A Quick Introduction to Numerical Data Manipulation with Python and NumPy

What is NumPy?

NumPy stands for numerical Python. It's the backbone of all kinds of scientific and numerical computing in Python.

And since machine learning is all about turning data into numbers and then figuring out the patterns, NumPy often comes into play.

a 6 step machine learning framework along will tools you can use for each step

Why NumPy?

You can do numerical calculations using pure Python. In the beginning, you might think Python is fast but once your data gets large, you'll start to notice slow downs.

One of the main reasons you use NumPy is because it's fast. Behind the scenes, the code has been optimized to run using C. Which is another programming language, which can do things much faster than Python.

The benefit of this being behind the scenes is you don't need to know any C to take advantage of it. You can write your numerical computations in Python using NumPy and get the added speed benefits.

If your curious as to what causes this speed benefit, it's a process called vectorization. Vectorization aims to do calculations by avoiding loops as loops can create potential bottlenecks.

NumPy achieves vectorization through a process called broadcasting.

What does this notebook cover?

The NumPy library is very capable. However, learning everything off by heart isn't necessary. Instead, this notebook focuses on the main concepts of NumPy and the `ndarray` datatype.

You can think of the `ndarray` datatype as a very flexible array of numbers.

More specifically, we'll look at:

- NumPy datatypes & attributes

- Creating arrays
- Viewing arrays & matrices (indexing)
- Manipulating & comparing arrays
- Sorting arrays
- Use cases (examples of turning things into numbers)

After going through it, you'll have the base knolwedge of NumPy you need to keep moving forward.

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 NumPy 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):
    - NumPy documentation - The ground truth for everything NumPy, this resource covers all of the NumPy functionality.
    - 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 NumPy function might be:

> "how to find unique elements in a numpy array"

Searching this on Google leads to the NumPy documentation for the `np.unique()` function: https://numpy.org/doc/stable/reference/generated/numpy.unique.html

The next steps here are to read through the documentation, check the examples and see if they line up to the problem you're trying to solve.

If they do, **rewrite the code** to suit your needs, run it, and see what the outcomes are.

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](). Be as specific as possible and provide details on what you've tried.

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

What's most important is continually asking yourself, "what am I trying to do with the data?".

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

Let's get started.

0. Importing NumPy

To get started using NumPy, the first step is to import it.

The most common way (and method you should use) is to import NumPy as the abbreviation `np`.

If you see the letters `np` used anywhere in machine learning or data science, it's probably referring to the NumPy library.

```
import numpy as np

# Check the version
print(np.__version__)
```

```
1.25.2
```

1. DataTypes and attributes

> **Note:** Important to remember the main type in NumPy is `ndarray`, even seemingly different kinds of arrays are still `ndarray`'s. This means an operation you do on one array, will work on another.

```python
# 1-dimensonal array, also referred to as a vector
a1 = np.array([1, 2, 3])

# 2-dimensional array, also referred to as matrix
a2 = np.array([[1, 2.0, 3.3],
               [4, 5, 6.5]])

# 3-dimensional array, also referred to as a matrix
a3 = np.array([[[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]],
                [[10, 11, 12],
                 [13, 14, 15],
                 [16, 17, 18]]])
```

```python
a1.shape, a1.ndim, a1.dtype, a1.size, type(a1)
```

```
((3,), 1, dtype('int64'), 3, numpy.ndarray)
```

```python
a2.shape, a2.ndim, a2.dtype, a2.size, type(a2)
```

```
((2, 3), 2, dtype('float64'), 6, numpy.ndarray)
```

```python
a3.shape, a3.ndim, a3.dtype, a3.size, type(a3)
```

```
((2, 3, 3), 3, dtype('int64'), 18, numpy.ndarray)
```

```python
a1
```

```
array([1, 2, 3])
```

```python
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```python
a3
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

Anatomy of an array

anatomy of a numpy array

Key terms:

- **Array** - A list of numbers, can be multi-dimensional.
- **Scalar** - A single number (e.g. `7`).
- **Vector** - A list of numbers with 1-dimension (e.g. `np.array([1, 2, 3])`).
- **Matrix** - A (usually) multi-dimensional list of numbers (e.g. `np.array([[1, 2, 3], [4, 5, 6]])`).

pandas DataFrame out of NumPy arrays

This is to examplify how NumPy is the backbone of many other libraries.

```
import pandas as pd
df = pd.DataFrame(np.random.randint(10, size=(5, 3)),
                                    columns=['a', 'b', 'c'])
df
```

	a	b	c
0	2	3	6
1	1	5	6
2	7	0	2
3	2	1	3
4	8	0	7

```
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```
df2 = pd.DataFrame(a2)
df2
```

	0	1	2
0	1.0	2.0	3.3
1	4.0	5.0	6.5

2. Creating arrays

- `np.array()`
- `np.ones()`

- `np.zeros()`
- `np.random.rand(5, 3)`
- `np.random.randint(10, size=5)`
- `np.random.seed()` - pseudo random numbers
- Searching the documentation example (finding `np.unique()` and using it)

```
# Create a simple array
simple_array = np.array([1, 2, 3])
simple_array
```

```
array([1, 2, 3])
```

```
simple_array = np.array((1, 2, 3))
simple_array, simple_array.dtype
```

```
(array([1, 2, 3]), dtype('int64'))
```

```
# Create an array of ones
ones = np.ones((10, 2))
ones
```

```
array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])
```

```
# The default datatype is 'float64'
ones.dtype
```

```
dtype('float64')
```

```
# You can change the datatype with .astype()
ones.astype(int)
```

```
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]])
```

```
# Create an array of zeros
zeros = np.zeros((5, 3, 3))
zeros
```

```
array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]])
```

```
zeros.dtype
```

```
dtype('float64')
```

```
# Create an array within a range of values
range_array = np.arange(0, 10, 2)
range_array
```

```
array([0, 2, 4, 6, 8])
```

```
# Random array
random_array = np.random.randint(10, size=(5, 3))
random_array
```

```
array([[1, 7, 2],
       [7, 0, 2],
       [8, 8, 8],
       [2, 5, 2],
       [4, 8, 6]])
```

```
# Random array of floats (between 0 & 1)
np.random.random((5, 3))
```

```
array([[0.09607892, 0.034903  , 0.47743753],
       [0.51703027, 0.90409121, 0.54436342],
       [0.8095754 , 0.60294712, 0.71141937],
       [0.50802295, 0.57255717, 0.99090604],
       [0.66225284, 0.87588103, 0.25643785]])
```

