Lecture 4: Random numbers & plotting

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

February 4, 2025

See GitHub repository for notebooks and data:

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

Contents

1 Random number generation

In this section, we examine how to generate random numbers for various probability distributions in NumPy.

1.1 Random number generators

Currently, there are several ways to draw random numbers in Python:

1. The *new* programming interface implemented in NumPy, introduced in version 1.17 (the current version as of this writing is 2.2).

2. The *legacy* programming interface implemented in NumPy.

   While these functions have been superseded by the new implementation, they continue to work. If you are familiar with the legacy interface, you can read about what has changed in the new interface here.

3. The Python standard library itself also includes random number generators in the `random` module.

   We won't be using this implementation at all, since for our purposes `numpy.random` is preferable as it supports NumPy arrays.

The programming interface for generating random numbers in NumPy changed substantially in release 1.17. We discuss the new interface in this unit since it offers several advantages, including faster algorithms for some distributions. Moreover, one would expect the legacy interface to be removed at some point in the future. However, many examples you will find in textbooks and on the internet are likely to use the old variant.

A note on random number generation

Computers usually cannot draw truly random numbers, so we often talk about *pseudo-random number generators* (PRNG). Given an initial seed, these PRNGs will always produce the same sequence of "random" numbers, at least if run using the same software, the same underlying algorithm, etc. For scientific purposes this is actually desirable as it allows us to create reproducible results. For simplicity, we will nevertheless be using the terms "random number" and "random number generator" (RNG), omitting the "pseudo" prefix.

1.1.1 Simple random data generation

Before we can generate any random numbers using the new interface, we need to obtain an RNG instance. We can get the default RNG by calling `default_rng()` as follows:

[1]:
```
# import function that returns the default RNG
from numpy.random import default_rng

# get an instance of the default RNG
rng = default_rng()
```

Drawing random floats

Let's begin with the simplest case, which uses the `random()` function to draw numbers that are uniformly distributed on the half-open interval $[0, 1)$.

[2]:
```
from numpy.random import default_rng
rng = default_rng()         # obtain default RNG implementation

rng.random(5)               # return array of 5 random numbers
```

[2]:
```
array([0.78240243, 0.51743546, 0.25704371, 0.89384046, 0.92706731])
```

Calling `random()` this way will return a different set of numbers each time (this might, for example, depend on the system time). To obtain the same draw each time, we can pass an initial *seed* when creating an instance of the RNG like this:

[3]:
```
seed = 123
rng = default_rng(seed)     # obtain default RNG implementation,
                            # initialize seed

rng.random(5)               # return array of 5 random numbers
```

[3]:
```
array([0.68235186, 0.05382102, 0.22035987, 0.18437181, 0.1759059 ])
```

The `seed` argument needs to be an integer or an array of integers. This way, each call gives the same numbers as can easily be illustrated with a loop:

[4]:
```
seed = 123
for i in range(5):
    rng = default_rng(seed)
    print(rng.random(5))
```

```
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
[0.68235186 0.05382102 0.22035987 0.18437181 0.1759059 ]
```

You can remove the `seed` to verify that the set of number will then differ in each iteration.

Drawing random integers

Alternatively, we might want to draw random integers by calling `integers()`, which returns numbers from a "discrete uniform" distribution on a given interval:

[5]:
```
rng.integers(2, size=5)          # vector of 5 integers from set {0, 1}
                                 # here we specify only the (non-inclusive)
                                 # upper bound 2
```

[5]:
```
array([0, 1, 0, 1, 0])
```

Alternatively, we can specify the lower and upper bounds like this:

[6]:
```
rng.integers(1, 10, size=5)      # specify lower and upper bound
```

[6]:
```
array([3, 8, 8, 8, 9])
```

Following the usual convention in Python, the upper bound is not included by default. We can change this by additionally passing `endpoint=True`:

[7]:
```
rng.integers(1, 10, size=5, endpoint=True)      # include upper bound
```

[7]:
```
array([1, 6, 3, 3, 3])
```

We can create higher-order arrays by passing a list or tuple as the `size` argument:

[8]:
```
rng.random(size=(2, 5))              # Create 2x5 array of floats
                                     # on [0.0, 1.0)
```

[8]:
```
array([[0.21376296, 0.74146705, 0.6299402 , 0.92740726, 0.23190819],
       [0.79912513, 0.51816504, 0.23155562, 0.16590399, 0.49778897]])
```

[9]:
```
rng.integers(2, size=(2,3,4))        # Create 2x3x4 array of integers {0,1}
```

[9]:
```
array([[[1, 0, 1, 0],
        [0, 0, 0, 0],
        [0, 0, 1, 0]],

       [[1, 0, 1, 1],
        [1, 1, 1, 0],
        [0, 0, 1, 0]]])
```

Legacy interface

For completeness, let's look at how you would accomplish the same using the *legacy* NumPy interface.

To draw floats on the unit interval, we use `random_sample()`:

[10]:
```
from numpy.random import random_sample, randint, seed

# Set seed globally using legacy interface
seed(123)

# Draw random floats
random_sample(5)
```

[10]:
```
array([0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897])
```

Random integers can be generated using `randint()`:

[11]:
```
randint(2, size=5)      # draw random integers from {0,1}
```

[11]:
```
array([1, 1, 0, 1, 0])
```

The legacy interface defines global functions `seed()`, `random_sample()`, etc. within the `numpy.random` module, which are implicitly associated with a global RNG object. This implicit association has been removed in the new programming model and you now have to obtain an RNG instance explicitly, for example by using the `default_rng()` function, as demonstrated above.

1.1.2 Drawing random numbers from distributions

Often we want to draw random numbers from a specific distribution such as the normal or log-normal distributions. The RNGs in `numpy.random` support a multitude of distributions, including:

- `binomial()`
- `exponential()`
- `normal()`
- `lognormal()`
- `multivariate_normal()`
- `uniform()`
- `standard_t()`

and many others. For a complete list, see the official documentation.

Example: Drawing from a normal distribution

We can draw from the normal distribution with mean $\mu = 1.0$ and standard deviation $\sigma = 0.5$ using `normal()` as follows:

[12]:
```
from numpy.random import default_rng

# Get RNG instance with given seed
rng = default_rng(123)

# location and scale parameters of normal distribution
mu = 1.0
sigma = 0.5

# Draw 10 normal numbers;
# mean and std. are passed as loc and scale arguments
rng.normal(loc=mu, scale=sigma, size=10)
```

[12]:
```
array([0.50543932, 0.81610667, 1.64396263, 1.09698721, 1.46011545,
       1.2885519 , 0.68176818, 1.27097611, 0.84170227, 0.83880544])
```

Example: Drawing from a multivariate normal distribution

To draw from the multivariate normal, we need to specify a vector of means $\boldsymbol{\mu}$ and the variance-covariance matrix $\boldsymbol{\Sigma}$, which we set to

$$\boldsymbol{\mu} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}$$

with $\sigma_1 = 0.5$, $\sigma_2 = 1.0$ and $\rho = 0.5$. We call `multivariate_normal()` to draw a sample:

[13]:
```
import numpy as np
from numpy.random import default_rng

# Get RNG instance with given seed
rng = default_rng(123)

mu = np.array((0.0, 1.0))       # vector of means
sigma1 = 0.5                    # Std. dev. of first dimension
sigma2 = 1.0                    # Std. dev. of second dimension
rho = 0.5                       # Correlation coefficient

# Compute covariance
cov = rho * sigma1 * sigma2

# Create variance-covariance (VCV) matrix
vcv = np.array([[sigma1**2.0, cov],
                [cov, sigma2**2.0]])

# Print the variance covariance matrix
vcv
```

[13]:
```
array([[0.25, 0.25],
       [0.25, 1.  ]])
```

[14]:
```
# Draw MVN random numbers:
# each row represents one sample draw.
rng.multivariate_normal(mean=mu, cov=vcv, size=10)
```

[14]:
```
array([[-0.44424426,  0.06264099],
       [ 0.46459735,  2.25499666],
       [ 0.50717927,  1.84365041],
       [ 0.02526624,  0.30264359],
       [-0.22397535,  0.72473434],
       [-0.58053624,  1.28106799],
       [ 0.09014951,  2.26460055],
       [ 0.35510451,  1.97642901],
       [ 0.19674056,  2.6006256 ],
       [ 0.0412573 ,  0.64963054]])
```

Your turn. Use the `uniform()` function from NumPy to draw a sample of 10 uniformly distributed numbers on the interval $[-1, 1]$. Use a seed of 456 for this exercise.

In the next section, we'll study how we can visualize draws of random numbers using scatter plots and histograms.

2 Plotting with Matplotlib

In this section, we study how to plot numerical data. Python itself does not have any built-in plotting capabilities, so we will be using `matplotlib` (MPL), 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
```

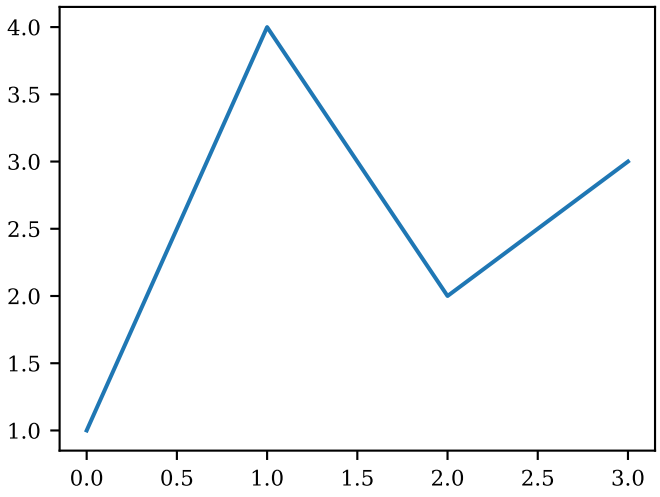
Note that there is an additional high-level plotting library called seaborn which builds on top of Matplotlib with a focus on providing convenient functions to create statistical graphs.

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.

[15]:
```
# import matplotlib library
import matplotlib.pyplot as plt

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

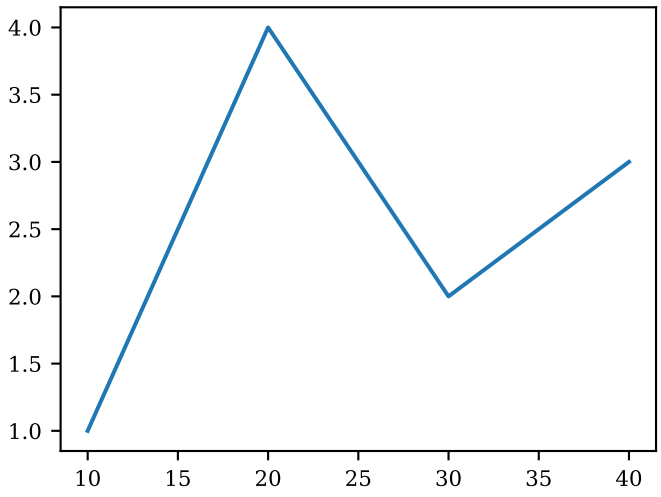
[15]:
```
[<matplotlib.lines.Line2D at 0x7f5041d20710>]
```



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

[16]:
```
# explicitly specify x-values
xvalues = [10, 20, 30, 40]
plt.plot(xvalues, yvalues)
```

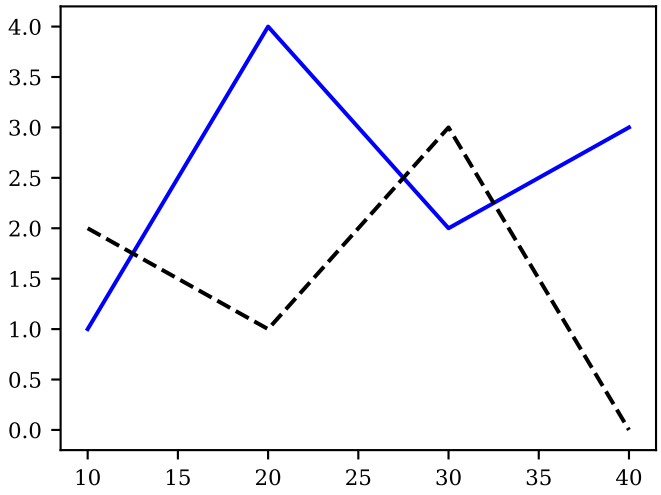
[16]:
```
[<matplotlib.lines.Line2D at 0x7f5041d4f0e0>]
```



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

[17]:
```
yvalues2 = [2.0, 1.0, 3.0, 0.0]
plt.plot(xvalues, yvalues, 'b-', xvalues, yvalues2, 'k--')
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f503216cce0>,
 <matplotlib.lines.Line2D at 0x7f503216cc20>]
```



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

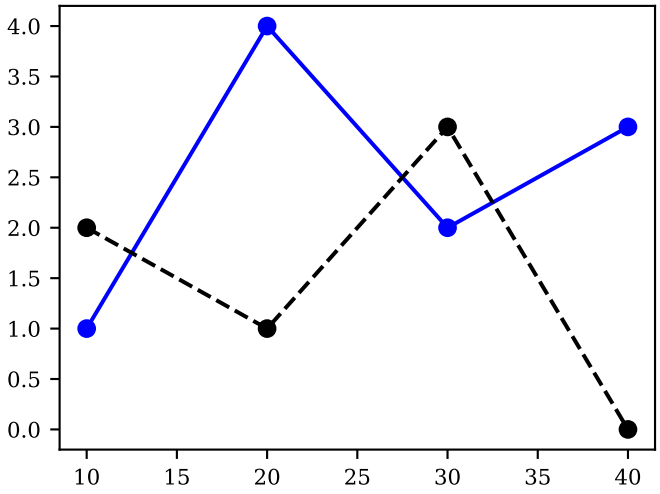
- `o`: circle
- `s`: square
- `*`: star
- `x`: x
- `d`: (thin) diamond

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.

