  86.080002  224.929993  86.400002
2023-01-09   87.360001  226.058380  87.360001  227.119995  89.480003

                            Low                  Open             \
                  MSFT     AMZN        MSFT      AMZN        MSFT
Date
2023-01-03  245.750000  84.209999  237.399994  85.459999  243.080002
2023-01-04  232.869995  83.360001  225.960007  86.550003  232.279999
2023-01-05  227.550003  83.070000  221.759995  85.330002  227.199997
2023-01-06  225.759995  81.430000  219.350006  83.029999  223.000000
2023-01-09  231.240005  87.080002  226.410004  87.459999  226.449997

              Volume
                AMZN      MSFT
Date
2023-01-03  76706000  25740000
2023-01-04  68885100  50623400
2023-01-05  67930800  39585600
2023-01-06  83303400  43613600
2023-01-09  65266100  27369800
```

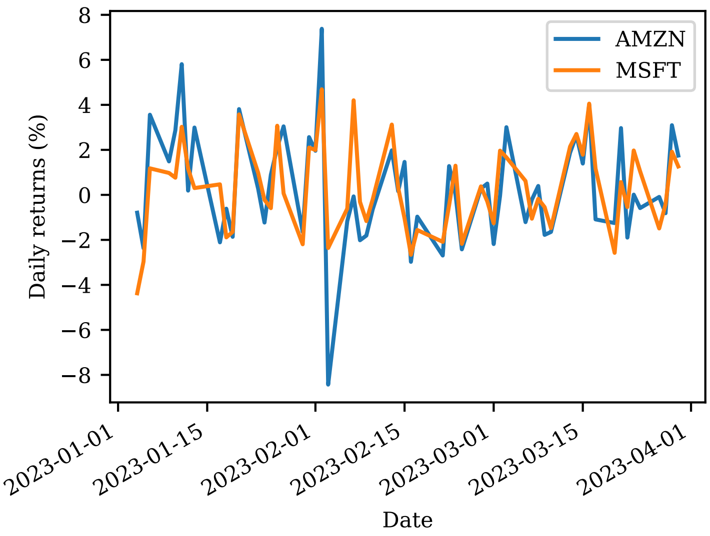
To extract data for a particular symbol, we have to take into account the hierarchical column index:

[15]:
```
# Use hierarchical indexing to get data for Amazon
data[('Close', 'AMZN')].head()
```

[15]:
```
Date
2023-01-03    85.820000
2023-01-04    85.139999
2023-01-05    83.120003
2023-01-06    86.080002
2023-01-09    87.360001
Name: (Close, AMZN), dtype: float64
```

[16]:
```
# Plot daily returns for both stocks
returns = data['Close'].pct_change() * 100.0
returns.plot(y=['AMZN', 'MSFT'], ylabel='Daily returns (%)')
```

[16]:
```
<Axes: xlabel='Date', ylabel='Daily returns (%)'>
```



7.3.2 Pandas Datareader

`pandas-datareader` is a Python library that fetches online data from multiple sources and returns them as pandas `DataFrame` objects. Despite its name, this library is not included in `pandas` and may need to be installed separately, e.g., by running

