Part 2 – Lecture 3: Plotting

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn

NHH Norwegian School of Economics

October 8, 2025

Contents

1 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 along-side 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.
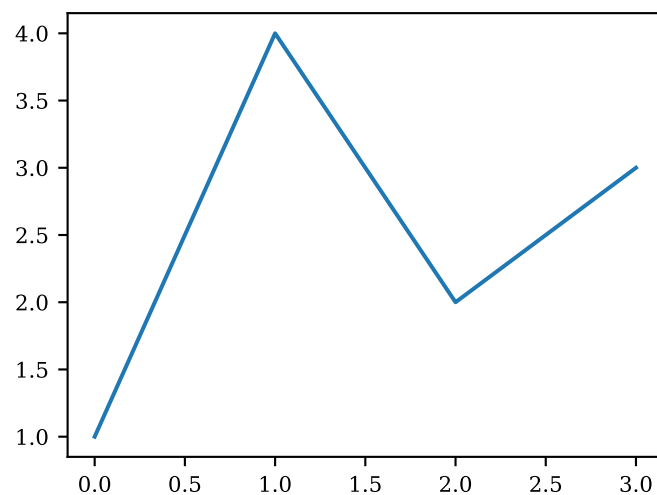
1.1 Line plots

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)
```
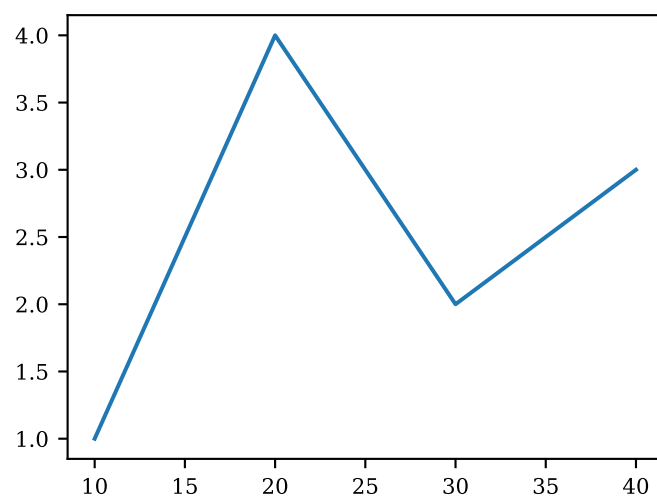
```
[1]: [<matplotlib.lines.Line2D at 0x7f6dbdc65f90>]
```


We didn't even have to specify the corresponding x-values, as Matplotlib automatically assumes them to be $[0, 1, 2, \dots]$. 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)
```

```
[2]: [<matplotlib.lines.Line2D at 0x7f6dbcd29590>]
```

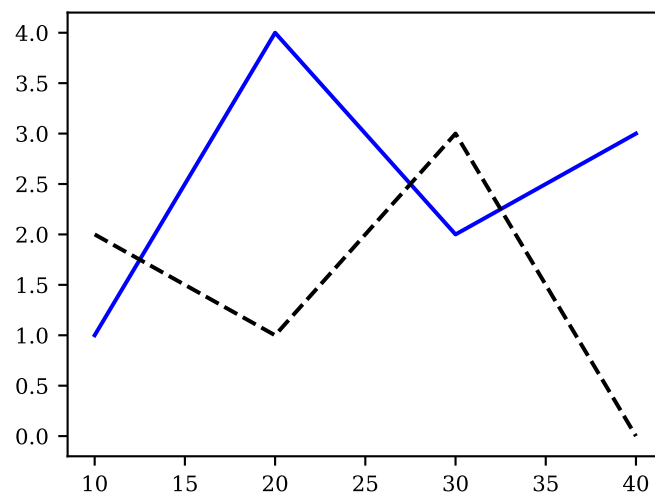> **Your turn.**   Plot the quadratic function $y = x^2$ on the interval [-1, 1]:
>
>   1. Create a grid of 50 x-values using `np.linspace()`.
>
>   2. Compute the corresponding y-values and plot them.

1.1.1 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--')
```

```
[3]: [<matplotlib.lines.Line2D at 0x7f6dbcdccf50>,
      <matplotlib.lines.Line2D at 0x7f6dbcdcd090>]
```


1.1.2 Specifying plot styles

The characters following each set of y-values are style specifications. The letters are short-hand 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')
```

