Open in Colab

View source code | Read notebook in online book format

A Quick Introduction to Matplotlib

In [1]:
```
import datetime
print(f"Last updated: {datetime.datetime.now()}")
```

```
Last updated: 2024-04-26 12:34:50.042711
```

What is matplotlib?

Matplotlib is a visualization library for Python.

As in, if you want to display something in a chart or graph, matplotlib can help you do that programmatically.

Many of the graphics you'll see in machine learning research papers or presentations are made with matplotlib.

Table of contents

What is matplotlib?

Why matplotlib?

What does this notebook cover?

Where can I get help?

0. Importing matplotlib

1. 2 ways of creating plots

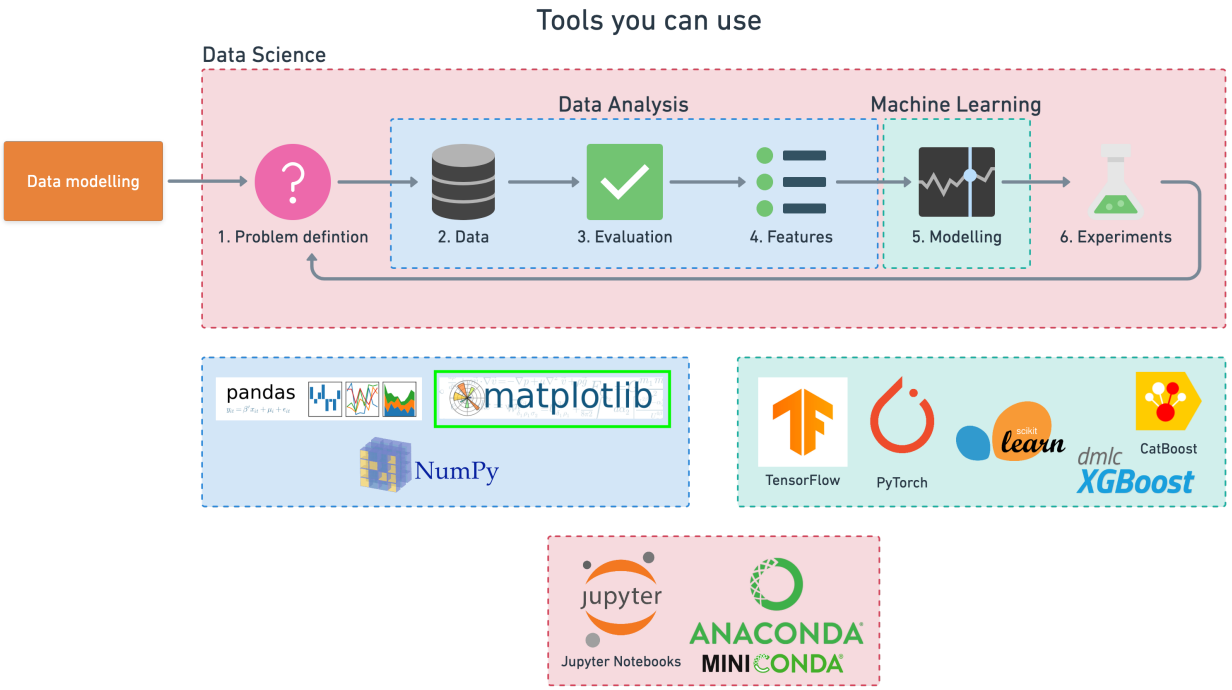
- Anatomy of a Matplotlib Figure
- A quick Matplotlib Workflow

2. Making the most common type of plots using NumPy arrays

- Creating a line plot
- Creating a scatter plot
- Creating bar plots
- Creating a histogram plot
- Creating Figures with multiple Axes with Subplots

3. Plotting data directly with pandas

- Line plot from a pandas DataFrame
- Working with actual data



Scatter plot from a pandas DataFrame

Bar plot from a pandas

Why matplotlib?

Matplotlib is part of the standard Python data stack (pandas, NumPy, matplotlib, Jupyter).

It has terrific integration with many other Python libraries.

pandas uses matplotlib as a backend to help visualize data in DataFrames.

What does this notebook cover?

A central idea in matplotlib is the concept of a "plot" (hence the name).

So we're going to practice making a series of different plots, which is a way to visually represent data.

Since there are basically limitless ways to create a plot, we're going to focus on a making and customizing (making them look pretty) a few common types of plots.

Where can I get help?

If you get stuck or think of something you'd like to do which this notebook doesn't cover, don't fear!

The recommended steps you take are:

1. **Try it** - Since matplotlib is very friendly, your first step should be to use what you know and try figure out the answer to your own question (getting it wrong is part of the process). If in doubt, run your code.
2. **Search for it** - If trying it on your own doesn't work, since someone else has probably tried to do something similar, try searching for your problem in the following places (either via a search engine or direct):
    - matplotlib documentation - the best place for learning all of the vast functionality of matplotlib. **Bonus:** You can see a series of matplotlib cheatsheets on the matplotlib website.

- Stack Overflow - this is the developers Q&A hub, it's full of questions and answers of different problems across a wide range of software development topics and chances are, there's one related to your problem.
- ChatGPT - ChatGPT is very good at explaining code, however, it can make mistakes. Best to verify the code it writes first before using it. Try asking "Can you explain the following code for me? {your code here}" and then continue with follow up questions from there.

An example of searching for a matplotlib feature might be:

> "how to colour the bars of a matplotlib plot"

Searching this on Google leads to this documentation page on the matplotlib website: https://matplotlib.org/stable/gallery/lines_bars_and_markers/bar_colors.html

The next steps here are to read through the post and see if it relates to your problem. If it does, great, take the code/information you need and **rewrite it** to suit your own problem.

3. **Ask for help** - If you've been through the above 2 steps and you're still stuck, you might want to ask your question on Stack Overflow or in the ZTM Discord chat. Remember to be specific as possible and provide details on what you've tried.

Remember, you don't have to learn all of these functions off by heart to begin with.

What's most important is remembering to continually ask yourself, "what am I trying to visualize?"

Start by answering that question and then practicing finding the code which does it.

Let's get to visualizing some data!

0. Importing matplotlib

We'll start by importing `matplotlib.pyplot`.

Why `pyplot`?

Because `pyplot` is a submodule for creating interactive plots programmatically.

`pyplot` is often imported as the alias `plt`.

> **Note:** In older notebooks and tutorials of matplotlib, you may see the magic command `%matplotlib inline`. This was required to view plots inside a notebook, however, as of 2020 it is mostly no longer required.

In [2]:
```
# Older versions of Jupyter Notebooks and matplotlib required this magic command
# %matplotlib inline

# Import matplotlib and matplotlib.pyplot
import matplotlib
import matplotlib.pyplot as plt

print(f"matplotlib version: {matplotlib.__version__}")
```

```
matplotlib version: 3.6.3
```

1. 2 ways of creating plots

There are two main ways of creating plots in matplotlib.

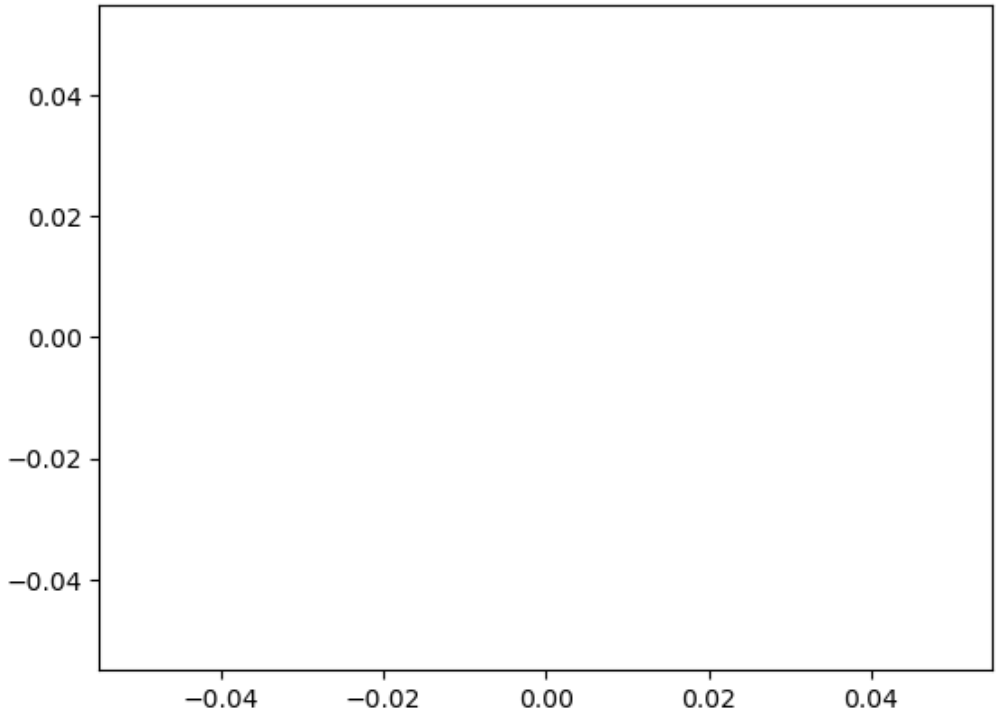
1. `matplotlib.pyplot.plot()` - Recommended for simple plots (e.g. x and y).
2. `matplotlib.pyplot.XX` (where XX can be one of many methods, this is known as the object-oriented API) - Recommended for more complex plots (for example `plt.subplots()` to create multiple plots on the same Figure, we'll get to this later).

Both of these methods are still often created by building off `import matplotlib.pyplot as plt` as a base.

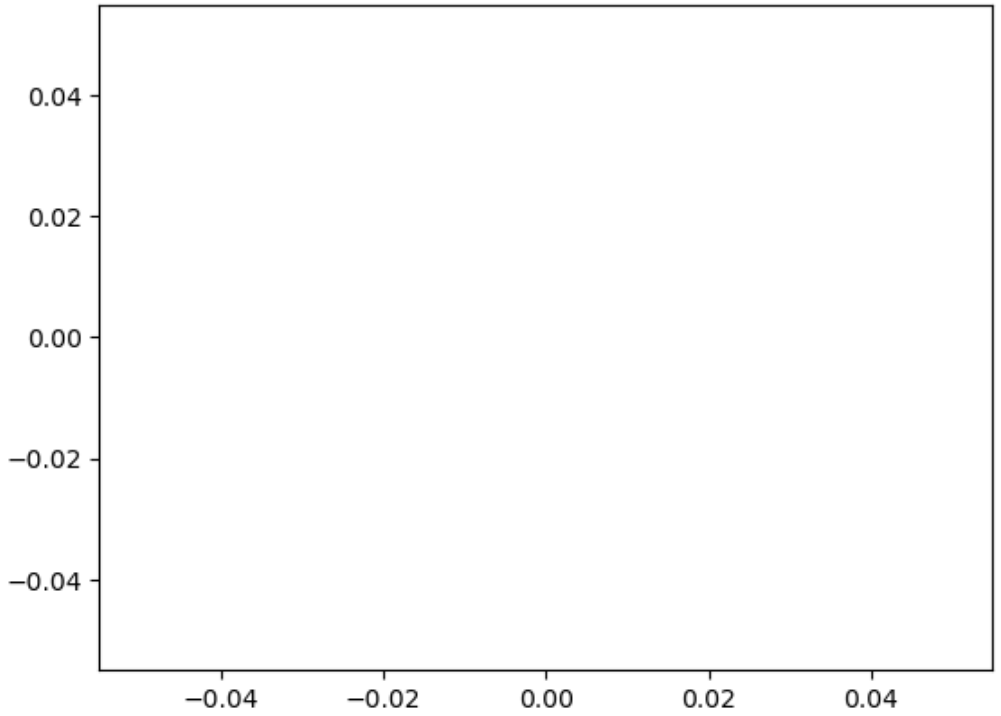
Let's start simple.