[18]:
```
# Plot two lines by calling plot() twice
plt.plot(xvalues, yvalues, 'b-o')
plt.plot(xvalues, yvalues2, 'k--o')
```

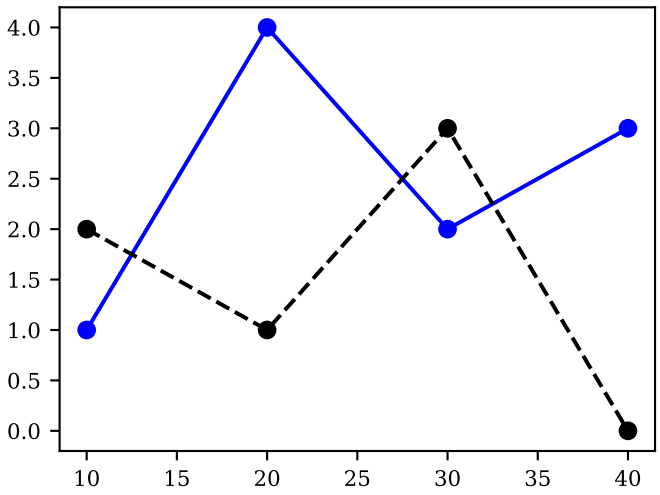
[18]:
```
[<matplotlib.lines.Line2D at 0x7f502f715ca0>]
```



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

[19]:
```
# 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')
```

[19]:
```
[<matplotlib.lines.Line2D at 0x7f502f7858b0>]
```



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:

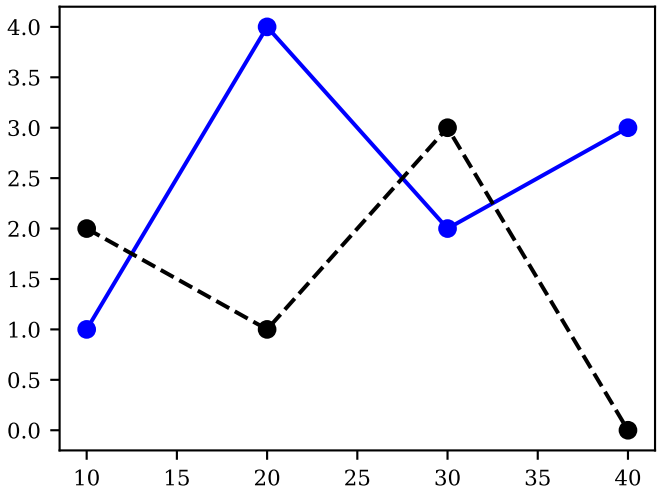
- `c` or `color`
- `ls` or `linestyle`
- `lw` or `linewidth`
- `ms` or `markersize`

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

We can write thus rewrite the above code as follows:

[20]:
```
# abbreviate plot style keywords
plt.plot(xvalues, yvalues, c='blue', ls='-', marker='o')
plt.plot(xvalues, yvalues2, c='black', ls='--', marker='o')
```

[20]:
```
[<matplotlib.lines.Line2D at 0x7f502cd19d90>]
```



Your turn. Consider the standard CRRA utility function given by $u(c) = c^{1-\gamma}/(1-\gamma)$. Create a graph that shows function over the interval $[0.1, 2]$ containing two lines for two distinct values of γ:

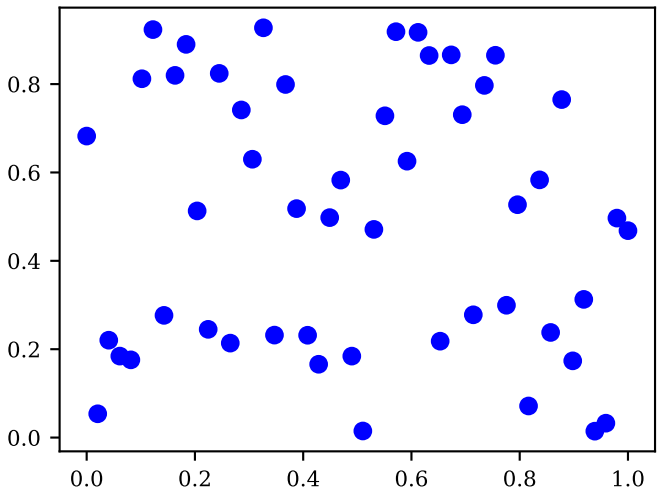
1. $\gamma = 1.1$ using a blue dashed line with a line width of 0.5.
2. $\gamma = 2$ using an orange line with line width of 0.75.

2.2 Scatter plots

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

[21]:
```
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
yvalues = np.random.default_rng(123).random(N)

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

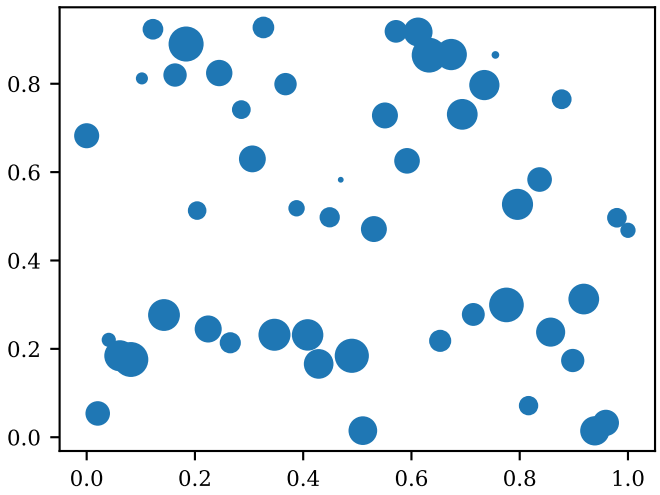
[21]:
```
<matplotlib.collections.PathCollection at 0x7f502cd6b920>
```



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.

[22]:
```
# Draw random marker sizes
size = np.random.default_rng(456).random(N) * 150.0

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

[22]:
```
<matplotlib.collections.PathCollection at 0x7f502a3107a0>
```



Your turn. Recall the example on how to draw a sample from a multivariate normal distribution covered in the previous section.

1. Repeat the exercise, but now draw a sample of size 100.
2. Visualize the first vs. the second dimension of this random sample as a scatter plot.

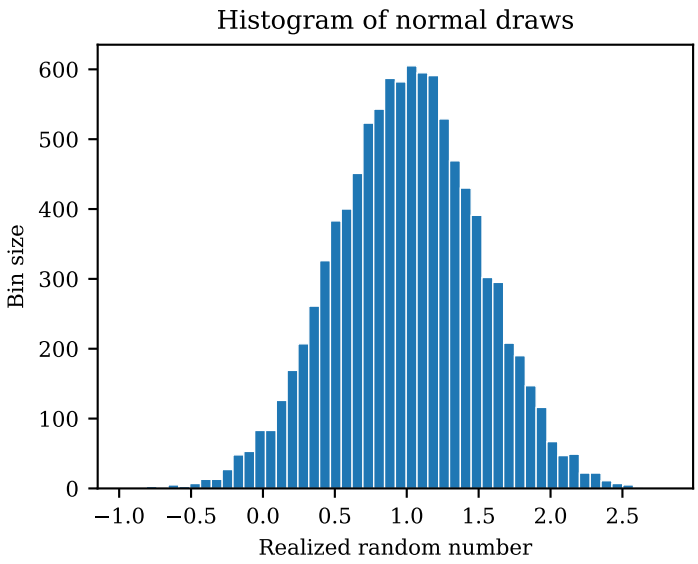
2.3 Histograms

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

To illustrate, we revisit the random sample drawn from a normal distribution studied in the previous section and visualize it using a histogram:

[23]:
```
from numpy.random import default_rng
import matplotlib.pyplot as plt

# Get RNG instance
rng = default_rng(123)

# location and scale parameters of normal distribution
mu = 1.0
sigma = 0.5

# Draw 10000 normal numbers;
# mean and std. are passed as loc and scale arguments
x = rng.normal(loc=mu, scale=sigma, size=10000)

# 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')
```