[4]: [<matplotlib.lines.Line2D at 0x7f6dbcc68f50>]
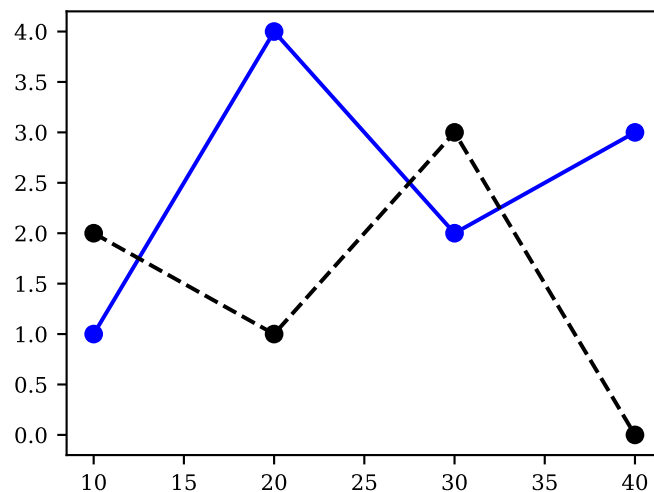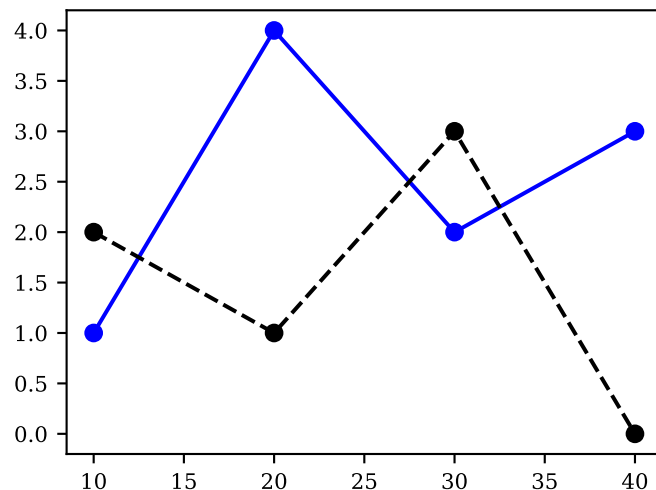

Individual calls to `plot()` also allow us to specify styles more explicitly using keyword arguments:

```
[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')
```

[5]: [<matplotlib.lines.Line2D at 0x7f6dbccdce10>]

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:

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

[6]: `[<matplotlib.lines.Line2D at 0x7f6dbcb84cd0>]`
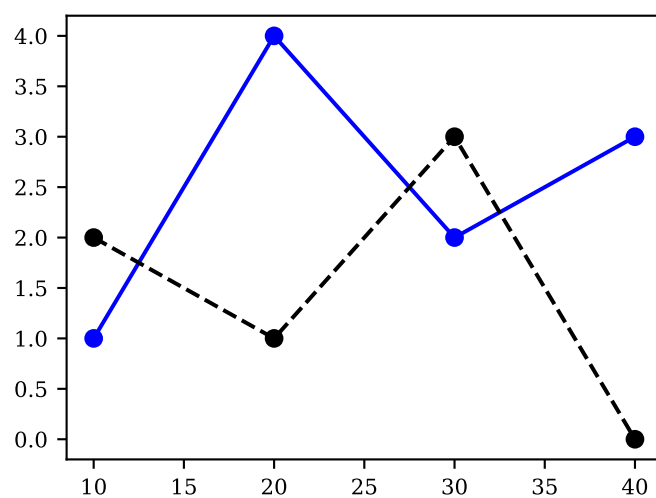
> **Your turn.** Use the data files located in the folder `../../data/FRED` to perform the following tasks:
>
> 1. Load the data from `FRED_annual.csv`. The file contains annual observations on selected macroeconomic variables for the US.
>
> 2. Plot the unemployment rate (column `UNRATE`) using a blue 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.

1.2 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.
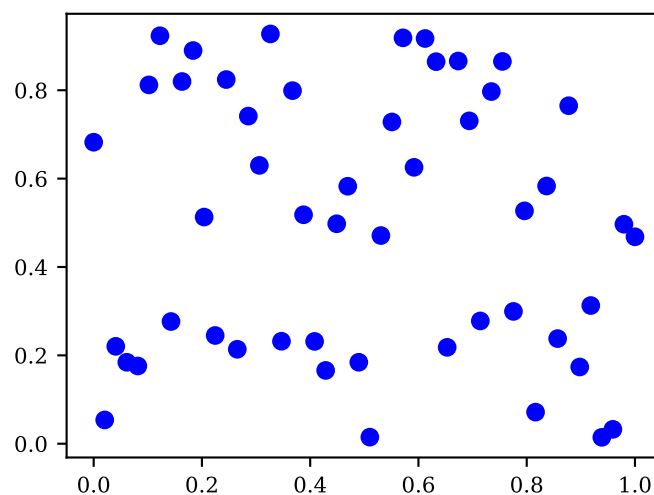
```python
import matplotlib.pyplot as plt
import numpy as np

# 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')
```