In [3]:
```
# Create a simple plot, without the semi-colon
plt.plot()
```

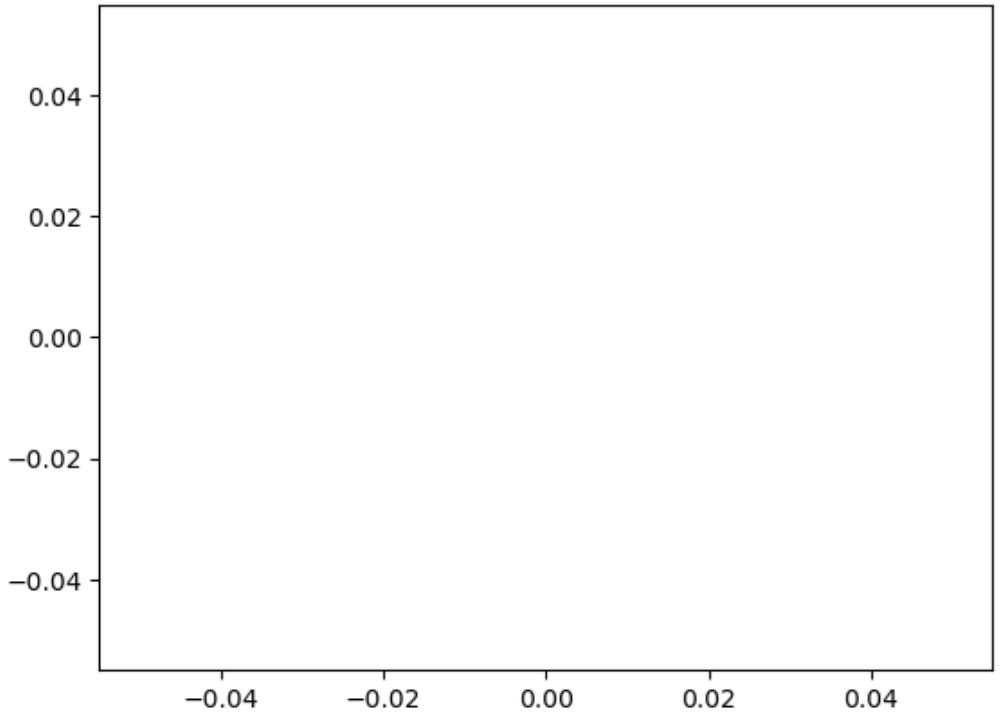
Out[3]:
```
[]
```



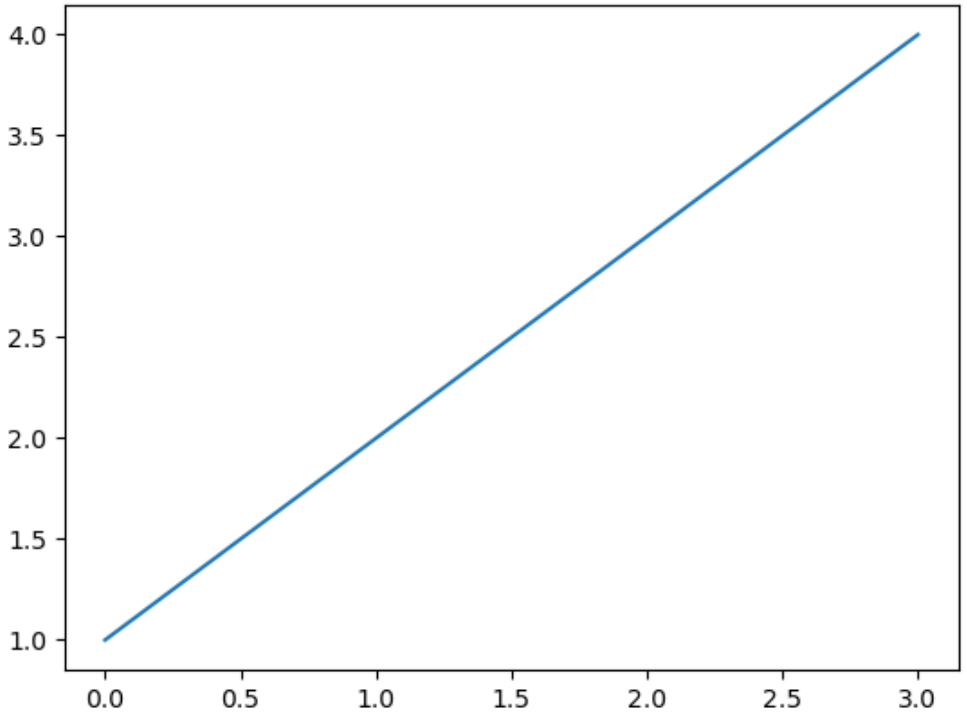
In [4]:
```
# With the semi-colon
plt.plot();
```



In [5]:
```
# You could use plt.show() if you want
plt.plot()
plt.show()
```



In [6]:
```
# Let's add some data
plt.plot([1, 2, 3, 4]);
```



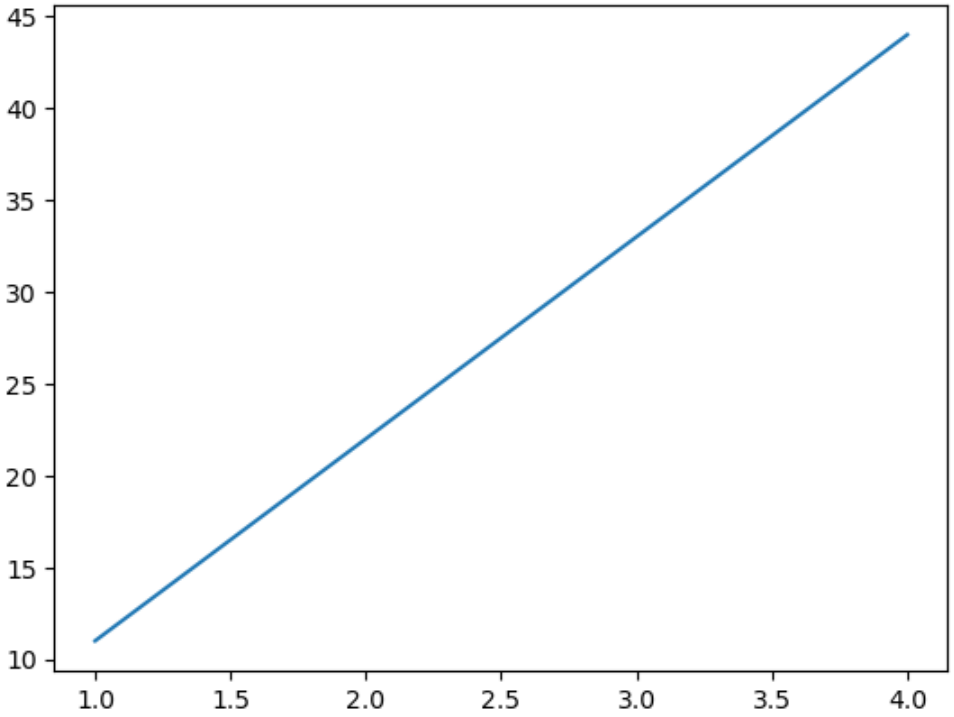
In [7]:
```
# Create some data
x = [1, 2, 3, 4]
y = [11, 22, 33, 44]
```

A few quick things about a plot:

- `x` is the horizontal axis.
- `y` is the vertical axis.
- In a data point, `x` usually comes first, e.g. `(3, 4)` would be `(x=3, y=4)`.

- The same is happens in `matplotlib.pyplot.plot()`, `x` comes before `y`, e.g. `plt.plot(x, y)`.

In [8]:
```
# Now a y-value too!
plt.plot(x, y);
```

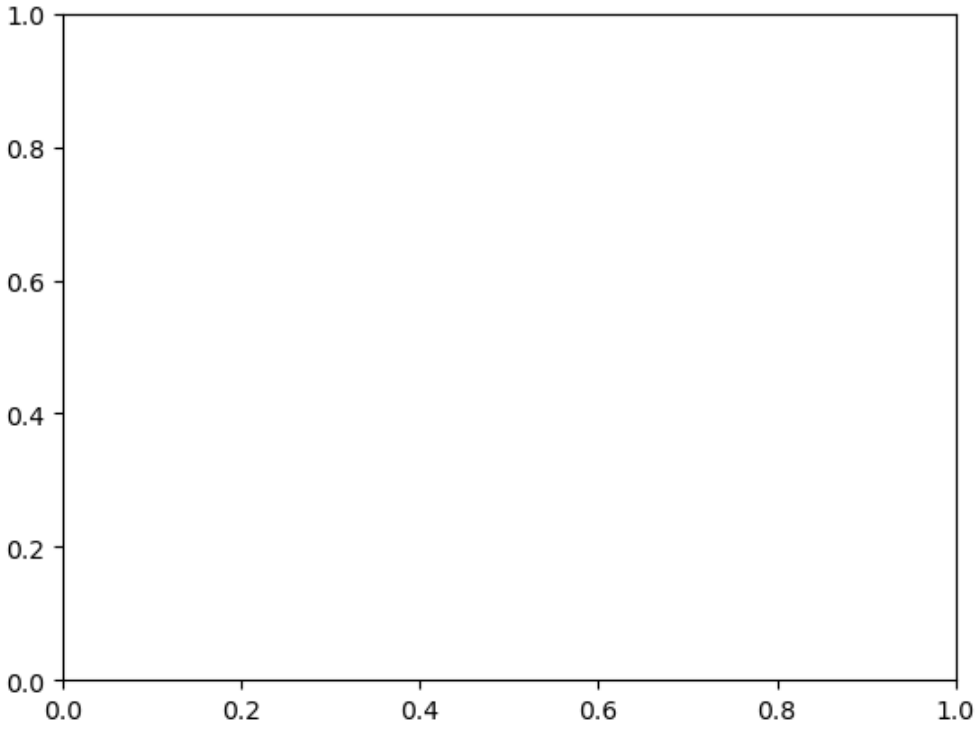


Now let's try using the object-orientated version.

We'll start by creating a figure with `plt.figure()`.

And then we'll add an axes with `add_subplot`.

In [9]:
```
# Creating a plot with the object-orientated verison
fig = plt.figure() # create a figure
ax = fig.add_subplot() # add an axes
plt.show()
```



A note on the terminology:

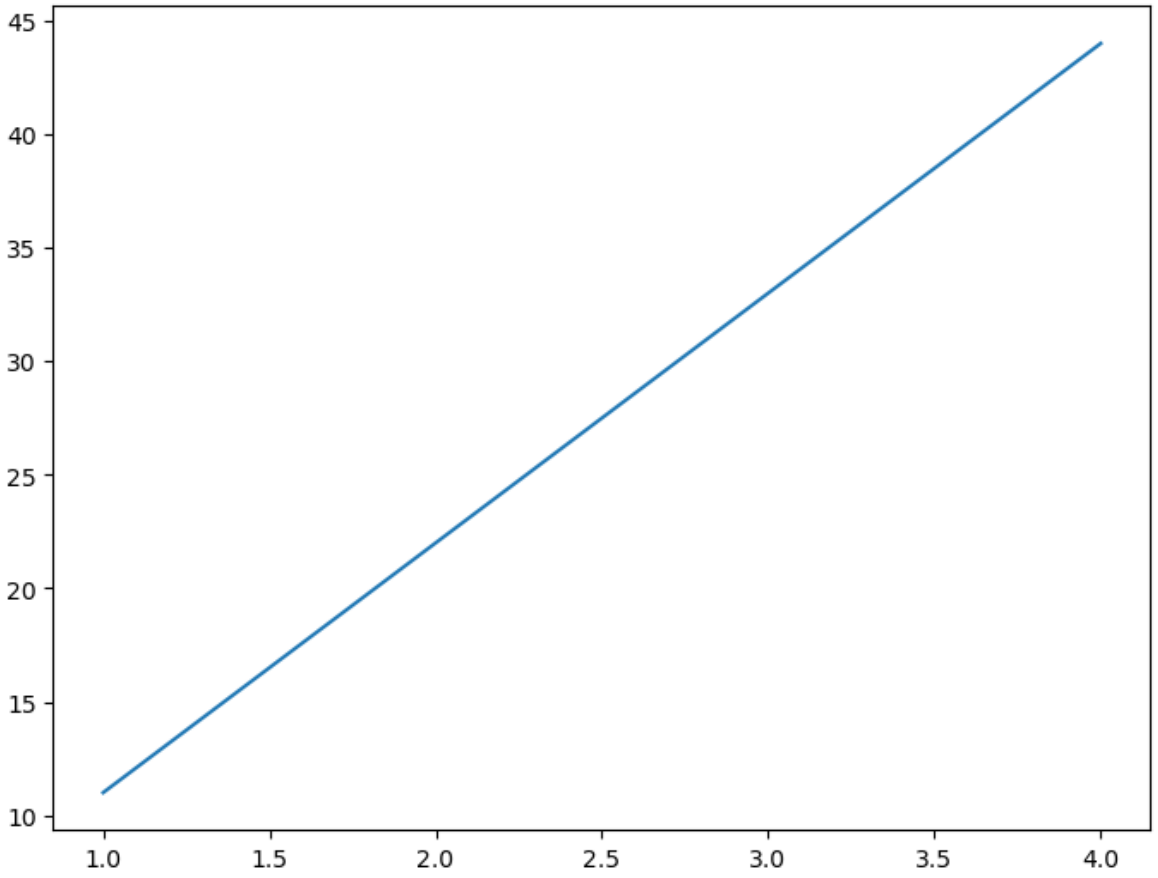
- A **`Figure`** (e.g. `fig = plt.figure()`) is the final image in matplotlib (and it may contain one or more `Axes`), often shortened to `fig`.
- The **`Axes`** are an individual plot (e.g. `ax = fig.add_subplot()`), often shorted to `ax`.

