Lecture 9: Introduction to pandas

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn

NHH Norwegian School of Economics

March 11, 2025

See GitHub repository for notebooks and data:

`https://github.com/richardfoltyn/FIE463-V25`

Contents

1 Introduction to pandas

1.1 Motivation

So far, we have encountered built-in Python containers (`list`, `tuple`, `dict`) and NumPy arrays as the only way to store data. However, while NumPy arrays are great for storing *homogenous* data without any particular structure, they are somewhat limited when we want to use them for data analysis.

For example, we usually want to process data sets with

1. several variables;
2. multiple observations, which need not be identical across variables (some values may be missing);
3. non-homogenous data types: for examples, names need to be stored as strings, birthdays as dates and income as a floating-point number.

While NumPy can in principle handle such situations, it puts all the burden on the user. Most users would prefer to not have to deal with such low-level details.

Pandas was created to offer more versatile data structures that are straightforward to use for storing, manipulating and analyzing heterogeneous data:

1. Data is clearly organized in *variables* and *observations*, similar to econometrics programs such as Stata and R using `data.frame`.
2. Each variable is permitted to have a *different* data type.
3. We can use *labels* to select observations instead of having to use a linear numerical index as with NumPy.

   We could, for example, index a data set using National Insurance Numbers or time stamps for time series data.
4. Pandas offers many convenient data aggregation and reduction routines that can be applied to subsets of data.

   For example, we can easily group observations by city and compute average incomes.
5. Pandas also offers many convenient data import / export functions that go beyond what's in NumPy.

Should we be using pandas at all times, then? No!

- For low-level tasks where performance is essential, use NumPy.
- For homogenous data without any particular data structure, use NumPy.
- On the other hand, if data is heterogeneous, needs to be imported from an external data source and cleaned or transformed before performing computations, use pandas.

There are numerous tutorials on pandas on the internet. Useful additional material includes:

- The official user guide.
- The official pandas cheat sheet which nicely illustrates the most frequently used operations.
- The official API reference with details on every pandas object and function.
- There are numerous tutorials (including videos) available on the internet. See here for a list.

1.2 Creating pandas data structures

Pandas has two main data structures:

1. `Series` represents observations of a *single* variable.
2. `DataFrame` is a container for *several* variables. You can think of each individual column of a `DataFrame` as a `Series`, and each row represents one observation.

The easiest way to create a `Series` or `DataFrame` is to create them from pre-existing data.

To access pandas data structures and routines, we need to import them first. The near-universal convention is to make pandas available using the name `pd`:

```
import pandas as pd
```

Example: Create Series from 1-dimensional NumPy array

[1]:
```
import numpy as np
import pandas as pd             # universal convention: import using pd

data = np.arange(5, 10)

# Create pandas Series from 1d array
pd.Series(data)
```

[1]:
```
0    5
1    6
2    7
```

```
3    8
4    9
dtype: int64
```

Example: Create DataFrame from NumPy array

We can create a `DataFrame` from a NumPy array:

[2]:
```
# Create matrix of data
data = np.arange(15).reshape((-1, 3))

# Define variable (or column) names
varnames = ['A', 'B', 'C']

# Create pandas DataFrame from matrix
pd.DataFrame(data, columns=varnames)
```

[2]:
```
    A   B   C
0   0   1   2
1   3   4   5
2   6   7   8
3   9  10  11
4  12  13  14
```

This code creates a `DataFrame` of three variables called `A`, `B` and `C` with 5 observations each.

Example: Create DataFrame from dictionary

Alternatively, we can create a `DataFrame` from non-homogenous data as follows:

[3]:
```
# Names (strings)
names = ['Alice', 'Bob']

# Birth dates (datetime objects)
bdates = pd.to_datetime(['1985-01-01', '1997-05-12'])

# Incomes (floats)
incomes = np.array([600000, np.nan])         # code missing income as NaN

# create DataFrame from dictionary
pd.DataFrame({'Name': names, 'Birthdate': bdates, 'Income': incomes})
```

[3]:
```
    Name  Birthdate    Income
0  Alice 1985-01-01  600000.0
1    Bob 1997-05-12       NaN
```

If data types differ across columns, as in the above example, it is often convenient to create the `DataFrame` by passing a dictionary as an argument. Each key represents a column name and each corresponding value contains the data for that variable.

1.3 Importing data

1.3.1 Loading data with NumPy & its limitations

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 functions to read text data are:

- `np.loadtxt()`: load data from a text file.
- `np.genfromtxt()`: load data from a text file and handle missing data.