`[7]:` `<matplotlib.collections.PathCollection at 0x7f6dbce51d30>`

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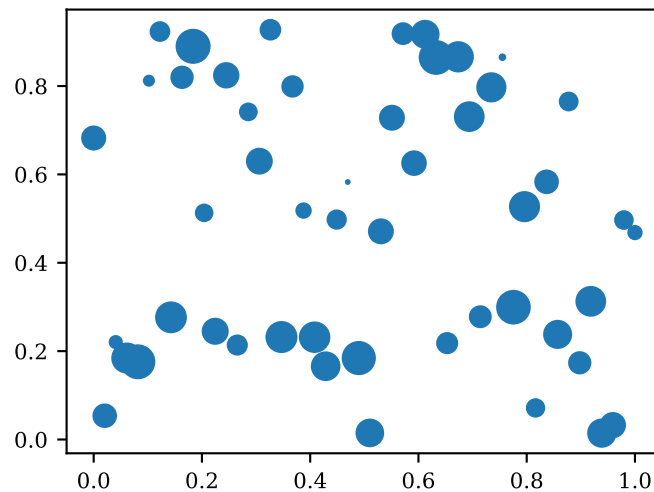
To illustrate, the following code assigns a different (randomly drawn) marker size to each dot:

```python
# 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)
```

[8]: `<matplotlib.collections.PathCollection at 0x7f6dbca8e210>`

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.

1.3 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]:
```python
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')
```

[9]: `Text(0.5, 1.0, 'Histogram of normal draws')`

Histogram of normal draws
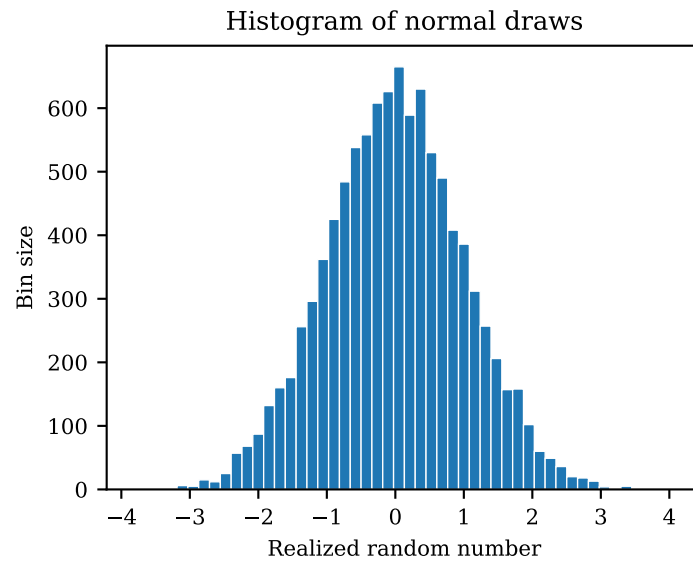
1.4 Plotting categorical data

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:

```python
import matplotlib.pyplot as plt

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

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

# Add overall title
plt.title('Population')
plt.ylabel('In thousand')
```
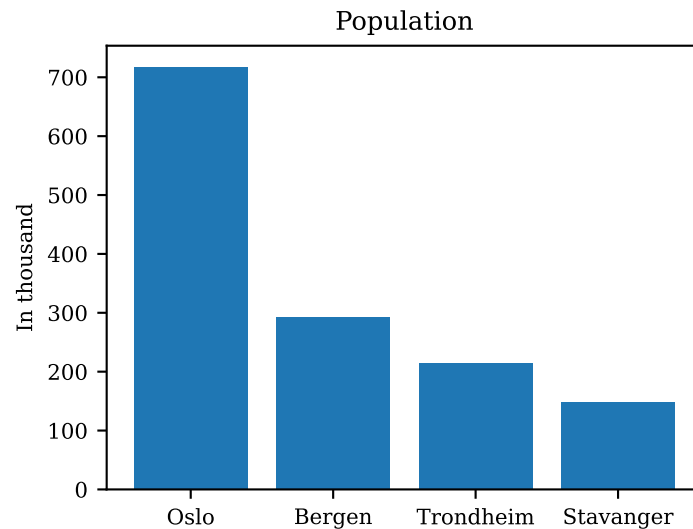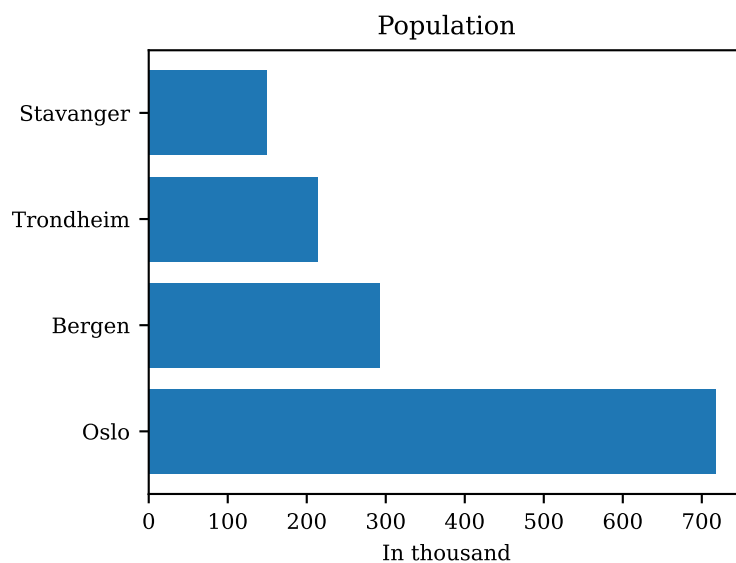
[10]: Text(0, 0.5, 'In thousand')

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

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

```
[11]: Text(0.5, 0, 'In thousand')
```


1.5 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 (called "axes" in Matplotlib).

