TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
NHH Norwegian School of Economics

August 2026

See GitHub repository for notebooks and data:
https://github.com/richardfoltyn/TECH2-H26

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 *homogeneous* 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-homogeneous 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 organized in *variables* and *observations*, similar to econometrics programs such as Stata, or R using `data.frame` objects.
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 homogeneous 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

# Create array of integers from 5 to 9
data = np.arange(5, 10)

# Create pandas Series from 1-dimensional NumPy 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 also create a `DataFrame` from a 2-dimensional NumPy array. To this end, we first create a 1-dimensional array of increasing integers from 0 to 14 and then use the `reshape()` method to reshape it to a 2-dimensional array.

[2]:
```
# Create 5-by-3 matrix of data
data = np.arange(15).reshape((-1, 3))

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

# Create pandas DataFrame from matrix (2-dimensional NumPy array)
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-homogeneous 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.

> **Your turn**
>
> Create a pandas `Series` which contains the characters `'a'`, `'b'`, and `'c'`.

1.3 Importing data

1.3.1 Loading data with NumPy & its limitations (optional)

Before turning to pandas, let's examine how one could import data using NumPy function. This also highlights the limitations of this approach.

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 NumPy 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 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'
```

```
# Alternatively, load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H26/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.
- `read_json()`, `to_json()`: Read or write JSON files (lightweight machine-readable text files, often produced by LLMs) 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 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.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    str
 4   Sex          891 non-null    str
 5   Age          714 non-null    float64
 6   Ticket       891 non-null    str
 7   Fare         891 non-null    float64
 8   Cabin        204 non-null    str
 9   Embarked     889 non-null    str
dtypes: float64(2), int64(3), str(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.

> **Your turn**
>
> Using the Titanic data set, tabulate the number of passengers by the port in which they boarded the ship (variable `Embarked`). How many observations have missing values for this variable?

1.4.1 Setting Pandas display options

It is possible to configure how much data Pandas displays by default using `set_option()`. There is a large number of display options you can set; see
here for the full list.

For our purposes, the following are the most important:

[15]:
```
# Set the line width to 120 characters (default: 80)
pd.set_option( 'display.width', 120 )
# Show up to 100 rows (default: 60)
pd.set_option('display.max_rows', 100)
```

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

[16]:
```
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
```

[16]:
```
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: str
```

[17]:
```
df[['Name', 'Sex']]  # select multiple columns using a list
```

[17]:
```
                                                  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

[18]:
```
df[1:4]
```

[18]:
```
   PassengerId  Survived  Pclass
Name     Sex   Age            Ticket  \
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs
Th...  female  38.0          PC 17599
2            3         1       3                              Heikkinen, Miss
Laina  female  26.0  STON/O2. 3101282
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May
Peel)  female  35.0            113803

      Fare Cabin Embarked
1  71.2833   C85        C
2   7.9250   NaN        S
3  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]`:

[19]:
```
import pandas as pd

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

[19]:
```
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:

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

[20]:
```
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:

[21]:
```
# Create DataFrame with 2 columns
df = pd.DataFrame({'A': [10, 20, 30], 'B': ['a', 'b', 'c']})

df
```

[21]:
```
    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()` to set the index to the values from a column, for example column `B`:

[22]:
```
# Use column 'B' as index, store result in new DataFrame
df2 = df.set_index('B')

# Display updated DataFrame
df2
```

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

Note that pandas operations often return a copy with the operation applied to that copy, leaving the original object unmodified. In the previous example, only `df2` uses column `B` as the index, whereas the original `df` remains unchanged:

[23]:
```
df
```

[23]:
```
    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:

[24]:
```
# Set index in-place, i.e.,  df is modified
df.set_index('B', inplace=True)

df
```

[24]:
```
    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`:

[25]:
```
# print first 2 rows using labels
df['a':'b']  # This is the same as df[:2]
```

[25]:
```
   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:

[26]:
```
# Reset index labels to default value (integers 0, 1, 2, ...) and print
# first three rows
df.reset_index(drop=True).head(3)
```

[26]:
```
   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.
>
> Perform the tasks using `inplace=False` and `inplace=True`. What's the difference?
> 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:

[27]:
```
# 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))

# Create data dictionary using a list comprehension
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. For example, to select the single row corresponding to the label `'b'` we specify:

[28]:
```
df.loc['b']
```

[28]:
```
X    X1
Y    Y1
Z    Z1
Name: b, dtype: str
```

To select a specific element by row and column labels, we proceed as follows:

[29]:
```
df.loc['b', 'X']
```

[29]:
```
'X1'
```

If you want to select multiple rows or columns, you can also pass lists as arguments to `.loc[]`:

[30]:
```
# Select rows 'b' and 'd', and columns 'X' and 'Z'
df.loc[['b', 'd'], ['X', 'Z']]
```

[30]:

	X	Z
b	X1	Z1
d	X3	Z3

Finally, it is possible to perform slicing on rows or columns using the `:` notation which you might have encountered when slicing lists, tuples, or NumPy arrays:

[31]:
```
# Select rows 'b' to 'e'
df.loc['b':'e']
```

[31]:

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

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

[32]:
```
# Select rows 'b' to 'e', and columns 'X' to 'Z'
df.loc['b':'e', 'X':'Z']
```

[32]:

	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...

[33]:
```
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:

[34]:
```
df = df.reset_index(drop=True)  # reset index labels to integers,
# drop original index
df.loc[0:4]
```

[34]:
```
    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[]`:

[35]:
```
df.iloc[0:4, 0:2]  # select first 4 rows, first 2 columns
```

[35]:
```
    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:

[36]:
```
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'
```

[36]:
```
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:

[37]:
```
df.loc[df['Embarked'] == 'S', 'Name':'Age']
```

[37]:
```
                                                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:

[38]:
```
df[df['Embarked'] == 'S']
```

[38]:
```
     PassengerId  Survived  Pclass                                               Name
Sex   Age            Ticket  \
0              1         0       3                            Braund, Mr. Owen Harris
male  22.0         A/5 21171
2              3         1       3                              Heikkinen, Miss Laina
female  26.0  STON/O2. 3101282
3              4         1       1    Futrelle, Mrs. Jacques Heath (Lily May Peel)
female  35.0            113803
4              5         0       3                           Allen, Mr. William Henry
male  35.0            373450
6              7         0       1                            McCarthy, Mr. Timothy J
male  54.0             17463
..           ...       ...     ...                                               ...
...   ...               ...
883          884         0       2                    Banfield, Mr. Frederick James
male  28.0  C.A./SOTON 34068
884          885         0       3                           Sutehall, Mr. Henry Jr
male  25.0   SOTON/OQ 392076
886          887         0       2                            Montvila, Rev. Juozas
male  27.0            211536
887          888         1       1                     Graham, Miss Margaret Edith
female  19.0            112053
888          889         0       3          Johnston, Miss Catherine Helen "Carrie"
female   NaN        W./C. 6607

        Fare Cabin Embarked
0     7.2500   NaN        S
2     7.9250   NaN        S
3    53.1000  C123        S
4     8.0500   NaN        S
6    51.8625   E46        S
..       ...   ...      ...
883  10.5000   NaN        S
```

```
884   7.0500   NaN        S
886  13.0000   NaN        S
887  30.0000   B42        S
888  23.4500   NaN        S

[644 rows x 10 columns]
```

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

[39]:
```
# Select men who embarked in Southampton
df.loc[(df['Embarked'] == 'S') & (df['Sex'] == 'male'), ['Name', 'Embarked', 'Sex']]
```

[39]:
```
                              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:

[40]:
```
# Select passengers who embarked in Southampton or Queenstown
df.loc[df['Embarked'].isin(('S', 'Q')), ['Name', 'Embarked']]
```

[40]:
```
                                          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.

[41]:
```
# Select passengers who embarked in Southampton and were above age 70
df.query('Embarked == "S" & Age > 70')
```

[41]:
```
     PassengerId  Survived  Pclass                                  Name   Sex
Age  Ticket    Fare Cabin Embarked
630          631         1       1  Barkworth, Mr. Algernon Henry Wilson  male
80.0   27042  30.000   A23        S
851          852         0       3                  Svensson, Mr. Johan  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 10^{th} to 20^{th} (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 2025, i.e., the period 2025-01-01 to 2025-03-31 using the `date_range()` function (we use the date format `YYYY-MM-DD` in this section, but pandas also supports other date formats).

[42]:
```
import numpy as np
import pandas as pd

# Create sequence of dates from 2025-01-01 to 2025-03-31
# at daily frequency
index = pd.date_range(start='2025-01-01', end='2025-03-31', 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)
```

[42]:
```
2025-01-01    0
2025-01-02    1
2025-01-03    2
2025-01-04    3
2025-01-05    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[]`:

[43]:
```
# Select single observation by date
data.loc['2025-01-01']
```

[43]:
```
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):

[44]:
```
# Select first 5 days
data.loc['2025-01-01':'2025-01-05']
```

[44]:
```
2025-01-01    0
2025-01-02    1
2025-01-03    2
2025-01-04    3
2025-01-05    4
Freq: D, dtype: int64
```

A particularly useful way to index time periods is to pass a partial index. For example, if we want to select all observations from January 2025, we could use the range `'2025-01-01':'2025-01-31'`, but it is much easier to specify the partial index `'2025-01'` instead which includes all observations from January.

[45]:
```
# Select all observations from January 2025
data.loc['2025-01']
```

[45]:
```
2025-01-01     0
2025-01-02     1
2025-01-03     2
2025-01-04     3
2025-01-05     4
              ..
2025-01-27    26
2025-01-28    27
2025-01-29    28
2025-01-30    29
2025-01-31    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`:

[46]:
```
# Lag observations by 1 period
data.shift(1).head(5)
```

[46]:
```
2025-01-01    NaN
2025-01-02    0.0
2025-01-03    1.0
2025-01-04    2.0
2025-01-05    3.0
Freq: D, dtype: float64
```

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

[47]:
```
# Compute difference between consecutive observations
data.diff(1).head(5)
```

[47]:
```
2025-01-01    NaN
2025-01-02    1.0
2025-01-03    1.0
2025-01-04    1.0
2025-01-05    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).

[48]:
```
# Compute percentage change vs.  previous period
data.pct_change().head(5)
```

[48]:
```
2025-01-01         NaN
2025-01-02         inf
2025-01-03    1.000000
2025-01-04    0.500000
2025-01-05    0.333333
Freq: D, dtype: float64
```

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

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

Your turn

Use the data from the `data/FRED` folder to perform the following task:

1. Read in the file `FRED_annual.csv` and set the column `Year` as the index.
2. Compute annual inflation as the percentage change of the consumer price index (column `CPI`).

1.7 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [[go to section]] links to the relevant section in this notebook.

1.7.1 Creating pandas containers from existing data

- `pd.Series()` — create a series containing observations of one variable [go to section]
- `pd.DataFrame()` — create a DataFrame containing observations of multiple variables [go to section]

1.7.2 Loading existing data

- `pd.read_csv()` — read CSV text file [go to section]
- `pd.read_excel()` — read Excel file [go to section]

1.7.3 Inspecting data

- `info()` — print information about observation count, columns, and data types [go to section]
- `head()` — print first few rows [go to section]
- `tail()` — print last few rows [go to section]
- `describe()` — print summary statistics for *numerical* data [go to section]

- `value_counts()` — tabulate observation counts for categorical data [go to section]

1.7.4 Manipulating the index

- `pd.Series(..., index=...)` — create Series with a custom index [go to section]
- `pd.DataFrame(..., index=...)` — create DataFrame with a custom index [go to section]
- `set_index()` — set existing columns as the index [go to section]
- `reset_index()` — reset index to default value [go to section]

1.7.5 Selecting observations

- `loc[]` — select elements by *label* (values of the index) [go to section]
- `iloc[]` — select elements by *position* [go to section]
- `isin()` — return whether elements are contained in a set of values [go to section]
- `query()` — select subset of observations according to some criteria [go to section]

1.7.6 Time series

- `date_range()` — create sequence of dates [go to section]
- `shift()` — shift the index by desired number of observations or time periods [go to section]
- `diff()` — calculate *absolute* difference vs. a previous/next element [go to section]
- `pct_change()` — calculate *relative* difference vs. a previous/next element [go to section]

2 Plotting with Matplotlib

In this lecture, we study how to plot numerical data. Python itself does not have any built-in plotting capabilities, so we will be using `matplotlib`, the most popular graphics library for Python.

- For details on a particular plotting function, see the official documentation.
- There is an official introductory tutorial which you can use alongside the material presented here.

In order to access the functions and objects from Matplotlib, we first need to import them. The general convention is to use the namespace `plt` for this purpose:

```
import matplotlib.pyplot as plt
```

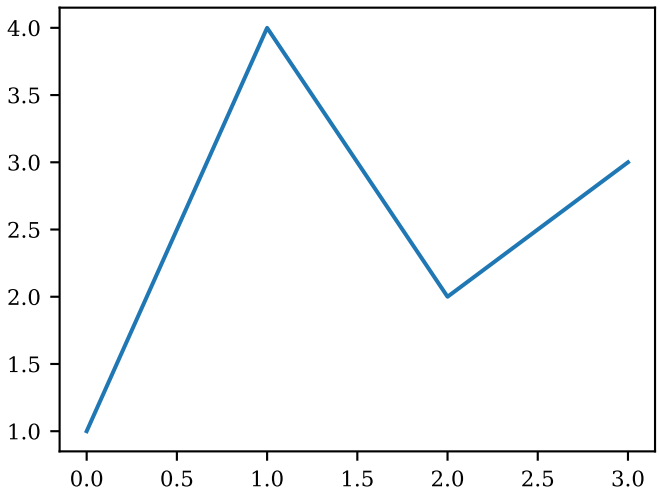
Note that there is an additional high-level plotting library called seaborn which builds on top of Matplotlib with a focus on providing convenient functions to create statistical graphs.

2.1 Line plots

One of the simplest plots we can generate using the `plot()` function is a line defined by a list of y-values.