```
pip install pandas-datareader
```

The aim is to provide a uniform API to access data from multiple sources, including those covered in other sections in this unit. See the official documentation for supported data sources and how to access them.

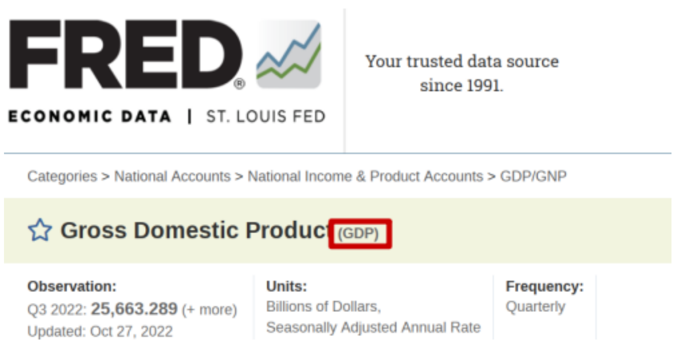
[17]:
```
# Uncomment and execute the following line if running in Google Colab
# ! pip install pandas-datareader
```

Example: Downloading data from FRED

As a first illustration, we fetch macroeconomic data from FRED, or Federal Reserve Economic Data. FRED is provided by the Federal Reserve of St. Louis and is one of the most important macroeconomic online databases (at least for US-centric data).

An alternative (but more complicated) way to access this data is via the `fredapi` library, which we examine below. With `pandas-datareader`, no API key is required to access FRED which makes using it a little simpler than `fredapi`.

In order to retrieve any data, we first need to identify the series name. This is easiest done by searching for the data on FRED using your browser and copying the series name, highlighted in red in the screenshot below.



For example, if we want to retrieve the US Gross Domestic Product (GDP), the corresponding series name is `GDP`. The FRED web page contains additional useful information such as the time period for which the data is available, the data frequency (monthly, quarterly, annual) and whether it's seasonally adjusted.

[18]:
```
# The convention is to import this library as web
import pandas_datareader.data as web

# define start and end dates
start_date = "2000-01-01"
end_date = "2021-12-31"

# Specify series name as first and 'fred' data source as second argument
gdp = web.DataReader('GDP',
    data_source='fred',
    start=start_date,
    end=end_date
)

# Show first 3 observations
gdp.head(3)
```

[18]:
```
                  GDP
DATE
2000-01-01  10002.179
2000-04-01  10247.720
2000-07-01  10318.165
```

We can also fetch multiple series at the same time, for example the CPI (`CPIAUCSL`) and the unemployment rate (`UNRATE`).

[19]:
```
# without an explicit end argument, the latest available data
# will be retrieved
data = web.DataReader(['CPIAUCSL', 'UNRATE'],
    data_source='fred',
    start='2020-01-01'
)

data.head(3)
```

[19]:
```
            CPIAUCSL  UNRATE
DATE
2020-01-01   259.037     3.5
2020-02-01   259.248     3.5
2020-03-01   258.124     4.4
```

7.3.3 FRED: Federal Reserve Economic Data (optional)

FRED, provided by the Federal Reserve of St. Louis, is one of the most important macroeconomic online databases (at least for US-centric data). `fredapi` is a Python API for the FRED data which provides a wrapper for the FRED web service (see also the project's GitHub page).

Before accessing FRED, you need to install `fredapi` into your Python environment as follows:

```
pip install --no-deps fredapi
```

Important:

- The `--no-deps` argument might be required for Anaconda users as otherwise the conda-provided versions of `numpy` and `pandas` could be overwritten.

- Anaconda users should *not* use the `fredapi` package provided in `conda-forge` as at the time of this writing it is outdated and will not work.

To use FRED, you additional need an API key which can be requested at https://fred.stlouisfed.org/docs/api/api_key.html. Unlike with some other APIs we discuss below, it is not possible to make a request without a key. Once you have a key, you can specify it in several ways:

1. On your local machine, set the environment variable `FRED_API_KEY` to store the key and it will be picked up automatically. This only works if you run a Python environment locally.
2. Store it in a file and pass the file name when creating a `Fred` instance:

   ```
   from fredapi import Fred
   fred = Fred(api_key_file='path_to_file')
   ```

3. Pass the string containing the API key as a parameter:

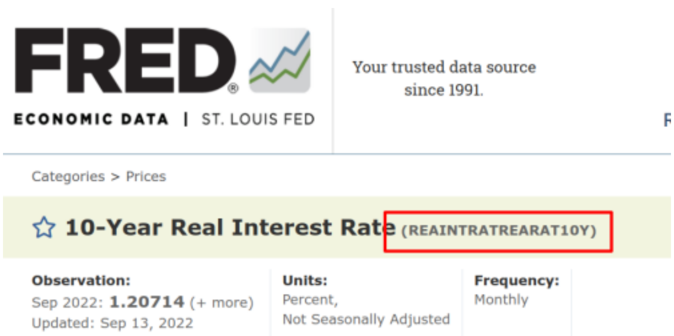
   ```
   from fredapi import Fred
   fred = Fred(api_key='INSERT API KEY HERE')
   ```

The following code assumes that the `FRED_API_KEY` variable has been set up and might not work in your environment if that is not the case.

Example: Retrieve the 10-year real interest rate

[20]:
```
# Uncomment the following to install fredapi in your local or
# cloud-hosted Python environment (e.g.,  Google Colab)