- One `Figure` can contain one or more `Axes`.

- The **`Axis`** are x (horizontal), y (vertical), z (depth).

Now let's add some data to our pevious plot.

In [10]:
```
# Add some data to our previous plot
fig = plt.figure()
ax = fig.add_axes([1, 1, 1, 1])
ax.plot(x, y)
plt.show()
```

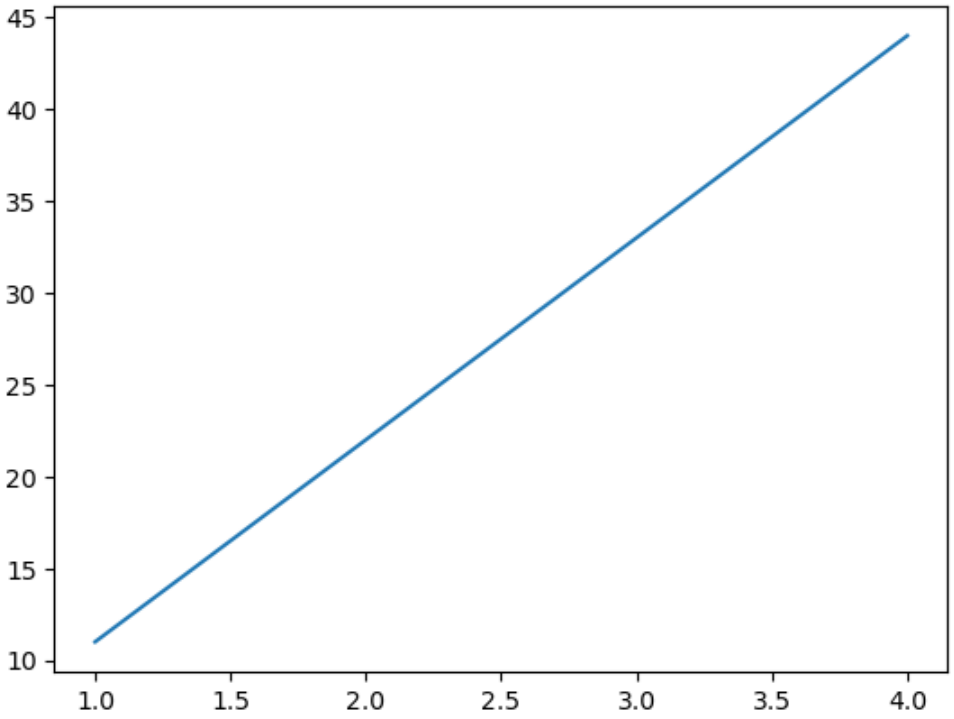


But there's an easier way we can use `matplotlib.pyplot` to help us create a `Figure` with multiple potential `Axes`.

And that's with `plt.subplots()`.

In [11]:
```
# Create a Figure and multiple potential Axes and add some data
fig, ax = plt.subplots()
```

```
ax.plot(x, y);
```



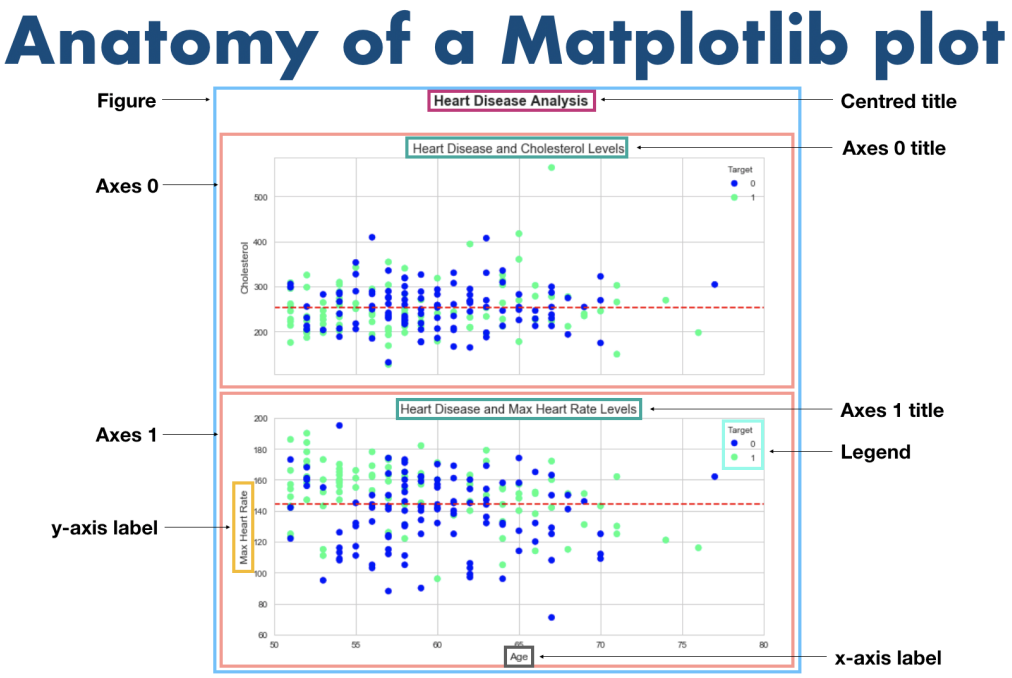
Anatomy of a Matplotlib Figure

Matplotlib offers almost unlimited options for creating plots.

However, let's break down some of the main terms.

- **Figure** - The base canvas of all matplotlib plots. The overall thing you're plotting is a Figure, often shortened to `fig`.

- **Axes** - One Figure can have one *or* multiple Axes, for example, a Figure with multiple suplots could have 4 Axes (2 rows and 2 columns). Often shortened to `ax`.
- **Axis** - A particular dimension of an Axes, for example, the x-axis or y-axis.



In [12]:
```
# This is where the object orientated name comes from
type(fig), type(ax)
```

Out[12]:
```
(matplotlib.figure.Figure, matplotlib.axes._subplots.AxesSubplot)
```

A quick Matplotlib Workflow

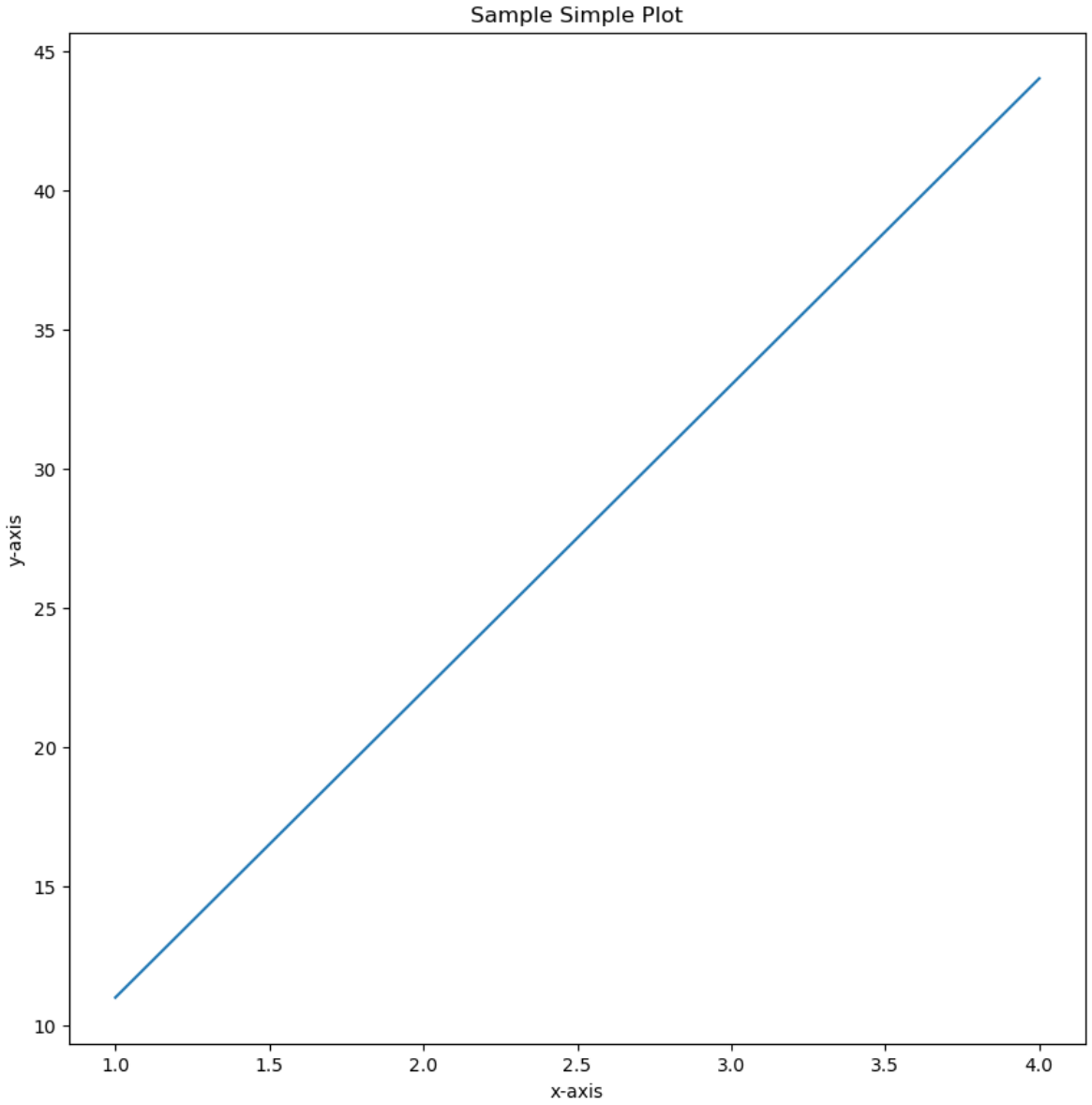
The following workflow is a standard practice when creating a matplotlib plot:

0. **Import matplotlib** - For example, `import matplotlib.pyplot as plt`).
1. **Prepare data** - This may be from an existing dataset (data analysis) or from the outputs of a machine learning model (data science).
2. **Setup the plot** - In other words, create the Figure and various Axes.
3. **Plot data to the Axes** - Send the relevant data to the target Axes.
4. **Cutomize the plot** - Add a title, decorate the colours, label each Axis.
5. **Save (optional) and show** - See what your masterpiece looks like and save it to file if necessary.

In [13]:

```
# A matplotlib workflow

# 0. Import and get matplotlib ready
# %matplotlib inline # Not necessary in newer versions of Jupyter (e.g. 2022 onwarc
import matplotlib.pyplot as plt

# 1. Prepare data
x = [1, 2, 3, 4]
y = [11, 22, 33, 44]

# 2. Setup plot (Figure and Axes)
fig, ax = plt.subplots(figsize=(10,10))

# 3. Plot data
ax.plot(x, y)
```

```python
# 4. Customize plot
ax.set(title="Sample Simple Plot", xlabel="x-axis", ylabel="y-axis")

# 5. Save & show
fig.savefig("../images/simple-plot.png")
```



2. Making the most common type of plots using NumPy arrays

Most of figuring out what kind of plot to use is getting a feel for the data, then seeing what kind of plot suits it best.

Matplotlib visualizations are built on NumPy arrays. So in this section we'll build some of the most common types of plots using NumPy arrays.

- Line plot - `ax.plot()` (this is the default plot in matplotlib)
- Scatter plot - `ax.scatter()`
- Bar plot - `ax.bar()`
- Histogram plot - `ax.hist()`

We'll see how all of these can be created as a method from `matplotlob.pyplot.subplots()`.

> **Resource:** Remember you can see many of the different kinds of matplotlib plot types in the documentation.