[1]:
```
# import matplotlib library
import matplotlib.pyplot as plt

# Plot list of integers
yvalues = [1, 4, 2, 3]
plt.plot(yvalues)
```

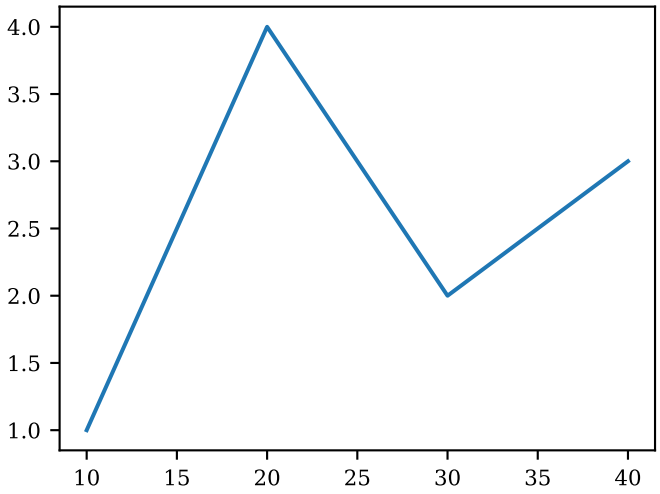
[1]:
```
[<matplotlib.lines.Line2D at 0x7fd34122f8c0>]
```



We didn't even have to specify the corresponding x-values, as Matplotlib automatically assumes them to be `[0, 1, 2, ...]`. Usually, we want to plot for a given set of x-values like this:

[2]:
```
# explicitly specify x-values
xvalues = [10, 20, 30, 40]
```

```
_ = plt.plot(xvalues, yvalues)
```



Your turn

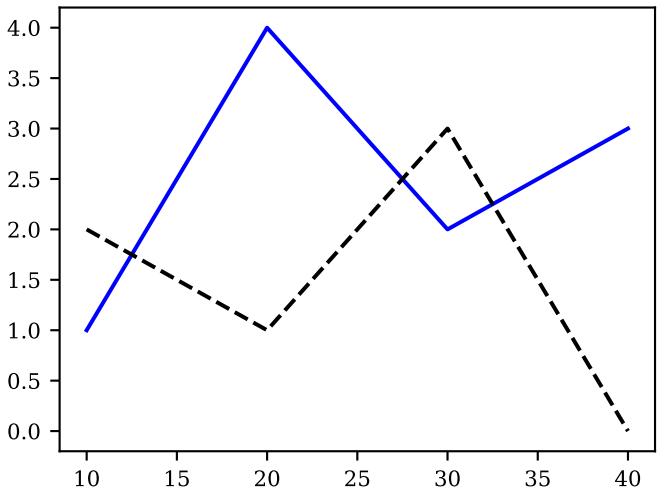
Plot the quadratic function $y = x^2$ on the interval `[-1, 1]`:

1. Create a grid of 50 x-values using `np.linspace()`.
2. Compute the corresponding y-values and plot them.

2.1.1 Plotting multiple lines

We can also specify multiple lines to be plotted in a single graph:

[3]:
```
yvalues2 = [2.0, 1.0, 3.0, 0.0]
_ = plt.plot(xvalues, yvalues, 'b-', xvalues, yvalues2, 'k--')
```



2.1.2 Specifying plot styles

The characters following each set of y-values are style specifications. The letters are shorthand notations for colors (see here for details):

- `b`: blue
- `g`: green
- `r`: red
- `c`: cyan
- `m`: magenta
- `y`: yellow
- `k`: black
- `w`: white

The remaining characters set the line styles. Valid values are:

- `-` solid line
- `--` dashed line
- `-.` dash-dotted line
- `:` dotted line

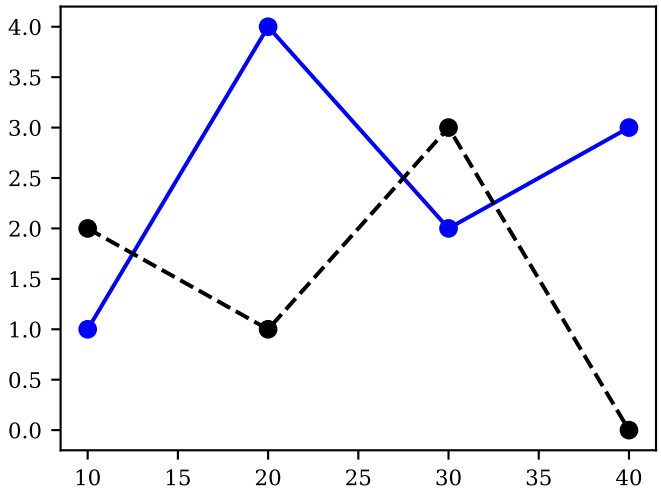
Additionally, we can append marker symbols to the style specification. The most frequently used ones are:

- `o`: circle
- `s`: square
- `*`: star
- `x`: x
- `d`: (thin) diamond

The whole list of supported symbols can be found here.

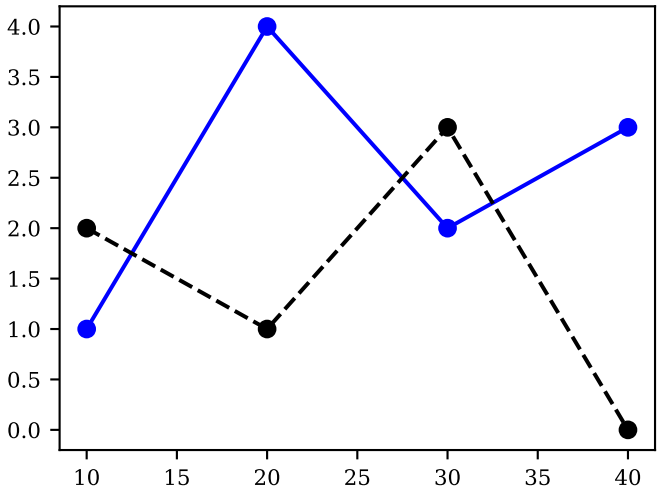
Instead of passing multiple values to be plotted at once, we can also repeatedly call `plot()` to add additional elements to a graph. This is more flexible since we can pass additional arguments which are specific to one particular set of data, such as labels displayed in legends.

[4]:
```
# Plot two lines by calling plot() twice
plt.plot(xvalues, yvalues, 'b-o')
_ = plt.plot(xvalues, yvalues2, 'k--o')
```



Individual calls to `plot()` also allow us to specify styles more explicitly using keyword arguments:

[5]:
```
# pass plot styles as explicit keyword arguments
plt.plot(xvalues, yvalues, color='blue', linestyle='-', marker='o')
_ = plt.plot(xvalues, yvalues2, color='black', linestyle='--', marker='o')
```



Note that in the example above, we use named colors such as `'red'` or `'blue'` (see here for the complete list of named colors).

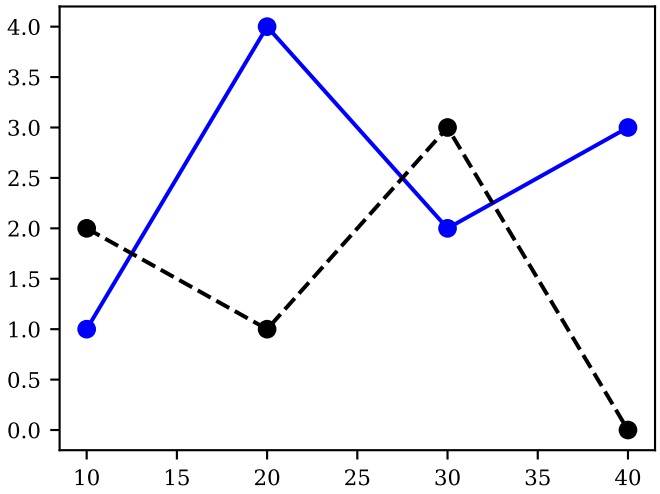
Matplotlib accepts abbreviations for the most common style definitions using the following shortcuts:

- `c` or `color`
- `ls` or `linestyle`
- `lw` or `linewidth`
- `ms` or `markersize`

See the section on *Other Parameters* in the `plot()` documentation for a complete list of arguments and their abbreviations.

We can thus rewrite the above code as follows:

[6]:
```
# abbreviate plot style keywords
plt.plot(xvalues, yvalues, c='blue', ls='-', marker='o')
_ = plt.plot(xvalues, yvalues2, c='black', ls='--', marker='o')
```



Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

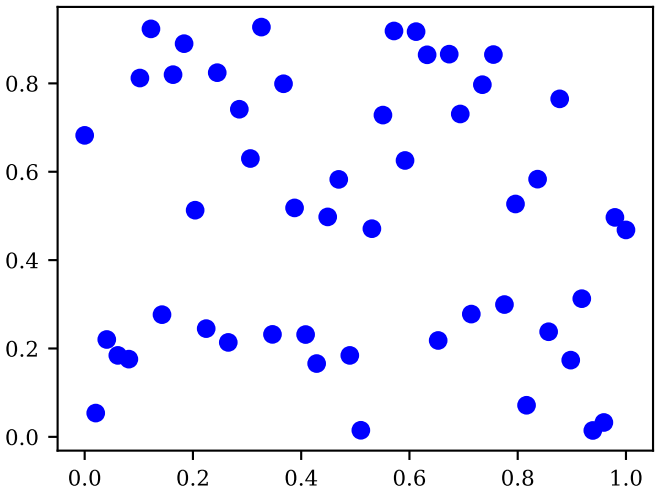
1. Load the data from `FRED_annual.csv`. The file contains annual observations on selected macroeconomic variables for the US.
2. Plot the unemployment rate (column `UNRATE`) using a blue dashed line with line width 0.5 and the inflation rate (column `INFLATION`) using an orange line with line width 0.75 in the *same* figure.

2.2 Scatter plots

We use the `scatter()` function to create scatter plots in a similar fashion to line plots.

To illustrate, we first generate some random data using the NumPy Random Number Generator (RNG) interface. We won't go into detail how to generate random numbers with NumPy, so you can just take that code as give.

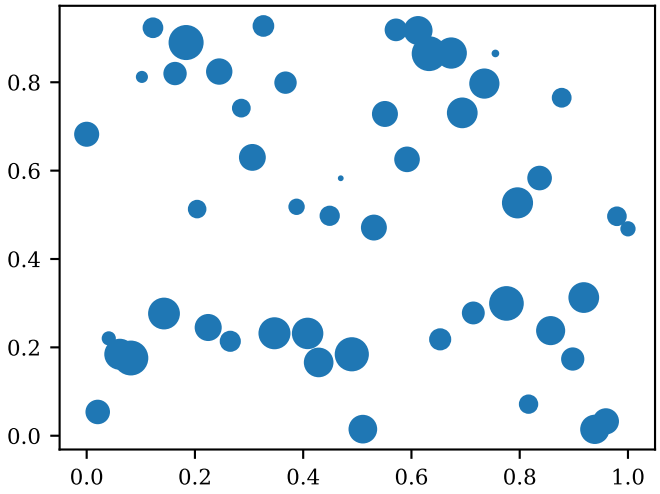
[7]:
```
import matplotlib.pyplot as plt
import numpy as np

# Number of points
N = 50

# Create 50 uniformly-spaced values on the unit interval
xvalues = np.linspace(0.0, 1.0, N)
# Draw random numbers from interval [0, 1)
yvalues = np.random.default_rng(seed=123).random(N)

_ = plt.scatter(xvalues, yvalues, color='blue')
```



We could in principle create scatter plots using `plot()` by turning off the connecting lines. However, `scatter()` allows us to specify the color and marker size as collections, so we can vary these for every point. `plot()`, on the other hand, imposes the same style on all points plotted in that particular function call.

To illustrate, the following code assigns a different (randomly drawn) marker size to each dot:

[8]:
```
# Draw random marker sizes
size = np.random.default_rng(seed=456).random(N) * 150.0

# plot with point-specific marker sizes
_ = plt.scatter(xvalues, yvalues, s=size)
```



Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

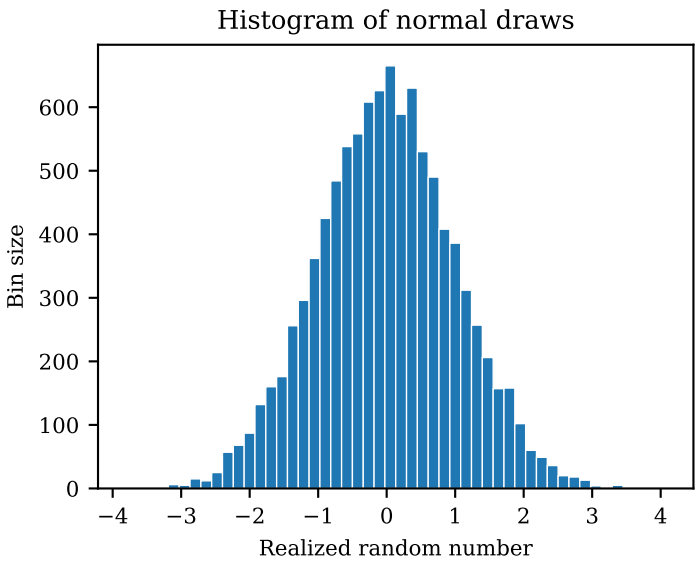
1. Load the data in `FRED_monthly.csv`. The file contains monthly observations on selected macroeconomic variables for the US.
2. Create a scatter plot of the real interest rate (column `REALRATE`) on the y-axis against the Federal Funds rate (column `FEDFUNDS`) on the x-axis. Specify the arguments `edgecolors='blue'` and `color='none'` to plot the data as blue rings.

2.3 Histograms

We use the `hist()` function to create histograms which can be used to nonparametrically visualize the distribution of some sample data.