There are a few other input/output 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

Consider the following tabular data from FRED stored in the file FRED_annual.csv where the first two rows look as follows:

Year	GDP	CPI	UNRATE	FEDFUNDS	INFLATION
1954	2877.7	26.9	5.6	1.0	
1955	3083.0	26.8	4.4	1.8	-0.4

Note that the inflation column does has a missing value for the year 1954.

These data are stored as character-separated values (CSV). To load this CSV file as a NumPy array, we use `loadtxt()`. 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.

[4]:
```
# 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/FIE463-V25/main/data'
```

[5]:
```
import numpy as np

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

# load CSV, skip header row and first row with missing data
data = np.loadtxt(file, skiprows=2, delimiter=',')

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

[5]:
```
array([[ 1.9550e+03,  3.0830e+03,  2.6800e+01,  4.4000e+00,  1.8000e+00,
        -4.0000e-01],
       [ 1.9560e+03,  3.1488e+03,  2.7200e+01,  4.1000e+00,  2.7000e+00,
         1.5000e+00]])
```

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.
- `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 the FRED data set including the first data row, we get the following error:

[6]:
```
# Attempt to load CSV
data = np.loadtxt(file, skiprows=1, delimiter=',')
```

```
ValueError: could not convert string '' to float64 at row 0, column 6.
```

This code fails because `loadtxt()` does not support files with missing values. One can use the more flexible function `np.genfromtxt()` which allows us to parse files with missing values:

```
[7]: # Load CSV file using genfromtxt() instead of loadtxt()
     data = np.genfromtxt(file, skip_header=True, delimiter=',')

     # Display first rows
     data[:1]
```

```
[7]: array([[1.9540e+03, 2.8777e+03, 2.6900e+01, 5.6000e+00, 1.0000e+00,
                    nan]])
```

However, it is usually not worthwhile to figure out how to load complex data with NumPy as this is much easier with pandas.

1.3.2 Loading data with Pandas

Pandas's input/output 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 functions 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.

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

To illustrate, we repeat the above examples using pandas's `read_csv()`:

```
[8]: import pandas as pd

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

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

```
[8]:    Year     GDP   CPI  UNRATE  FEDFUNDS  INFLATION
     0  1954  2877.7  26.9     5.6       1.0        NaN
     1  1955  3083.0  26.8     4.4       1.8       -0.4
```

Your turn. Use the pandas functions listed above to import data from the following files located in the data folder:

1. `titanic.csv`
2. `FRED/FRED_annual.xlsx`

To load Excel files, you need to have the package `openpyxl` installed.

1.4 Viewing data

With large data sets, you hardly ever want to print the entire `DataFrame`. Pandas by default limits the amount of data shown. You can use the `head()` and `tail()` methods to explicitly display a specific number of rows from the top or the end of a `DataFrame`.

To illustrate, we use a data set of passengers on board of the Titanic's maiden voyage stored in `titanic.csv` which contains the following columns:

1. `PassengerId`
2. `Survived`: indicator whether the person survived
3. `Pclass`: accommodation class (first, second, third)
4. `Name`: Name of passenger (last name, first name)
5. `Sex`: `male` or `female`
6. `Age`
7. `Ticket`: Ticket number
8. `Fare`: Fare in pounds
9. `Cabin`: Deck + cabin number
10. `Embarked`: Port at which passenger embarked: `C` - Cherbourg, `Q` - Queenstown, `S` - Southampton

We can read in the data stored in the file `titanic.csv` like this:

[9]:
```
import pandas as pd

# URL to CSV file in GitHub repository
file = f'{DATA_PATH}/titanic.csv'

# Load sample data set of Titanic passengers. Individual fields are separated
# using a comma, which is the default.
df = pd.read_csv(file, sep=',')
```

We can now display the first and last three rows:

[10]:
```
df.head(3)      # show first three rows
```

[10]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3

                                                Name     Sex   Age  \
0                            Braund, Mr. Owen Harris    male  22.0
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss Laina  female  26.0

             Ticket     Fare Cabin Embarked
0         A/5 21171   7.2500   NaN        S
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
```

[11]:
```
df.tail(3)      # show last three rows
```

[11]:
```
     PassengerId  Survived  Pclass                                      Name  \
888          889         0       3  Johnston, Miss Catherine Helen "Carrie"
889          890         1       1                     Behr, Mr. Karl Howell
890          891         0       3                       Dooley, Mr. Patrick
```