To make sure we have access to NumPy, we'll import it as `np`.

```
In [14]: import numpy as np
```

Creating a line plot

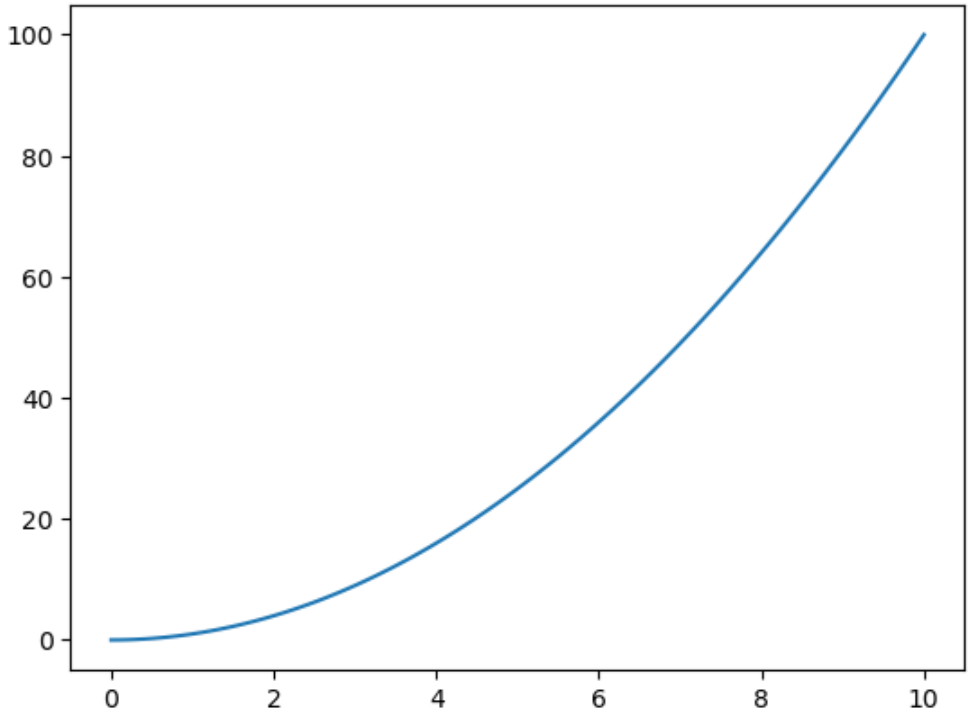
Line is the default type of visualization in Matplotlib. Usually, unless specified otherwise, your plots will start out as lines.

Line plots are great for seeing trends over time.

In [15]:
```
# Create an array
x = np.linspace(0, 10, 100)
x[:10]
```

Out[15]:
```
array([0.        , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,
       0.50505051, 0.60606061, 0.70707071, 0.80808081, 0.90909091])
```

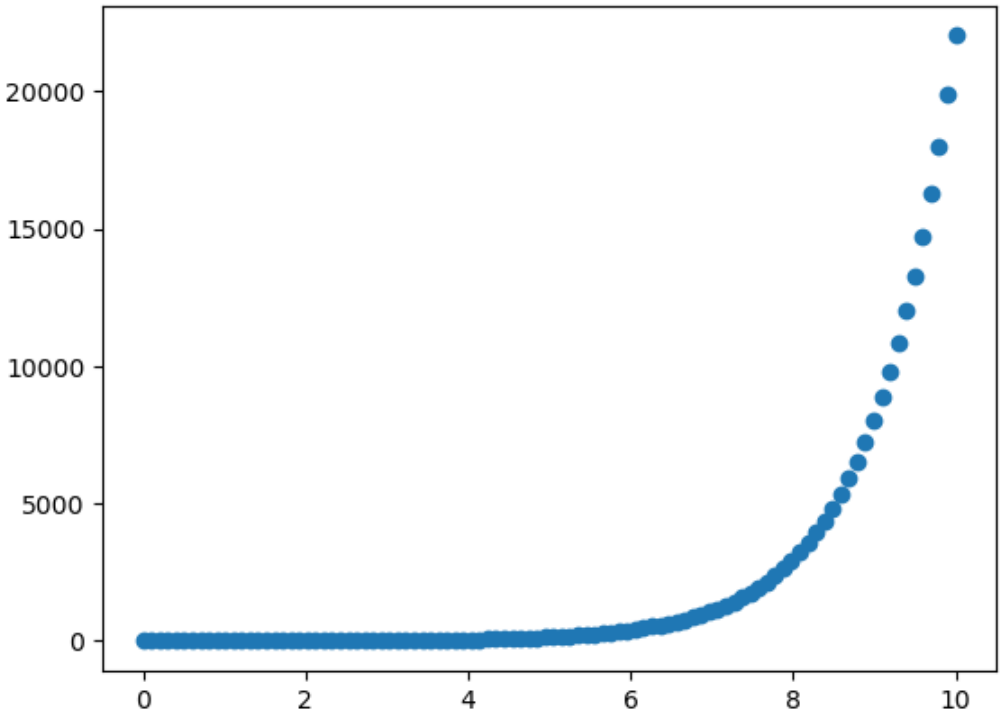
In [16]:
```
# The default plot is line
fig, ax = plt.subplots()
ax.plot(x, x**2);
```



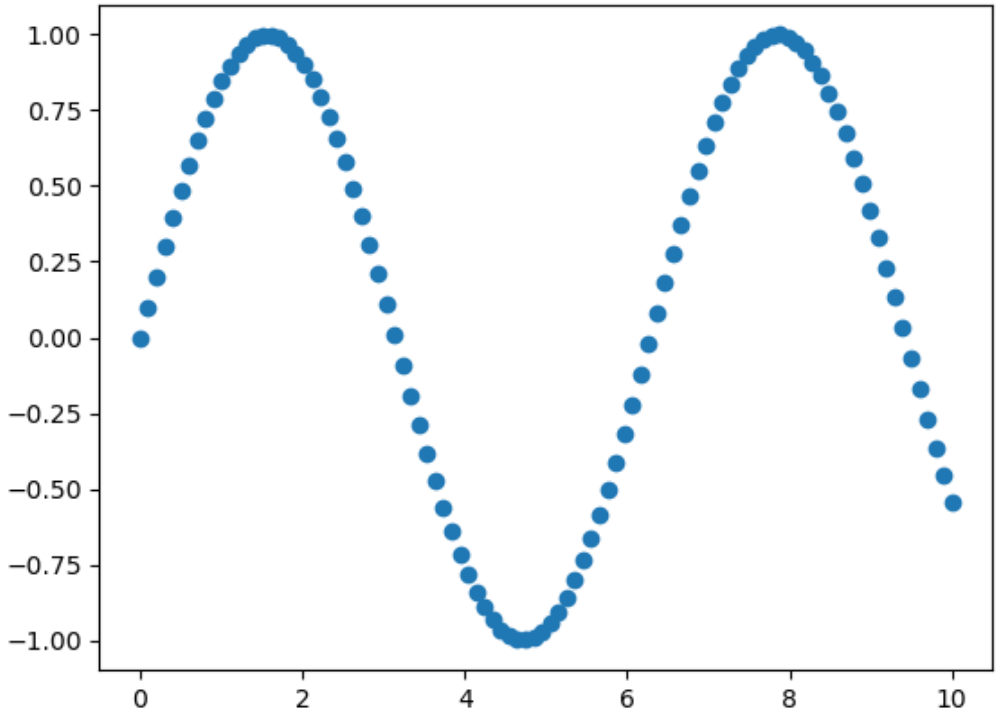
Creating a scatter plot

Scatter plots can be great for when you've got many different individual data points and you'd like to see how they interact with eachother without being connected.

In [17]:
```
# Need to recreate our figure and axis instances when we want a new figure
fig, ax = plt.subplots()
ax.scatter(x, np.exp(x));
```



In [18]:
```
fig, ax = plt.subplots()
ax.scatter(x, np.sin(x));
```



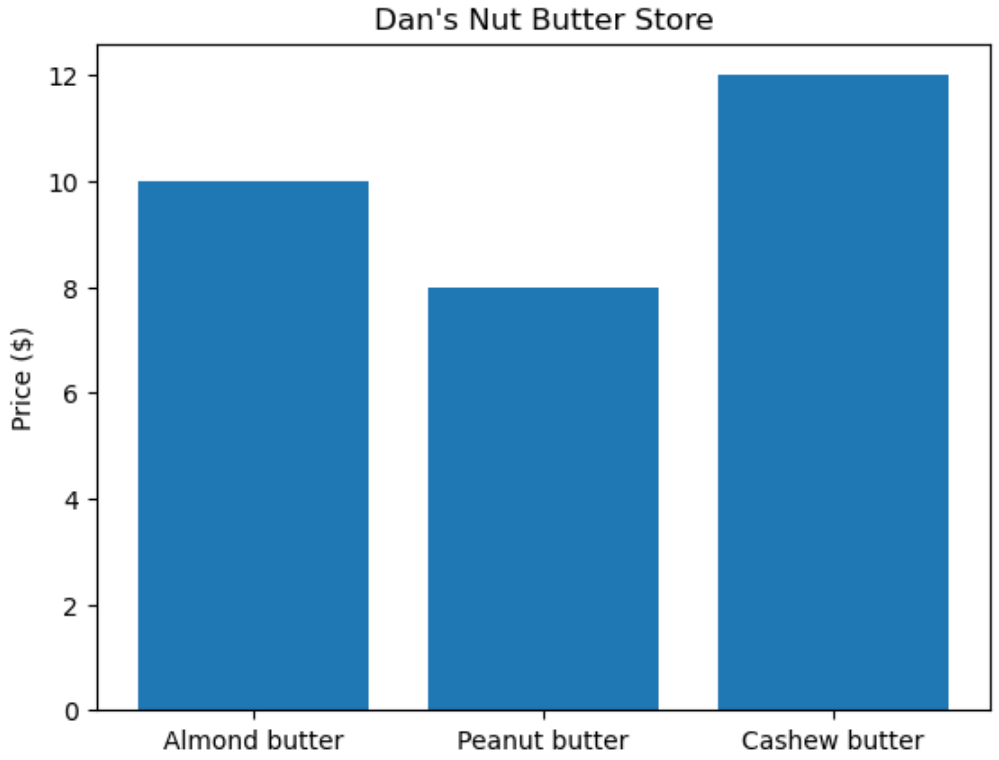
Creating bar plots

Bar plots are great to visualize different amounts of similar themed items.

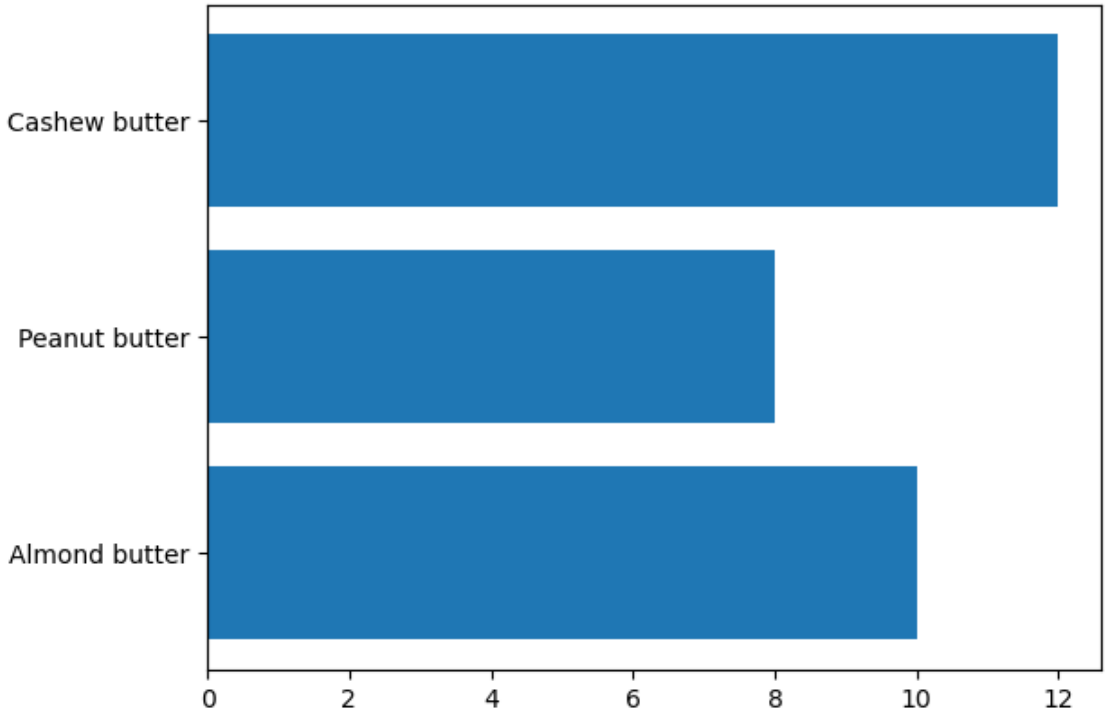
For example, the sales of items at a Nut Butter Store.

You can create **vertical** bar plots with `ax.bar()` and **horizontal** bar plots with `ax.barh()`.