To illustrate, we draw a random sample from the standard-normal distribution and visualize it using a histogram:

[9]:
```
import matplotlib.pyplot as plt
import numpy as np

# Draw 10,000 standard-normal numbers
x = np.random.default_rng(seed=1234).normal(size=10_000)

# Plot the results as a histogram
plt.hist(x, bins=50, linewidth=0.5, edgecolor='white')
plt.xlabel('Realized random number')
plt.ylabel('Bin size')
_ = plt.title('Histogram of normal draws')
```

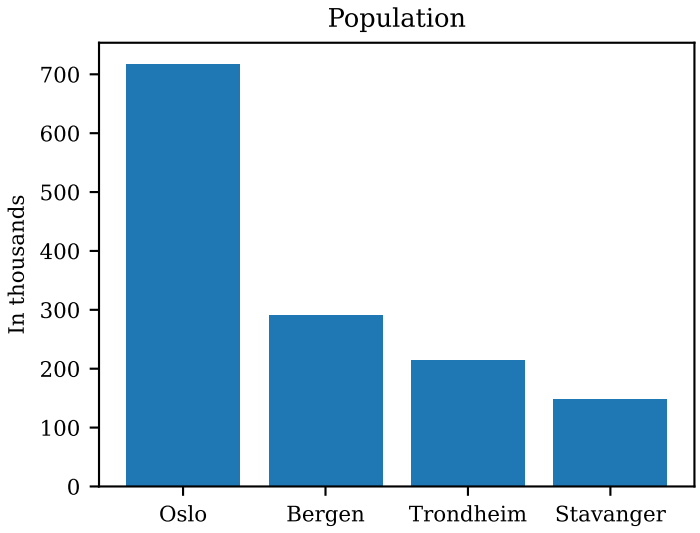


2.4 Plotting categorical data

Instead of continuous numerical values on the x-axis, we can also plot categorical variables as bar charts using the function `bar()`.

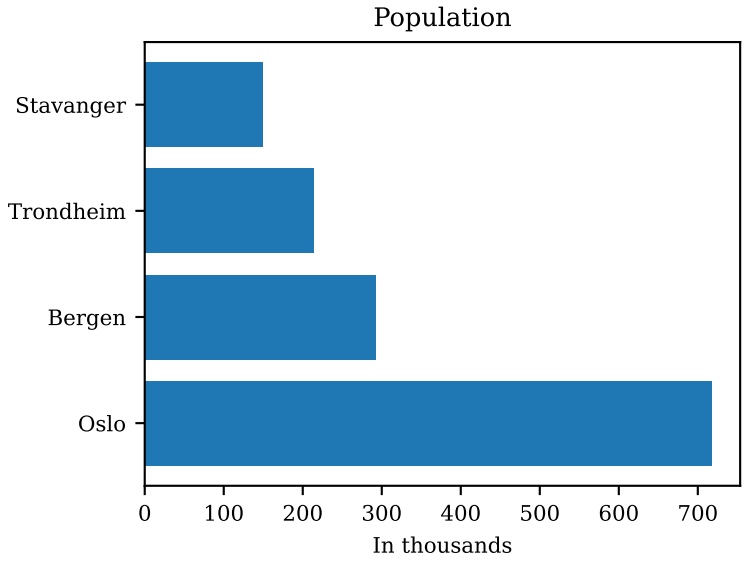
To illustrate, the following example creates a bar chart showing the population numbers of the four largest municipalities in Norway:

[10]:
```
import matplotlib.pyplot as plt

# Define category labels
municipality = ['Oslo', 'Bergen', 'Trondheim', 'Stavanger']
# Population in thousandss
population = np.array([717710, 291940, 214565, 149048]) / 1000

# Create bar chart
plt.bar(municipality, population)

# Add overall title
plt.title('Population')
_ = plt.ylabel('In thousands')
```



We use `barh()` to create *horizontal* bars:

[11]:
```
plt.barh(municipality, population)
plt.title('Population')
_ = plt.xlabel('In thousands')
```



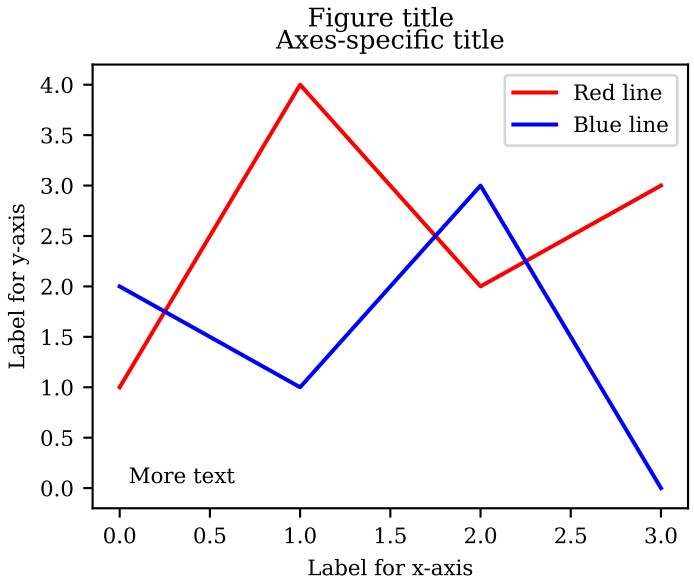
2.5 Adding labels and annotations

Matplotlib has numerous functions to add labels and annotations:

- Use `title()` and `suptitle()` to add titles. The latter adds a title for the whole figure, which might span multiple plots (axes).
- We can add axis labels by calling `xlabel()` and `ylabel()`.
- To add a legend, call `legend()`, which in its most simple form takes a list of labels which are in the same order as the plotted data.
- Use `text()` to add additional text at arbitrary locations.

Example: Add title, labels, and a legend to some demo data

[12]:
```
import matplotlib.pyplot as plt

# Demo data used for plotting
xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]

plt.plot(xvalues, yvalues, color='red')
plt.plot(xvalues, yvalues2, color='blue')

# Add titles, labels, legend, and text
plt.suptitle('Figure title')
plt.title('Axes-specific title')
plt.xlabel('Label for x-axis')
plt.ylabel('Label for y-axis')
plt.legend(['Red line', 'Blue line'])
# Adds text at data coordinates (0.05, 0.05)
_ = plt.text(0.05, 0.05, 'More text')
```



2.6 Plot limits, ticks and tick labels

We adjust the plot limits, ticks and tick labels as follows:

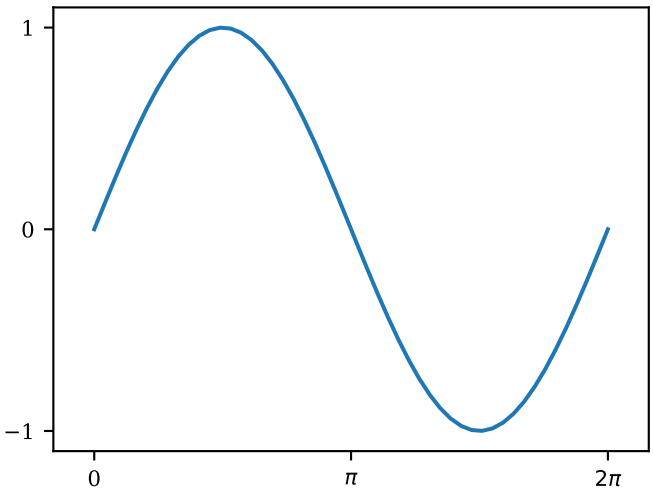
- Plotting limits are set using the `xlim()` and `ylim()` functions. Each accepts a tuple `(min,max)` to set the desired range.
- Ticks and tick labels can be set by calling `xticks()` or `yticks()`.

Example: Adding horizontal and vertical lines

We demonstrate the use of these functions by plotting the sine function on the interval $[0, 2\pi]$:

[13]:
```
import matplotlib.pyplot as plt
import numpy as np

# Create data using the sine function
xvalues = np.linspace(0.0, 2 * np.pi, 50)
```

```
yvalues = np.sin(xvalues)

plt.plot(xvalues, yvalues)

# Set major ticks and labels for x-axis
#   1. Define tick positions
xticks = [0.0, np.pi, 2 * np.pi]
#   2. Define tick labels (can use LaTeX code!)
xtick_labels = ['0', r'$\pi$', r'$2\pi$']
plt.xticks(xticks, xtick_labels)

# Set major ticks for y-axis
plt.yticks([-1.0, 0.0, 1.0])

# Adjust plot limits in x and y direction
plt.xlim((-0.5, 2 * np.pi + 0.5))
_ = plt.ylim((-1.1, 1.1))
```



2.7 Adding straight lines

Quite often, we want to add horizontal or vertical lines to highlight a particular value. We can do this using the following functions:

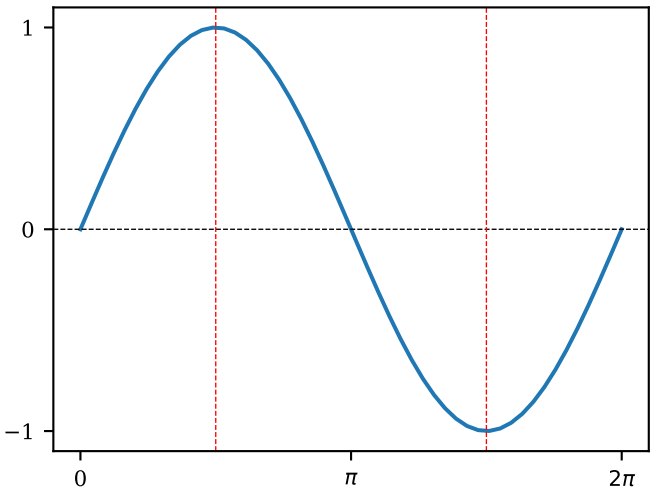
- `axhline()` adds a *horizontal* line at a given y-value.
- `axvline()` adds a *vertical* line at a given x-value.
- `axline()` adds a line defined by two points or by a single point and a slope.

Example: Adding horizontal and vertical lines

Consider the sine function from above. We can add a horizontal line at 0 and two vertical lines at the points where the function attains its minimum and maximum as follows:

[14]:
```
import matplotlib.pyplot as plt
import numpy as np

# Plot sine function (same as above)
xvalues = np.linspace(0.0, 2 * np.pi, 50)
yvalues = np.sin(xvalues)

plt.plot(xvalues, yvalues)
```

```
# Set major ticks and labels for x-axis
#   1. Define tick positions
xticks = [0.0, np.pi, 2 * np.pi]
#   2. Define tick labels (can use LaTeX code!)
xtick_labels = ['0', r'$\pi$', r'$2\pi$']
plt.xticks(xticks, xtick_labels)

# Set major ticks for y-axis
plt.yticks([-1.0, 0.0, 1.0])

# Add black dashed horizontal line at y-value 0
plt.axhline(0.0, lw=0.5, ls='--', c='black')

# Add red dashed vertical lines at maximum / minimum points
plt.axvline(0.5 * np.pi, lw=0.5, ls='--', c='red')
_ = plt.axvline(1.5 * np.pi, lw=0.5, ls='--', c='red')
```



Your turn

Plot the function $y = (x-1)^2$ on the interval $[0, 2]$:

1. Create a grid of 50 x-values using `np.linspace()`.
2. Compute and plot the y-values.
3. Label the x- and y-axes with "x" and "y".
4. Add the title `"y = (x-1)^2"`.
5. Add a vertical line at $x = 1$ using a dashed line style.

2.8 Object-oriented interface

So far, we have only used the so-called `pyplot` interface which involves calling *global* plotting functions from `matplotlib.pyplot`. This interface is intended to be similar to Matlab, but is also somewhat limited and less clean.

We can instead use the object-oriented interface (called this way because we call methods of the `Figure` and `Axes` objects). While there is not much point in using the object-oriented interface in a Jupyter

notebook when we want to create a single graph, it should be the preferred method when writing reusable code in Python files or when creating a figure with multiple subplots.

To use the object-oriented interface, we need to get figure and axes objects. The easiest way to accomplish this is using the `subplots()` function, like this:

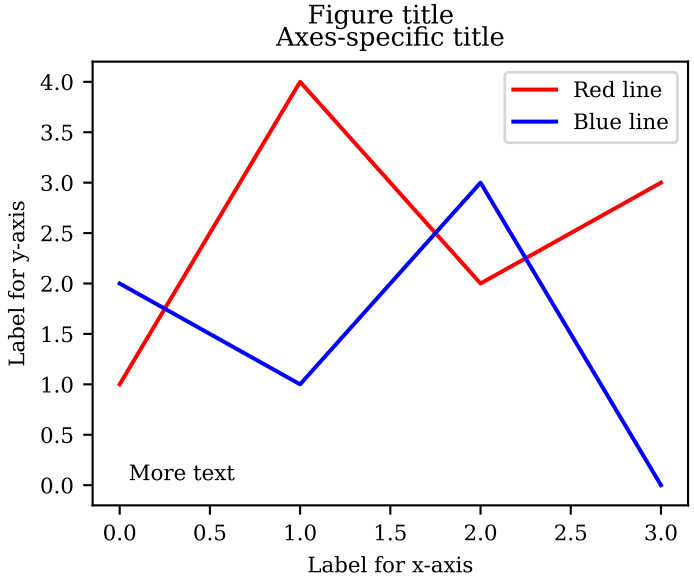
```
fig, ax = plt.subplots()
```

We then use methods of the `Axes` object returned by `subplots()` instead of the functions we have used so far. For example, instead of `plot()`, we use the `Axes.plot()` method.