```python
np.random.random((5, 3))
```

```
array([[0.42800066, 0.76816054, 0.14858447],
       [0.48390262, 0.3708042 , 0.231316  ],
       [0.29166801, 0.64327528, 0.18039386],
       [0.89010443, 0.51218751, 0.31543512],
       [0.38781697, 0.25729731, 0.66219967]])
```

```python
# Random 5x3 array of floats (between 0 & 1), similar to above
np.random.rand(5, 3)
```

```
array([[0.28373526, 0.10074198, 0.24643463],
       [0.8268303 , 0.48672847, 0.57633359],
       [0.77867161, 0.38490598, 0.53343872],
       [0.67396616, 0.15888354, 0.47710898],
       [0.92319417, 0.19133444, 0.51837588]])
```

```python
np.random.rand(5, 3)
```

```
array([[0.73585424, 0.83359732, 0.93900774],
       [0.27563836, 0.55971665, 0.26819222],
       [0.29253202, 0.64152402, 0.90479721],
       [0.6585366 , 0.36165565, 0.37515932],
       [0.82890572, 0.54502359, 0.48398256]])
```

NumPy uses pseudo-random numbers, which means, the numbers look random but aren't really, they're predetermined.

For consistency, you might want to keep the random numbers you generate similar throughout experiments.

To do this, you can use `np.random.seed()`.

What this does is it tells NumPy, "Hey, I want you to create random numbers but keep them aligned with the seed."

Let's see it.

```python
# Set random seed to 0
np.random.seed(0)

# Make 'random' numbers
np.random.randint(10, size=(5, 3))
```

```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2],
       [4, 7, 6],
       [8, 8, 1]])
```

With `np.random.seed()` set, every time you run the cell above, the same random numbers will be generated.

What if `np.random.seed()` wasn't set?

Every time you run the cell below, a new set of numbers will appear.

```
# Make more random numbers
np.random.randint(10, size=(5, 3))
```

```
array([[6, 7, 7],
       [8, 1, 5],
       [9, 8, 9],
       [4, 3, 0],
       [3, 5, 0]])
```

Let's see it in action again, we'll stay consistent and set the random seed to 0.

```
# Set random seed to same number as above
np.random.seed(0)

# The same random numbers come out
np.random.randint(10, size=(5, 3))
```

```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2],
       [4, 7, 6],
       [8, 8, 1]])
```

Because `np.random.seed()` is set to 0, the random numbers are the same as the cell with `np.random.seed()` set to 0 as well.

Setting `np.random.seed()` is not 100% necessary but it's helpful to keep numbers the same throughout your experiments.

For example, say you wanted to split your data randomly into training and test sets.

Every time you randomly split, you might get different rows in each set.

If you shared your work with someone else, they'd get different rows in each set too.

Setting `np.random.seed()` ensures there's still randomness, it just makes the randomness repeatable. Hence the 'pseudo-random' numbers.

```
np.random.seed(0)
df = pd.DataFrame(np.random.randint(10, size=(5, 3)))
df
```

	0	1	2
0	5	0	3
1	3	7	9
2	3	5	2
3	4	7	6
4	8	8	1

What unique values are in the array a3?

Now you've seen a few different ways to create arrays, as an exercise, try find out what NumPy function you could use to find the unique values are within the `a3` array.

You might want to search some like, "how to find the unqiue values in a numpy array".

```
# Your code here
```

3. Viewing arrays and matrices (indexing)

Remember, because arrays and matrices are both `ndarray`'s, they can be viewed in similar ways.

Let's check out our 3 arrays again.

```
a1
```

```
array([1, 2, 3])
```

```
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```
a3
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

Array shapes are always listed in the format `(row, column, n, n, n...)` where `n` is optional extra dimensions.

```
a1[0]
```

```
1
```

```
a2[0]
```

```
array([1. , 2. , 3.3])
```

```
a3[0]
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
# Get 2nd row (index 1) of a2
a2[1]
```

```
array([4. , 5. , 6.5])
```

```
# Get the first 2 values of the first 2 rows of both arrays
a3[:2, :2, :2]
```

```
array([[[ 1,  2],
        [ 4,  5]],

       [[10, 11],
        [13, 14]]])
```

This takes a bit of practice, especially when the dimensions get higher. Usually, it takes me a little trial and error of trying to get certain values, viewing the output in the notebook and trying again.

NumPy arrays get printed from outside to inside. This means the number at the end of the shape comes first, and the number at the start of the shape comes last.

```
a4 = np.random.randint(10, size=(2, 3, 4, 5))
a4
```

```
array([[[[6, 7, 7, 8, 1],
         [5, 9, 8, 9, 4],
         [3, 0, 3, 5, 0],
         [2, 3, 8, 1, 3]],

        [[3, 3, 7, 0, 1],
         [9, 9, 0, 4, 7],
         [3, 2, 7, 2, 0],
         [0, 4, 5, 5, 6]],

        [[8, 4, 1, 4, 9],
         [8, 1, 1, 7, 9],
```

```
          [9, 3, 6, 7, 2],
          [0, 3, 5, 9, 4]]],


        [[[4, 6, 4, 4, 3],
          [4, 4, 8, 4, 3],
          [7, 5, 5, 0, 1],
          [5, 9, 3, 0, 5]],

         [[0, 1, 2, 4, 2],
          [0, 3, 2, 0, 7],
          [5, 9, 0, 2, 7],
          [2, 9, 2, 3, 3]],

         [[2, 3, 4, 1, 2],
          [9, 1, 4, 6, 8],
          [2, 3, 0, 0, 6],
          [0, 6, 3, 3, 8]]]])
```

```
a4.shape
```

```
(2, 3, 4, 5)
```

```
# Get only the first 4 numbers of each single vector
a4[:, :, :, :4]
```

```
array([[[[6, 7, 7, 8],
         [5, 9, 8, 9],
         [3, 0, 3, 5],
         [2, 3, 8, 1]],

        [[3, 3, 7, 0],
         [9, 9, 0, 4],
         [3, 2, 7, 2],
         [0, 4, 5, 5]],

        [[8, 4, 1, 4],
         [8, 1, 1, 7],
         [9, 3, 6, 7],
         [0, 3, 5, 9]]],


       [[[4, 6, 4, 4],
         [4, 4, 8, 4],
         [7, 5, 5, 0],
         [5, 9, 3, 0]],

        [[0, 1, 2, 4],
         [0, 3, 2, 0],
         [5, 9, 0, 2],
         [2, 9, 2, 3]],

        [[2, 3, 4, 1],
         [9, 1, 4, 6],
         [2, 3, 0, 0],
         [0, 6, 3, 3]]]])
```