#! pip install --no-deps fredapi
```

In order to retrieve any data, we first need to identify the series name. This is easiest done by searching for the data on FRED using your browser and copying the series name, highlighted in red in the screenshot below.



For example, if we want to retrieve the 10-year real interest rate, the corresponding series name is `REAINTRATREARAT10Y`. The FRED web page contains additional useful information such as the time period for which the data is available, the data frequency (monthly, quarterly, annual) and whether it's seasonally adjusted.

To download and plot the 10-year real interest rate, we proceed as follows:

[21]:
```
from fredapi import Fred

# Create instance assuming API key is stored as environment variable
fred = Fred()

# or specify API key directly
# fred = Fred(api_key='INSERT API KEY HERE')

# Download observations starting from the year 2000 onward
series = fred.get_series('REAINTRATREARAT10Y',
```

```
    observation_start='2000-01-01'
)
```

[22]:
```
# Print first 5 observations
series.head(5)
```

[22]:
```
2000-01-01    3.411051
2000-02-01    3.513343
2000-03-01    3.440347
2000-04-01    3.202967
2000-05-01    3.360531
dtype: float64
```

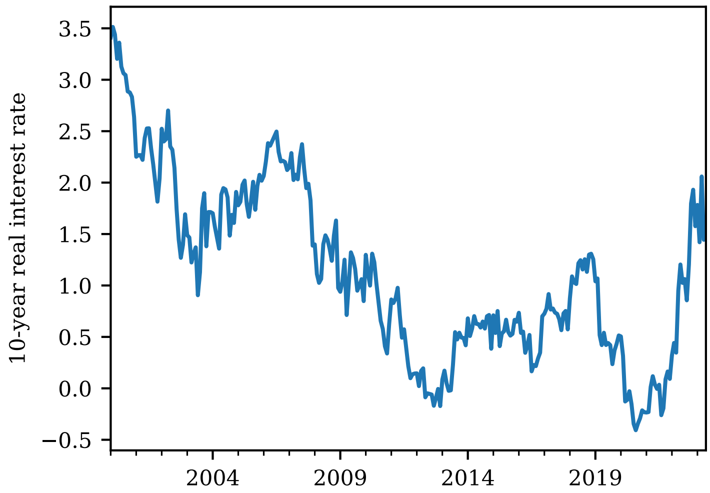
The data is returned as a pandas `Series` object with the corresponding dates set as the index.

[23]:
```
series.info()
```

```
<class 'pandas.core.series.Series'>
DatetimeIndex: 281 entries, 2000-01-01 to 2023-05-01
Series name: None
Non-Null Count  Dtype
--------------  -----
281 non-null    float64
dtypes: float64(1)
memory usage: 4.4 KB
```

[24]:
```
# Plot interest rate time series
series.plot(ylabel='10-year real interest rate')
```

[24]:
```
<Axes: ylabel='10-year real interest rate'>
```



Other popular time series available on FRED are the CPI, real GDP and the unemployment rate.

7.3.4 NASDAQ data API (optional)

The NASDAQ stock exchange provides an open-source Python library hosted on GitHub to access various types of financial data (not only those traded on NASDAQ), see here for details. The detailed API documentation can be found at here. This data API was formerly known as quandl which is no longer actively maintained but might still work.

Before using this service, you need to make sure that the Python package is installed. Depending on how you launched this notebook, you may need to execute the following code to install `nasdaq-data-link`:

```
pip install nasdaq-data-link
```

Various types of data are available via this service and can be found using the online search at https://data.nasdaq.com/search.

- Data come from various data provides. To select a data set, you usually have to specify a string of the form `'PROVIDER/SERIES'` where `'PROVIDER'` is the name of the provider (e.g., `'FRED'` or `'BOE'`) and `'SERIES'` is the name of the time series.
- Most of these data require a subscription or at least a free NASDAQ account. Once you have an account, you will need to get an API key and specify it when retrieving data. See the above links for details.
- Some commercial data series include sample data that can be used without a subscription but requires a free NASDAQ account.
- Some data series are freely available without a subscription or an account. These are often taken from other freely available data sets such as FRED or blockchain.com. We'll be using these to demonstrate how the API works.

  Important: Even for freely available data, NASDAQ imposes a cap of 50 web requests per day. You need to register to get around this.

The data is returned as pandas `DataFrame` object (or alternatively as an NumPy array).

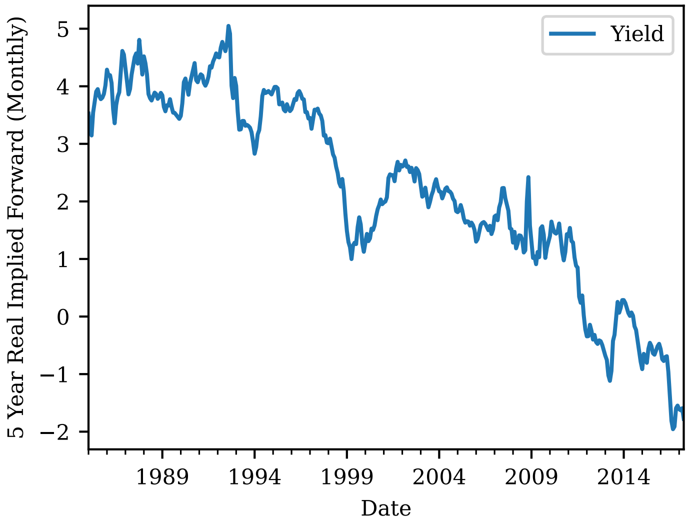
Example: Data from the Bank of England

Let's start by retrieving some macroeconomic times series from the Bank of England (BOE). It's not always straightforward to find the name of the time series one is looking for, but you can see some of the available time series here. The name will vary depending on the type of data (interest rate, exchange rate), the frequency and how it is aggregated (daily, last day of the month, monthly average) and a currency pair, if applicable.

[25]:
```
# When running via Google Colab, uncomment and execute the following line
#! pip install nasdaq-data-link
```

[26]:
```
# Retrieve 5-year real implied yield on UK government bonds
import nasdaqdatalink as ndl
df = ndl.get('BOE/IUMASRIF')

# Rename column which is always called 'Value'
df = df.rename(columns={'Value': 'Yield'})

# Plot time series
df.plot(ylabel='5 Year Real Implied Forward (Monthly)')
```

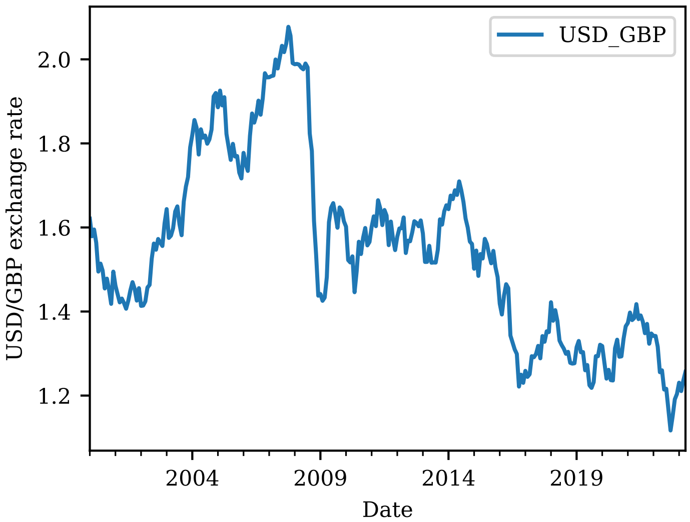
[26]:
```
<Axes: xlabel='Date', ylabel='5 Year Real Implied Forward (Monthly)'>
```



As another example, we retrieve the US dollar / Sterling exchange rate at a monthly frequency (this is determined by the name of the time series used where `ML` requests the monthly series, using the last observation for each month). Note that we can pass additional arguments, for example restricting the time period we want to retrieve using `start_date` and `end_date`.

[27]:
```
# Get USD / GDP exchange rate using the last observation for each month.
df = ndl.get('BOE/XUMLUSS', start_date='2000-01-31')
df = df.rename(columns={'Value': 'USD_GBP'})