```
        Sex   Age      Ticket   Fare Cabin Embarked
888  female   NaN  W./C. 6607  23.45   NaN        S
889    male  26.0      111369  30.00  C148        C
890    male  32.0      370376   7.75   NaN        Q
```

To quickly compute some descriptive statistics for the *numerical* variables in the `DataFrame`, we use `describe()`:

[12]:
```
df.describe()
```

[12]:
```
       PassengerId    Survived      Pclass         Age        Fare
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118   32.204208
std     257.353842    0.486592    0.836071   14.526497   49.693429
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    7.910400
50%     446.000000    0.000000    3.000000   28.000000   14.454200
75%     668.500000    1.000000    3.000000   38.000000   31.000000
max     891.000000    1.000000    3.000000   80.000000  512.329200
```

Note that this automatically ignores the columns `Name`, `Sex`, `Ticket` and `Cabin` as they contain strings, and computing the mean, standard deviation, etc. of a string variable does not make sense.

For categorical data, we can use `value_counts()` to tabulate the number of unique values of a variable. For example, the following code tabulates passengers by sex:

[13]:
```
df['Sex'].value_counts()
```

[13]:
```
Sex
male      577
female    314
Name: count, dtype: int64
```

Lastly, to see low-level information about the data type used in each column and the number of non-missing observations, we call `info()`:

[14]:
```
df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 10 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   Ticket       891 non-null    object 
 7   Fare         891 non-null    float64
 8   Cabin        204 non-null    object 
 9   Embarked     889 non-null    object 
dtypes: float64(2), int64(3), object(5)
memory usage: 69.7+ KB
```

Pandas automatically discards missing information in computations. For example, the age column has several missing values, so the number of reported `Non-Null` values is lower than for the other columns.

1.5 Indexing

Pandas supports two types of indexing:

1. Indexing by position. This is basically identical to the indexing of other Python and NumPy containers.
2. Indexing by label, i.e., by the values assigned to the row or column index. These labels need not be integers in increasing order, as is the case for NumPy. We will see how to assign labels below.

Pandas indexing is performed either by using brackets `[]`, or by using `.loc[]` for label indexing, or `.iloc[]` for positional indexing.

Indexing via `[]` can be somewhat confusing:

- specifying `df['name']` returns the column `name` as a `Series` object.
- On the other hand, specifying a range such as `df[5:10]` returns the *rows* associated with the *positions* 5,...,9.

Example: Selecting columns

[15]:
```
import pandas as pd

# Set option to limit the number of rows displayed
pd.set_option('display.max_rows', 10)

# Load sample data of Titanic passengers
df = pd.read_csv(f'{DATA_PATH}/titanic.csv')
df['Name']                  # select a single column
```

[15]:
```
0                                  Braund, Mr. Owen Harris
1        Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                   Heikkinen, Miss Laina
3             Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                                 Allen, Mr. William Henry
                              ...
886                                 Montvila, Rev. Juozas
887                           Graham, Miss Margaret Edith
888               Johnston, Miss Catherine Helen "Carrie"
889                                 Behr, Mr. Karl Howell
890                                   Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

[16]:
```
df[['Name', 'Sex']]     # select multiple columns using a list
```

[16]:
```
                                                  Name     Sex
0                              Braund, Mr. Owen Harris    male
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female
2                               Heikkinen, Miss Laina  female
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
4                             Allen, Mr. William Henry    male
..                                                 ...     ...
886                             Montvila, Rev. Juozas    male
887                       Graham, Miss Margaret Edith  female
888           Johnston, Miss Catherine Helen "Carrie"  female
889                             Behr, Mr. Karl Howell    male
890                               Dooley, Mr. Patrick    male

[891 rows x 2 columns]
```

Note: In order to select multiple columns you *must* specify these as a `list`, not a `tuple`.

Example: Selecting rows by position

To return the rows at positions 1, 2 and 3 we use

[17]:
```
df[1:4]
```

[17]:
```
   PassengerId  Survived  Pclass  \
1            2         1       1
2            3         1       3
3            4         1       1

                                                Name     Sex   Age  \
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss Laina  female  26.0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0

             Ticket     Fare Cabin Embarked
1          PC 17599  71.2833   C85        C
2  STON/O2. 3101282   7.9250   NaN        S
3            113803  53.1000  C123        S
```

Pandas follows the Python convention that indices are 0-based, and the endpoint of a slice is not included.

1.5.1 Creating and manipulating indices

Pandas uses *labels* to index and align data. These can be integer values starting at 0 with increments of 1 for each additional element, which is the default, but they need not be. The three main methods to create/manipulate indices are:

1. Create a new `Series` or `DataFrame` object with a custom index using the `index` argument.
2. `set_index(keys=['column1', ...])` uses the values of `column1` and optionally additional columns as indices, discarding the current index.
3. `reset_index()` resets the index to its default value, a sequence of increasing integers starting at 0.