`a4` 's shape is (2, 3, 4, 5), this means it gets displayed like so:

- Inner most array = size 5
- Next array = size 4
- Next array = size 3
- Outer most array = size 2

4. Manipulating and comparing arrays

- Arithmetic
  - `+`, `-`, `*`, `/`, `//`, `**`, `%`
  - `np.exp()`
  - `np.log()`
  - [Dot product](#) - `np.dot()`
  - Broadcasting
- Aggregation
  - `np.sum()` - faster than Python's `.sum()` for NumPy arrays
  - `np.mean()`
  - `np.std()`
  - `np.var()`
  - `np.min()`
  - `np.max()`
  - `np.argmin()` - find index of minimum value
  - `np.argmax()` - find index of maximum value
  - These work on all `ndarray` 's
    - `a4.min(axis=0)` -- you can use axis as well
- Reshaping
  - `np.reshape()`
- Transposing
  - `a3.T`
- Comparison operators
  - `>`
  - `<`
  - `<=`
  - `>=`
  - `x != 3`
  - `x == 3`
  - `np.sum(x > 3)`

Arithmetic

```
a1
```

```
array([1, 2, 3])
```

```
ones = np.ones(3)
ones
```

```
array([1., 1., 1.])
```

```
# Add two arrays
a1 + ones
```

```
array([2., 3., 4.])
```

```
# Subtract two arrays
a1 - ones
```

```
array([0., 1., 2.])
```

```
# Multiply two arrays
a1 * ones
```

```
array([1., 2., 3.])
```

```
# Multiply two arrays
a1 * a2
```

```
array([[ 1. ,  4. ,  9.9],
       [ 4. , 10. , 19.5]])
```

```
a1.shape, a2.shape
```

```
((3,), (2, 3))
```

```
# This will error as the arrays have a different number of dimensions (2, 3) v
a2 * a3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[49], line 2
      1 # This will error as the arrays have a different number of dimensions (2, 3) 
vs. (2, 3, 3)
----> 2 a2 * a3

ValueError: operands could not be broadcast together with shapes (2,3) (2,3,3) 
```

```
a3
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

Broadcasting

- What is broadcasting?
  - Broadcasting is a feature of NumPy which performs an operation across multiple dimensions of data without replicating the data. This saves time and space. For example, if you have a 3x3 array (A) and want to add a 1x3 array (B), NumPy will add the row of (B) to every row of (A).
- Rules of Broadcasting
  1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
  2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
  3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

The broadcasting rule: In order to broadcast, the size of the trailing axes for both arrays in an operation must be either the same size or one of them must be one.

```
a1
```

```
array([1, 2, 3])
```

```
a1.shape
```

```
(3,)
```

```
a2.shape
```

```
(2, 3)
```

```python
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```python
a1 + a2
```

```
array([[2. , 4. , 6.3],
       [5. , 7. , 9.5]])
```

```python
a2 + 2
```

```
array([[3. , 4. , 5.3],
       [6. , 7. , 8.5]])
```

```python
# Raises an error because there's a shape mismatch (2, 3) vs. (2, 3, 3)
a2 + a3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[57], line 2
      1 # Raises an error because there's a shape mismatch (2, 3) vs. (2, 3, 3)
----> 2 a2 + a3

ValueError: operands could not be broadcast together with shapes (2,3) (2,3,3)
```

```python
# Divide two arrays
a1 / ones
```

```
array([1., 2., 3.])
```

```python
# Divide using floor division
a2 // a1
```

```
array([[1., 1., 1.],
       [4., 2., 2.]])
```

```python
# Take an array to a power
a1 ** 2
```

```
array([1, 4, 9])
```

```python
# You can also use np.square()
np.square(a1)
```

```
array([1, 4, 9])
```

```
# Modulus divide (what's the remainder)
a1 % 2
```

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

You can also find the log or exponential of an array using `np.log()` and `np.exp()`.

```
# Find the log of an array
np.log(a1)
```

```
array([0.        , 0.69314718, 1.09861229])
```

```
# Find the exponential of an array
np.exp(a1)
```

```
array([ 2.71828183,  7.3890561 , 20.08553692])
```

Aggregation

Aggregation - bringing things together, doing a similar thing on a number of things.

```
sum(a1)
```

```
6
```

```
np.sum(a1)
```

```
6
```

Tip: Use NumPy's `np.sum()` on NumPy arrays and Python's `sum()` on Python `list`s.

```
massive_array = np.random.random(100000)
massive_array.size, type(massive_array)
```

```
(100000, numpy.ndarray)
```

```
%timeit sum(massive_array) # Python sum()
%timeit np.sum(massive_array) # NumPy np.sum()
```

```
4.38 ms ± 119 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
20.3 µs ± 110 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Notice `np.sum()` is faster on the Numpy array (`numpy.ndarray`) than Python's `sum()`.

Now let's try it out on a Python list.

```
import random
massive_list = [random.randint(0, 10) for i in range(100000)]
len(massive_list), type(massive_list)
```

```
(100000, list)
```

```
massive_list[:10]
```

```
[0, 4, 5, 9, 7, 0, 1, 7, 8, 1]
```

```
%timeit sum(massive_list)
%timeit np.sum(massive_list)
```

```
598 µs ± 959 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
2.72 ms ± 10.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

NumPy's `np.sum()` is still fast but Python's `sum()` is faster on Python `list` s.