[23]:
```
Text(0.5, 1.0, 'Histogram of normal draws')
```



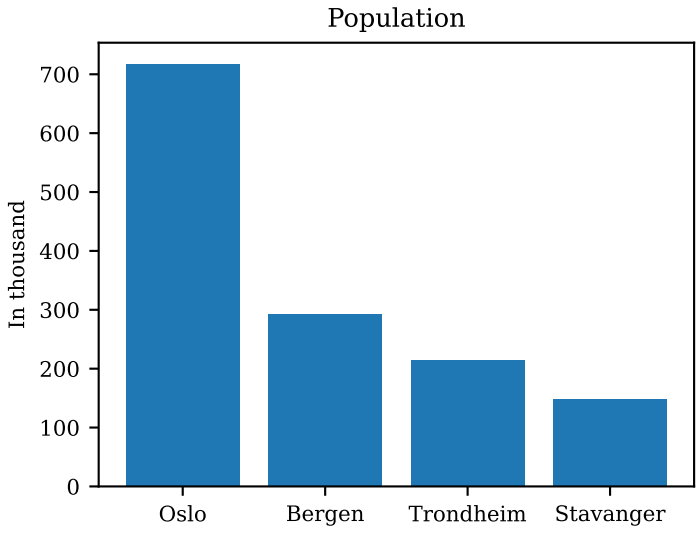
2.4 Plotting categorical data

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

For example, assume we have four categories and each has an associated numerical value:

[24]:
```
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')
```

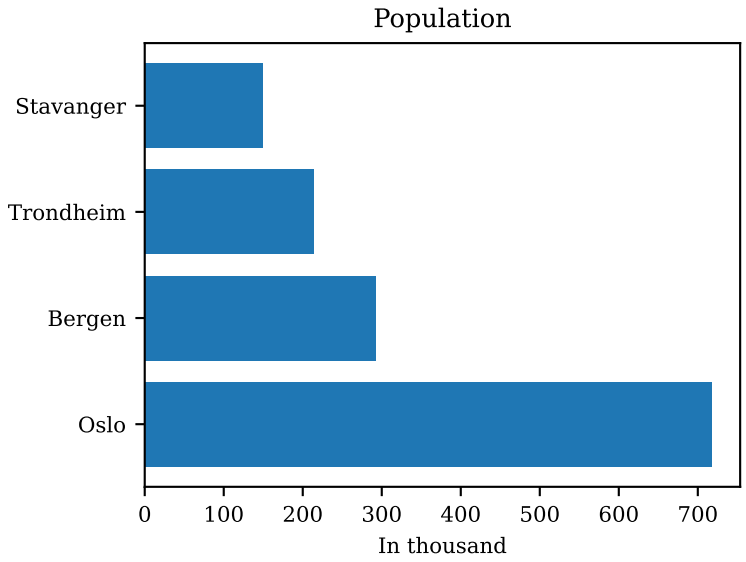
[24]:
```
Text(0, 0.5, 'In thousand')
```



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

[25]:
```
plt.barh(municipality, population)
plt.title('Population')
plt.xlabel('In thousand')
```

[25]:
```
Text(0.5, 0, 'In thousand')
```



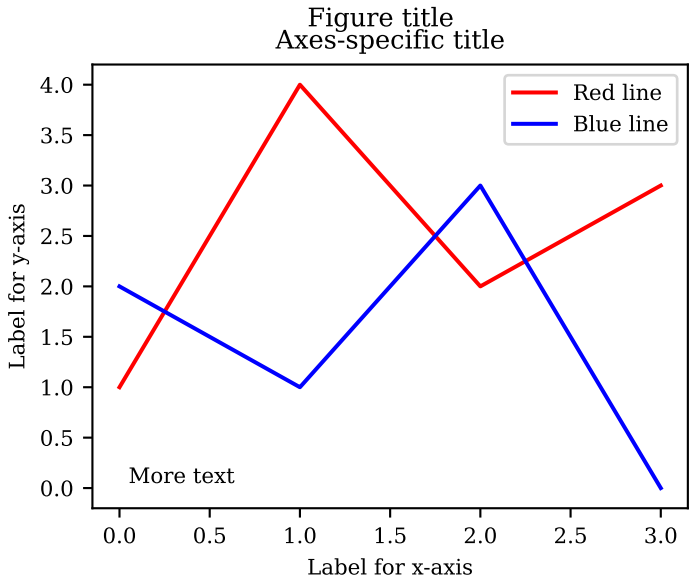
2.5 Adding labels and annotations

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.

[26]:
```
import matplotlib.pyplot as plt

xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]

plt.plot(xvalues, yvalues, 'r', xvalues, yvalues2, 'b')
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')
```

[26]:
```
Text(0.05, 0.05, 'More text')
```



2.6 Plot limits, ticks and tick labels

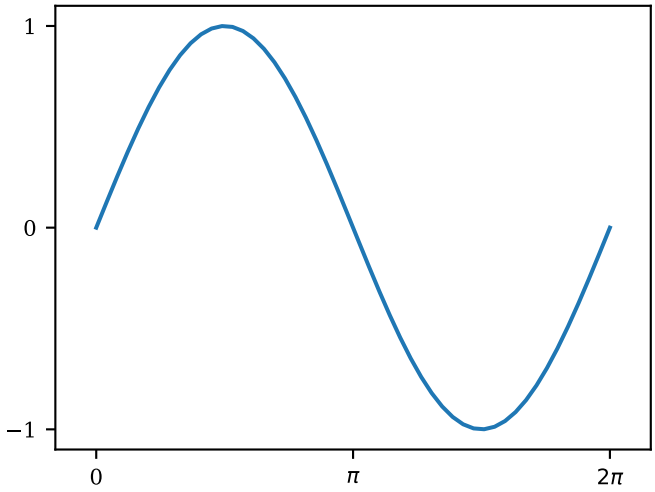
We adjust the plot limits, ticks and tick labels as follows:

- 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()`.

[27]:
```
import matplotlib.pyplot as plt
import numpy as np

xvalues = np.linspace(0.0, 2*np.pi, 50)
plt.plot(xvalues, np.sin(xvalues))

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

[27]: `(-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:

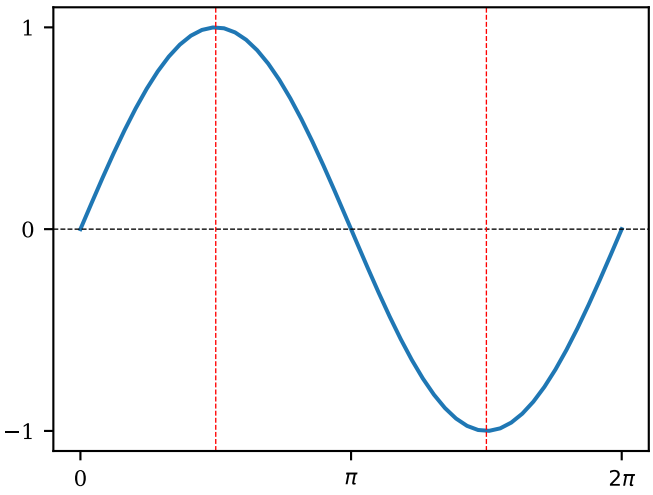
- `axhline()` adds a *horizontal* line at a given y-value.
- `axvline()` adds a *vertical* line at a given x-value.
- `axline()` adds a line defined by two points or by a single point and a slope.

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:

[28]:
```
import matplotlib.pyplot as plt
import numpy as np