Creating custom indices

First, consider the following code which creates a `Series` with three elements `[10, 20, 30]` using the default index `[0, 1, 2]`:

[18]:
```
import pandas as pd

# Create Series with default integer index
pd.Series([10, 20, 30])
```

[18]:
```
0    10
1    20
2    30
dtype: int64
```

We can use the `index` argument to specify a custom index, for example one containing the lower-case characters `a`, `b`, `c` as follows:

[19]:
```
# Create Series with custom index [a, b, c]
pd.Series([10, 20, 30], index=['a', 'b', 'c'])
```

[19]:
```
a    10
b    20
c    30
```

```
dtype: int64
```

Manipulating indices

To modify the index of an *existing* `Series` or `DataFrame` object, we use the methods `set_index()` and `reset_index()`. Note that these return a new object and leave the original `Series` or `DataFrame` unchanged. If we want to change the existing object, we need to pass the argument `inplace=True`.

For example, we can replace the row index and use the Roman lower-case characters `a`, `b`, `c`, ... as labels instead of integers:

[20]:
```
# Create DataFrame with 2 columns
df = pd.DataFrame({'A': [10, 20, 30], 'B': ['a', 'b', 'c']})

df
```

[20]:
```
    A  B
0  10  a
1  20  b
2  30  c
```

Since we did not specify any index, the default index $[0, 1, ...]$ is used. We can use `set_index()` set the index to the values from a column, for example column `B`:

[21]:
```
# Use column 'B' as index, store result in new DataFrame
df2 = df.set_index('B')

# Display updated DataFrame
df2
```

[21]:
```
    A
B
a  10
b  20
c  30
```

Note that pandas operations are usually not in place, so only `df2` uses column `B` as the index, whereas the original `df` remains unchanged:

[22]:
```
df
```

[22]:
```
    A  B
0  10  a
1  20  b
2  30  c
```

We can use the `inplace=True` argument to `set_index()` to update the index in-place, even though the pandas project usually does not encourage users to change things in place:

[23]:
```
# Set index in-place, i.e.,  df is modified
df.set_index('B', inplace=True)

df
```

[23]:
```
    A
B
a  10
b  20
c  30
```

Importantly, when changing things in-place, pandas functions usually don't return anything (the return value is `None`), so it is a mistake to attempt to assign the return value to a variable.

We can now use these new labels to select records in the DataFrame:

[24]:
```
# print first 2 rows using labels
df['a':'b']              # This is the same as df[:2]
```

[24]:
```
    A
B
a  10
b  20
```

Note that when specifying a range in terms of labels, the last element *is* included! Hence the row with index c in the above example is shown.

We can reset the index to its default integer values using the `reset_index()` method:

[25]:
```
# Reset index labels to default value (integers 0, 1, 2, ...) and print
# first three rows
df.reset_index(drop=True).head(3)
```

[25]:
```
    A
0  10
1  20
2  30
```

The `drop=True` argument tells pandas to throw away the old index values instead of storing them as a column of the resulting `DataFrame`.

Your turn. Read in the following data files from the `data/FRED` folder and manipulate the dataframe index:

1. Read in the file `FRED_annual.csv` and set the column `Year` as the index.
2. Read in the file `FRED_monthly.csv` and set the columns `Year` and `Month` as the index

Experiment what happens if you use the `inplace=True` and `append=True` options of `set_index()`. Restore the original (default) index after you are done.

1.5.2 Selecting elements

To more clearly distinguish between selection by label and by position, pandas provides the `.loc[]` and `.iloc[]` methods of indexing. To make your intention obvious, you should therefore adhere to the following rules:

1. Use `df['name']` only to select *columns* and nothing else.
2. Use `.loc[]` to select by label.
3. Use `.iloc[]` to select by position.

Selection by label

To illustrate, using `.loc[]` unambiguously indexes by label. First we create a demo data set with 3 columns and 5 rows:

[26]:
```
# Create demo data with 3 columns and 5 rows

# Column labels
columns = ['X', 'Y', 'Z']
# Row labels
rows = ['a', 'b', 'c', 'd', 'e']

values = np.arange(len(rows))
```

```python
# Create data dictionary
data = {col: [f'{col}{val}' for val in values] for col in columns}

# Create DataFrame from dictionary
df = pd.DataFrame(data, index=rows)
```

We now use `.loc[]` to select rows and columns by label:

[27]:
```python
# Select rows 'b' to 'e', and columns 'X' and 'Y'
df.loc["b":"e", ["X", "Y"]]
```

[27]:

	X	Y
b	X1	Y1
c	X2	Y2
d	X3	Y3
e	X4	Y4

With `.loc[]` we can even perform slicing on column names, which is not possible with the simpler `df[]` syntax:

[28]:
```python
df.loc['b':'e', 'X':'Z']
```

[28]:

	X	Y	Z
b	X1	Y1	Z1
c	X2	Y2	Z2
d	X3	Y3	Z3
e	X4	Y4	Z4

This includes all the columns between `X` and `Z`, where the latter is included since we are slicing by label.