As an example, we recreate the graph from the section on labels and annotations using the object-oriented interface:

[15]:
```
# Demo data used for plotting
xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]
```

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

# Create Figure and Axes objects
fig, ax = plt.subplots()

ax.plot(xvalues, yvalues, color='red', label='Red line')
ax.plot(xvalues, yvalues2, color='blue', label='Blue line')

# Add titles, labels, legend, and text
fig.suptitle('Figure title')  # Alternative: plt.suptitle()
ax.set_title('Axes-specific title')  # Alternative: plt.title()
ax.set_xlabel('Label for x-axis')  # Alternative: plt.xlabel()
ax.set_ylabel('Label for y-axis')  # Alternative: plt.ylabel()
ax.legend(['Red line', 'Blue line'])  # Alternative: plt.legend()
# Adds text at data coordinates (0.05, 0.05)
_ = ax.text(0.05, 0.05, 'More text')  # Alternative: plt.text()
```



The code is quite similar to the previous section, except that attributes are set using the `set_xxx()` methods of the `ax` object. For example, instead of calling `xlim()`, we use `ax.set_xlim()`.

Your turn

Recall the example from above in which we drew from a normal distribution and visualized the sample using a histogram.

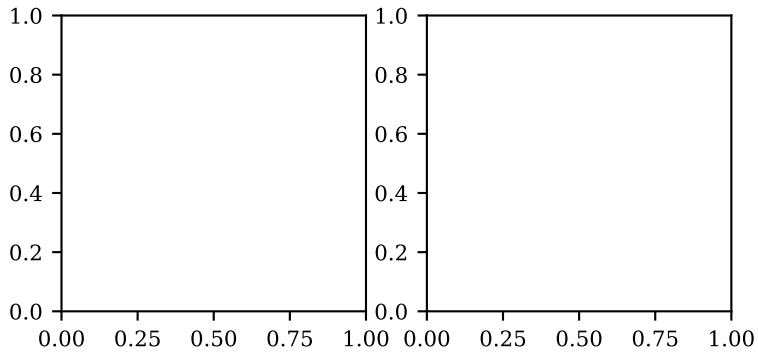
1. Repeat these steps, but create the histogram using the object-oriented Matplotlib interface.
2. Set the x-axis label to 'Realized random number' and the y-axis label to 'Number of observations'.
3. Add the title 'Histogram of normal draws'.

2.9 Working with multiple plots (axes)

The object-oriented interface becomes particularly useful if we want to create multiple axes (or figures). This can also be achieved using the `pyplot` programming model but is somewhat more obscure.

For example, to create a row with two subplots, we use:

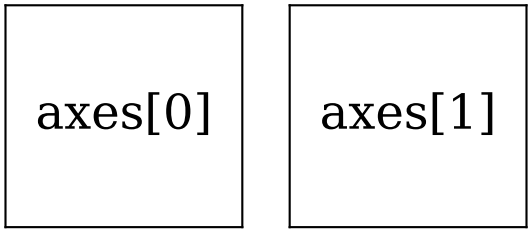
[17]:
```
import matplotlib.pyplot as plt

# Create one figure with 2 axes objects, arranged as two columns in a single row
fig, axes = plt.subplots(1, 2, figsize=(4.5, 2.0))
```



With multiple axes objects in a single figure (as in the above example), the `axes` object returned by `subplots()` is a NumPy array. Its elements map to the individual panels within the figure in a natural way.

We can visualize this mapping for the case of a single row and two columns as follows:

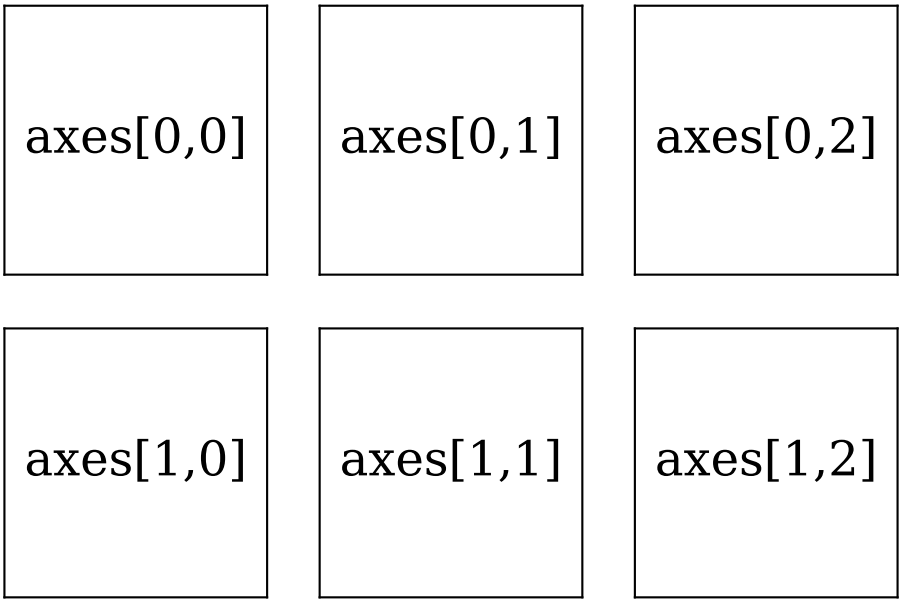
[18]:
```
fig, axes = plt.subplots(1, 2, figsize=(3.5, 1.5))

for i, ax in enumerate(axes):
    # Turn off ticks of both axes
    ax.set_xticks(())
    ax.set_yticks(())
    # Label axes object
    text = f'axes[{i}]'
    ax.text(0.5, 0.5, text, va='center', ha='center', fontsize=18)
```



Don't worry about the details of how this graph is generated; the only takeaway here is how axes objects are mapped to the panels in the figure.

If we request panels in two dimensions, the `axes` object will be a 2-dimensional array, and the mapping of axes objects to panels will look like this instead:

[19]:
```
# Create figure with 2 rows, 3 columns
nrow = 2
ncol = 3
fig, axes = plt.subplots(nrow, ncol, figsize=(6, 4))

for i in range(nrow):
    for j in range(ncol):
        # Get reference to current Axes object
        ax = axes[i, j]
        # Turn off ticks of both axes
        ax.set_xticks(())
        ax.set_yticks(())
        # Label axes object
        text = f'axes[{i},{j}]'
        ax.text(0.5, 0.5, text, va='center', ha='center', fontsize=18)
```

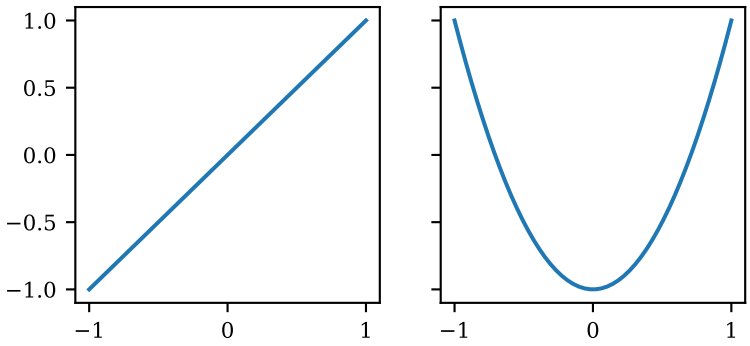


Example: Create a plot with 2 panels

We can use the elements of `axes` to plot into individual panels:

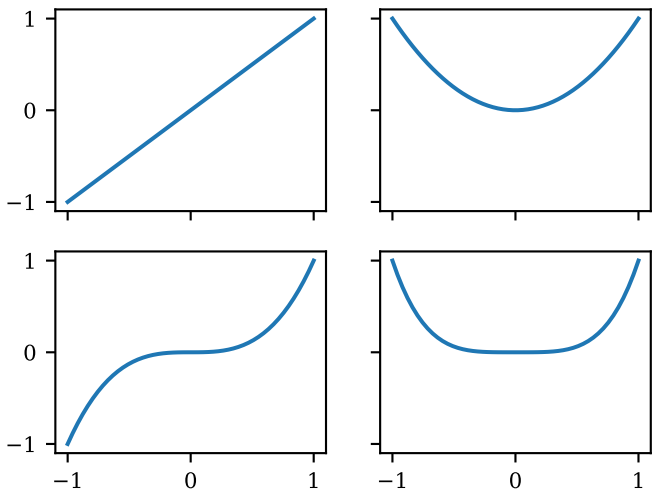
[20]:
```
import matplotlib.pyplot as plt
import numpy as np

# Request a figure with 2 panels (1 row, 2 columns)
```

```
fig, axes = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(4.5, 2.0))

xvalues = np.linspace(-1.0, 1.0, 50)

# Plot into first column
axes[0].plot(xvalues, xvalues)

# Plot into second column
_ = axes[1].plot(xvalues, 2 * xvalues**2.0 - 1)
```



Example: Create a figure with 2 rows and 2 columns

[21]:
```
# create figure with 2 rows, 2 columns
nrow = 2
ncol = 2
fig, axes = plt.subplots(nrow, ncol, sharex=True, sharey=True)

xvalues = np.linspace(-1.0, 1.0, 50)

# Plot the first four powers of x
exponent = 1
for i in range(nrow):
    for j in range(ncol):
        yvalues = xvalues**exponent
        axes[i, j].plot(xvalues, yvalues)
        # Increment exponent for next subplot
        exponent += 1
```



Note the use of `sharex=True` and `sharey=True`. This tells Matplotlib that all axes share the same plot limits, so the tick labels can be omitted in the figure's interior to preserve space.

Your turn

Create a figure with 3 columns (on a single row) and plot the following functions on the interval $[0, 6]$:

1. Subplot 1: $y = \sin(x)$
2. Subplot 2: $y = \sin(2 \cdot x)$
3. Subplot 3: $y = \sin(4 \cdot x)$

Hint: The sine function can be imported from NumPy as `np.sin()`.

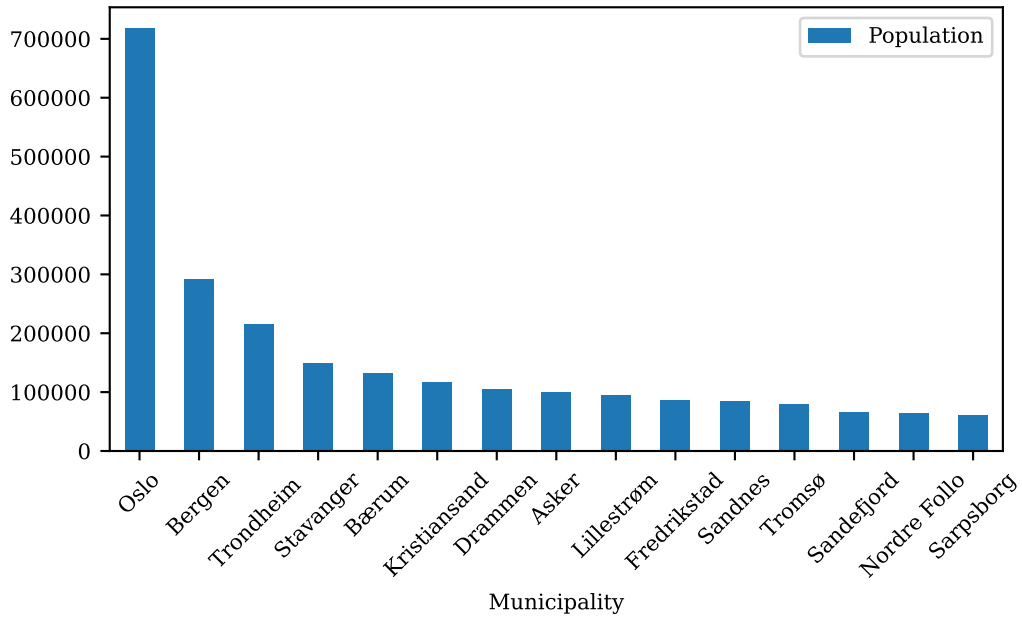
2.10 Plotting with pandas

Pandas does not implement its own graphics library, but provides convenient wrappers around Matplotlib functions that can be used to quickly visualize data stored in DataFrames. Alternatively, we can extract the numerical data and pass it to Matplotlib's routines manually.

2.10.1 Bar charts

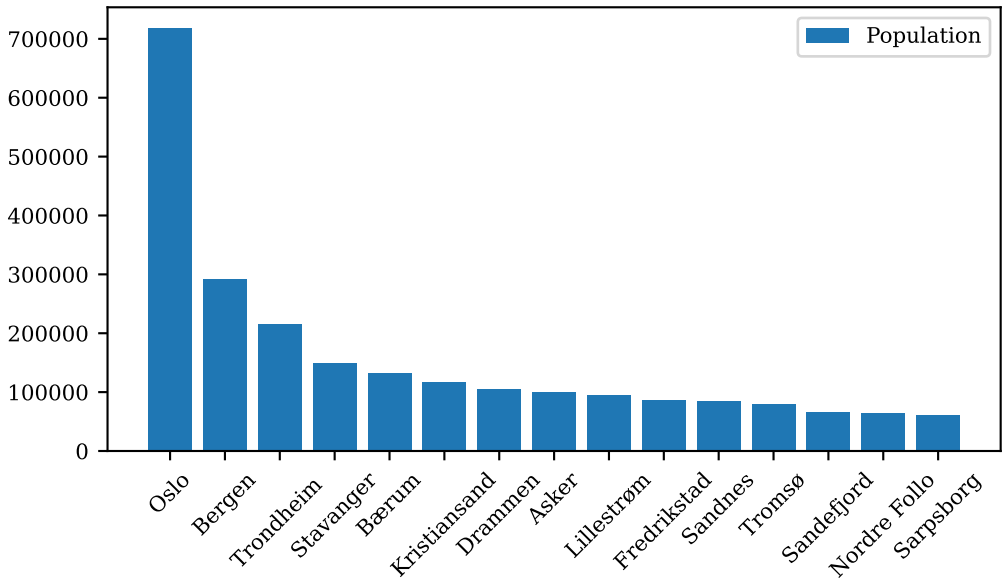
Let's return to our municipality population data. To plot population numbers as a bar chart, we can directly use pandas's `plot.bar()`:

[22]:
```
import pandas as pd

# Path to local data/ folder
DATA_PATH = '../../data'

# Path to population data
filepath = f'{DATA_PATH}/population_norway.csv'

# Read in population data for Norwegian municipalities
df = pd.read_csv(filepath)

# Keep only the first 15 observations
df = df.iloc[:15]

# Create bar chart, specify figure size, rotate x-axis tick labels by 45 degrees
_ = df.plot.bar(x='Municipality', y='Population', rot=45, figsize=(6, 3))
```



Alternatively, we can construct the graph ourselves using Matplotlib:

[23]:
```
import matplotlib.pyplot as plt