- 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.

9

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')
```

```
[12]: Text(0.05, 0.05, 'More text')
```
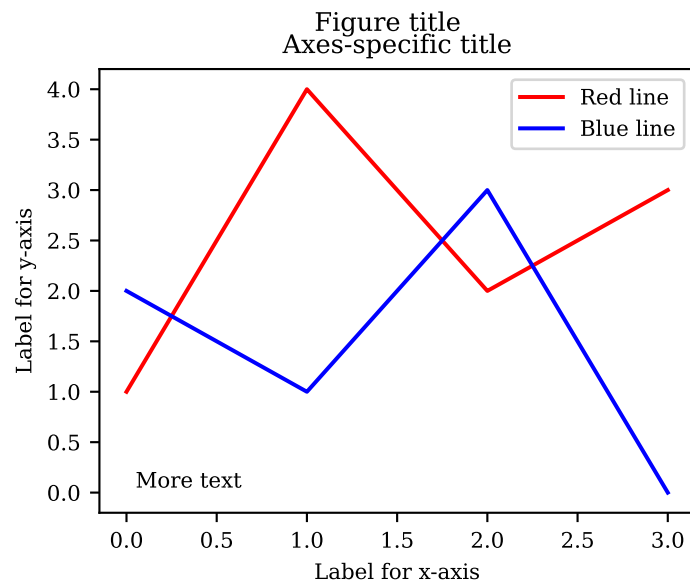

1.6 Plot limits, ticks and tick labels

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: Plotting the sine function

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))
```

[13]: (-1.1, 1.1)
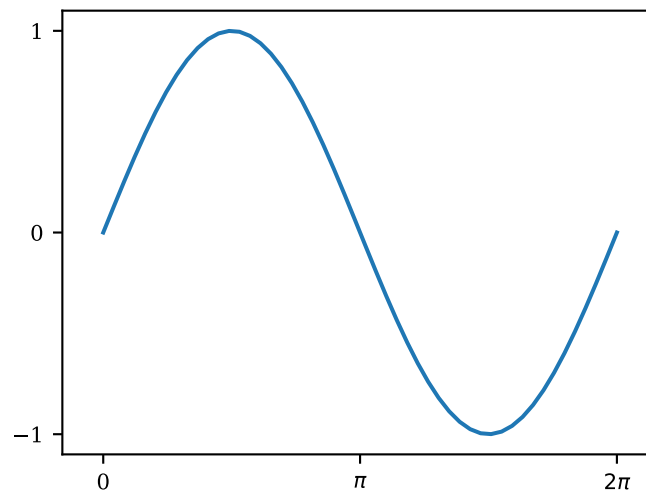
1.7 Adding straight lines

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')
```

`[14]:` `<matplotlib.lines.Line2D at 0x7f6dbc6dbed0>`
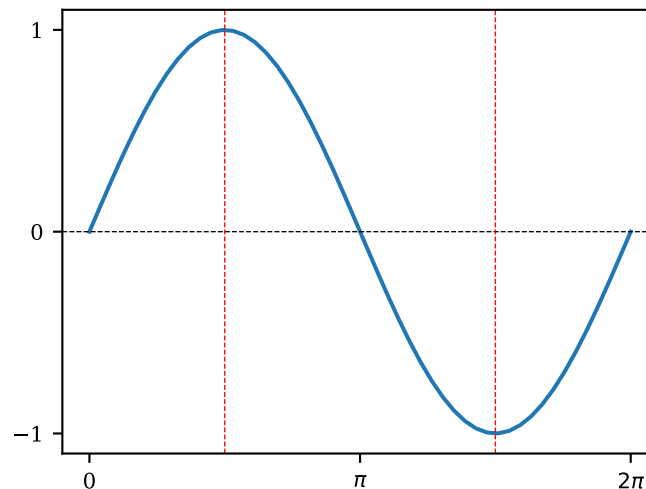
Your turn.   Plot the function $y = (x-1)^2$ on the interval $[0, 2]$:

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.

1.8 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 re-usable 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()
```

```
[16]:  Text(0.05, 0.05, 'More text')
```
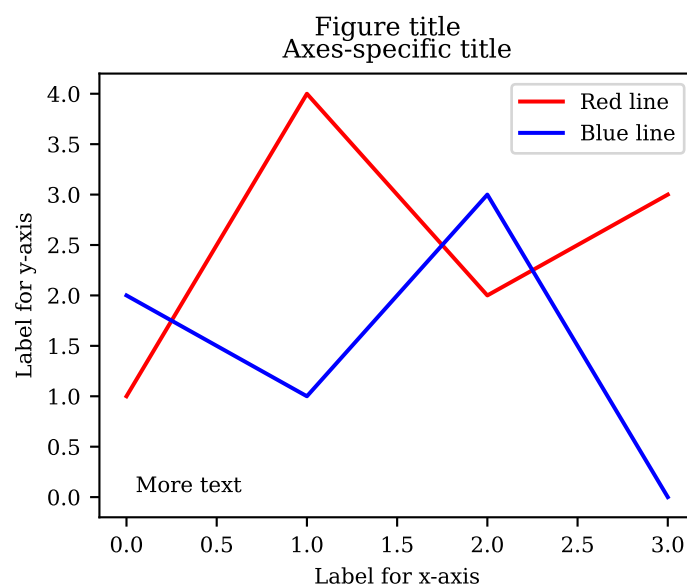