Trying to pass in positional arguments will return an error for the given `DataFrame` since the index labels are `a`, `b`, `c`,... and not 0, 1, 2...

[29]:
```python
df.loc[0:4]
```

```
TypeError: cannot do slice indexing on Index with these indexers [0] of type int
```

However, we can reset the index to its default value. Then the index labels are integers and coincide with their position, so that `.loc[]` works:

[30]:
```python
df = df.reset_index(drop=True)      # reset index labels to integers,
                                    # drop original index
df.loc[0:4]
```

[30]:

	X	Y	Z
0	X0	Y0	Z0
1	X1	Y1	Z1
2	X2	Y2	Z2
3	X3	Y3	Z3
4	X4	Y4	Z4

Again, the end point with label `4` is included because we are selecting by label.

Indexing via `.loc[]` supports a few more types of arguments, see the official documentation for details.

Selection by position

Conversely, if we want to select items exclusively by their position and ignore their labels, we use `.iloc[]`:

[31]:
```python
df.iloc[0:4, 0:2]           # select first 4 rows, first 2 columns
```

[31]:
```
   X   Y
0  X0  Y0
1  X1  Y1
2  X2  Y2
3  X3  Y3
```

Again, `.iloc[]` supports a multitude of other arguments, see the official documentation for details.

Boolean indexing

Similar to NumPy, pandas allows us to select a subset of rows in a `Series` or `DataFrame` if they satisfy some condition. The simplest use case is to create a column of boolean values (`True` or `False`) as a result of some logical operation:

This even works without explicitly using the `.loc[]` attribute:

[32]:
```python
import pandas as pd

# Read in Titanic passenger data
df = pd.read_csv(f'{DATA_PATH}/titanic.csv')

# Check whether passenger embarked in Southampton
df['Embarked'] == "S"
```

[32]:
```
0       True
1      False
2       True
3       True
4       True
       ...
886     True
887     True
888     True
889    False
890    False
Name: Embarked, Length: 891, dtype: bool
```

Such boolean arrays can be used to select a subset of entries:

[33]:
```python
df.loc[df['Embarked'] == 'S', 'Name':'Age']
```

[33]:
```
                                           Name     Sex   Age
0                       Braund, Mr. Owen Harris    male  22.0
2                        Heikkinen, Miss Laina  female  26.0
3     Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
4                      Allen, Mr. William Henry    male  35.0
6                       McCarthy, Mr. Timothy J    male  54.0
..                                          ...     ...   ...
883               Banfield, Mr. Frederick James    male  28.0
884                      Sutehall, Mr. Henry Jr    male  25.0
886                       Montvila, Rev. Juozas    male  27.0
887                Graham, Miss Margaret Edith  female  19.0
888     Johnston, Miss Catherine Helen "Carrie"  female   NaN

[644 rows x 3 columns]
```

Boolean indexing also works directly with `[]` without having to specify `.loc[]`, but then it is not possible to also select a subset of columns at the same time:

[34]:
```python
df[df['Embarked'] == 'S']
```

[34]:
```
     PassengerId  Survived  Pclass  \
0              1         0       3
2              3         1       3
```

```
3              4       1      1
4              5       0      3
6              7       0      1
..           ...     ...    ...
883          884       0      2
884          885       0      3
886          887       0      2
887          888       1      1
888          889       0      3

                                             Name     Sex   Age  \
0                         Braund, Mr. Owen Harris    male  22.0
2                          Heikkinen, Miss Laina  female  26.0
3    Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
4                        Allen, Mr. William Henry    male  35.0
6                         McCarthy, Mr. Timothy J    male  54.0
..                                            ...     ...   ...
883                     Banfield, Mr. Frederick James    male  28.0
884                        Sutehall, Mr. Henry Jr    male  25.0
886                         Montvila, Rev. Juozas    male  27.0
887                  Graham, Miss Margaret Edith  female  19.0
888      Johnston, Miss Catherine Helen "Carrie"  female   NaN

               Ticket     Fare Cabin Embarked
0           A/5 21171   7.2500   NaN        S
2    STON/O2. 3101282   7.9250   NaN        S
3              113803  53.1000  C123        S
4              373450   8.0500   NaN        S
6               17463  51.8625   E46        S
..                ...      ...   ...      ...
883  C.A./SOTON 34068  10.5000   NaN        S
884   SOTON/OQ 392076   7.0500   NaN        S
886            211536  13.0000   NaN        S
887            112053  30.0000   B42        S
888        W./C. 6607  23.4500   NaN        S

[644 rows x 10 columns]
```

Multiple conditions can be combined using the `&` (logical and) or `|` (logical or) operators:

[35]:
```
# Select men who embarked in Southampton
df.loc[(df['Embarked'] == 'S') & (df['Sex'] == 'male'), ['Name', 'Embarked', 'Sex']]
```

[35]:
```
                              Name Embarked   Sex
0          Braund, Mr. Owen Harris        S  male
4         Allen, Mr. William Henry        S  male
6          McCarthy, Mr. Timothy J        S  male
7    Palsson, Master Gosta Leonard        S  male
12   Saundercock, Mr. William Henry        S  male
..                             ...      ...   ...
878              Laleff, Mr. Kristo        S  male
881              Markun, Mr. Johann        S  male
883   Banfield, Mr. Frederick James        S  male
884          Sutehall, Mr. Henry Jr        S  male
886           Montvila, Rev. Juozas        S  male

[441 rows x 3 columns]
```

If we want to include rows where an observation takes on one of multiple values, the `isin()` method can be used:

[36]:
```
# Select passengers who embarked in Southampton or Queenstown
df.loc[df['Embarked'].isin(('S', 'Q')), ['Name', 'Embarked']]
```

```
[36]:                                                Name Embarked
0                                 Braund, Mr. Owen Harris        S
2                                  Heikkinen, Miss Laina        S
3     Futrelle, Mrs. Jacques Heath (Lily May Peel)        S
4                                Allen, Mr. William Henry        S
5                                        Moran, Mr. James        Q
..                                                    ...      ...
885          Rice, Mrs. William (Margaret Norton)        Q
886                                 Montvila, Rev. Juozas        S
887                          Graham, Miss Margaret Edith        S
888        Johnston, Miss Catherine Helen "Carrie"        S
890                                   Dooley, Mr. Patrick        Q

[721 rows x 2 columns]
```

Finally, `DataFrame` implements a `query()` method which allows us to combine multiple conditions in a single string in an intuitive fashion. Column names can be used directly within this string to put restrictions on their values.

```
[37]: # Select passengers who embarked in Southampton and were above age 70
df.query('Embarked == "S" & Age > 70')
```

```
[37]:      PassengerId  Survived  Pclass                                  Name  \
630          631         1       1  Barkworth, Mr. Algernon Henry Wilson
851          852         0       3                   Svensson, Mr. Johan

      Sex   Age  Ticket    Fare Cabin Embarked
630  male  80.0   27042  30.000   A23        S
851  male  74.0  347060   7.775   NaN        S
```

Your turn. Load the Titanic passenger data set `data/titanic.csv` and select the following subsets of data:

1. Select all passengers with passenger IDs from 10 to 20
2. Select the 10th to 20th (inclusive) row of the dataframe
3. Using `query()`, select the sub-sample of female passengers aged 30 to 40. Display only the columns Name, Age, and Sex (in that order)
4. Repeat the last exercise without using `query()`
5. Select all men who embarked in Queenstown or Cherbourg

1.6 Working with time series data

In economics and finance, we frequently work with time series data, i.e., observations that are associated with a particular point in time (time stamp) or a time period. pandas offers comprehensive support for such data, in particular if the time stamp or time period is used as the index of a `Series` or `DataFrame`. This section presents a few of the most important concepts, see the official documentation for a comprehensive guide.

To illustrate, let's construct a set of daily data for the first three months of 2024, i.e., the period 2024-01-01 to 2024-03-31 using the `date_range()` function (we use the data format `YYYY-MM-DD` in this section, but pandas also supports other date formats).

```
[38]: import pandas as pd
import numpy as np

# Create sequence of dates from 2024-01-01 to 2024-03-31
```

```
# at daily frequency
index = pd.date_range(start="2024-01-01", end="2024-03-31", freq="D")

# Use date range as index for Series with some artificial data
data = pd.Series(np.arange(len(index)), index=index)

# Print first 5 observations
data.head(5)
```

[38]:
```
2024-01-01    0
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
Freq: D, dtype: int64
```

1.6.1 Indexing with date/time indices

pandas implements several convenient ways to select observations associated with a particular date or a set of dates. For example, if we want to select one specific date, we can pass it as a string to `.loc[]`:

[39]:
```
# Select single observation by date
data.loc["2024-01-01"]
```

[39]:
```
np.int64(0)
```

It is also possible to select a time period by passing a start and end point (where the end point is included, as usual with label-based indexing in pandas):

[40]:
```
# Select first 5 days
data.loc["2024-01-01":"2024-01-05"]
```

[40]:
```
2024-01-01    0
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
Freq: D, dtype: int64
```

A particularly useful way to index time periods is a to pass a partial index. For example, if we want to select all observations from January 2024, we could use the range `'2024-01-01':'2024-01-31'`, but it is much easier to specify the partial index `'2024-01'` instead which includes all observations from January.