In [19]:
```
# You can make plots from a dictionary
nut_butter_prices = {"Almond butter": 10,
                     "Peanut butter": 8,
                     "Cashew butter": 12}
fig, ax = plt.subplots()
ax.bar(nut_butter_prices.keys(), nut_butter_prices.values())
ax.set(title="Dan's Nut Butter Store", ylabel="Price ($)");
```



In [20]:
```
fig, ax = plt.subplots()
ax.barh(list(nut_butter_prices.keys()), list(nut_butter_prices.values()));
```



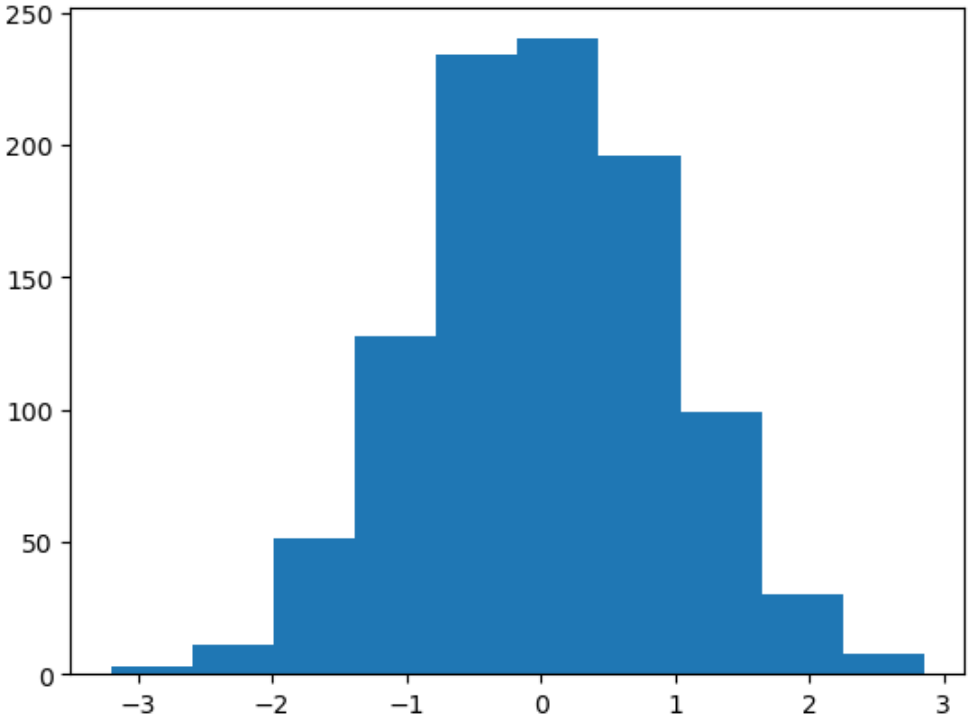
Creating a histogram plot

Histogram plots are excellent for showing the distribution of data.

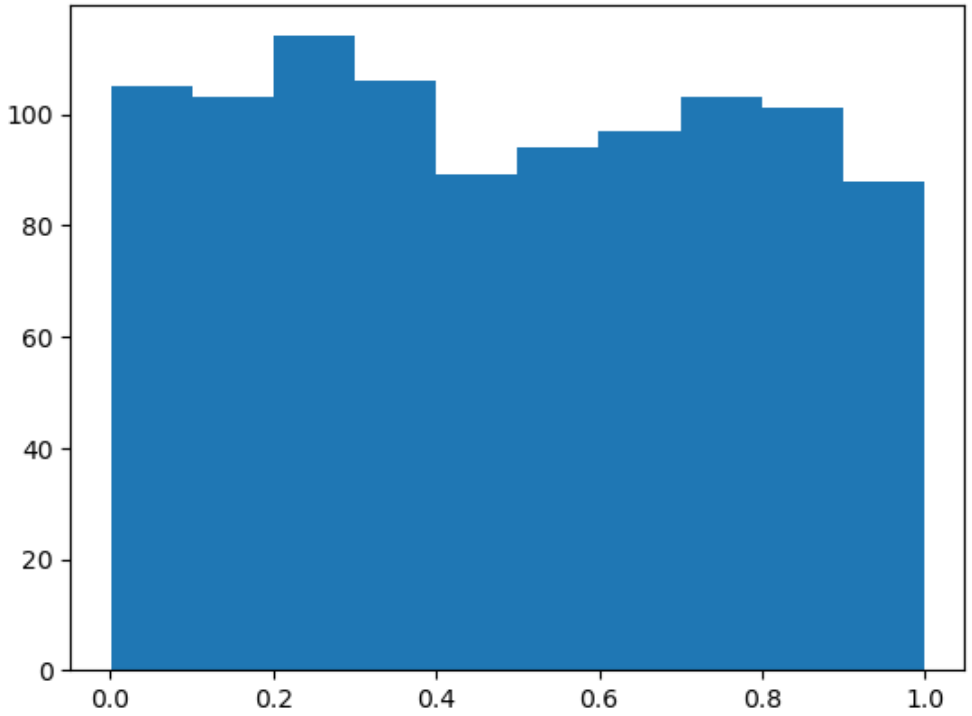
For example, you might want to show the distribution of ages of a population or wages of city.

In [21]:
```
# Make some data from a normal distribution
x = np.random.randn(1000) # pulls data from a normal distribution
```

```
fig, ax = plt.subplots()
ax.hist(x);
```



In [22]:
```
x = np.random.random(1000) # random data from random distribution

fig, ax = plt.subplots()
ax.hist(x);
```



Creating Figures with multiple Axes with Subplots

Subplots allow you to create multiple Axes on the same Figure (multiple plots within the same plot).

Subplots are helpful because you start with one plot per Figure but scale it up to more when necessary.

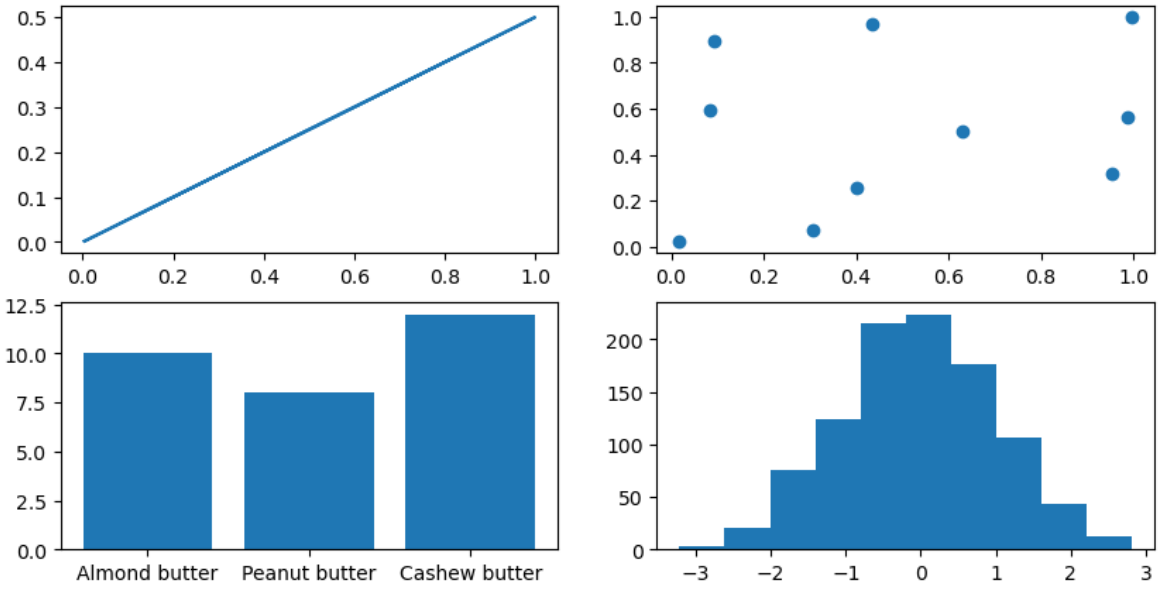
For example, let's create a subplot that shows many of the above datasets on the same Figure.

We can do so by creating multiple Axes with `plt.subplots()` and setting the `nrows` (number of rows) and `ncols` (number of columns) parameters to reflect how many Axes we'd like.

`nrows` and `ncols` parameters are multiplicative, meaning `plt.subplots(nrows=2, ncols=2)` will create `2*2=4` total Axes.

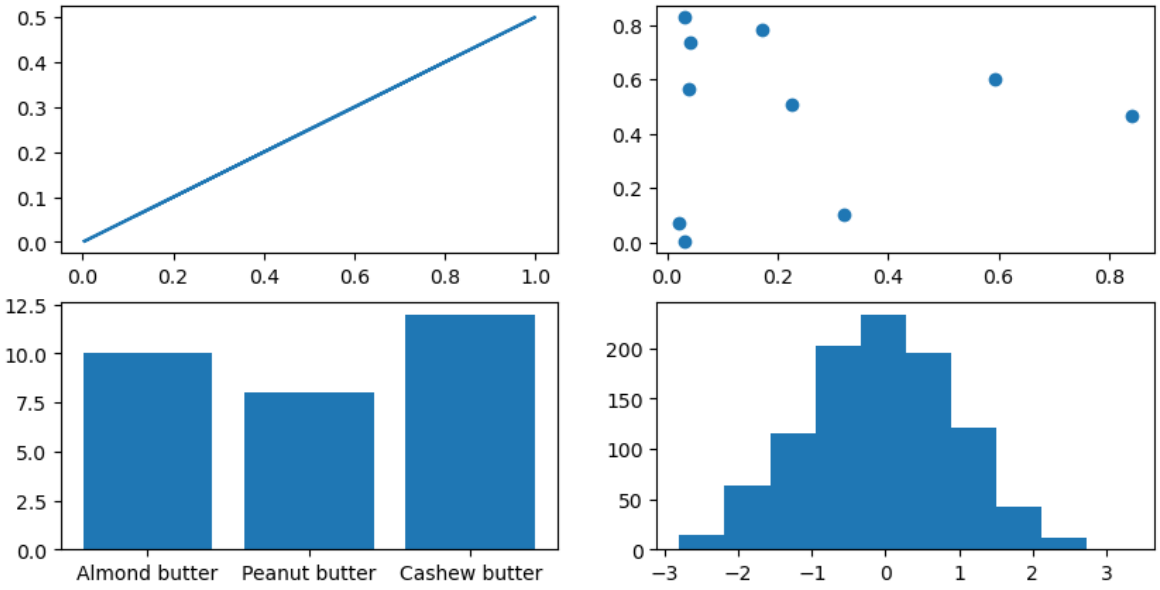
> **Resource:** You can see a sensational number of examples for creating Subplots in the matplotlib documentation.

In [23]:
```
# Option 1: Create 4 subplots with each Axes having its own variable name
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2,
                                             ncols=2,
                                             figsize=(10, 5))

# Plot data to each axis
ax1.plot(x, x/2);
ax2.scatter(np.random.random(10), np.random.random(10));
ax3.bar(nut_butter_prices.keys(), nut_butter_prices.values());
ax4.hist(np.random.randn(1000));
```



In [24]:

```
# Option 2: Create 4 subplots with a single ax variable
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))

# Index the ax variable to plot data
ax[0, 0].plot(x, x/2);
ax[0, 1].scatter(np.random.random(10), np.random.random(10));
ax[1, 0].bar(nut_butter_prices.keys(), nut_butter_prices.values());
ax[1, 1].hist(np.random.randn(1000));
```



3. Plotting data directly with pandas

Matplotlib has a tight integration with pandas too.

You can directly plot from a pandas DataFrame with `DataFrame.plot()`.

Let's see the following plots directly from a pandas DataFrame:

- Line
- Scatter
- Bar

- Hist

To plot data with pandas, we first have to import it as `pd`.

In [25]:
```
import pandas as pd
```

Now we need some data to check out.

In [26]:
```
# Let's import the car_sales dataset
car_sales = pd.read_csv("../data/car-sales.csv")
car_sales
```

Out[26]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
1	Honda	Red	87899	4	$5,000.00
2	Toyota	Blue	32549	3	$7,000.00
3	BMW	Black	11179	5	$22,000.00
4	Nissan	White	213095	4	$3,500.00
5	Toyota	Green	99213	4	$4,500.00
6	Honda	Blue	45698	4	$7,500.00
7	Honda	Blue	54738	4	$7,000.00
8	Toyota	White	60000	4	$6,250.00
9	Nissan	White	31600	4	$9,700.00

Line plot from a pandas DataFrame

To understand examples, I often find I have to repeat them (code them myself) rather than just read them.

To begin understanding plotting with pandas, let's recreate the a section of the pandas Chart visualization documents.

In [27]:
```
# Start with some dummy data
ts = pd.Series(np.random.randn(1000),
               index=pd.date_range('1/1/2024', periods=1000))

# Note: ts = short for time series (data over time)
ts
```

Out[27]:
```
2024-01-01   -0.195994
2024-01-02   -1.022610
2024-01-03   -0.202821
2024-01-04    0.640333
2024-01-05   -0.999877
                ...
2026-09-22    0.096283
2026-09-23    1.466828
2026-09-24   -0.149209
2026-09-25   -0.161122
2026-09-26   -0.168698
Freq: D, Length: 1000, dtype: float64
```

Great! We've got some random values across time.