# Plot USD/GBP time series
df.plot(ylabel='USD/GBP exchange rate')
```

[27]:
```
<Axes: xlabel='Date', ylabel='USD/GBP exchange rate'>
```



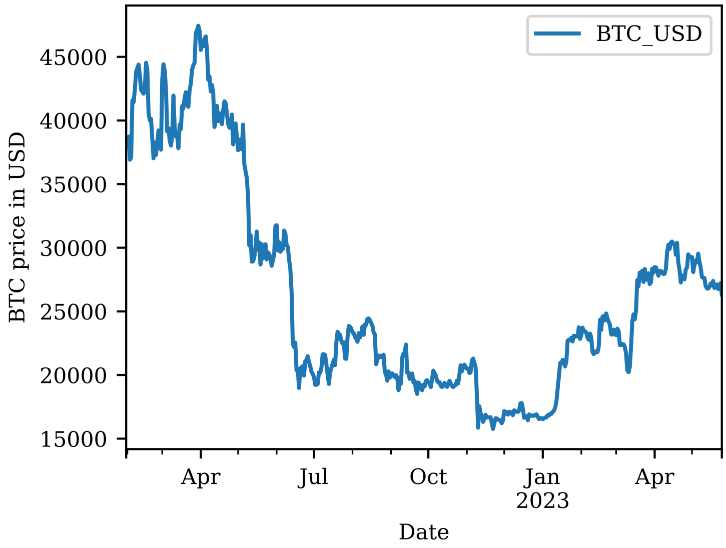
Example: Data from blockchain.com

The NASDAQ data link also supports retrieving data on cryptocurrencies. For example, there is a freely accessible time series for the price of Bitcoin in USD.

[28]:
```
import nasdaqdatalink as ndl

# Retrieve price of BTC in USD for 2022
```

```
df = ndl.get('BCHAIN/MKPRU', start_date='2022-01-31')

# Change column name to something more descriptive
df = df.rename(columns={'Value': 'BTC_USD'})

# Plot time series
df.plot(ylabel='BTC price in USD')
```

[28]: <Axes: xlabel='Date', ylabel='BTC price in USD'>



Example: Historical stock data

As a final example, we obtain the trading data for the stock of Apple (ticker symbol AAPL) for the year 2001. Such data is often not available without a subscription or a login, but it works if the requested time period is sufficiently far in the past!

[29]:
```
# Retrieve stock data for Apple (ticker symbol AAPL)
df = ndl.get("WIKI/AAPL",
    start_date='2000-01-01',
    end_date='2000-12-31'
)
```

Unlike in the previous examples, this data contains not only a single value, but a whole range of variables including the opening and closing price, the trading volume, etc.:

[30]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 252 entries, 2000-01-03 to 2000-12-29
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Open         252 non-null    float64
 1   High         252 non-null    float64
 2   Low          252 non-null    float64
 3   Close        252 non-null    float64
 4   Volume       252 non-null    float64
 5   Ex-Dividend  252 non-null    float64
 6   Split Ratio  252 non-null    float64
 7   Adj. Open    252 non-null    float64
 8   Adj. High    252 non-null    float64
```

```
 9   Adj. Low     252 non-null    float64
 10  Adj. Close   252 non-null    float64
 11  Adj. Volume  252 non-null    float64
dtypes: float64(12)
memory usage: 25.6 KB
```

To plot a specific column, we can use the `y=...` argument to `DataFrame.plot()`.

[31]:
```
df.plot(y='Close', ylabel='Stock prive of AAPL')
```

[31]:
```
<Axes: xlabel='Date', ylabel='Stock prive of AAPL'>
```