```
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```
# Find the mean
np.mean(a2)
```

```
3.6333333333333333
```

```
# Find the max
np.max(a2)
```

```
6.5
```

```
# Find the min
np.min(a2)
```

```
1.0
```

```
# Find the standard deviation
np.std(a2)
```

```
1.8226964152656422
```

```
# Find the variance
np.var(a2)
```

```
3.3222222222222224
```

```
# The standard deviation is the square root of the variance
np.sqrt(np.var(a2))
```

```
1.8226964152656422
```

What's mean?

Mean is the same as average. You can find the average of a set of numbers by adding them up and dividing them by how many there are.

What's standard deviation?

Standard deviation is a measure of how spread out numbers are.

What's variance?

The variance is the averaged squared differences of the mean.

To work it out, you:

1. Work out the mean
2. For each number, subtract the mean and square the result
3. Find the average of the squared differences

```
# Demo of variance
high_var_array = np.array([1, 100, 200, 300, 4000, 5000])
low_var_array = np.array([2, 4, 6, 8, 10])

np.var(high_var_array), np.var(low_var_array)
```

```
(4296133.472222221, 8.0)
```

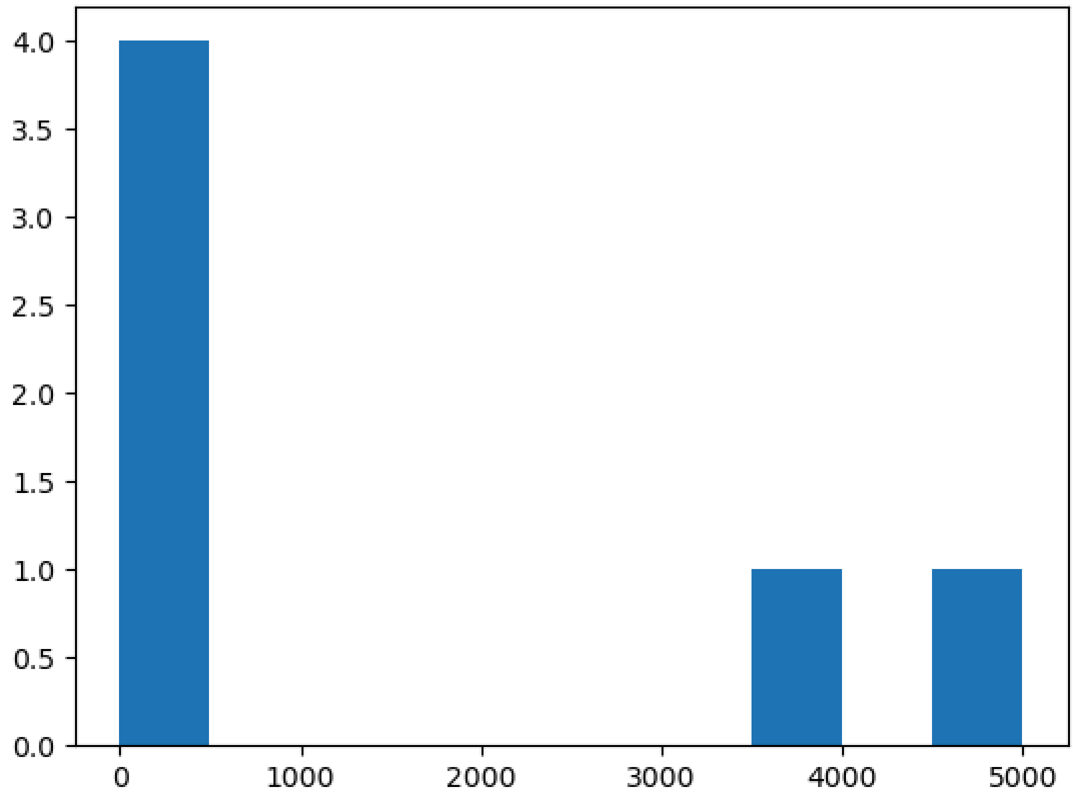
```
np.std(high_var_array), np.std(low_var_array)
```

```
(2072.711623024829, 2.8284271247461903)
```

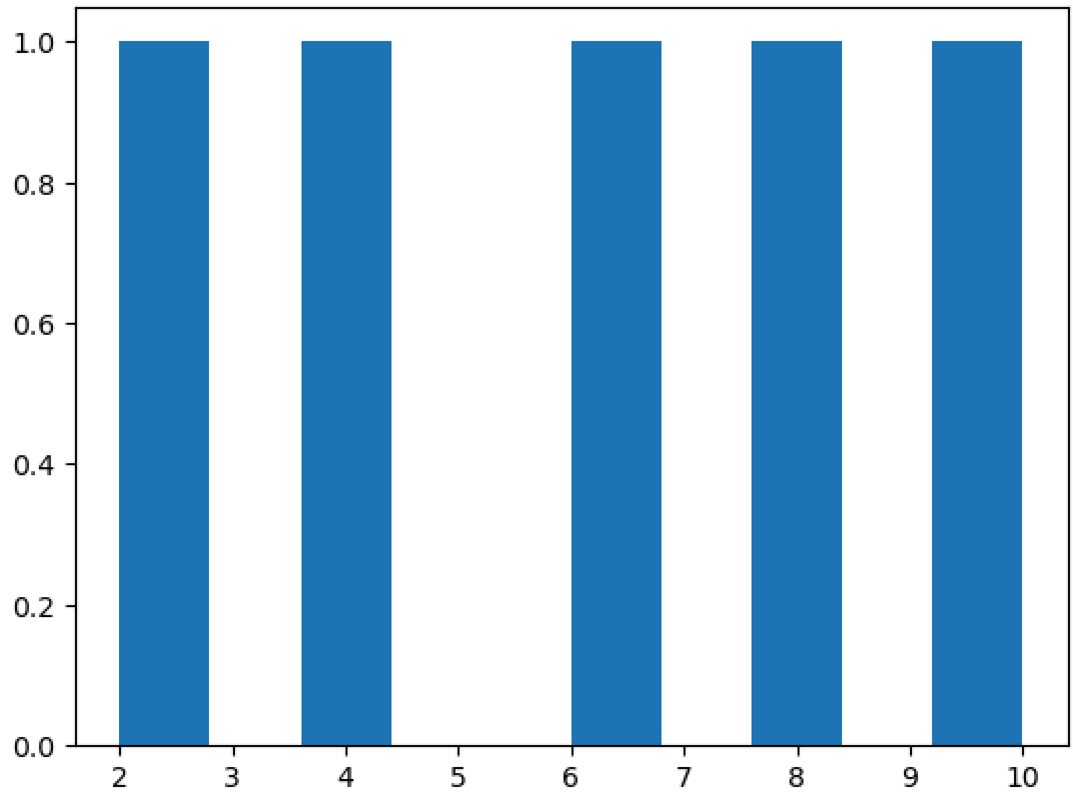
```
# The standard deviation is the square root of the variance
np.sqrt(np.var(high_var_array))
```

```
2072.711623024829
```

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(high_var_array)
plt.show()
```




```
plt.hist(low_var_array)
plt.show()
```



Reshaping

```
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```
a2.shape
```

```
(2, 3)
```

```
a2 + a3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[86], line 1
----> 1 a2 + a3

ValueError: operands could not be broadcast together with shapes (2,3) (2,3,3)
```

```
a2.reshape(2, 3, 1)
```

```
a2.reshape(2, 3, 1) + a3
```

Transpose

A tranpose reverses the order of the axes.