The code is quite similar to the previous section, except that most attributes are set using the `set_xxx()` methods of the `ax` object. For example, instead of calling `xlim()`, we use `ax.set_xlim()`.

1.9 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:

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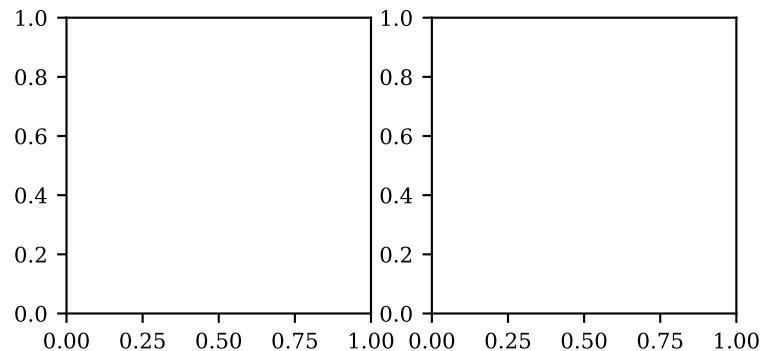
With multiple axes objects in a single figure (as in the above example), the `axes` returned by `subplots()` is a NumPy array. Its elements map to the individual panels within the figure in a natural way.

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

```python
[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)
```
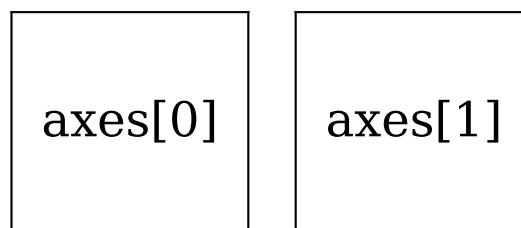
Don't worry about the details of how this graph is generated, the only take-away here is how axes objects are mapped to the panels in the figure.

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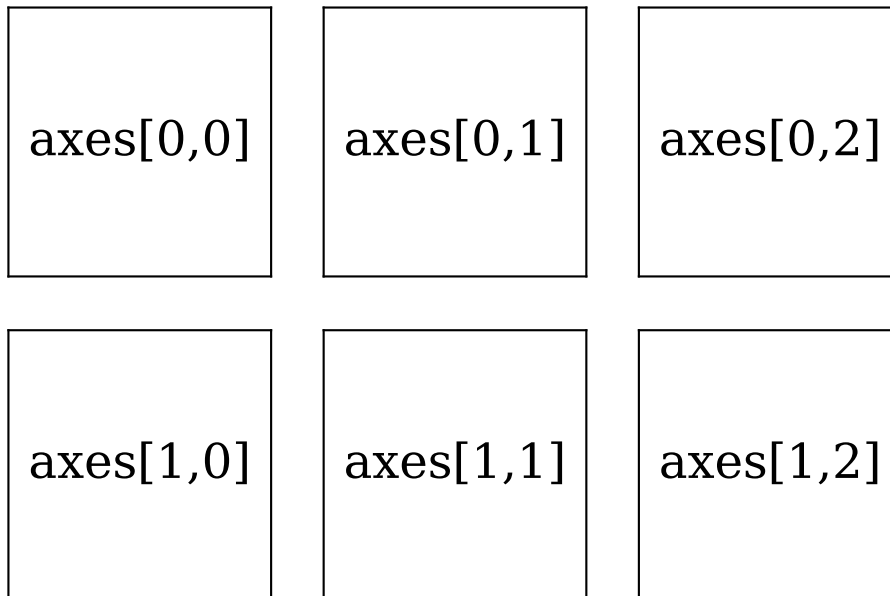
```python
[19]: # Create figure with 2 rows, 3 columns
      nrow = 2
      ncol = 3
      fig, axes = plt.subplots(nrow, ncol, figsize=(6,4))

      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

      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)
```
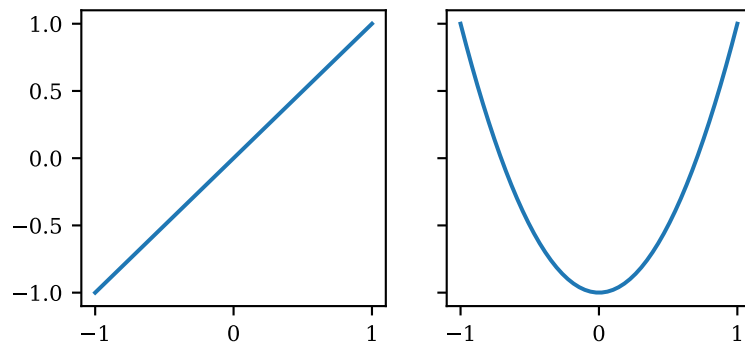
```
[20]: [<matplotlib.lines.Line2D at 0x7f6dbc3c79d0>]
```