# Extract municipality names
labels = df['Municipality']

# Extract population numbers
values = df['Population']

# Create new figure with desired size
plt.figure(figsize=(6, 3))

# Create bar chart
plt.bar(labels, values)

# Add legend
plt.legend(['Population'])

# Rotate tick labels by 45 degrees
plt.tick_params(axis='x', labelrotation=45)
```

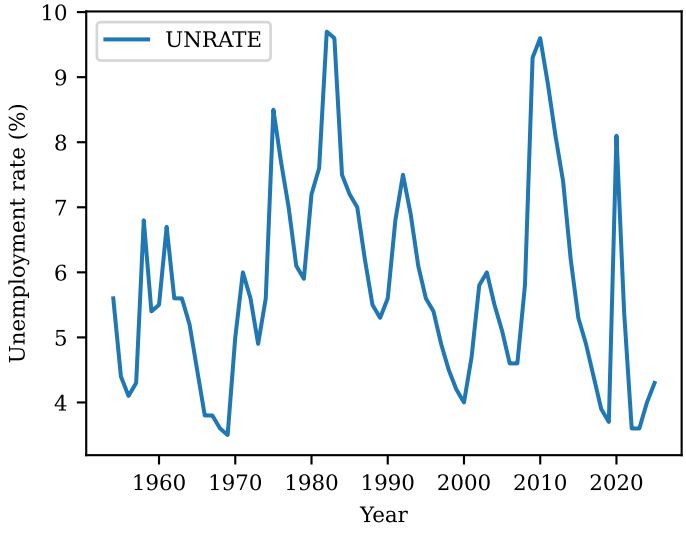


Matplotlib's functions usually work directly with pandas' data structures. In cases where they don't, we can convert a `DataFrame` or `Series` object to a NumPy array using the `to_numpy()` method.

2.10.2 Plotting time series data

To plot time series data, we can use the `DataFrame.plot()` method which optionally accepts arguments to specify which columns should be used for the x-axis and which for the y-axis. We illustrate this using the US unemployment rate at annual frequency.

[24]:
```
import pandas as pd

# Path to FRED data; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED/FRED_annual.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

# Plot unemployment rate by year
_ = df.plot(x='Year', y='UNRATE', ylabel='Unemployment rate (%)')
```



Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

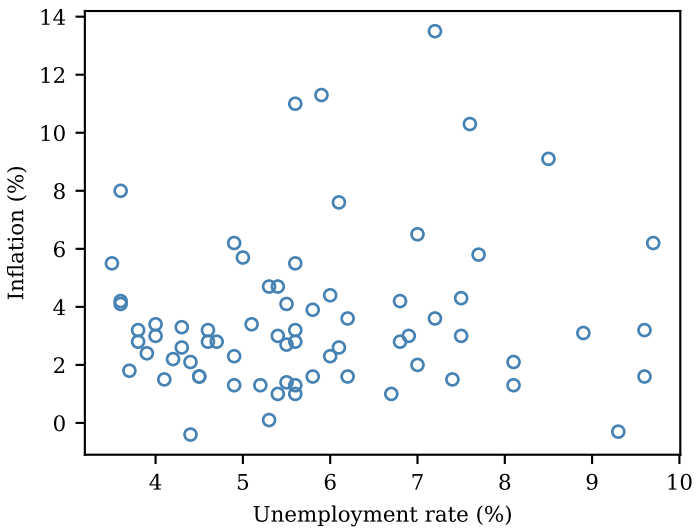
1. Load the macroeconomic time series data from `FRED_monthly_all.csv`. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'], index_col='DATE')` to automatically parse strings stored in the `DATE` column as dates and set `DATE` as the index.
2. Create a line plot, showing both the unemployment rate `UNRATE` and the inflation rate `INFLATION` in a single graph.

2.10.3 Scatter plots

Using the `DataFrame.plot.scatter()` method, we can generate scatter plots, plotting one column against another. To illustrate, we plot the US unemployment rate against inflation in any given year over the post-war period.

Note that you can pass additional arguments (for example `edgecolors`) to pandas' version of `scatter()` which are passed on to Matplotlib's `scatter()`.

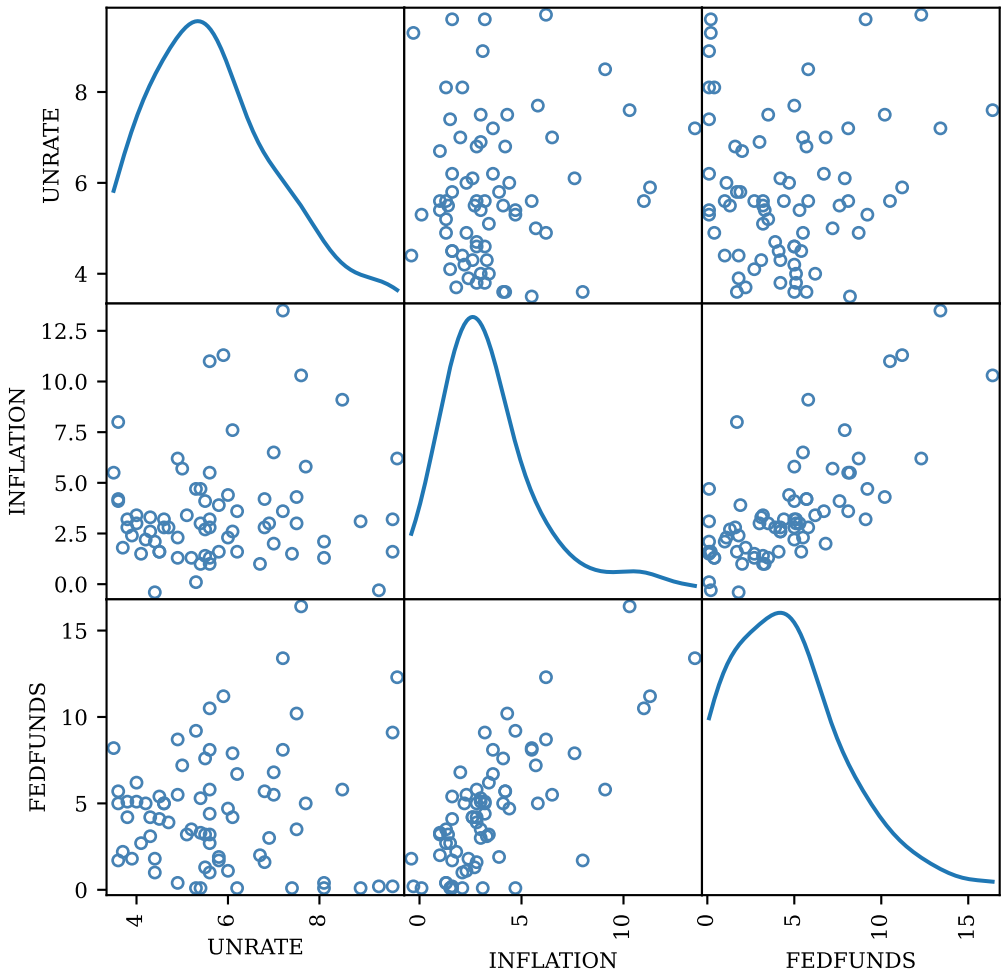
[25]:
```
# Path to FRED.csv; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED/FRED_annual.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

_ = df.plot.scatter(
    x='UNRATE',  # plot unemployment rate on x-axis
    y='INFLATION',  # plot inflation rate on y-axis
    color='none',
    edgecolors='steelblue',
    xlabel='Unemployment rate (%)',
    ylabel='Inflation (%)',
)
```



Pandas also offers the convenience function `scatter_matrix()` which lets us easily create pairwise scatter plots for more than two variables:

[26]:
```
from pandas.plotting import scatter_matrix
```

```
# Columns to include in plot
columns = ['UNRATE', 'INFLATION', 'FEDFUNDS']

# Use argument diagonal='kde' to plot kernel density estimate
# in diagonal panels
ax = scatter_matrix(
    df[columns],
    figsize=(6, 6),
    diagonal='kde',  # plot kernel density along diagonal
    s=70,  # marker size
    color='none',
    edgecolors='steelblue',
    alpha=1.0,
)
```

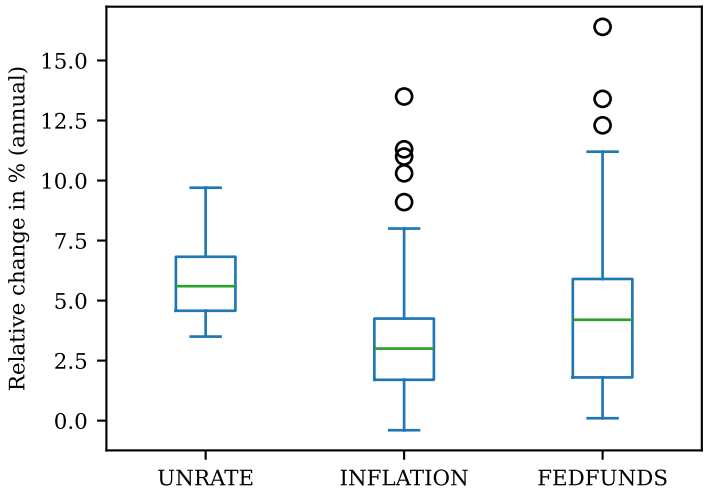


2.10.4 Box plots

To quickly plot some descriptive statistics, we can use the `DataFrame.plot.box()` provided by pandas. This plot shows the median, the interquartile range (25th to 75th percentile) and the outliers of some underlying data.

We demonstrate this by plotting the distribution of the unemployment rate, inflation and the Federal Funds Rate in the US:

[27]:
```
import pandas as pd

# Path to FRED data; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED/FRED_annual.csv'

# Read CSV data
df = pd.read_csv(filepath, sep=',')

# Include only the following columns in plot
columns = ['UNRATE', 'INFLATION', 'FEDFUNDS']

# Create box plot. Alternatively, use df.plot(kind='box')
_ = df[columns].plot.box(ylabel='Relative change in % (annual)')
```



2.11 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [go to section] links to the relevant section in this notebook.

2.11.1 Plotting functions

- `plot()` — plot data using lines [go to section]
- `scatter()` — create scatter plots [go to section]
- `hist()` — create histograms [go to section]
- `bar()` — create (vertical) bar charts [go to section]
- `barh()` — create *horizontal* bar charts [go to section]
- `axhline()` — adds a *horizontal* line at a given y-value [go to section]
- `axvline()` — adds a *vertical* line at a given x-value [go to section]

2.11.2 Plotting with pandas

- `DataFrame.plot()` — create a line plot (the default) from DataFrame contents [go to section]
- `DataFrame.plot.bar()` — create a bar chart from DataFrame contents [go to section]
- `DataFrame.plot.scatter()` — create a scatter plot from DataFrame contents [go to section]
- `scatter_matrix()` — create a n-by-n matrix of scatter plots from DataFrame contents [go to section]
- `DataFrame.plot.box()` — create a box plot from DataFrame contents [go to section]

2.11.3 Figures with multiple plots

- `subplots()` — create figure and (one or multiple) axes objects [go to section]

2.11.4 Annotations (labels, legends, . . .)

- `title()` — add a title for a single sub-figure [go to section]
- `suptitle()` — add a title for the whole figure [go to section]
- `xlabel()` — add a label for the x-axis [go to section]
- `ylabel()` — add a label for the y-axis [go to section]
- `legend()` — add a legend [go to section]
- `text()` — add arbitrary text at some location [go to section]
- `xticks()` — specify ticks and tick labels for the x-axis [go to section]
- `yticks()` — specify ticks and tick labels for the y-axis [go to section]

2.11.5 Other customizations

- `xlim()` — specify plotting limits (lower and upper bounds) for the x-axis [go to section]
- `ylim()` — specify plotting limits (lower and upper bounds) for the y-axis [go to section]

3 Grouping and aggregation with pandas

3.1 Aggregation and reduction

Similar to NumPy, pandas supports data aggregation and reduction functions such as computing sums or averages. By *"aggregation"* or *"reduction"* we mean that the result of a computation has a lower dimension than the original data: for example, the mean reduces a series of observations (1 dimension) into a scalar value (0 dimensions).

Unlike NumPy, these operations can be applied to subsets of the data which have been grouped according to some criterion.

Such operations are often referred to as *split-apply-combine* (see the official user guide) as they involve these three steps:

1. *Split* data into groups based on some criteria;
2. *Apply* some function to each group separately; and
3. *Combine* the results into a single `DataFrame` or `Series`.

See also the pandas cheat sheet for an illustration of such operations.

We first set the path pointing to the folder which contains the data files used in this lecture.

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

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H26/main/data'
```

3.1.1 Aggregations of whole Series or DataFrames

The simplest way to perform data reduction is to invoke the desired function on the entire `DataFrame`. We illustrate this using the Titanic dataset which we have encountered in the previous lectures.

We first load the data and tabulate the columns present in the `DataFrame`:

[2]:
```
import pandas as pd

# Path to Titanic passenger data CSV file
file = f'{DATA_PATH}/titanic.csv'

# Read in Titanic passenger data, set PassenderId column as index
df = pd.read_csv(f'{DATA_PATH}/titanic.csv', index_col='PassengerId')

# Tabulate columns and number of observations
df.info(show_counts=True)
```

```
<class 'pandas.DataFrame'>
RangeIndex: 891 entries, 1 to 891
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  891 non-null    int64
 1   Pclass    891 non-null    int64
```

```
 2   Name      891 non-null    str
 3   Sex       891 non-null    str
 4   Age       714 non-null    float64
 5   Ticket    891 non-null    str
 6   Fare      891 non-null    float64
 7   Cabin     204 non-null    str
 8   Embarked  889 non-null    str
dtypes: float64(2), int64(2), str(5)
memory usage: 62.8 KB
```

We can now apply the `mean()` method to all numerical columns to compute the average for each column:

[3]:
```
# Compute mean of all numerical columns
df.mean(numeric_only=True)
```

[3]:
```
Survived     0.383838
Pclass       2.308642
Age         29.699118
Fare        32.204208
dtype: float64
```

Methods such as `mean()` are by default applied column-wise to each column. The `numeric_only=True` argument is used to discard all non-numeric columns (depending on the version of pandas, `mean()` will issue a warning if there are non-numerical columns in the `DataFrame`).

One big advantage over NumPy is that missing values (represented by `np.nan`) are automatically ignored, as the following code demonstrates:

[4]:
```
import numpy as np

# mean() automatically drops missing observations
mean_pandas = df['Age'].mean()

# np.mean() returns NaN since some ages are missing (coded as NaN)
mean_numpy = np.mean(df['Age'].to_numpy())

print(f'Mean using Pandas: {mean_pandas:.1f}')
print(f'Mean using NumPy:  {mean_numpy:.1f}')
```

```
Mean using Pandas: 29.7
Mean using NumPy:  nan
```

For this reason, NumPy implements an additional set of aggregation functions which drop NaNs, for example `np.nanmean()`.