Now let's add up the data cumulatively overtime with `DataFrame.cumsum()` (`cumsum` is short for cumulative sum or continaully adding one thing to the next and so on).

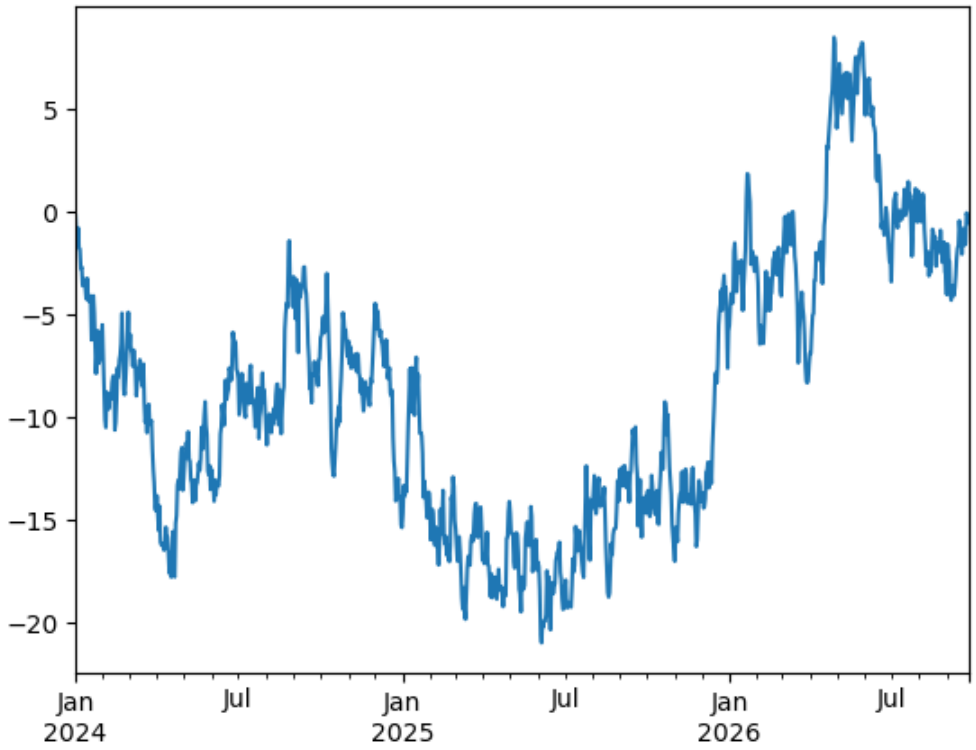
In [28]:
```
# Add up the values cumulatively
ts.cumsum()
```

Out[28]:
```
2024-01-01   -0.195994
2024-01-02   -1.218604
2024-01-03   -1.421425
2024-01-04   -0.781092
2024-01-05   -1.780968
                ...
2026-09-22   -1.518964
2026-09-23   -0.052136
2026-09-24   -0.201345
2026-09-25   -0.362467
2026-09-26   -0.531165
Freq: D, Length: 1000, dtype: float64
```

We can now visualize the values by calling the `plot()` method on the DataFrame and specifying the kind of plot we'd like with the `kind` parameter.

In our case, the kind we'd like is a line plot, hence `kind="line"` (this is the default for the `plot()` method).

In [29]:
```
# Plot the values over time with a line plot (note: both of these will return the s
# ts.cumsum().plot() # kind="line" is set by default
ts.cumsum().plot(kind="line");
```



Working with actual data

Let's do a little data manipulation on our `car_sales` DataFrame.

In [30]:
```
# Import the car sales data
car_sales = pd.read_csv("../data/car-sales.csv")

# Remove price column symbols
```

```
car_sales["Price"] = car_sales["Price"].str.replace('[\$\,\.]', '',
                                                    regex=True) # Tell pandas to
car_sales
```

Out[30]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	400000
1	Honda	Red	87899	4	500000
2	Toyota	Blue	32549	3	700000
3	BMW	Black	11179	5	2200000
4	Nissan	White	213095	4	350000
5	Toyota	Green	99213	4	450000
6	Honda	Blue	45698	4	750000
7	Honda	Blue	54738	4	700000
8	Toyota	White	60000	4	625000
9	Nissan	White	31600	4	970000

In [31]:

```
# Remove last two zeros
car_sales["Price"] = car_sales["Price"].str[:-2]
car_sales
```

Out[31]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	4000
1	Honda	Red	87899	4	5000
2	Toyota	Blue	32549	3	7000
3	BMW	Black	11179	5	22000
4	Nissan	White	213095	4	3500
5	Toyota	Green	99213	4	4500

6	Honda	Blue	45698	4	7500
7	Honda	Blue	54738	4	7000
8	Toyota	White	60000	4	6250
9	Nissan	White	31600	4	9700

In [32]:
```
# Add a date column
car_sales["Sale Date"] = pd.date_range("1/1/2024", periods=len(car_sales))
car_sales
```

Out[32]:

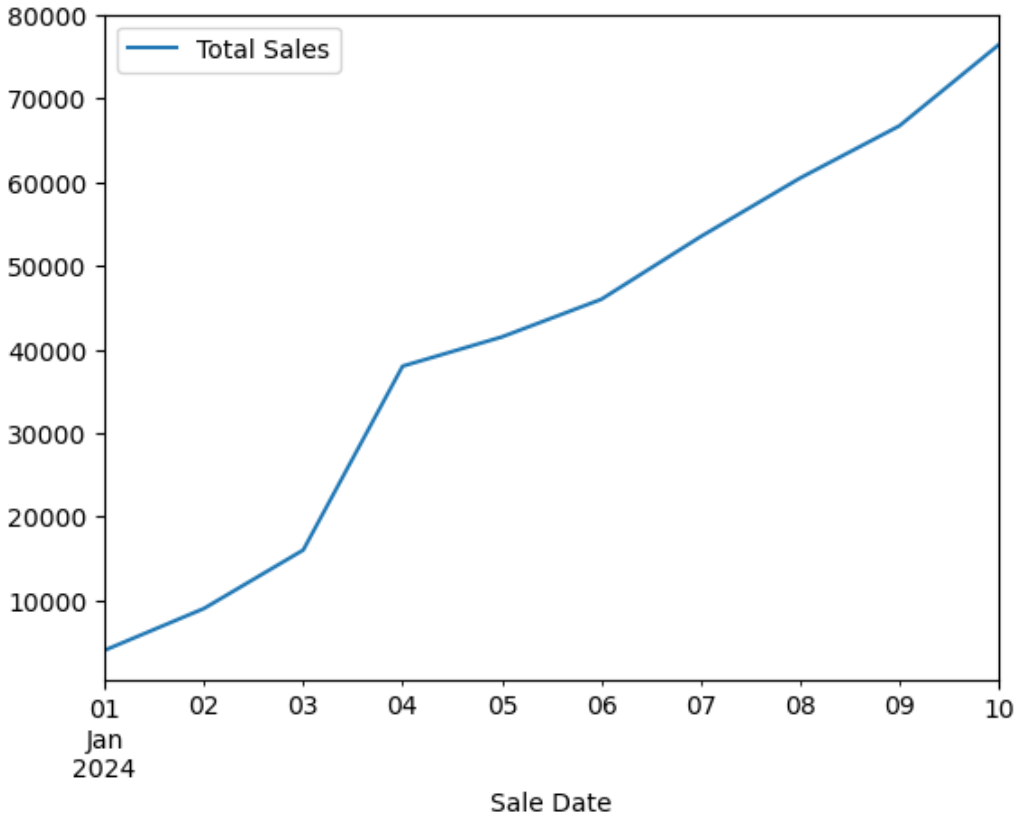
	Make	Colour	Odometer (KM)	Doors	Price	Sale Date
0	Toyota	White	150043	4	4000	2024-01-01
1	Honda	Red	87899	4	5000	2024-01-02
2	Toyota	Blue	32549	3	7000	2024-01-03
3	BMW	Black	11179	5	22000	2024-01-04
4	Nissan	White	213095	4	3500	2024-01-05
5	Toyota	Green	99213	4	4500	2024-01-06
6	Honda	Blue	45698	4	7500	2024-01-07
7	Honda	Blue	54738	4	7000	2024-01-08
8	Toyota	White	60000	4	6250	2024-01-09
9	Nissan	White	31600	4	9700	2024-01-10

In [33]:
```
# Make total sales column (doesn't work, adds as string)
#car_sales["Total Sales"] = car_sales["Price"].cumsum()

# Oops... want them as int's not string
car_sales["Total Sales"] = car_sales["Price"].astype(int).cumsum()
car_sales
```

Out[33]:

	Make	Colour	Odometer (KM)	Doors	Price	Sale Date	Total Sales
0	Toyota	White	150043	4	4000	2024-01-01	4000
1	Honda	Red	87899	4	5000	2024-01-02	9000
2	Toyota	Blue	32549	3	7000	2024-01-03	16000
3	BMW	Black	11179	5	22000	2024-01-04	38000
4	Nissan	White	213095	4	3500	2024-01-05	41500
5	Toyota	Green	99213	4	4500	2024-01-06	46000
6	Honda	Blue	45698	4	7500	2024-01-07	53500
7	Honda	Blue	54738	4	7000	2024-01-08	60500
8	Toyota	White	60000	4	6250	2024-01-09	66750
9	Nissan	White	31600	4	9700	2024-01-10	76450

In [34]:
```
car_sales.plot(x='Sale Date', y='Total Sales');
```

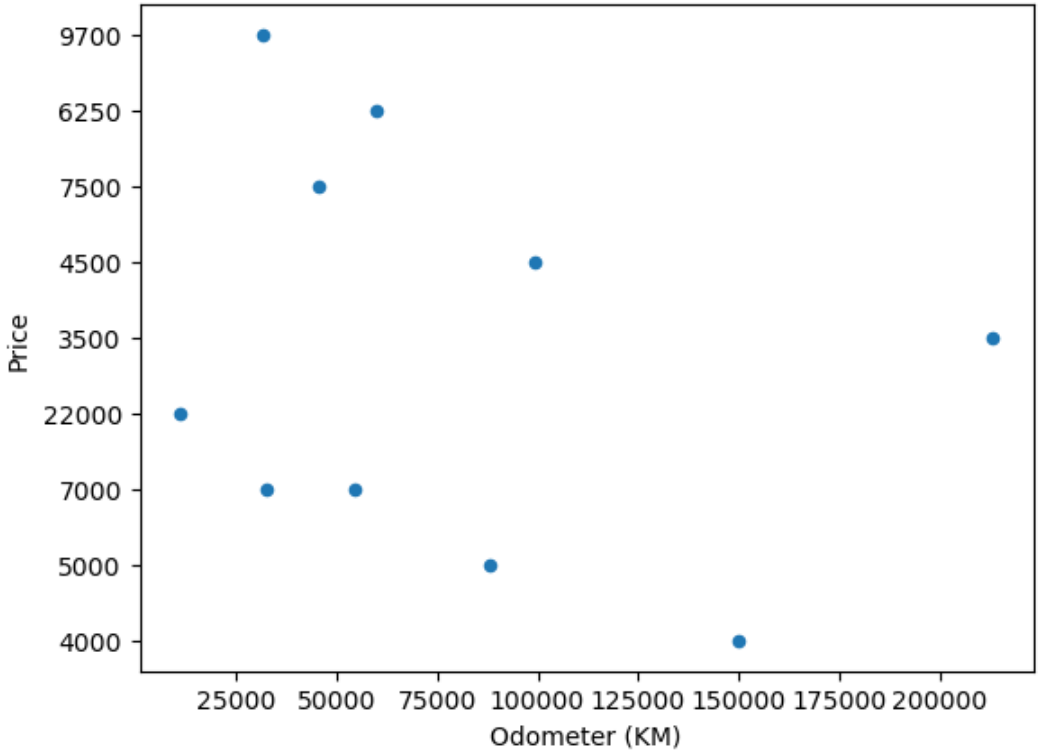


Scatter plot from a pandas DataFrame

You can create scatter plots from a pandas DataFrame by using the `kind="scatter"` parameter.