Example: Create a figure with 2 rows and 2 columns

```python
[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
```
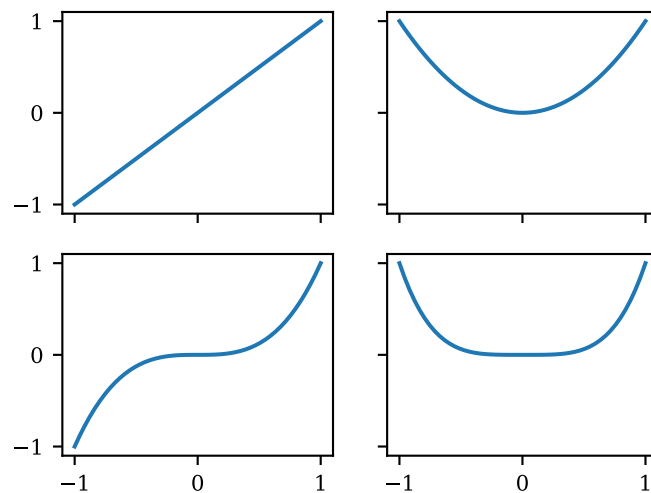

Note the use of `sharex=True` and `sharey=True`. This tells Matplotlib that all axes share the same plot limits, so the tick labels can be omitted in the figure's interior to preserve space.

> **Your turn.** Create a figure with 3 columns (on a single row) and plot the following functions on the interval $[0, 6]$:
>
> - Subplot 1: $y = sin(x)$
>
> - Subplot 2: $y = sin(2x)$
>
> - Subplot 3: $y = sin(4x)$
>
> *Hint:* The sine function can be imported from NumPy as `np.sin()`.

2 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.

2.1 Bar charts with pandas

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

```python
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))
```
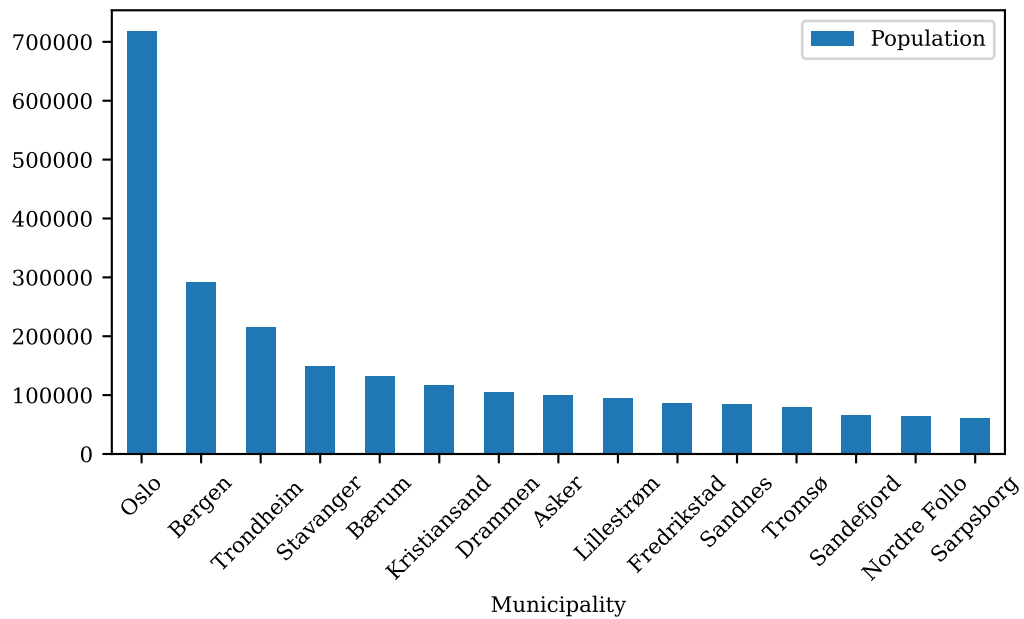
[22]:

[22]: `<Axes: xlabel='Municipality'>`

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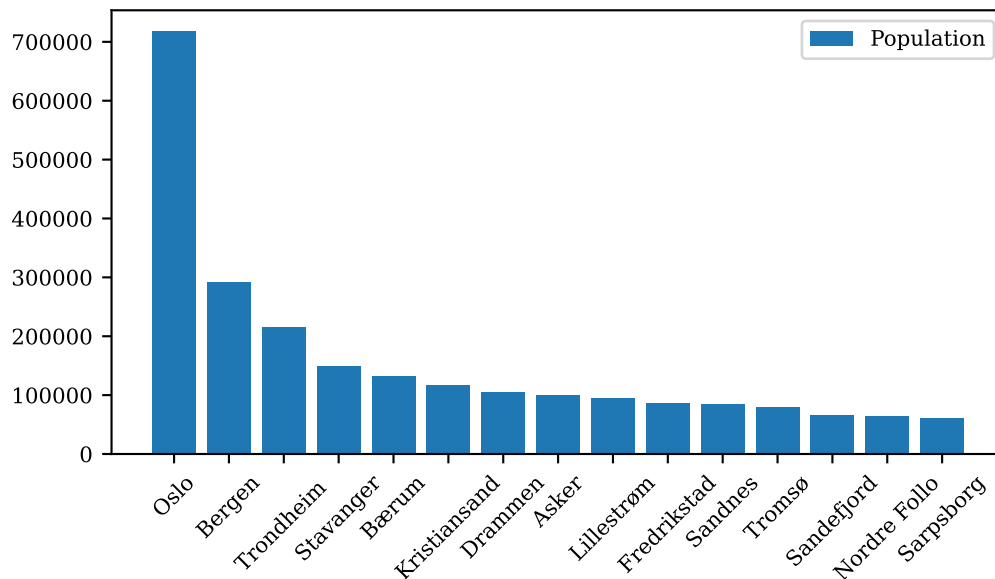
Matplotlib's functions usually directly work with pandas's data structures. In cases where they don't, we can convert a `DataFrame` or `Series` object to a NumPy array using the `to_numpy()` method.