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

# Set major ticks and labels for x-axis
plt.xticks([0.0, np.pi, 2*np.pi], ['0', r'$\pi$', r'$2\pi$'])
# 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')
```

[28]: `<matplotlib.lines.Line2D at 0x7f5024fc0e00>`



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 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:

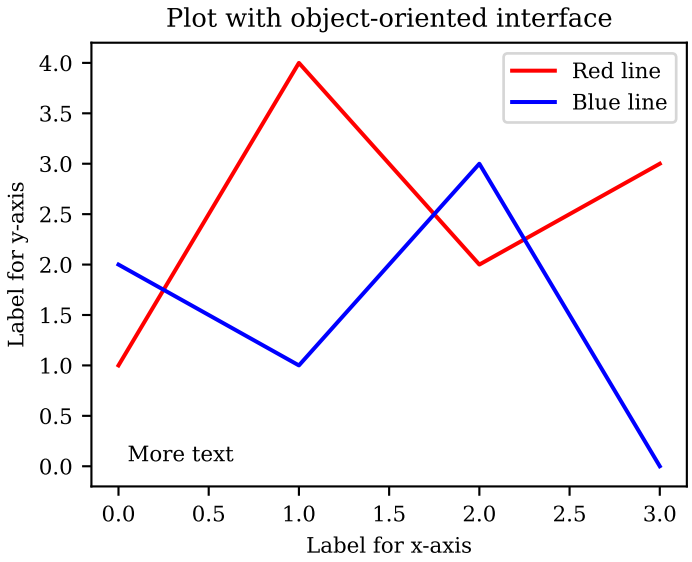
```
fig, ax = plt.subplots()
```

We then use methods of the `Axes` object returned by `subplots()` instead of the functions we have used so for. 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:

[29]:
```
import matplotlib.pyplot as plt

xvalues = [0, 1, 2, 3]
yvalues = [1, 4, 2, 3]
yvalues2 = [2.0, 1.0, 3.0, 0.0]

fig, ax = plt.subplots()
ax.plot(xvalues, yvalues, color='red', label='Red line')
ax.plot(xvalues, yvalues2, color='blue', label='Blue line')
ax.set_xlabel('Label for x-axis')
ax.set_ylabel('Label for y-axis')
ax.legend()
ax.set_title('Plot with object-oriented interface')
ax.text(0.05, 0.05, 'More text')
```

[29]:
```
Text(0.05, 0.05, 'More 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.

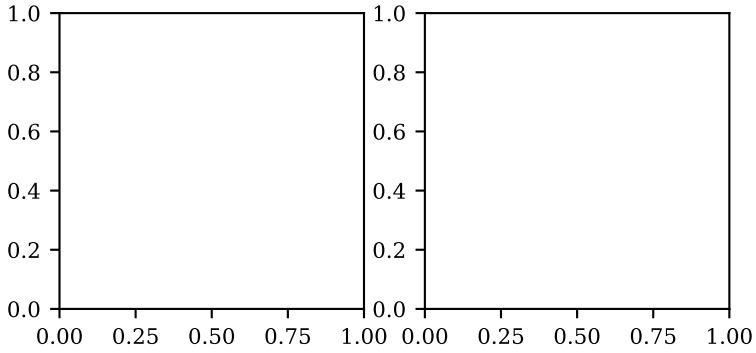
1. Repeat these steps, but create the histogram using the object-oriented Matplotlib interface.
2. Set the x-axis label to 'Realized random number' and the y-axis label to 'Bin size'.
3. Add the title 'Histogram of normal draws'.

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:

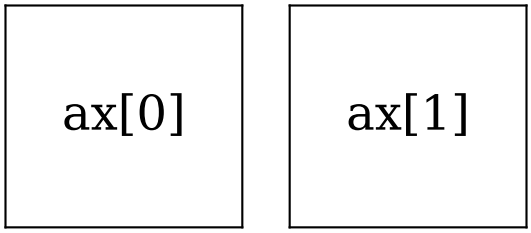
[30]:
```
import matplotlib.pyplot as plt

# 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 `ax` 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:

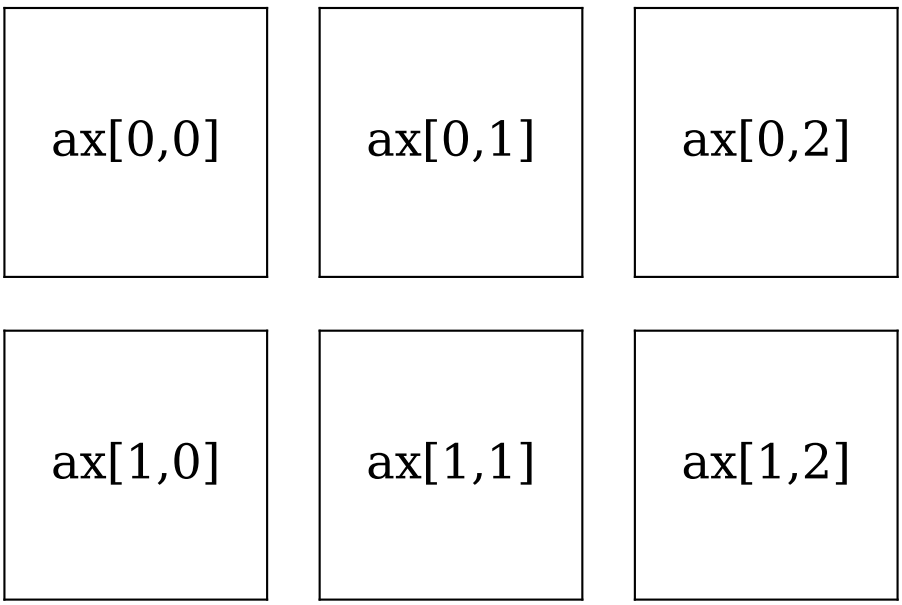
[31]:
```
fig, axes = plt.subplots(1, 2, figsize=(3.5,1.5))

for i, ax in enumerate(axes):
    # Turn off ticks of both axes
    ax.set_xticks(())
    ax.set_yticks(())
    # Label axes object
    text = f'ax[{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 `ax` object will be a 2-dimensional array, and the mapping of axes objects to panels will look like this instead:

[32]:
```
# Request figure with 2 rows, 3 columns
fig, axes = plt.subplots(2, 3, figsize=(6,4))

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

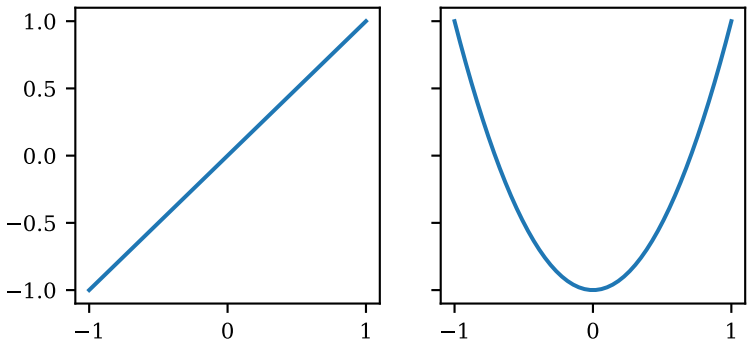


Example: Create a plot with 2 panels

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

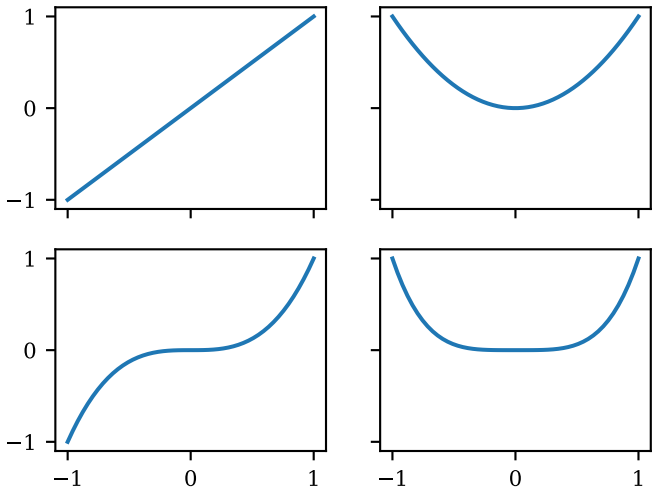
[33]:
```
import matplotlib.pyplot as plt
import numpy as np

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)
```

[33]:
```
[<matplotlib.lines.Line2D at 0x7f5022569010>]
```



Example: Create a figure with 2 rows and 2 columns

[34]:
```
# create figure with 2 rows, 2 columns
fig, axes = plt.subplots(2, 2, 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(2):
    for j in range(2):
        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: $y = \sin(2x)$
3. Subplot 3: $y = \sin(4x)$

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

2.10 Plotting with pandas

Pandas is a library to process and analyze data in Python. We cover pandas later in the course, but this section summarizes how plotting can be done directly with pandas. You can skip it until we have introduced 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.

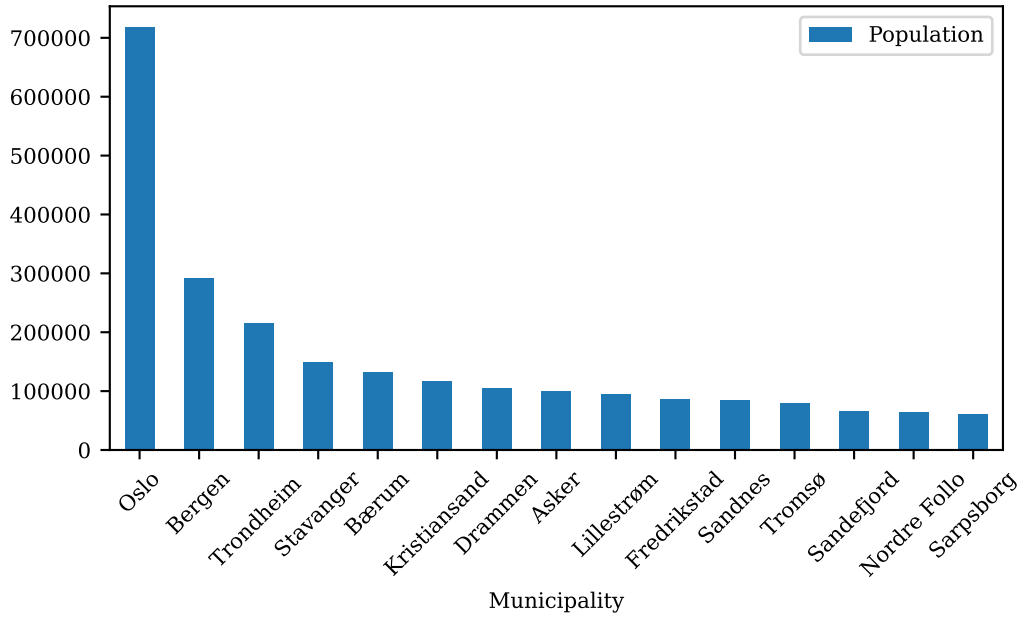
2.10.1 Bar charts

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

[35]:
```
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
```

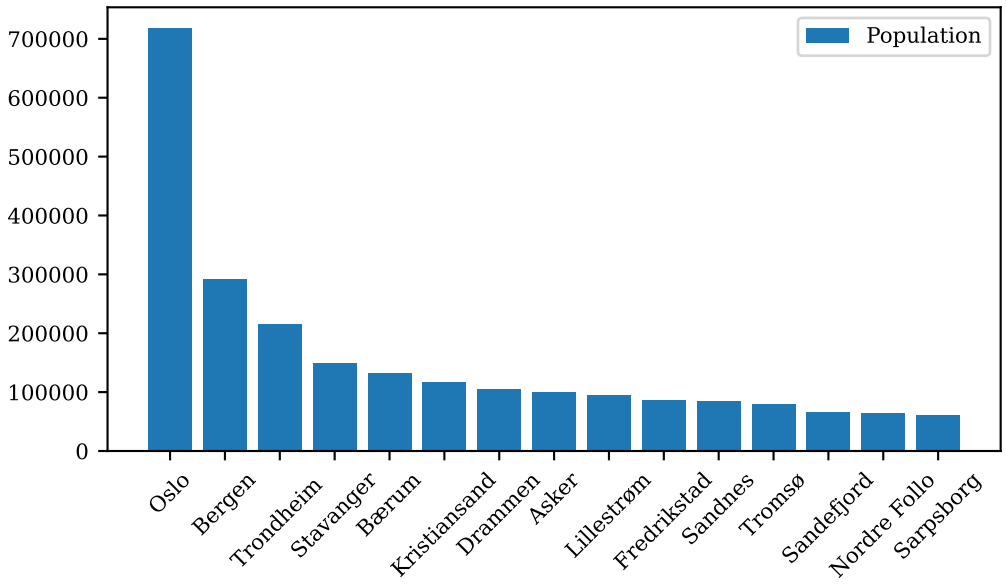
```
df = df.iloc[:15]

# 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))
```

[35]: <Axes: xlabel='Municipality'>



Alternatively, we can construct the graph ourselves using Matplotlib:

[36]:
```
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)
```



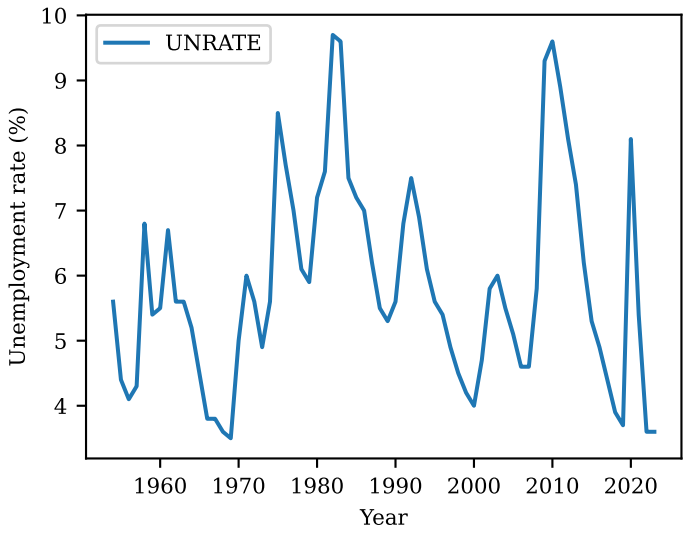
Matplotlib's functions usually directly work with pandas's data structures, In cases where they don't, we can convert a `DataFrame` or `Series` object to a NumPy array using the `to_numpy()` method.