> **Your turn**
>
> Use the Titanic dataset to perform the following aggregations:
>
> 1. Compute the median age (column `Age`).
> 2. Compute the fraction of female passengers (use the information in column `Sex`).

3.1.2 Aggregations of subsets of data (grouping)

Applying aggregation functions to the entire `DataFrame` is similar to what we can do with NumPy. The added flexibility of pandas becomes obvious once we want to apply these functions to subsets of data, i.e., groups which we define based on column values or index labels.

For example, consider the categorical variable `Pclass` (passenger class). We can tabulate the categories and their corresponding observation count as follows:

[5]:
```
df['Pclass'].value_counts().sort_index()
```

[5]:
```
Pclass
1    216
2    184
3    491
Name: count, dtype: int64
```

We now want to compute the averages of numerical variables (`Age`, etc.) separately for each passenger class. Instead of looping over categories, this can be achieved more elegantly with `groupby()`:

[6]:
```
# Group observations by accommodation class (first, second, third)
groups = df.groupby(['Pclass'])
```

The return value `groups` is a special pandas object which can subsequently be used to obtain group-specific data. To compute the group-wise averages, we can simply run

[7]:
```
groups.mean(numeric_only=True)
```

[7]:
```
        Survived        Age       Fare
Pclass
1       0.629630  38.233441  84.154687
2       0.472826  29.877630  20.662183
3       0.242363  25.140620  13.675550
```

Groups support column indexing: if we want to only compute the total fare paid by passengers in each class, we can do this as follows:

[8]:
```
groups['Fare'].sum()
```

[8]:
```
Pclass
1    18177.4125
2     3801.8417
3     6714.6951
Name: Fare, dtype: float64
```

Built-in aggregations

There are numerous routines to aggregate grouped data, for example:

- `mean()`: compute average within each group
- `sum()`: sum values within each group
- `std()`, `var()`: within-group standard deviation and variance
- `median()`: compute median within each group
- `quantile()`: compute quantiles within each group
- `size()`: number of observations in each group
- `count()`: number of non-missing observations in each group
- `first()`, `last()`: first and last elements in each group
- `min()`, `max()`: minimum and maximum elements within a group

See the official documentation for a complete list.

Example: Number of elements within each group

We use `size()` and `count()` to compute group sizes or get the number of non-missing observations:

[9]:
```
groups.size()  # return number of elements in each group
```

[9]:
```
Pclass
1    216
2    184
3    491
dtype: int64
```

Note that `size()` and `count()` are two *different* functions. The former returns the group sizes (and the return value is a `Series`), whereas `count()` returns the number of non-missing observations for *each* column:

[10]:
```
groups.count()  # return number of non-missing observations for each column
```

[10]:
```
        Survived  Name  Sex  Age  Ticket  Fare  Cabin  Embarked
Pclass
1            216   216  216  186     216   216    176       214
2            184   184  184  173     184   184     16       184
3            491   491  491  355     491   491     12       491
```

Example: Return first observation of each group

We use `first()` to select the first observation within each group. This can be useful if we are working with time series data and want to get the first observation in a period, or if some variable is constant within groups and we need to select a single value per group.

[11]:
```
groups[
    ['Survived', 'Age', 'Sex', 'Fare']
].first()  # return first observation in each group
```

[11]:
```
        Survived   Age     Sex     Fare
Pclass
1              1  38.0  female  71.2833
2              1  14.0  female  30.0708
3              0  22.0    male   7.2500
```

Your turn

Use the Titanic dataset to perform the following aggregations:

1. Compute the average survival rate by sex (stored in the `Sex` column). You need to use the survival indicator stored in the column `Survived` for this.
2. Count the number of passengers aged 50+. Compute the average survival rate by sex for this group.
3. Count the number of passengers below the age of 20 by class and sex. Compute the average survival rate for this group by class and sex.

Writing custom aggregations

We can create custom aggregation routines by calling `agg()` (shorthand for `aggregate()`) on the grouped object. These functions operate on one column at a time, so it is only possible to use observations from that column for computations.

For example, we could alternatively call the built-in aggregation functions listed above (`mean()`, `std()`, etc.) via `agg()`:

[12]:
```
# Calculate group means in needlessly complicated way
groups['Age'].agg('mean')

# More direct approach:
# groups["Age"].mean()
```

[12]:
```
Pclass
1    38.233441
2    29.877630
3    25.140620
Name: Age, dtype: float64
```

There is no advantage of doing this over directly calling `mean()` in this particular case, but `agg()` will come in handy if we need to apply *multiple* operations in a single call, as we'll see below.

On the other hand, we *have to* use `agg()` if there is no built-in function to perform the desired aggregation. To illustrate, imagine that we want to compute the fraction of passengers aged 40+ in each class. There is no built-in function to achieve this, so we could use `agg()` combined with a custom function to perform the desired aggregation:

[13]:
```
import numpy as np

# Apply a custom aggregation using a lambda expression
groups['Age'].agg(lambda x: np.sum(x >= 40) / len(x))
```

[13]:
```
Pclass
1    0.375000
2    0.201087
3    0.091650
Name: Age, dtype: float64
```

In this example, we use a `lambda` expression to define the custom aggregation function in place. This is a shorthand notation which is equivalent to defining a custom function first:

[14]:
```
# Define a custom aggregation function
def fcn(x):
    return np.sum(x >= 40) / len(x)


# Apply the custom aggregation function using a function
groups['Age'].agg(fcn)
```

[14]:
```
Pclass
1    0.375000
2    0.201087
3    0.091650
Name: Age, dtype: float64
```

Note that we called `agg()` only on the column `Age`, otherwise the function would be applied to every column separately, which is not what we want.

Applying multiple functions at once

It is possible to apply multiple functions in a single call by passing a list of functions. These can be passed as strings or as callables (functions).

*Example: Applying multiple functions to a **single** column*

To compute the mean and median passenger age by class, we proceed as follows:

[15]:
```
groups['Age'].agg(['mean', 'median'])
```

[15]:
```
              mean  median
Pclass
1       38.233441    37.0
2       29.877630    29.0
3       25.140620    24.0
```

Note that we could have also specified these functions by passing references to the corresponding NumPy functions instead:

[16]:
```
groups['Age'].agg([np.mean, np.median])
```

[16]:
```
              mean  median
Pclass
1       38.233441     NaN
2       29.877630     NaN
3       25.140620     NaN
```

The following more advanced syntax allows us to create new column names using existing columns and some operation:

```
groups.agg(
    new_column_name1=('column_name1', 'operation1'),
    new_column_name2=('column_name2', 'operation2'),
    ...
)
```

This is called "named aggregation" as the keywords determine the output column *names*.

*Example: Applying multiple functions to **multiple** columns*

The following code computes the average age and the highest fare in a single aggregation:

[17]:
```
groups.agg(average_age=('Age', 'mean'), max_fare=('Fare', 'max'))
```

[17]:
```
        average_age  max_fare
Pclass
1         38.233441  512.3292
2         29.877630   73.5000
3         25.140620   69.5500
```

Finally, the most flexible aggregation method is `apply()` which calls a given function, passing the *entire* group-specific subset of data (including all columns) as an argument. You need to use `apply()` if data from more than one column is required to compute a statistic of interest.

Your turn

Use the Titanic dataset to perform the following aggregations:

1. Compute the minimum, maximum and average age by embarkation port (stored in the column `Embarked`) in a single `agg()` operation. There are several ways to solve this problem.
2. Compute the number of passengers, the average age and the fraction of women by embarkation port in a single `agg()` operation. *Hint:* To compute the fraction of women, you can either use a `lambda` expression, or you first create a numerical indicator variable for females (as we did in the workshop).

3.2 Transformations

In the previous section, we combined grouping and reduction, i.e., data at the group level was reduced to a single statistic such as the mean. Alternatively, we can combine grouping with the `transform()`

function which assigns the result of a computation to each observation within a group and consequently leaves the number of observations unchanged.

For example, for *each* observation we could compute the average fare by class as follows:

[18]:
```
df['Avg_Fare'] = df.groupby('Pclass')[['Fare']].transform('mean')

# Print class, fare and average fare by class for first 10 passengers
df[['Pclass', 'Fare', 'Avg_Fare']].head(10)
```

[18]:

PassengerId	Pclass	Fare	Avg_Fare
1	3	7.2500	13.675550
2	1	71.2833	84.154687
3	3	7.9250	13.675550
4	1	53.1000	84.154687
5	3	8.0500	13.675550
6	3	8.4583	13.675550
7	1	51.8625	84.154687
8	3	21.0750	13.675550
9	3	11.1333	13.675550
10	2	30.0708	20.662183

As you can see, instead of collapsing the `DataFrame` to only 3 observations (one for each class), the number of observations remains the same, and the average fare is constant within each class.

When would we want to use `transform()` instead of aggregation? Such use cases arise whenever we want to perform computations that include the individual value as well as an aggregate statistic.

Example: Deviation from average fare

Assume that we want to compute how much each passenger's fare differed from the average fare in their respective class. We could compute this using `transform()` as follows:

[19]:
```
# Compute difference of passenger's fare and avg. fare paid within class
df['Fare_Diff'] = df['Fare'] - df.groupby('Pclass')['Fare'].transform('mean')

# Print relevant columns
df[['Pclass', 'Fare', 'Fare_Diff']].head(10)
```

[19]:

PassengerId	Pclass	Fare	Fare_Diff
1	3	7.2500	-6.425550
2	1	71.2833	-12.871387
3	3	7.9250	-5.750550
4	1	53.1000	-31.054687
5	3	8.0500	-5.625550
6	3	8.4583	-5.217250
7	1	51.8625	-32.292187
8	3	21.0750	7.399450
9	3	11.1333	-2.542250
10	2	30.0708	9.408617

Your turn

Use the Titanic dataset to perform the following aggregations:

1. Compute the *excess* fare paid by each passenger relative to the minimum fare by embarkation port and class, i.e., compute $Fare - min(Fare)$ by port and class.

3.3 Resampling and aggregation

We discussed how to handle time series data in pandas in the previous lecture. This basically comes down to specifying an index which is a date or time stamp. Such an index allows us to easily perform operations such as computing leads, lags, and differences over time.

Another useful feature of the time series support in pandas is *resampling* which is used to group observations by time period and apply some aggregation function. This can be accomplished using the `resample()` method which in its simplest form takes a string argument that describes how observations should be grouped (`'YE'` for aggregation to years, `'QE'` for quarters, `'ME'` for months, `'W'` for weeks, etc.).

To illustrate, we load a dataset that contains daily observations on the value of the NASDAQ stock market index at close:

[20]:
```
# Path to NASDAQ data file
file = f'{DATA_PATH}/stockmarket/NASDAQ.csv'

# Read in NASDAQ data, set Date column as index
df = pd.read_csv(file, index_col='Date', parse_dates=True)

# Keep observations for 2025
df = df.loc['2025']

# Print first few rows
df.head()
```

[20]:
```
             NASDAQ
Date
2025-01-02  19280.8
2025-01-03  19621.7
2025-01-06  19865.0
2025-01-07  19489.7
2025-01-08  19478.9
```

For example, if we want to aggregate this daily data to monthly frequency, we would use `resample('ME')`. This returns an object which is very similar to the one returned by `groupby()` we studied previously, and we can call various aggregation methods such as `mean()`:

[21]:
```
# Resample to monthly frequency, aggregate to mean of daily observations
# within each month
df.resample('ME').mean()
```

[21]:
```
                NASDAQ
Date
2025-01-31  19565.045000
2025-02-28  19547.284211
2025-03-31  17828.028571
2025-04-30  16678.466667
2025-05-31  18642.361905
2025-06-30  19673.300000
2025-07-31  20784.900000
2025-08-31  21383.590476
2025-09-30  22188.600000
2025-10-31  23003.134783
2025-11-30  23023.626316
2025-12-31  23383.771429
```

Similarly, we can use `resample('W')` to resample to weekly frequency. Below, we combine this with the aggregator `last()` to return the last observation of each week (weeks by default start on Sundays):

[22]:
```
# Return last observation of each week, print first 10 rows
df.resample('W').last().head(10)
```

[22]:
```
             NASDAQ
Date
2025-01-05  19621.7
2025-01-12  19161.6
2025-01-19  19630.2
2025-01-26  19954.3
2025-02-02  19627.4
2025-02-09  19523.4
2025-02-16  20026.8
2025-02-23  19524.0
2025-03-02  18847.3
2025-03-09  18196.2
```

Your turn

Use the daily NASDAQ data for 2025 and compute the percentage change from the first to the last trading day within each month.

3.4 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [go to section] links to the relevant section in this notebook.

3.4.1 Grouping data

- `groupby()` — group data based on values of columns or index [go to section]

3.4.2 Built-in aggregation functions

Most of these methods can be applied to whole `DataFrame` or `Series` objects, or to objects obtained via `groupby()`.

- `mean()` — compute average within each group [go to section]
- `sum()` — sum values within each group [go to section]
- `std()`, `var()` — within-group standard deviation and variance [go to section]
- `median()` — compute median within each group [go to section]
- `quantile()` — compute quantiles within each group [go to section]
- `size()` — number of observations in each group [go to section]
- `count()` — number of non-missing observations in each group [go to section]
- `first()`, `last()` — first and last elements in each group [go to section]
- `min()`, `max()` — minimum and maximum elements within a group [go to section]

3.4.3 Custom aggregation functions

- `agg()` — perform one or multiple (built-in or custom) aggregations *by column* [go to section]
- `apply()` — perform custom aggregation using *entire* `DataFrame`

3.4.4 Transformations

- `transform()` — perform (custom or built-in) transformation *by column* [go to section]

3.4.5 Aggregating time series data

- `resample()` — resample time series data to different frequency and apply aggregation [go to section]

4 Concatenating and merging data

More often than not, data sets come from various sources and need to be concatenated (the process of appending observations or variables) or merged as part of data pre-processing. Pandas offers several routines to accomplish such tasks which we study in this unit:

1. `pd.concat()` allows us to combine multiple Series or DataFrames by appending observations (rows) or columns.
2. `pd.merge()` allows us to match observations from one Series or DataFrame with observations from another Series or DataFrame and combine these into a *merged* DataFrame.