For example, an array with shape `(2, 3)` becomes `(3, 2)`.

```
a2.shape
```

```
a2.T
```

```
a2.transpose()
```

```
a2.T.shape
```

For larger arrays, the default value of a tranpose is to swap the first and last axes.

For example, `(5, 3, 3) -> (3, 3, 5)`.

```
matrix = np.random.random(size=(5, 3, 3))
matrix
```

```
matrix.shape
```

```
matrix.T
```

```
matrix.T.shape
```

```
# Check to see if the reverse shape is same as tranpose shape
matrix.T.shape == matrix.shape[::-1]
```

```
# Check to see if the first and last axes are swapped
matrix.T == matrix.swapaxes(0, -1) # swap first (0) and last (-1) axes
```

You can see more advanced forms of tranposing in the NumPy documentation under `numpy.transpose`.

Dot product

The main two rules for dot product to remember are:

1. The **inner dimensions** must match:
   - `(3, 2) @ (3, 2)` won't work
   - `(2, 3) @ (3, 2)` will work
   - `(3, 2) @ (2, 3)` will work
2. The resulting matrix has the shape of the **outer dimensions**:
   - `(2, 3) @ (3, 2) -> (2, 2)`
   - `(3, 2) @ (2, 3) -> (3, 3)`

Note: In NumPy, `np.dot()` and `@` can be used to acheive the same result for 1-2 dimension arrays. However, their behaviour begins to differ at arrays with 3+ dimensions.

```
np.random.seed(0)
mat1 = np.random.randint(10, size=(3, 3))
mat2 = np.random.randint(10, size=(3, 2))

mat1.shape, mat2.shape
```

```
mat1
```

```
mat2
```

```
np.dot(mat1, mat2)
```

```
# Can also achieve np.dot() with "@"
# (however, they may behave differently at 3D+ arrays)
mat1 @ mat2
```

```
np.random.seed(0)
mat3 = np.random.randint(10, size=(4,3))
mat4 = np.random.randint(10, size=(4,3))
mat3
```

```
mat4
```

```
# This will fail as the inner dimensions of the matrices do not match
np.dot(mat3, mat4)
```

```
mat3.T.shape
```

```
# Dot product
np.dot(mat3.T, mat4)
```

```
# Element-wise multiplication, also known as Hadamard product
mat3 * mat4
```

Dot product practical example, nut butter sales

```
np.random.seed(0)
sales_amounts = np.random.randint(20, size=(5, 3))
sales_amounts
```

```
weekly_sales = pd.DataFrame(sales_amounts,
                            index=["Mon", "Tues", "Wed", "Thurs", "Fri"],
                            columns=["Almond butter", "Peanut butter", "Cashew
weekly_sales
```

```
prices = np.array([10, 8, 12])
prices
```

```
array([10,  8, 12])
```

```
butter_prices = pd.DataFrame(prices.reshape(1, 3),
                             index=["Price"],
                             columns=["Almond butter", "Peanut butter", "Cashe
butter_prices.shape
```

```
(1, 3)
```

```
weekly_sales.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[89], line 1
----> 1 weekly_sales.shape

NameError: name 'weekly_sales' is not defined
```

```
# Find the total amount of sales for a whole day
total_sales = prices.dot(sales_amounts)
total_sales
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[90], line 2
      1 # Find the total amount of sales for a whole day
----> 2 total_sales = prices.dot(sales_amounts)
      3 total_sales

NameError: name 'sales_amounts' is not defined
```

The shapes aren't aligned, we need the middle two numbers to be the same.

```
prices
```

```
array([10,  8, 12])
```

```
sales_amounts.T.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[92], line 1
----> 1 sales_amounts.T.shape

NameError: name 'sales_amounts' is not defined
```

```
# To make the middle numbers the same, we can transpose
total_sales = prices.dot(sales_amounts.T)
total_sales
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[93], line 2
      1 # To make the middle numbers the same, we can transpose
----> 2 total_sales = prices.dot(sales_amounts.T)
      3 total_sales

NameError: name 'sales_amounts' is not defined
```

```
butter_prices.shape, weekly_sales.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[94], line 1
----> 1 butter_prices.shape, weekly_sales.shape

NameError: name 'weekly_sales' is not defined
```

```
daily_sales = butter_prices.dot(weekly_sales.T)
daily_sales
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[95], line 1
----> 1 daily_sales = butter_prices.dot(weekly_sales.T)
      2 daily_sales

NameError: name 'weekly_sales' is not defined
```

```
# Need to transpose again
weekly_sales["Total"] = daily_sales.T
weekly_sales
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[96], line 2
      1 # Need to transpose again
----> 2 weekly_sales["Total"] = daily_sales.T
      3 weekly_sales

NameError: name 'daily_sales' is not defined
```

Comparison operators

Finding out if one array is larger, smaller or equal to another.

```
a1
```

```
array([1, 2, 3])
```

```
a2
```

```
array([[1. , 2. , 3.3],
       [4. , 5. , 6.5]])
```

```
a1 > a2
```

```
array([[False, False, False],
       [False, False, False]])
```

```
a1 >= a2
```

```
array([[ True,  True, False],
       [False, False, False]])
```

```
a1 > 5
```

```
array([False, False, False])
```

```
a1 == a1
```

```
array([ True,  True,  True])
```

```
a1 == a2
```

```
array([[ True,  True, False],
       [False, False, False]])
```