2.2 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.
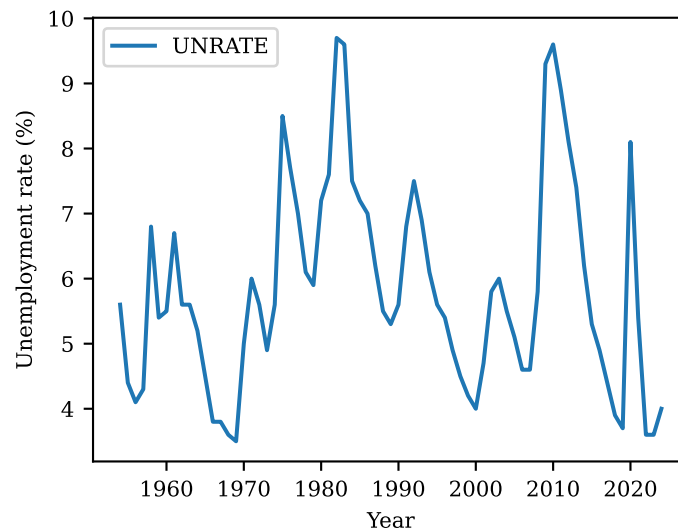
```
[24]: import pandas as pd

      # 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 (%)')
```

```
[24]: <Axes: xlabel='Year', 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 (column `UNRATE`) and the inflation rate (column `INFLATION`) in a single graph.

2.3 Scatter plots with pandas

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's 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 (%)'
       )
```

```
[25]:  <Axes: xlabel='Unemployment rate (%)', ylabel='Inflation (%)'>
```
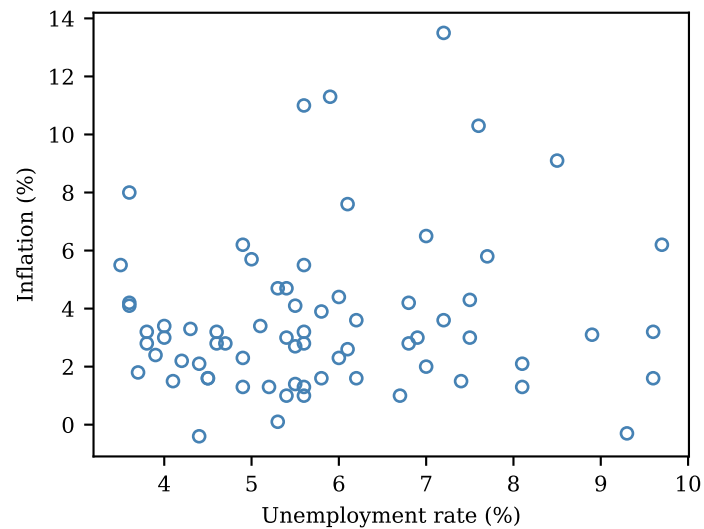
Pandas also offers the convenience function `scatter_matrix()` which lets us easily create pairwise scatter plots for more than two variables:

```
[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,
      )
```
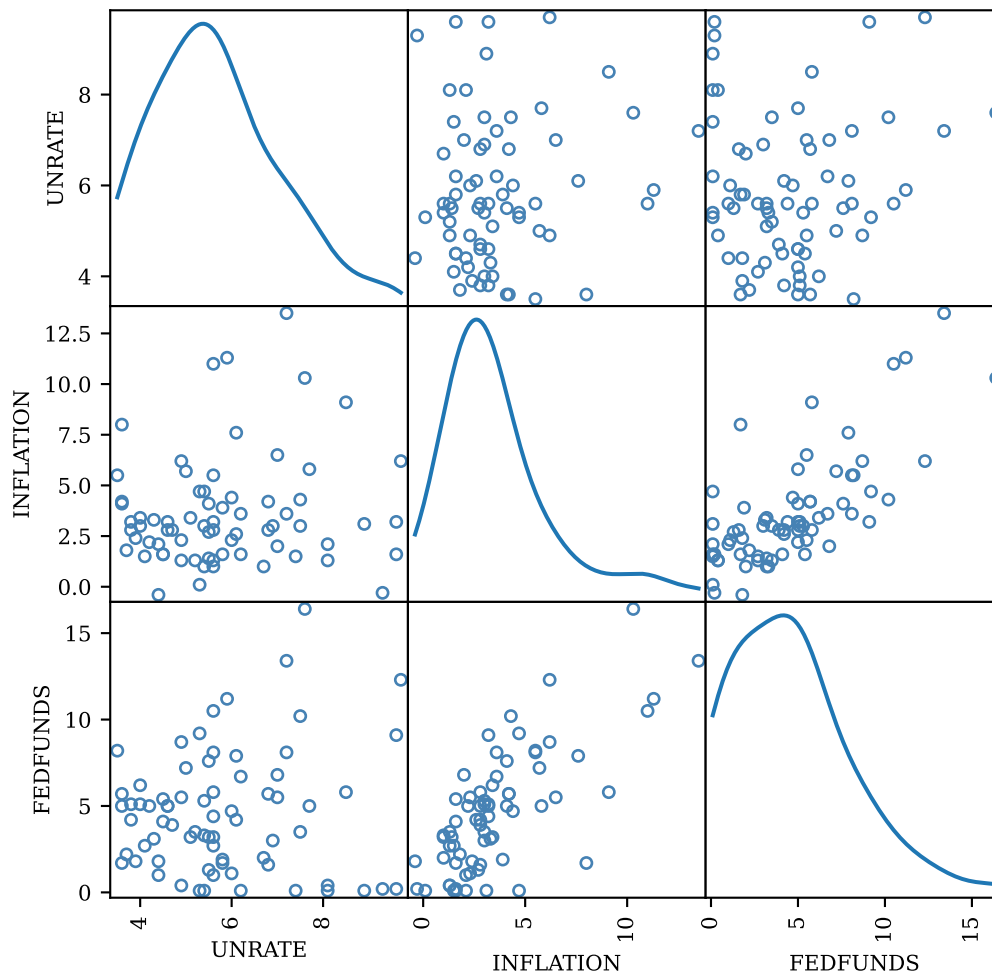
2.4 Box plots with pandas

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:
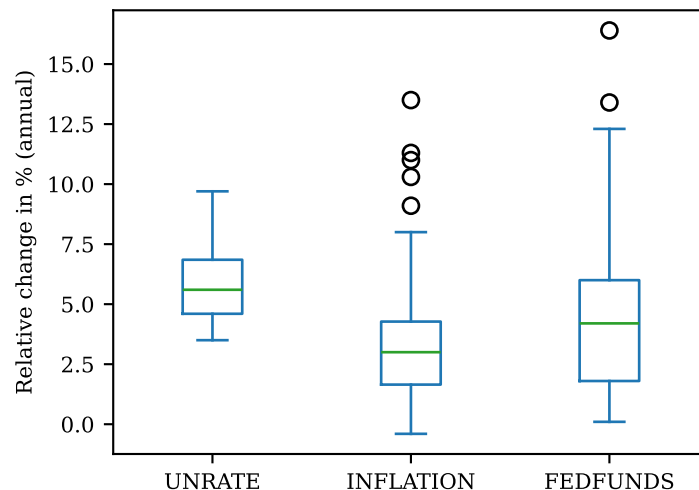
```python
import pandas as pd

# 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)')
```

[27]: <Axes: ylabel='Relative change in % (annual)'>

3 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.

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]

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]

Figures with multiple plots

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

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]

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]