2.10.2 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.

[37]:
```
import numpy as np
import pandas as pd

# Path to annual 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 (%)')
```

[37]:
```
<Axes: xlabel='Year', ylabel='Unemployment rate (%)'>
```



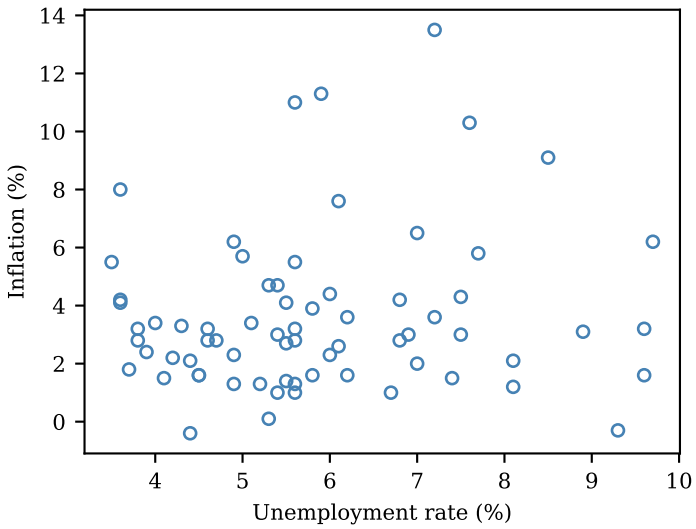
2.10.3 Scatter plots

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 to pandas's version of `scatter()` which are passed on to Matplotlib's `scatter()`, for example `edgecolors`.

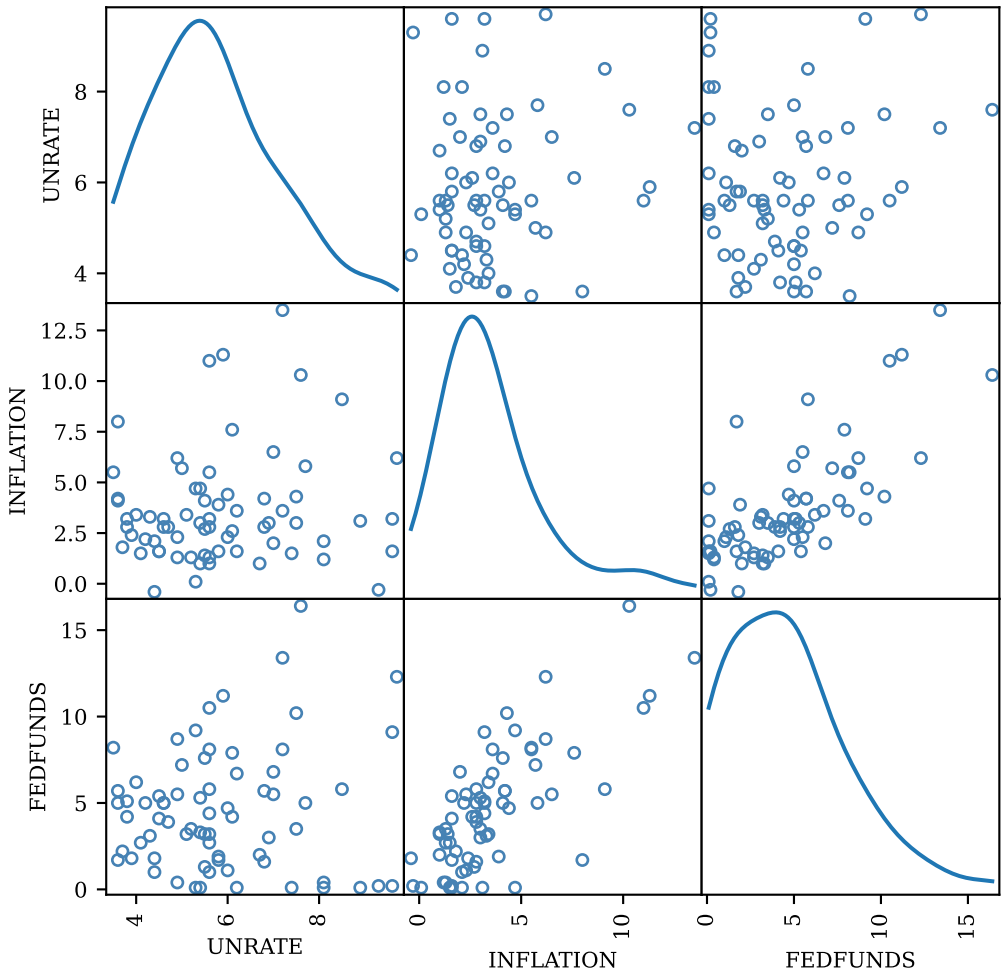
[38]:
```
# Path to annual FRED data; DATA_PATH variable was defined above!
filepath = f'{DATA_PATH}/FRED/FRED_annual.csv'

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

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 (%)'
)
```

[38]:
```
<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:

[39]:
```
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,
)
```



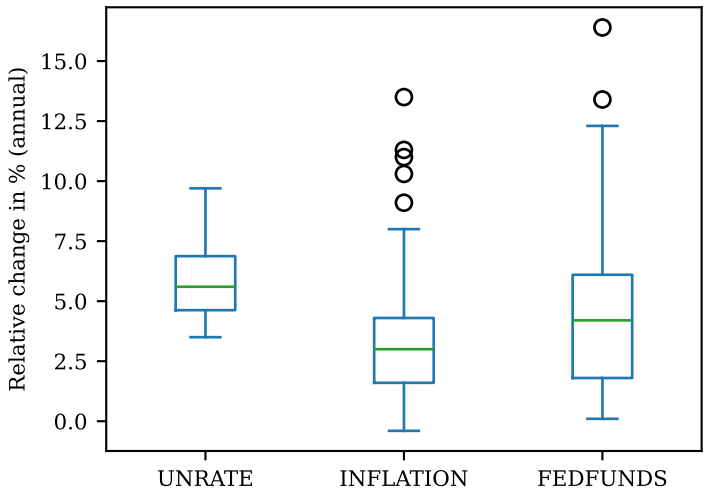
2.10.4 Box plots

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:

[40]:
```
import numpy as np
import pandas as pd

# Path to annual 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)')
```

[40]:
```
<Axes: ylabel='Relative change in % (annual)'>
```



2.11 Optional exercises

Exercise 1: Trigonometric functions

Plot the functions $\sin(x)$ and $\cos(x)$ on the interval $[-\pi, \pi]$, each in a separate graph. Include a legend for each plot, and add pretty tick labels at $[-\pi, 0, \pi]$ which use the LaTeX symbol for π. Add an overall title "Trigonometric functions".

Hint: NumPy defines the functions `np.sin()` and `np.cos()` as well as the value `np.pi`.

Exercise 2: Histograms for increasing sample sizes

In this exercise, we plot histograms against the actual PDF of a standard-t distributed random variable for increasing sample sizes.

Consider the standard-t distribution with 20 degrees of freedom (this is the only parameter of this distribution):

- To draw samples from this distribution, use NumPy RNG's `standard_t()` method.
- To plot the PDF of this distribution, use the `t` distribution from `scipy.stats`. You can import it as follows