5. Sorting arrays

- `np.sort()` - sort values in a specified dimension of an array.
- `np.argsort()` - return the indices to sort the array on a given axis.
- `np.argmax()` - return the index/indicies which gives the highest value(s) along an axis.
- `np.argmin()` - return the index/indices which gives the lowest value(s) along an axis.

```
random_array
```

```
array([[1, 7, 2],
       [7, 0, 2],
       [8, 8, 8],
       [2, 5, 2],
       [4, 8, 6]])
```

```
np.sort(random_array)
```

```
array([[1, 2, 7],
       [0, 2, 7],
       [8, 8, 8],
       [2, 2, 5],
       [4, 6, 8]])
```

```
np.argsort(random_array)
```

```
array([[0, 2, 1],
       [1, 2, 0],
       [0, 1, 2],
       [0, 2, 1],
       [0, 2, 1]])
```

```
a1
```

```
array([1, 2, 3])
```

```
# Return the indices that would sort an array
np.argsort(a1)
```

```
array([0, 1, 2])
```

```
# No axis
np.argmin(a1)
```

```
0
```

```
random_array
```

```
array([[1, 7, 2],
       [7, 0, 2],
       [8, 8, 8],
       [2, 5, 2],
       [4, 8, 6]])
```

```
# Down the vertical
np.argmax(random_array, axis=1)
```

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

```
# Across the horizontal
np.argmin(random_array, axis=0)
```

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

6. Use case

Turning an image into a NumPy array.

Why?

Because computers can use the numbers in the NumPy array to find patterns in the image and in turn use those patterns to figure out what's in the image.

This is what happens in modern computer vision algorithms.

Let's start with this beautiful image of a panda:

photo of a panda waving

```python
from matplotlib.image import imread

panda = imread('../images/numpy-panda.jpeg')
print(type(panda))
```

```
<class 'numpy.ndarray'>
```

```python
panda.shape
```

```
(2330, 3500, 3)
```

```python
panda
```

```
array([[[0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        ...,
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765]],

       [[0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        ...,
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765]],

       [[0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        [0.05490196, 0.10588235, 0.06666667],
        ...,
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765],
        [0.16470589, 0.12941177, 0.09411765]],

       ...,

       [[0.13333334, 0.07450981, 0.05490196],
        [0.12156863, 0.0627451 , 0.04313726],
        [0.10980392, 0.05098039, 0.03137255],
        ...,
```

```
          [0.02745098, 0.02745098, 0.03529412],
          [0.02745098, 0.02745098, 0.03529412],
          [0.02745098, 0.02745098, 0.03529412]],

         [[0.13333334, 0.07450981, 0.05490196],
          [0.12156863, 0.0627451 , 0.04313726],
          [0.12156863, 0.0627451 , 0.04313726],
          ...,
          [0.02352941, 0.02352941, 0.03137255],
          [0.02352941, 0.02352941, 0.03137255],
          [0.02352941, 0.02352941, 0.03137255]],

         [[0.13333334, 0.07450981, 0.05490196],
          [0.12156863, 0.0627451 , 0.04313726],
          [0.12156863, 0.0627451 , 0.04313726],
          ...,
          [0.02352941, 0.02352941, 0.03137255],
          [0.02352941, 0.02352941, 0.03137255],
          [0.02352941, 0.02352941, 0.03137255]]], dtype=float32)
```

photo of a car

```
car = imread("../images/numpy-car-photo.png")
car.shape
```

```
(431, 575, 4)
```

```
car[:,:,:3].shape
```

```
(431, 575, 3)
```

photo a dog

```
dog = imread("../images/numpy-dog-photo.png")
dog.shape
```

```
(432, 575, 4)
```

```
dog
```

```
array([[[0.70980394, 0.80784315, 0.88235295, 1.        ],
        [0.72156864, 0.8117647 , 0.8862745 , 1.        ],
        [0.7411765 , 0.8156863 , 0.8862745 , 1.        ],
        ...,
        [0.49803922, 0.6862745 , 0.8392157 , 1.        ],
        [0.49411765, 0.68235296, 0.8392157 , 1.        ],
        [0.49411765, 0.68235296, 0.8352941 , 1.        ]],

       [[0.69411767, 0.8039216 , 0.8862745 , 1.        ],
        [0.7019608 , 0.8039216 , 0.88235295, 1.        ],
        [0.7058824 , 0.80784315, 0.88235295, 1.        ],
        ...,
        [0.5019608 , 0.6862745 , 0.84705883, 1.        ],
```

```
 [0.49411765, 0.68235296, 0.84313726, 1.        ],
 [0.49411765, 0.68235296, 0.8392157 , 1.        ]],

[[0.6901961 , 0.8       , 0.88235295, 1.        ],
 [0.69803923, 0.8039216 , 0.88235295, 1.        ],
 [0.7058824 , 0.80784315, 0.88235295, 1.        ],
 ...,
 [0.5019608 , 0.6862745 , 0.84705883, 1.        ],
 [0.49803922, 0.6862745 , 0.84313726, 1.        ],
 [0.49803922, 0.6862745 , 0.84313726, 1.        ]]
```

Pandas

importing Pandas and getting dtaaframe by importing a csv file

```
import pandas as pd
```

```
car_sales = pd.read_csv('/content/car-sales.csv')
car_sales
```

	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

Describing data

```
#Function()
car_sales.describe()
```

	Odometer (KM)	Doors
count	10.000000	10.000000
mean	78601.400000	4.000000
std	61983.471735	0.471405
min	11179.000000	3.000000
25%	35836.250000	4.000000
50%	57369.000000	4.000000
75%	96384.500000	4.000000
max	213095.000000	5.000000

```
car_sales.info() #it wll show how many null objects present in the DF
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   Make           10 non-null     object
 1   Colour         10 non-null     object
 2   Odometer (KM)  10 non-null     int64 
 3   Doors          10 non-null     int64 
 4   Price          10 non-null     object
dtypes: int64(2), object(3)
memory usage: 528.0+ bytes
```

```
car_sales.Doors.sum()
```

```
40
```

```
#Attribute
car_sales.dtypes
```

```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
```

```
Price            object
dtype: object
```

```
car_sales.columns
```

```
Index(['Make', 'Colour', 'Odometer (KM)', 'Doors', 'Price'], dtype='object')
```

```
car_sales.index
```

```
RangeIndex(start=0, stop=10, step=1)
```

```
len(car_sales.Doors)
```

```
10
```