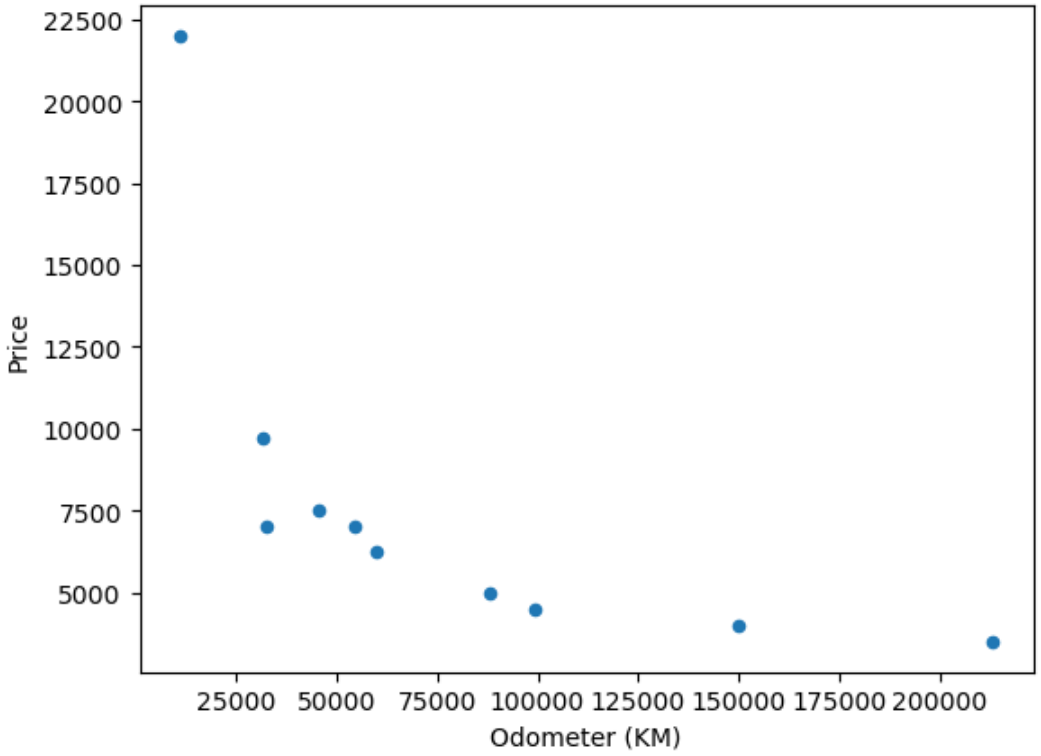
However, you'll often find that certain plots require certain kinds of data (e.g. some plots require certain columns to be numeric).

In [35]:
```
# Note: In previous versions of matplotlib and pandas, have the "Price" column as a
# return an error
car_sales["Price"] = car_sales["Price"].astype(str)

# Plot a scatter plot
car_sales.plot(x="Odometer (KM)", y="Price", kind="scatter");
```



Having the `Price` column as an `int` returns a much better looking y-axis.

In [36]:
```
# Convert Price to int
car_sales["Price"] = car_sales["Price"].astype(int)

# Plot a scatter plot
car_sales.plot(x="Odometer (KM)", y="Price", kind='scatter');
```



Bar plot from a pandas DataFrame

Let's see how we can plot a bar plot from a pandas DataFrame.

First, we'll create some data.

In [37]:
```
# Create 10 random samples across 4 columns
x = np.random.rand(10, 4)
x
```

Out[37]:
```
array([[0.65745479, 0.42745471, 0.61990211, 0.01218935],
       [0.10699156, 0.6546944 , 0.5915984 , 0.55011077],
       [0.50720269, 0.2725063 , 0.95817204, 0.67309876],
       [0.33016817, 0.85921522, 0.02778741, 0.36043001],
       [0.8850031 , 0.82582603, 0.58275893, 0.10393635],
       [0.70596769, 0.15698541, 0.43727796, 0.03307697],
       [0.55611843, 0.86959028, 0.49525034, 0.06849191],
       [0.19340766, 0.69988787, 0.89546643, 0.368045  ],
       [0.01834179, 0.74501467, 0.06589424, 0.58463789],
       [0.31159084, 0.4001198 , 0.59601375, 0.64712406]])
```

In [82]:
```
# Turn the data into a DataFrame
df = pd.DataFrame(x, columns=['a', 'b', 'c', 'd'])
df
```

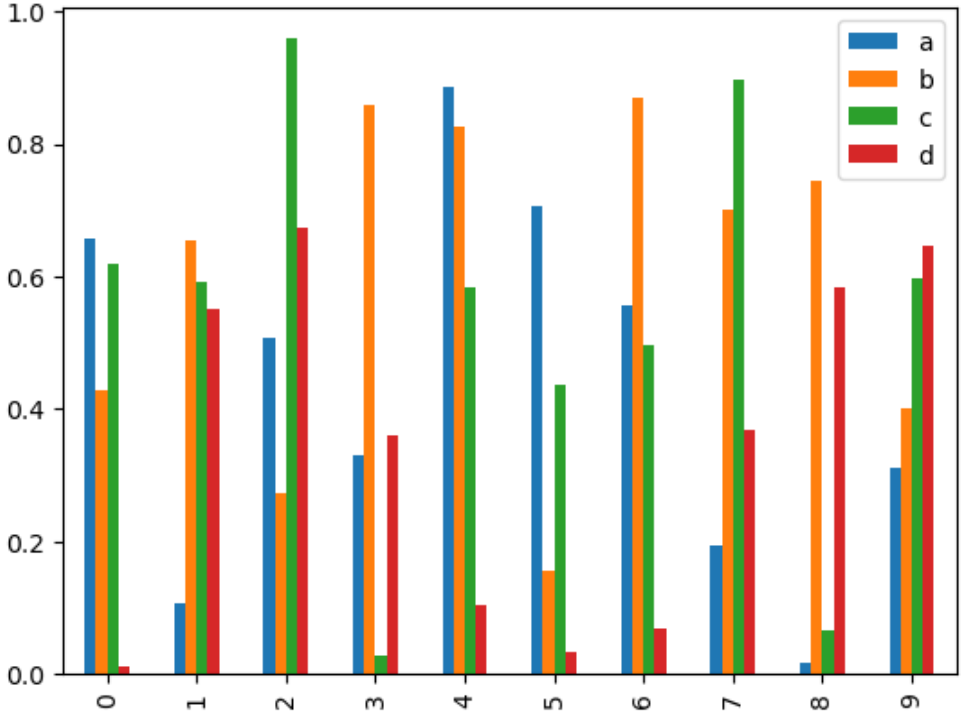
Out[82]:

	a	b	c	d
0	1.326093	-1.179144	-1.228776	0.320765
1	-0.473547	-0.226671	-0.784430	0.192451
2	2.288607	-1.090920	-0.204312	0.486072
3	1.591945	0.320072	2.949674	-1.306000
4	-1.873583	1.132770	1.423901	0.928743
5	-1.121281	-0.640948	-0.527283	0.242460
6	1.302475	-0.295322	3.141830	0.558532

7	-1.663926	1.767556	-0.558923	0.750767
8	-0.658601	0.278021	0.854262	0.012043
9	-0.734160	-1.011017	0.842804	-0.008819

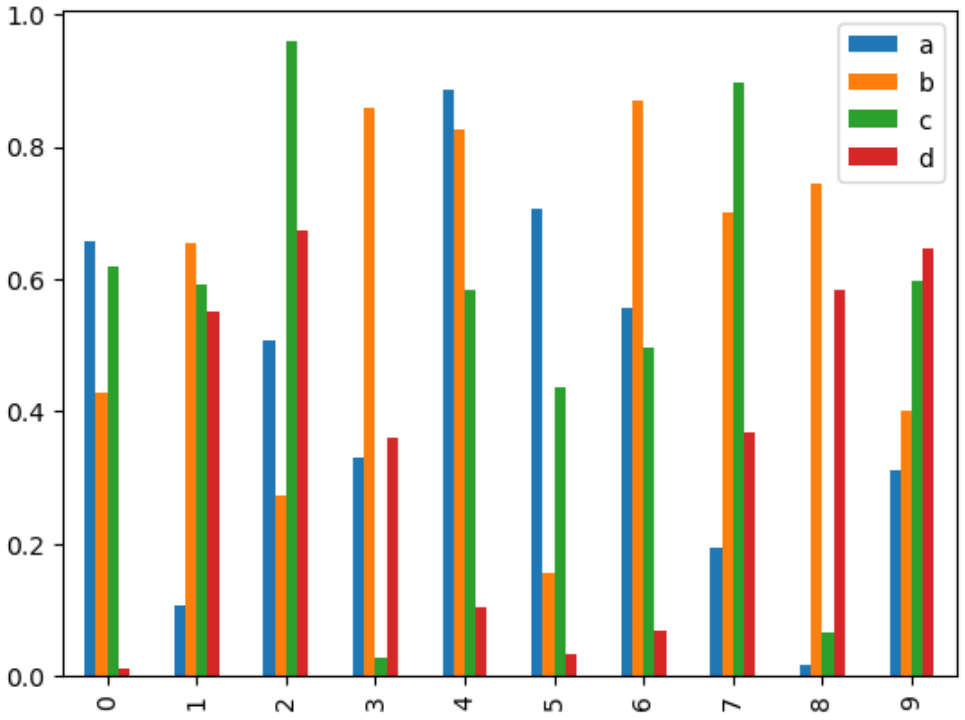
We can plot a bar chart directly with the `bar()` method on the DataFrame.

In [39]:
```
# Plot a bar chart
df.plot.bar();
```



And we can also do the same thing passing the `kind="bar"` parameter to `DataFrame.plot()`.

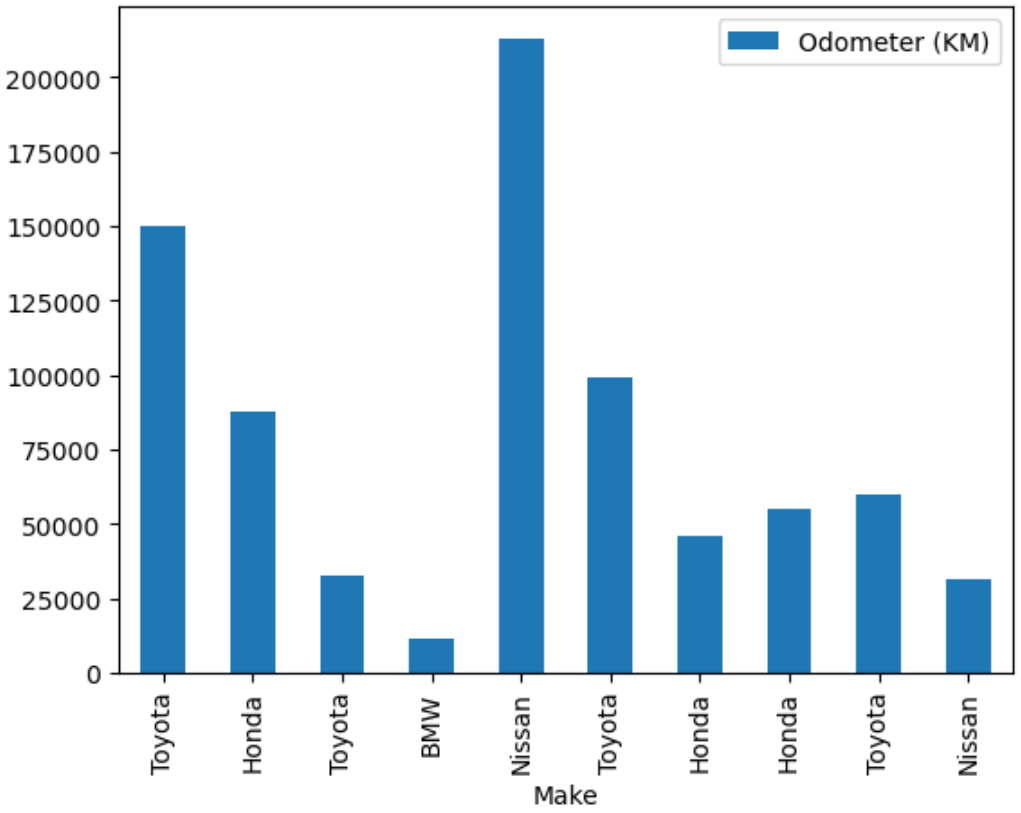
In [40]:
```
# Plot a bar chart with the kind parameter
df.plot(kind='bar');
```



Let's try a bar plot on the `car_sales` DataFrame.

This time we'll specify the `x` and `y` axis values.