  ```
  from scipy.stats import t as standard_t
  ```

  It is a good idea to assign more descriptive names to imported symbols than a `t`.

Perform the following tasks:

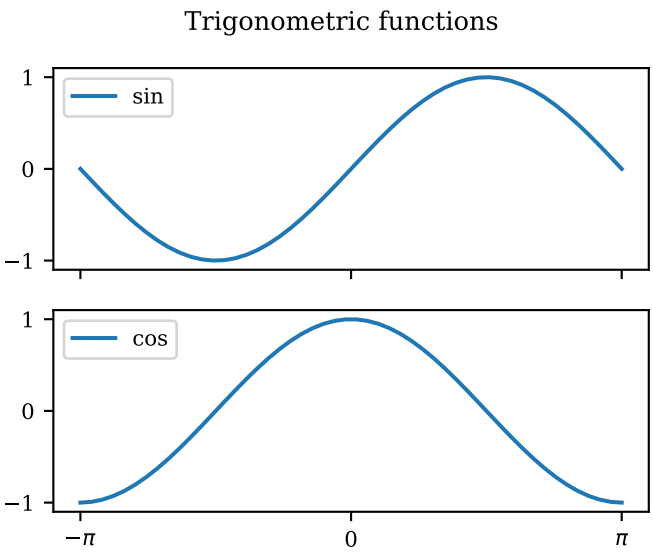
1. Draw random samples from the standard-t distribution for a sequence of increasing sample sizes of 50, 100, 500, 1000, 5000 and 10000.
2. Create a single figure with 6 panels in which you plot a histogram of the samples you have drawn. Use matplotlib's `hist()` function to do this, and pass the argument `bins = 50` so that each panel uses the same number of bins.
3. Add the actual PDF of the standard-t distribution to each panel. To evaluate the PDF, use the `pdf()` method of the `t` distribution you imported from `scipy.stats`, see here.

2.12 Solutions

Solution for exercise 1

[41]:
```
import matplotlib.pyplot as plt
import numpy as np

xvalues = np.linspace(-np.pi, np.pi, 50)
# Create figure with two rows, one column
fig, ax = plt.subplots(2, 1, sharey=True, sharex=True)

xticks = [-np.pi, 0.0, np.pi]
# Tick labels use LaTeX notation for pi, which is \pi and has to be
# surrounded by $$.
xticklabels = [r'$-\pi$', '0', r'$\pi$']
yticks = [-1.0, 0.0, 1.0]

# Create sin() plot using first axes object
ax[0].plot(xvalues, np.sin(xvalues), label='sin')
ax[0].set_xticks(xticks)
ax[0].set_xticklabels(xticklabels)
ax[0].set_yticks(yticks)
ax[0].legend(loc='upper left')

# Create cos() plot using second axes object
ax[1].plot(xvalues, np.cos(xvalues), label='cos')
ax[1].set_xticks(xticks)
ax[1].set_xticklabels(xticklabels)
ax[1].set_yticks(yticks)
ax[1].legend(loc='upper left')

# Add overall figure title (this is not axes-specific)
fig.suptitle('Trigonometric functions')
```

[41]:
```
Text(0.5, 0.98, 'Trigonometric functions')
```



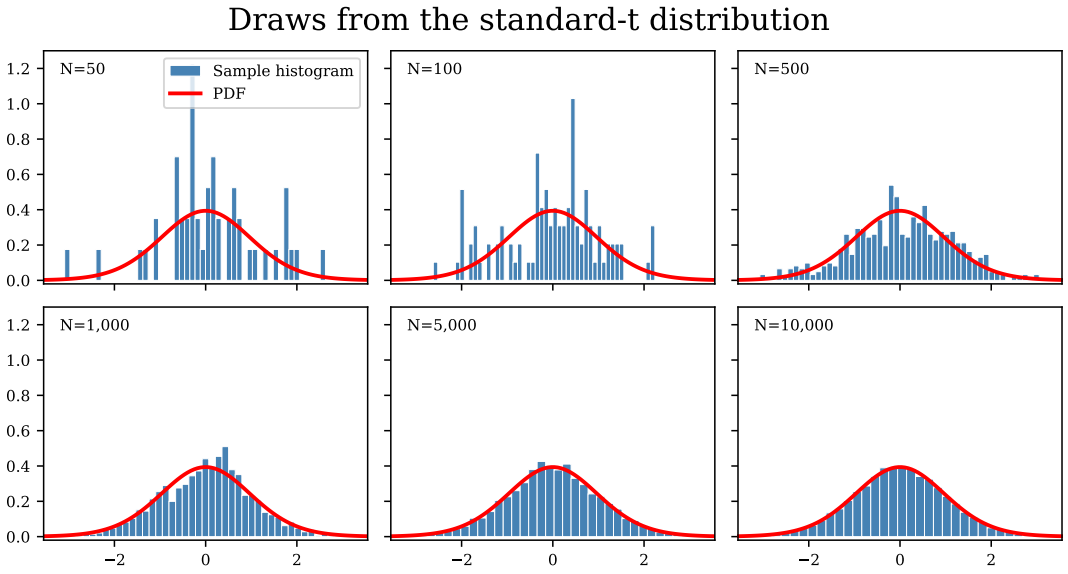
Solution for exercise 2

In the following solution, we create a figure with six panels (axes) and iterate over these axes. In each iteration, we

1. draw a random sample for the given (increasing) size;
2. plot the histogram using the current axes object; and
3. overlay the actual PDF.

[42]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import default_rng
from scipy.stats import t as standard_t

# Sample sizes
Nobs = np.array((50, 100, 500, 1000, 5000, 10000))

# degrees of freedom
df = 20

# Determine xlims such that we cover the (0.1, 99.9) percentiles
# of the distribution.
xmin, xmax = standard_t.ppf((0.001, 0.999), df=df)

xvalues = np.linspace(xmin, xmax, 100)
pdf = standard_t.pdf(xvalues, df=df)

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True, figsize=(8, 4))

# initialize default RNG
rng = default_rng(123)

for i, ax in enumerate(ax.flatten()):
    # Sample size to be plotted in current panel
    N = Nobs[i]
    # Draw sample of size N
    data = rng.standard_t(df=df, size=N)

    # plot histogram of given sample
    ax.hist(data, bins=50, linewidth=0.5, edgecolor='white',
              color='steelblue', density=True, label='Sample histogram')

    # overlay actual PDF
    ax.plot(xvalues, pdf, color='red', lw=2.0, label='PDF')

    # create text with current sample size
    ax.text(0.05, 0.95, f'N={N:,d}', transform=ax.transAxes, va='top')

    ax.set_xlim((xmin, xmax))
    ax.set_ylim((-0.02, 1.3))

    # plot legend only for the first panel
    if i == 0:
        ax.legend(loc='upper right')

# compress space between individual panels
fig.tight_layout()
# Add overall title
fig.suptitle('Draws from the standard-t distribution', fontsize=16, y=1.05)
```

[42]:
```
Text(0.5, 1.05, 'Draws from the standard-t distribution')
```


Draws from the standard-t distribution
N=50
Sample histogram
PDF
N=100
N=500
N=1,000
N=5,000
N=10,000
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−2
0
2