Viewing and Selecting Data

```
car_sales.head()
```

	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

```
car_sales.tail()
```

	Make	Colour	Odometer (KM)	Doors	Price
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

```
#loc & iloc

animal = pd.Series(['lion','tiger','bear','deer','eagle','cheetah'],index= [0,2,3,4,3,4])
animal
```

```
0       lion
2      tiger
3       bear
4       deer
3      eagle
4    cheetah
dtype: object
```

```
animal.loc[3] # loc is used to grab index number which may have duplicate index numbers
```

```
3     bear
3    eagle
dtype: object
```

```
car_sales.loc[4]
```

```
Make                Nissan
Colour               White
Odometer (KM)       213095
Doors                    4
Price            $3,500.00
Name: 4, dtype: object
```

```
animal
```

```
0       lion
2      tiger
```

```
3       bear
4       deer
3      eagle
4    cheetah
dtype: object
```

```
animal.iloc[3] #iloc refers to the position only not the index number
```

```
'deer'
```

```
animal.iloc[:3]
```

```
0     lion
2    tiger
3     bear
dtype: object
```

```
car_sales.iloc[:3]
```

	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

When choosing or transitioning between loc and iloc, there is one "gotcha" worth keeping in mind, which is that the two methods use slightly different indexing schemes.

iloc uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So 0:10 will select entries 0,...,9. loc, meanwhile, indexes inclusively. So 0:10 will select entries 0,...,10.

This is particularly confusing when the DataFrame index is a simple numerical list, e.g. 0,...,1000. In this case df.iloc[0:1000] will return 1000 entries, while df.loc[0:1000] return 1001 of them! To get 1000 elements using loc, you will need to go one lower and ask for df.loc[0:999].

```
#Boolean indexing
```

```
car_sales[car_sales['Make']== 'Toyota']
```

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	150043	4	$4,000.00
2	Toyota	Blue	32549	3	$7,000.00
5	Toyota	Green	99213	4	$4,500.00
8	Toyota	White	60000	4	$6,250.00

```
# To compare two columns we can use Crosstab
```

```
pd.crosstab(car_sales['Make'],car_sales['Doors'])
```

Doors	3	4	5
Make			
BMW	0	0	1
Honda	0	3	0
Nissan	0	2	0
Toyota	1	3	0

```
# To compare more column we should groupby the categorical columns
```

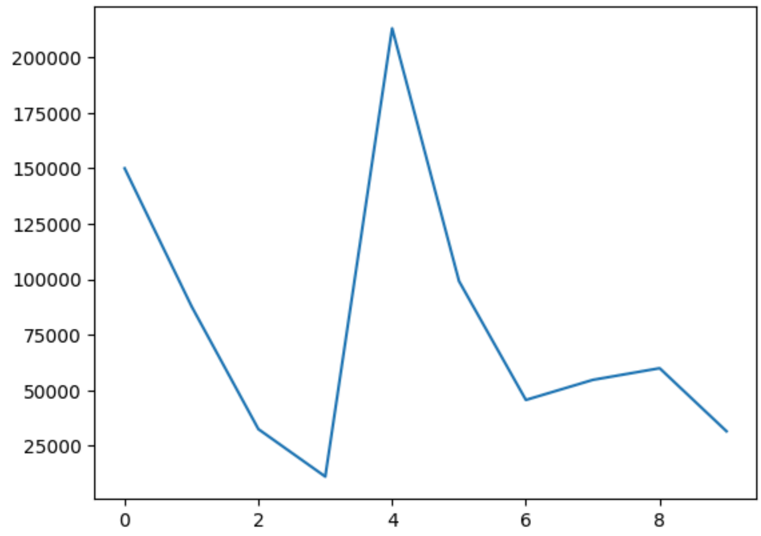
```
car_sales.groupby(['Make'])
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fccc505f040>
```

```
%matplotlib inline
import matplotlib.pyplot as plt
```

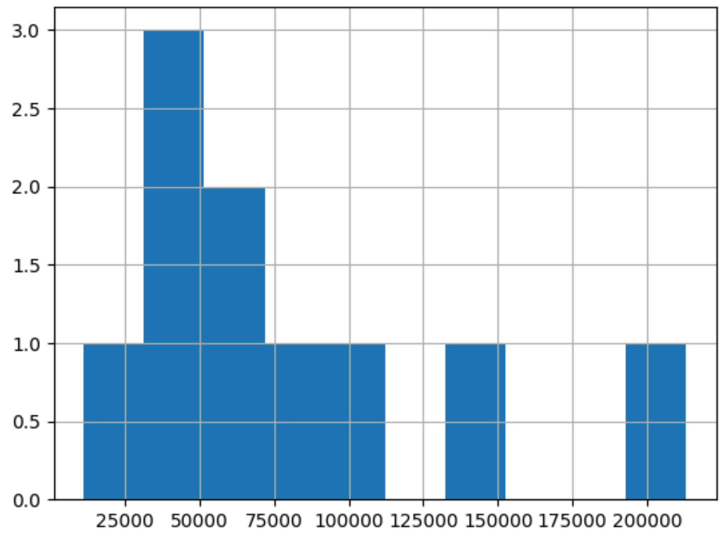
```
car_sales[("Odometer (KM)")].plot()
```

```
<Axes: >
```




```
car_sales[("Odometer (KM)")].hist()
```

```
<Axes: >
```




```
car_sales[("Price")].plot()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-43fb2b4fe724> in <cell line: 1>()
----> 1 car_sales[("Price")].plot()

                              3 frames

/usr/local/lib/python3.10/dist-packages/pandas/plotting/_matplotlib/core.py in
_compute_plot_data(self)
    630         # no non-numeric frames or series allowed
    631         if is_empty:
--> 632             raise TypeError("no numeric data to plot")
    633
    634         self.data = numeric_data.apply(self._convert_to_ndarray)

TypeError: no numeric data to plot
```