You can also consult the official user guide and the pandas cheat sheet for more information.

4.1 Concatenation

Concatenation with `pd.concat()` is used to combine multiple data sets along the row or column axes. This function can be called with both `Series` and `DataFrame` arguments, as we illustrate below.

4.1.1 Concatenating Series

We begin with the simplest case of combining two `Series` to obtain a new `Series` which contains observations from both.

Example: Concatenating two Series along the row axis

[1]:
```
import pandas as pd

# Create first series of 3 observations
a = pd.Series(['A0', 'A1', 'A2'])
a
```

[1]:
```
0    A0
1    A1
2    A2
dtype: str
```

[2]:
```
# Data for second series (5 observations)
data_b = [f'B{i}' for i in range(5)]

# Create second series
b = pd.Series(data_b)
b
```

[2]:
```
0    B0
1    B1
2    B2
3    B3
4    B4
dtype: str
```

To concatenate `a` and `b` along the first dimension, we call `pd.concat()` as follows:

[3]:
```
# Call concat() with the default value for axis, which is axis=0
s = pd.concat((a, b))

# Alternatively, make explicit that we are concatenating along the row axis
# s = pd.concat((a, b), axis=0)
s
```

[3]:
```
0    A0
1    A1
2    A2
0    B0
1    B1
2    B2
3    B3
4    B4
dtype: str
```

As you can see, `pd.concat()` also concatenates the index, which has the undesirable effect that the index values are no longer unique. As a rule, you never want to operate with non-unique indices in pandas as this can cause problems in some scenarios which are hard to diagnose. We can rectify this with the `reset_index()` method that we encountered in previous units:

[4]:
```
# Reset index to get rid of duplicates
s = s.reset_index(drop=True)
s
```

[4]:
```
0    A0
1    A1
2    A2
3    B0
4    B1
5    B2
6    B3
7    B4
dtype: str
```

Alternatively, we can pass the `ignore_index=True` argument to `pd.concat()` and the return value will use the default index:

[5]:
```
# Concatenate two series, use the default index for the return value
pd.concat((a, b), ignore_index=True)
```

[5]:
```
0    A0
1    A1
2    A2
3    B0
4    B1
5    B2
6    B3
7    B4
dtype: str
```

Example: Concatenating along the column axis

It is also possible to concatenate `Series` along the column dimension by specifying `axis=1`. We would usually use this only for `Series` of equal length, as the result otherwise contains `NaN` values if the Series have different indices (e.g., because they differ in the number of observations).

[6]:
```
s = pd.concat((a, b), axis=1)
s
```

[6]:
```
     0   1
0   A0  B0
1   A1  B1
2   A2  B2
3  NaN  B3
4  NaN  B4
```

If the `Series` in question have no names, pandas assigns the values `0`, `1`, ... as column names. This can be avoided by explicitly passing the desired column names using the `keys` argument:

[7]:
```
s = pd.concat((a, b), axis=1, keys=['Variable1', 'Variable2'])
s
```

[7]:
```
  Variable1 Variable2
0        A0        B0
1        A1        B1
2        A2        B2
3       NaN        B3
4       NaN        B4
```

Your turn

1. Create a new `Series` with observations `['C0', 'C1']`.
2. Using the previously created `Series` `a` and `b`, concatenate all three objects along the row axis and create a new (unique) index.
3. Repeat the previous step, but now concatenate along the column axis. Assign the column names `'Column1'`, `'Column2'`, and `'Column3'`.

4.1.2 Concatenating DataFrames

Concatenating DataFrames works exactly the same way as for Series.

Concatenating along the column axis

Example: Concatenating two DataFrames along the column axis

In this example, we create two DataFrames with two and three columns, respectively.

[8]:
```
import numpy as np

# Create 2 x 2 array of string data
data_a = np.array([f'A{i}' for i in range(4)]).reshape((2, 2))

df_a = pd.DataFrame(data_a)
df_a
```

[8]:
```
    0   1
0  A0  A1
1  A2  A3
```

[9]:
```
# Create 2 x 3 array of string data
data_b = np.array([f'B{i}' for i in range(6)]).reshape((2, 3))

df_b = pd.DataFrame(data_b)
df_b
```

[9]:
```
    0   1   2
0  B0  B1  B2
1  B3  B4  B5
```

To create a new DataFrame which contains the columns from both `df_a` and `df_b`, we use `pd.concat(..., axis=1)`:

[10]:
```
# Concatenate along the column axis
df = pd.concat((df_a, df_b), axis=1)
df
```

[10]:
```
    0   1   0   1   2
0  A0  A1  B0  B1  B2
1  A2  A3  B3  B4  B5
```

As before, the resulting `DataFrame` can have non-unique column names which is undesirable. There is no `reset_index()` method for columns, but we can easily create unique column names as follows:

[11]:
```
# Reset column index to 0, 1, 2,...
df.columns = np.arange(len(df.columns))
df
```

[11]:
```
    0   1   2   3   4
0  A0  A1  B0  B1  B2
1  A2  A3  B3  B4  B5
```

Alternatively, we can pass the `ignore_index=True` argument to `pd.concat()` as we did earlier to have pandas create default column names.

[12]:
```
# Concatenate columns, return DataFrame with default column names
pd.concat((df_a, df_b), axis=1, ignore_index=True)
```

[12]:
```
    0   1   2   3   4
0  A0  A1  B0  B1  B2
1  A2  A3  B3  B4  B5
```

It is also possible to add an additional level of column names to the resulting `DataFrame` by specifying the `keys` argument:

[13]:
```
# Concatenate along column axis, add additional column index level [A, B]
df = pd.concat((df_a, df_b), axis=1, keys=['A', 'B'])
df
```

[13]:
```
    A       B
    0   1   0   1   2
0  A0  A1  B0  B1  B2
1  A2  A3  B3  B4  B5
```

The new `DataFrame` then has a so-called hierarchical column index.

Example: Concatenating a DataFrame and a Series

One can also concatenate DataFrames and Series object along the column axis. In that case, the `Series` is automatically converted to a `DataFrame` using the default column name.

[14]:
```
s = pd.Series(['C0', 'C1'])
s
```

[14]:
```
0    C0
1    C1
dtype: str
```

[15]:
```
# Concatenate DataFrame and Series
pd.concat((df_a, s), axis=1, ignore_index=True)
```

[15]:

	0	1	2
0	A0	A1	C0
1	A2	A3	C1

Concatenating along the row axis

We usually concatenate DataFrames along the row axis if we have observations on the same variables scattered across multiple data sets. Appending DataFrames with different columns will usually create `NaN` values and hence is often not useful.

Example: Concatenating rows with identical columns

[16]:
```
#  Concatenate 2x2 DataFrame and 3x2 DataFrame (note the transpose!)
df = pd.concat((df_a, df_b.T), axis=0, ignore_index=True)
df
```

[16]:

	0	1
0	A0	A1
1	A2	A3
2	B0	B3
3	B1	B4
4	B2	B5

Example: Concatenating rows with different columns

The DataFrames `df_a` and `df_b` have a different number of columns, so the resulting `DataFrame` will contain `NaN` for all observations of column `2` that were originally in `df_a`:

[17]:
```
# Concatenate DataFrame rows with different numbers of columns
df = pd.concat((df_a, df_b), axis=0, ignore_index=True)
df
```

[17]:

	0	1	2
0	A0	A1	NaN
1	A2	A3	NaN
2	B0	B1	B2
3	B3	B4	B5

Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `FRED_monthly_1950.csv` and `FRED_monthly_1960.csv` into two different DataFrames. The files contain monthly macroeconomic time series for the 1950s and 1960s, respectively. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.
2. Concatenate these DataFrames along the row dimension to get a total of 240 observations.
3. Set the column `DATE` as index for the newly created DataFrame.

4.2 Merging and joining data sets

4.2.1 Types of merges

While concatenation simply appends blocks of rows or columns from multiple data sets, merging allows for more fine-grained control over how data should be combined. The most common scenarios in empirical work are:

1. *one-to-one*: The observations in data sets `A` and `B` have a unique identifier (*"key"*), and each observation in `A` is matched with at most one observation in `B`. For example, we could have data on individuals from multiple sources, and each of these data sets identifies individuals by their social security number. Each observation in one data set corresponds to exactly one observation in the other data set.
2. *many-to-one*: Data set `A` contains unique identifiers but these can correspond to multiple observations in data set `B`. For example, we could have data at the ZIP-code (neighborhood) level in data set `A` and data on individuals in data set `B`. ZIP-codes are a unique identifier in `A`, but many individuals can live in the same neighboorhood, so each observation in `A` can reasonably be matched with many different observations in `B`.
3. *many-to-many*: Identifying keys are not unique in either data set, and the resulting data set is a Cartesian product of all possible key combinations from both data sets. This situation should usually be *avoided* as it tends to have surprising results and can potentially consume large amounts of memory.

4.2.2 Implementation in pandas

Merging in pandas can be performed in two different ways:

1. `pd.merge()` is a function that takes as argument the *two* DataFrames to be merged:

   ```
   result = pd.merge(df_A, df_B)
   ```

2. `df.merge()` is a method of a specific `DataFrame` object, and takes as an argument the other `DataFrame` to be merged:

   ```
   result = df_A.merge(df_B)
   ```

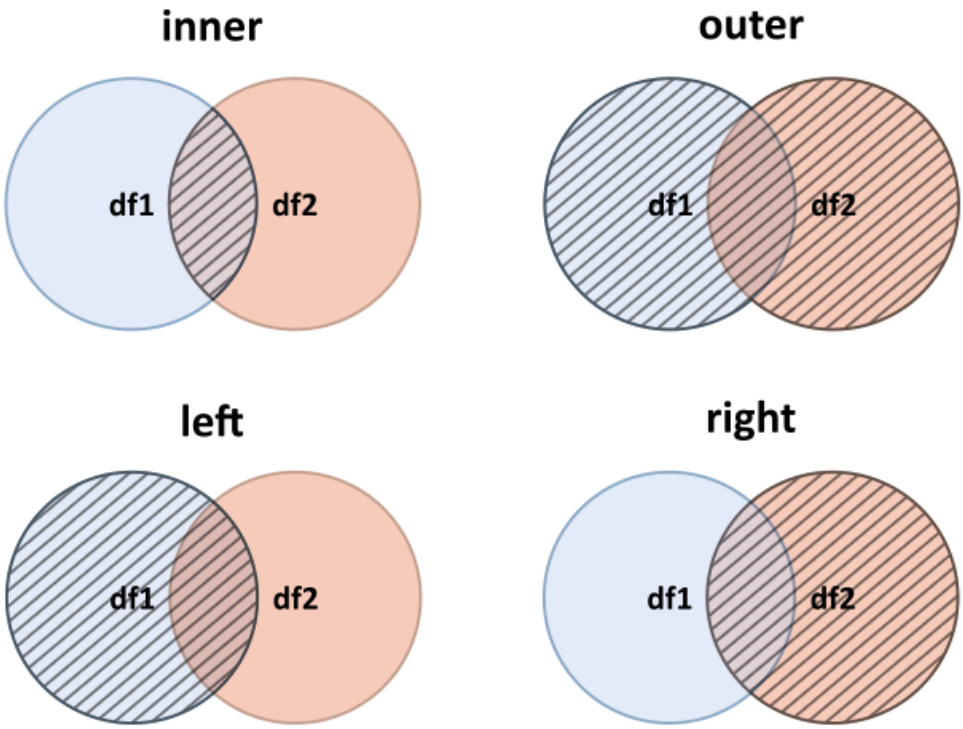
Both ways are equivalent and can be used interchangeably.

4.2.3 Controlling the resulting data set

Irrespective of whether we perform a *one-to-one* or a *many-to-one* merge, we frequently face the situation that some observations are present in one data set but not the other. We therefore need to control which subset of the data we want to retain in the final data set. This is accomplished using the `how` argument passed to `merge()`. There are several possible merge methods which were originally introduced in SQL, a data processing language for relational databases (see also the official user guide):

1. `how='inner'` performs a so-called *inner join*: the merged data contains only the *intersection* of keys that are present in *both* data sets.
2. `how='outer'` performs an *outer join*: the merged data contains the *union* of keys present in either of the data sets. Rows which are not present in both data sets will contain missing values.
3. `how='left'` performs a *left join*: all identifiers from the *left* data set are present in the merge result, but rows that are only present in the *right* data set are dropped.
4. `how='right'` performs a *right join*: all identifiers from the *right* data set are present in the merge result, but rows that are only present in the *left* data set are dropped.

The following figure illustrates these concepts graphically using Venn diagrams. Each circle represents the keys present in the left (`df1`) or right (`df2`) DataFrames. The merge method controls which subset of keys is retained in the merge result.



4.2.4 Merging with `pd.merge()`

One-to-one merges

We first create two data sets `A` and `B` used to demonstrate various merge methods. We use the column `key` as the identifier on which to perform merges.

[18]:
```
# Create first DataFrame with 2 rows
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})
df_a
```

[18]:
```
   key value_a
0    0      A0
1    1      A1
```

[19]:
```
# Create second DataFrame with 2 rows
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
df_b
```

[19]:
```
   key value_b
0    1      B1
1    2      B2
```

When merging two DataFrames, in most cases we need to specify the columns (or index levels) on which the merge should be performed. We do this using the argument `on` when calling `pd.merge()`.

[20]:
```
# Merge A and B on the identifier 'key' using an inner join
pd.merge(df_a, df_b, on='key', how='inner')
```

[20]:
```
   key value_a value_b
0    1      A1      B1
```

Note that in this case we could leave the `on` argument unspecified, as then `pd.merge()` by default merges on the intersection of columns present in both DataFrames (which in this case is just the column `key`). However, for clarity it is advisable to always specify `on` explicitly.

Moreover, `pd.merge()` performs an inner join by default, so we could have called the function as follows to get the same result:

[21]:
```
# Merge A and B on default key using default inner join
pd.merge(df_a, df_b)
```

[21]:
```
   key value_a value_b
0    1      A1      B1
```

Since we are performing an inner join, the merged data set contains only a single column corresponding to the identifier `1`, the only one present in both DataFrames.