[41]:
```
# Select all observations from January 2024
data.loc["2024-01"]
```

[41]:
```
2024-01-01     0
2024-01-02     1
2024-01-03     2
2024-01-04     3
2024-01-05     4
              ..
2024-01-27    26
2024-01-28    27
2024-01-29    28
2024-01-30    29
2024-01-31    30
Freq: D, Length: 31, dtype: int64
```

1.6.2 Lags, differences, and other useful transformations

When working with time series data, we often need to create lags or leads of a variable (e.g., if we want to include lagged values in a regression model). In pandas, this is done using `shift()` which shifts the index by the desired number of periods (default: 1). For example, invoking `shift(1)` creates lagged observations of each column in the `DataFrame`:

[42]:
```
# Lag observations by 1 period
data.shift(1).head(5)
```

[42]:
```
2024-01-01    NaN
2024-01-02    0.0
2024-01-03    1.0
2024-01-04    2.0
2024-01-05    3.0
Freq: D, dtype: float64
```

We can use the `diff()` method to compute differences over a given number of periods:

[43]:
```
# Compute difference between consecutive observations
data.diff(1).head(5)
```

[43]:
```
2024-01-01    NaN
2024-01-02    1.0
2024-01-03    1.0
2024-01-04    1.0
2024-01-05    1.0
Freq: D, dtype: float64
```

Note that `diff()` is identical to manually computing the difference with the lagged value like this:

```
data - data.shift()
```

Additionally, we can use `pct_change()` which computes the percentage change (the relative difference) over a given number of periods (default: 1).

[44]:
```
# Compute percentage change vs. previous period
data.pct_change().head(5)
```

[44]:
```
2024-01-01         NaN
2024-01-02         inf
2024-01-03    1.000000
2024-01-04    0.500000
2024-01-05    0.333333
Freq: D, dtype: float64
```

Again, this is just a convenience method that is a short-cut for manually computing the percentage change:

```
(data - data.shift()) / data.shift()
```

1.7 Retrieving data from the internet

1.7.1 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. There are two ways to achieve this:

1. Using the Terminal (or Anaconda Prompt on Windows), activate the conda environment you want to install `yfinance` into and enter the following command:

   ```
   pip install yfinance
   ```

2. In a Jupyter notebook, you can uncomment the code cell below to perform the installation. This needs to be done only once, so be sure to comment the line again the installation is complete.

[45]:
```
# Uncomment to install yfinance package
# ! 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.

[46]:
```
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:

[47]:
```
# 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: 4953.56 - 6147.43
```

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.

[48]:
```
# Retrieve daily index values data for 2024
daily = ticker.history(start='2024-01-01', end='2024-12-31')
```

The price history is returns as a `DataFrame` with several columns which include the price at open and clone, as well as the daily low and high:

[49]:
```
# Print first row
daily.head(1)
```

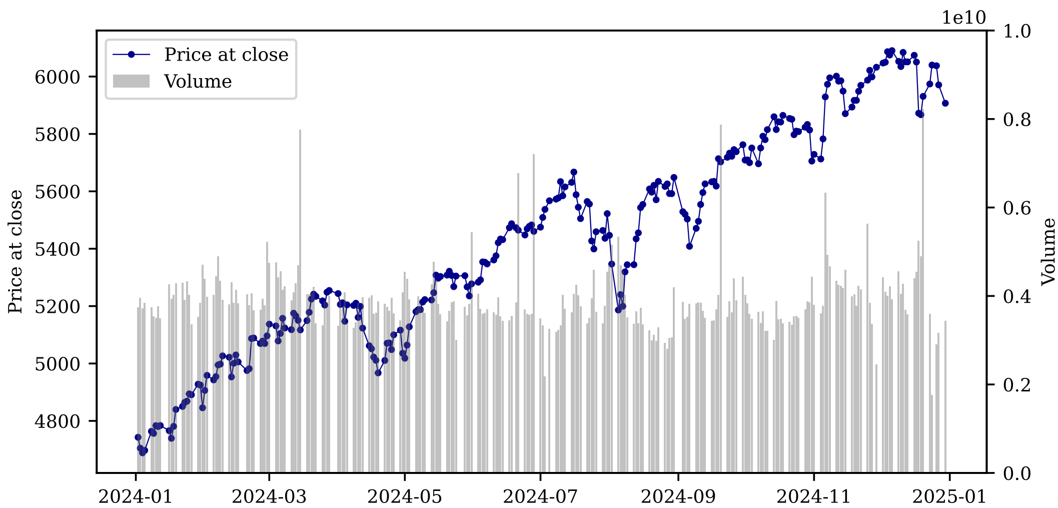
[49]:
```
                                  Open         High          Low        Close  \
Date
2024-01-02 00:00:00-05:00  4745.200195  4754.330078  4722.669922  4742.830078
```

```
                               Volume  Dividends  Stock Splits
Date
2024-01-02 00:00:00-05:00  3743050000        0.0           0.0
```