In [41]:
```
# Plot a bar chart from car_sales DataFrame
car_sales.plot(x="Make",
               y="Odometer (KM)",
               kind="bar");
```

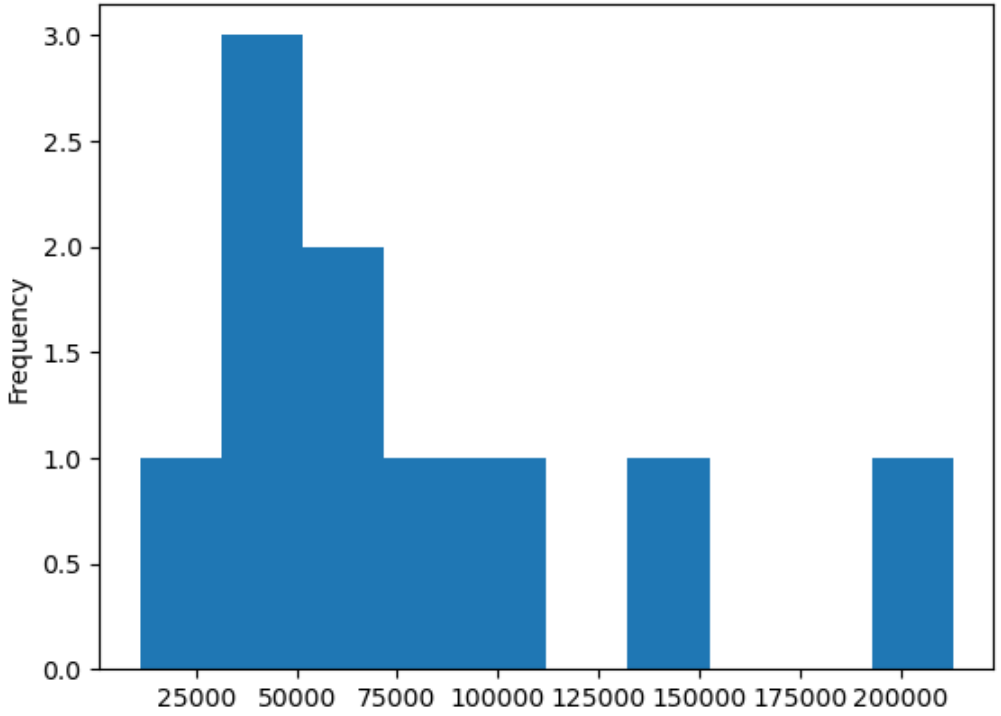


Histogram plot from a pandas DataFrame

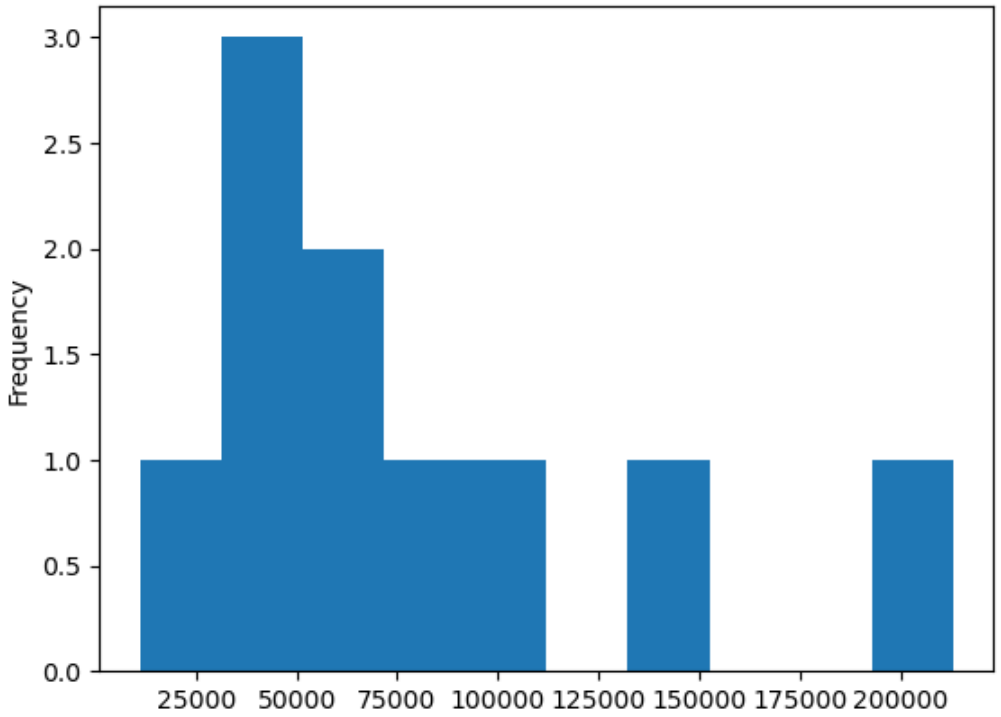
We can plot a histogram plot from our `car_sales` DataFrame using `DataFrame.plot.hist()` or `DataFrame.plot(kind="hist")`.

Histograms are great for seeing the distribution or the spread of data.

In [42]:
```
car_sales["Odometer (KM)"].plot.hist();
```



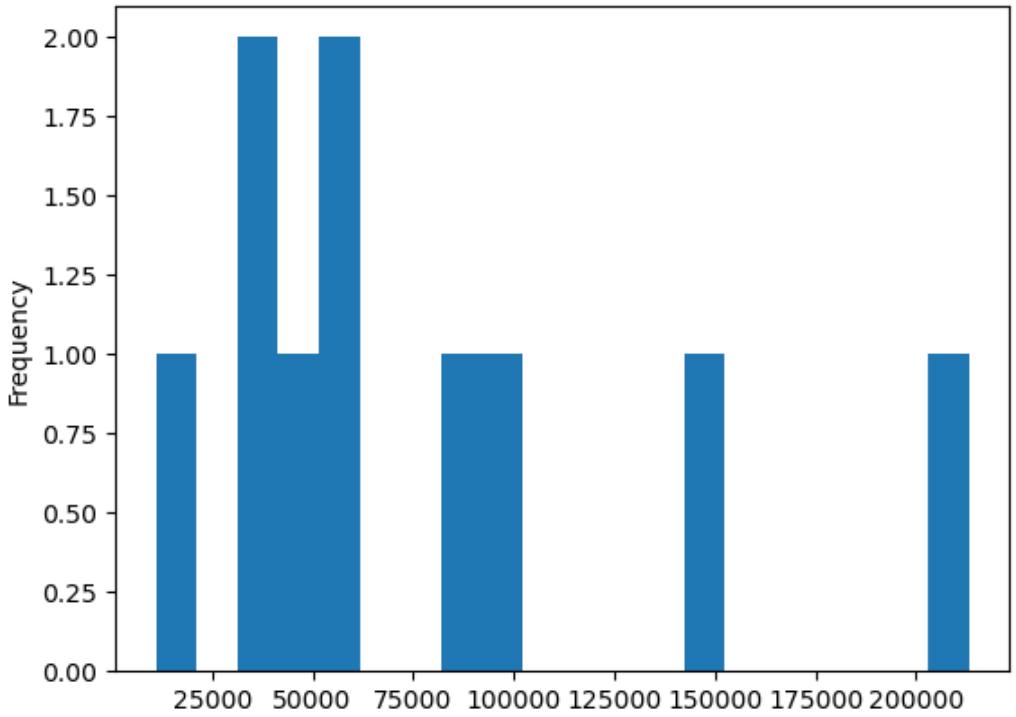
In [43]:
```
car_sales["Odometer (KM)"].plot(kind="hist");
```



Changing the `bins` parameter we can put our data into different numbers of collections.

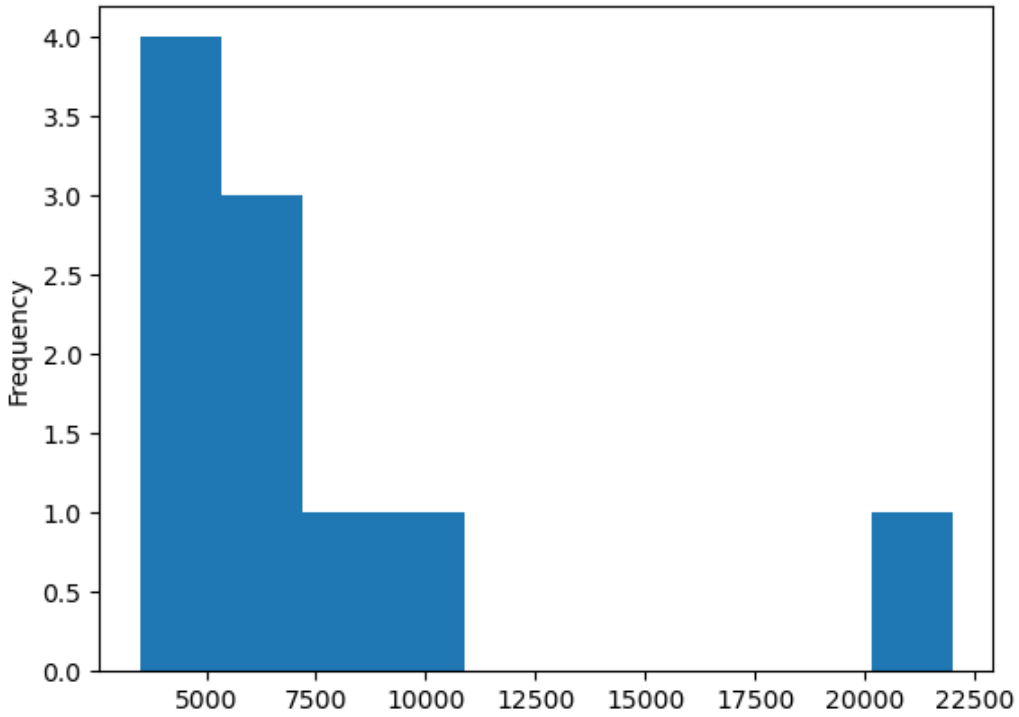
For example, by default `bins=10` (10 groups of data), let's see what happens when we change it to `bins=20`.

In [44]:
```
# Default number of bins is 10
car_sales["Odometer (KM)"].plot.hist(bins=20);
```



To practice, let's create a histogram of the `Price` column.

In [45]:
```
# Create a histogram of the Price column
car_sales["Price"].plot.hist(bins=10);
```



And to practice even further, how about we try another dataset?

Namely, let's create some plots using the heart disease dataset we've worked on before.

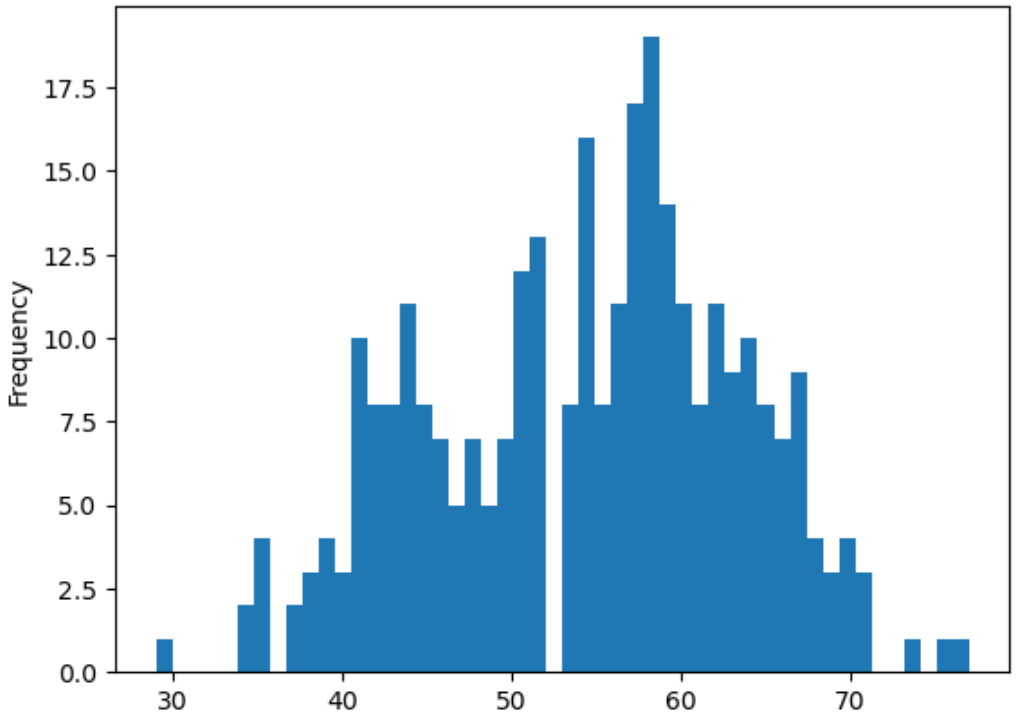
In [46]:
```
# Import the heart disease dataset
heart_disease = pd.read_csv("../data/heart-disease.csv")
heart_disease.head()
```

Out[46]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1

1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

In [47]:
```
# Create a histogram of the age column
heart_disease["age"].plot.hist(bins=50);
```



What does this tell you about the spread of heart disease data across different ages?