If we want to retain all observations, we achieve this using an outer join:

[22]:
```
# Merge A and B using outer join (keep union of observations)
pd.merge(df_a, df_b, on='key', how='outer')
```

[22]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2     NaN      B2
```

Since the keys `0` and `2` are not present in both DataFrames, the corresponding columns contain missing values.

We can also only retain the keys present in the left (i.e., the first argument) or the right (i.e., the second argument) DataFrame:

[23]:
```
# Merge A and B on the identifier 'key', keep left identifiers
pd.merge(df_a, df_b, on='key', how='left')
```

[23]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

[24]:
```
# Merge A and B on the identifier 'key', keep right identifiers
pd.merge(df_a, df_b, on='key', how='right')
```

[24]:
```
   key value_a value_b
0    1      A1      B1
1    2     NaN      B2
```

Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.
2. Merge the CPI and the GDP time series with `pd.merge()` without specifying the `how=` argument.
3. Merge the CPI and the GDP time series with `pd.merge()` using a left join (`how='left'`).
4. Merge the CPI and the GDP time series with `pd.merge()` using an inner join (`how='inner'`).

How do the results differ depending on the value of the `how` argument?

Many-to-one merges

To illustrate many-to-one merges, we create a DataFrame `A` which has non-unique values in the column `keys`. We want to merge this with another DataFrame `B` with unique values in the column `keys`. This operation is therefore a many-to-one merge.

[25]:
```
# Create first DataFrame with 4 rows, non-unique key
df_a = pd.DataFrame({'key': [0, 1, 0, 1], 'value_a': ['A0', 'A1', 'A2', 'A3']})
df_a
```

[25]:
```
   key value_a
0    0      A0
1    1      A1
2    0      A2
3    1      A3
```

[26]:
```
# Create second DataFrame with 2 rows, unique key
df_b = pd.DataFrame({'key': [0, 1], 'value_b': ['B0', 'B1']})
df_b
```

[26]:
```
   key value_b
0    0      B0
1    1      B1
```

[27]:
```
# Merge A and B on the identifier 'key', keep left identifiers
pd.merge(df_a, df_b, on='key')
```

[27]:
```
   key value_a value_b
0    0      A0      B0
1    1      A1      B1
2    0      A2      B0
3    1      A3      B1
```

As you can see, in the resulting DataFrame the values from `B` are matched with multiple rows of `A`.

4.2.5 Merging with `DataFrame.merge()`

As mentioned above, there is an alternative but equivalent way to merge DataFrames using the method `df.merge()`. In this context, the *left* `DataFrame` is the one on which `merge()` is being invoked, while the *right* `DataFrame` is the argument passed to `merge()`.

To illustrate, we recreate the DataFrames we used for the one-to-one merge examples above:

[28]:
```
# Create first DataFrame with 2 rows
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})

# Create second DataFrame with 2 rows
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
```

We now use the `merge()` method invoked on DataFrame `A` to merge it with DataFrame `B`:

[29]:
```
# Use DataFrame method to merge, keep only left identifiers
df_a.merge(df_b, on='key', how='left')
```

[29]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

[30]:
```
# Now df_a is the right DataFrame, set of final identifiers is the same as
# in the example above!
df_b.merge(df_a, on='key', how='right')
```

[30]:
```
   key value_b value_a
0    0     NaN      A0
1    1      B1      A1
```

Example: Merging with overlapping column names

Sometimes both DataFrames contain the same column names. If these columns are not used as keys in the merge operation, pandas automatically renames these columns in the resulting `DataFrame` to avoid naming clashes.

To illustrate, we rename the value columns to `'value'` in both DataFrames and then perform the merge:

[31]:
```
# Rename columns to common name 'value'
df_a = df_a.rename(columns={'value_a': 'value'})
df_b = df_b.rename(columns={'value_b': 'value'})
```

Note that once we have identical column names `['key', 'value']` in both DataFrames, we *must* specify the `on` argument to `merge()` as otherwise pandas by default merges on the intersection of column names in both DataFrames, i.e., in this case it merges on `['key', 'value']`:

[32]:
```
# Invoking merge() with default on argument has unintended consequences
df_a.merge(df_b)
```

[32]:
```
Empty DataFrame
Columns: [key, value]
Index: []
```

The merge result is empty because we are performing an *inner join* (the default), and there are no overlapping rows that have the same values for both `key` and `value` columns. We therefore need to explicitly specify `on='key'` to get the desired result:

[33]:
```
# Merge DataFrames with overlapping column 'value'
df_a.merge(df_b, on='key')
```

[33]:
```
   key value_x value_y
0    1      A1      B1
```

As you can see, pandas automatically appends the suffixes `'_x'` and `'_y'` to the column from the left and right DataFrames, respectively. We can change this default behavior by explicitly specifying the suffixes to be appended:

[34]:
```
df_a.merge(df_b, on='key', suffixes=('_left', '_right'))
```

[34]:
```
   key value_left value_right
0    1         A1          B1
```

Your turn

Repeat the previous exercise, but use the `DataFrame.merge()` method to perform the merge.

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.
2. Merge the DataFrame containing the CPI with the DataFrame containing GDP without specifying the `how=` argument.

3. Merge the DataFrame containing the CPI with the DataFrame containing GDP using a *left* join.
4. Merge the DataFrame containing the CPI with the DataFrame containing GDP using an *inner* join.

How do the results differ depending on the value of the `how` argument?

4.2.6 Merging with `join()`

The `DataFrame` method `join()` is a convenience wrapper around `pd.merge()` with the following subtle differences:

1. `join()` can be called *only* directly on the `DataFrame` object, i.e., `df.join()`, while for merge we have both the `pd.merge()` and the `df.merge()` variants.
2. `join()` always operates on the *index* of the other `DataFrame`, whereas `merge()` is more flexible and can operate on either the index or on columns.
3. `join()` by default performs a `left` join, whereas `merge()` performs an `inner` join.

As a rule of thumb, you should use `join()` if you want to join DataFrames which have a similar index.

Example: joining DataFrames

We first create two DataFrames to be joined. This time, we explicitly set an index for each of them which will be used to perform the `join()`.

[35]:
```
# Create first DataFrame with 2 rows
df_a = pd.DataFrame(['A0', 'A1'], columns=['value_a'], index=[0, 1])
df_a
```

[35]:
```
  value_a
0      A0
1      A1
```

[36]:
```
# Create second DataFrame with 2 rows
df_b = pd.DataFrame(['B1', 'B2'], columns=['value_b'], index=[1, 2])
df_b
```

[36]:
```
  value_b
1      B1
2      B2
```

[37]:
```
# Perform left join (the default option)
df_a.join(df_b)
```

[37]:
```
  value_a value_b
0      A0     NaN
1      A1      B1
```

[38]:
```
# Join with explicit inner join
df_a.join(df_b, how='inner')
```

[38]:
```
  value_a value_b
1      A1      B1
```

[39]:
```
# Perform an outer join
df_a.join(df_b, how='outer')
```

[39]:
```
  value_a value_b
0      A0     NaN
1      A1      B1
2     NaN      B2
```

Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.
2. Set the `DATE` column as the index for each of the two DataFrames.
3. Merge the CPI with the GDP time series with `join()`. Do this with both a left and an inner join.

4.3 Dealing with missing values

We already encountered missing values in earlier lectures. These are particularly likely to arise when merging or concatenating data if individual DataFrames lack some observations.

To illustrate, recall the example from above:

[40]:
```
# Create two DataFrames with partially overlapping keys
df_a = pd.DataFrame({'key': [0, 1], 'value_a': ['A0', 'A1']})
df_b = pd.DataFrame({'key': [1, 2], 'value_b': ['B1', 'B2']})
```

[41]:
```
# Perform outer merge, keep union of keys
pd.merge(df_a, df_b, on='key', how='outer')
```

[41]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2     NaN      B2
```

Since the keys in DataFrames `df_a` and `df_b` were only partially overlapping, the resulting DataFrame has missing values by construction. In what follows, we explore strategies on how to handle these missing data.

4.3.1 Dropping missing values

One strategy is to drop missing values outright, even though we might lose information that could be useful to perform data analysis if only some but not all columns contain missing values, as is the case above.

Missing values can be dropped by either

1. Using `dropna()`, potentially restricted to a subset of columns.
2. Selecting a subset of observations to keep with a boolean operation such as `notna()` or `isna()`.
3. Avoiding the missing values in the first place, e.g., by using `merge(..., how='inner')`.

Example: Dropping missing values

Consider the merged `DataFrame` from above. We can drop rows with missing values with `dropna()`, which by default drops all rows with *any* missing values. Alternatively, we can specify only a subset of columns to be checked for missing values.

[42]:
```
# Merge with outer join, thus creating missing values
df = pd.merge(df_a, df_b, on='key', how='outer')
```

[43]:
```
# Drop any row which contains at least one missing value
df.dropna()
```

[43]:
```
   key value_a value_b
1    1      A1      B1
```

[44]:
```
# Drop rows which contain missing values in column 'value_a', ignore missing
# values in 'value_b'
df.dropna(subset='value_a')
```

[44]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
```

> **Your turn**
>
> Instead of using `dropna()`, drop the missing observations using either `notna()` or `isna()`.
>
> 1. Drop all rows with any missing observations.
> 2. Drop all rows with missing observations in column `value_a`.

Example: Avoiding missing values in the first place

Of course the missing values in the example above arose only because we specified `how='outer'`. Merging with `how='inner'` drops keys which are not present in both DataFrames right away, avoiding the issue of missing values (unless these are already present in the original DataFrames):

[45]:
```
# Merge using inner join, drop keys not present in both DataFrames
pd.merge(df_a, df_b, on='key', how='inner')
```

[45]:
```
   key value_a value_b
0    1      A1      B1
```

4.3.2 Filling missing values

Instead of dropping data, we can impute missing values in various ways:

1. `fillna()` can be used to replace missing data with user-specified values.
2. `ffill()` and `bfill()` can be used to fill missing values forward or backward from adjacent non-missing observations.
3. `interpolate()` supports various interpolation methods such as linear interpolation based on non-missing values.

Example: Replacing missing values with `fillna()`

Consider the merged `DataFrame` we have created above:

[46]:
```
df = pd.merge(df_a, df_b, on='key', how='outer')
df
```

[46]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2     NaN      B2
```

We can use `fillna()` to replace missing values with some constant.

[47]:
```
# Replace ALL missing values with 'Some value'
df.fillna('Some value')
```

[47]:
```
   key     value_a     value_b
0    0          A0  Some value
1    1          A1          B1
2    2  Some value          B2
```

This might not be what you want as the provided non-missing value is imposed on *all* columns. It is therefore possible to specify a different value for each column using a dictionary as an argument.

[48]:
```
# Use different replacement values for columns 'value_a' and 'value_b'
df.fillna({'value_a': 'Missing A', 'value_b': 'Missing B'})
```

[48]:
```
   key    value_a    value_b
0    0         A0  Missing B
1    1         A1         B1
2    2  Missing A         B2
```

Example: forward- or backward-filling missing values

Another common imputation method is to use the previous (*"forward"*) or next (*"backward"*) non-missing value as replacement for missing data.

Continuing with the `DataFrame` from the previous example, we can apply these methods as follows:

[49]:
```
# Forward-fill missing values from previous observation
df.ffill()
```

[49]:
```
   key value_a value_b
0    0      A0     NaN
1    1      A1      B1
2    2      A1      B2
```

This inserts the value `'A1'` in the 3rd row of column `value_a`, but does not do anything about the missing value in column `value_b` since there is no preceding non-missing value.

Conversely, `bfill()` does the opposite and backfills the missing value in column `value_b`:

[50]:
```
df.bfill()
```

[50]:
```
   key value_a value_b
0    0      A0      B1
1    1      A1      B1
2    2     NaN      B2
```

Example: linear interpolation

Consider the following `Series` with numerical data (interpolation only makes sense for numerical data, not strings):

[51]:
```
s = pd.Series([1.0, 2.0, 3.0, np.nan, 5.0])
s
```

[51]:
```
0    1.0
1    2.0
2    3.0
3    NaN
4    5.0
dtype: float64
```

We can interpolate the missing data using `interpolate()`, for example by using linear interpolation (check the documentation for many other interpolation methods).

[52]:
```
# Interpolate missing values using linear interpolation
s.interpolate(method='linear')
```

[52]:
```
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
dtype: float64
```

Your turn

Use the data files located in the folder `../../data/FRED` to perform the following tasks:

1. Load the data in `CPI.csv` and `GDP.csv` into two different DataFrames. The files contain monthly data for the Consumer Price Index (CPI) and quarterly data for GDP, respectively. *Hint:* Use `pd.read_csv(..., parse_dates=['DATE'])` to automatically parse strings stored in the `DATE` column as dates.
2. Merge the CPI with the GDP time series with `merge()` using a left join. This creates missing values in the `GDP` column.
3. Impute the missing GDP values using `interpolate()` and replace the missing values in column `GDP`.

4.4 List of functions used in this lecture

The following list contains the central functions covered in this lecture. Each function name is linked to the official API documentation which you can consult for more details regarding function arguments and return values. The [go to section] links to the relevant section in this notebook.

4.4.1 Concatenating (appending) data

- `pd.concat()` — append Series or DataFrames along the row or column dimension [go to section]
- `reset_index()` — reset (potentially non-unique) index after concatenation [go to section]

4.4.2 Merging data

- `pd.merge()` — merge two Series or DataFrames along the column dimension based on some keys [go to section]
- `DataFrame.merge()` — merge a given (left) DataFrame with another (right) Series or DataFrame based on some keys [go to section]
- `DataFrame.join()` — merge a given (left) DataFrame with another (right) Series or DataFrame based on the index [go to section]

4.4.3 Dealing with missing values

- `notna()` — check whether (one or more) columns have non-missing values [go to section]
- `isna()` — check whether (one or more) columns have missing values [go to section]
- `dropna()` — drop rows with missing observations [go to section]
- `fillna()` — replace missing values with a given value [go to section]
- `ffill()` — fill missing values by propagating the last valid observation [go to section]
- `bfill()` — fill missing values by using the next valid observation to fill the gap [go to section]
- `interpolate()` — interpolate missing values from adjacent non-missing ones [go to section]