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

[50]:
```python
import matplotlib.pyplot as plt

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

# Plot closing price
price = ax.plot(daily.index, daily['Close'], color='darkblue', marker='o', ms=2, lw=0.5)
ax.set_ylabel('Price at close')

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

# Plot trading volume as bar chart
volume = 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')

# Add legend using handles returned by plot() and bar()
ax.legend([price[0], volume[0]], ['Price at close', 'Volume'])
```

[50]:
```
<matplotlib.legend.Legend at 0x7f195338c680>
```



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 last 3 months of 2024, we proceed as follows:

[51]:
```python
import yfinance as yf

# Ticker symbols for bulk download
tickers = 'AMZN', 'MSFT'
```

```
# Start date for data download
start = '2024-10-01'

# End date for data download
end = '2024-12-31'

# Get data for Amazon (AMZN) and Microsoft (MSFT) for first quarter of 2023
data = yf.download(tickers, start=start, end=end)
```

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

[52]:
```
# Inspect first few rows
data.head(2)
```

[52]:
```
Price            Close                    High                     Low   \
Ticker            AMZN        MSFT        AMZN        MSFT        AMZN
Date
2024-10-01  185.130005  419.009460  186.190002  426.768350  183.449997
2024-10-02  184.759995  415.463654  186.600006  421.130926  184.039993

Price                         Open                 Volume
Ticker            MSFT        AMZN        MSFT       AMZN      MSFT
Date
2024-10-01  417.136966  184.899994  426.738471  36044900  19092900
2024-10-02  415.045318  184.440002  420.891864  23704100  16582300
```

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

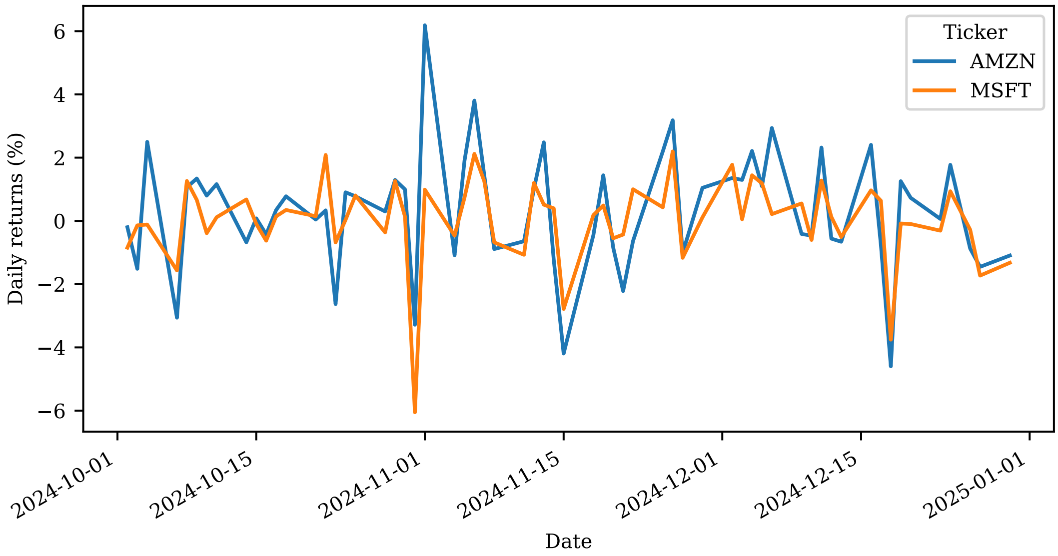
[53]:
```
# Use hierarchical indexing to get data for Amazon
data[('Close', 'AMZN')].head()
```

[53]:
```
Date
2024-10-01    185.130005
2024-10-02    184.759995
2024-10-03    181.960007
2024-10-04    186.509995
2024-10-07    180.800003
Name: (Close, AMZN), dtype: float64
```

To illustrate how this data can be used, we plot the daily returns for Amazon and Microsoft below:

[54]:
```
# Plot daily returns for both stocks
returns = data['Close'].pct_change() * 100.0
returns.plot(y=['AMZN', 'MSFT'], ylabel='Daily returns (%)', figsize=(7, 3.5))
```

[54]:
```
<Axes: xlabel='Date', ylabel='Daily returns (%)'>
```


Ticker
AMZN
MSFT
Daily returns (%)
6
4
2
0
−2
−4
−6
2024-10-01
2024-10-15
2024-11-01
2024-11-15
2024-12-01
2024-12-15
2025-01-01
Date