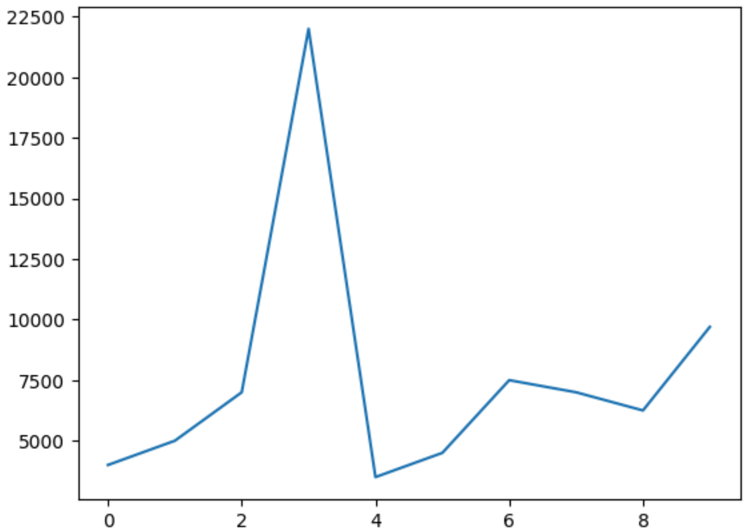
```
type(car_sales.Price) # Price column has '$' string, we need to replace and convert it into Int dtype
```

```
pandas.core.series.Series
def __init__(data=None, index=None, dtype: Dtype | None=None, name=None, copy:
bool | None=None, fastpath: bool=False) -> None

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object
supports both integer- and label-based indexing and provides a host of
methods for performing operations involving the index. Statistical
```

```
car_sales["Price"] = car_sales["Price"].str.replace('[\$\,]|\.\d*', '', regex=True).astype(int)
```

```
car_sales.Price.plot()
```

```
<Axes: >
```



Manipulating Data

```
car_sales.Make.str.lower() # this won't affect original dataframe. To make it possible, need to assign to a variable
```

```
0    toyota
1     honda
2    toyota
3       bmw
4    nissan
5    toyota
6     honda
7     honda
8    toyota
9    nissan
Name: Make, dtype: object
```

```
car_sales
```

	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

```
car_sales['Make'] = car_sales.Make.str.lower()
car_sales
```

	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

```
car_sales_m = pd.read_csv('car-sales-missing-data.csv')
car_sales_m
```

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	NaN	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	NaN	4.0	$4,500
6	Honda	NaN	NaN	4.0	$7,500
7	Honda	Blue	NaN	4.0	NaN
8	Toyota	White	60000.0	NaN	NaN
9	NaN	White	31600.0	4.0	$9,700

```
car_sales_m["Odometer"].fillna(car_sales_m["Odometer"].mean()) # this is tmeporary, to make it permanent add inplace= T
```

```
0    150043.000000
1     87899.000000
2     92302.666667
3     11179.000000
4    213095.000000
5     92302.666667
6     92302.666667
7     92302.666667
8     60000.000000
9     31600.000000
Name: Odometer, dtype: float64
```

```
car_sales_m
```

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.0	4.0	$4,000
1	Honda	Red	87899.0	4.0	$5,000
2	Toyota	Blue	NaN	3.0	$7,000
3	BMW	Black	11179.0	5.0	$22,000
4	Nissan	White	213095.0	4.0	$3,500
5	Toyota	Green	NaN	4.0	$4,500
6	Honda	NaN	NaN	4.0	$7,500
7	Honda	Blue	NaN	4.0	NaN
8	Toyota	White	60000.0	NaN	NaN
9	NaN	White	31600.0	4.0	$9,700

```
car_sales_m["Odometer"].fillna(car_sales_m["Odometer"].mean(),inplace=True)
car_sales_m

#or we can use

#car_sales_m["Odometer"] = car_sales_m["Odometer"].fillna(car_sales_m["Odometer"].mean(),inplace=True)
```

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500
6	Honda	NaN	92302.666667	4.0	$7,500
7	Honda	Blue	92302.666667	4.0	NaN
8	Toyota	White	60000.000000	NaN	NaN
9	NaN	White	31600.000000	4.0	$9,700

```
#To remove entire row containing missing values, use dropna()

car_sales_m.dropna()
```

	Make	Colour	Odometer	Doors	Price
0	Toyota	White	150043.000000	4.0	$4,000
1	Honda	Red	87899.000000	4.0	$5,000
2	Toyota	Blue	92302.666667	3.0	$7,000
3	BMW	Black	11179.000000	5.0	$22,000
4	Nissan	White	213095.000000	4.0	$3,500
5	Toyota	Green	92302.666667	4.0	$4,500

```
car_sales_dropped = car_sales_m.dropna()
car_sales_dropped.to_csv('car_sales_dropped.csv')
```

```
#Column from Series

seats = pd.Series([5,5,5,5,5])
car_sales['Seats'] = seats
```

```
car_sales
```

	Make	Colour	Odometer (KM)	Doors	Price	Seats
0	Toyota	White	150043	4	$4,000.00	5.0
1	Honda	Red	87899	4	$5,000.00	5.0
2	Toyota	Blue	32549	3	$7,000.00	5.0
3	BMW	Black	11179	5	$22,000.00	5.0
4	Nissan	White	213095	4	$3,500.00	5.0
5	Toyota	Green	99213	4	$4,500.00	NaN
6	Honda	Blue	45698	4	$7,500.00	NaN
7	Honda	Blue	54738	4	$7,000.00	NaN
8	Toyota	White	60000	4	$6,250.00	NaN
9	Nissan	White	31600	4	$9,700.00	NaN

```
car_sales['Seats'].fillna(5,inplace=True)
car_sales
```

	Make	Colour	Odometer (KM)	Doors	Price	Seats
0	Toyota	White	150043	4	$4,000.00	5.0
1	Honda	Red	87899	4	$5,000.00	5.0
2	Toyota	Blue	32549	3	$7,000.00	5.0
3	BMW	Black	11179	5	$22,000.00	5.0
4	Nissan	White	213095	4	$3,500.00	5.0
5	Toyota	Green	99213	4	$4,500.00	5.0
6	Honda	Blue	45698	4	$7,500.00	5.0
7	Honda	Blue	54738	4	$7,000.00	5.0
8	Toyota	White	60000	4	$6,250.00	5.0
9	Nissan	White	31600	4	$9,700.00	5.0

```
#To drop a colum,
```

```
car_sales.drop('Seats', axis=1,inplace=True)
car_sales
```

	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

```
car_sales_shuffled = car_sales.sample(frac=1) #gives just a sample with 100% of data
car_sales_shuffled
```

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

```
car_sales_shuffled.sample(frac=0.2)
```

	Make	Colour	Odometer (KM)	Doors	Price
8	Toyota	White	60000	4	$6,250.00
3	BMW	Black	11179	5	$22,000.00

```
car_sales_shuffled.reset_index(drop=True ,inplace=True)
```

```
car_sales_shuffled
```

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

```
#apply() applies the function provided to specified col

car_sales['Odometer (KM)'] = car_sales['Odometer (KM)'].apply(lambda x: x/2)

car_sales
```

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	White	75021.5	4	$4,000.00
1	Honda	Red	43949.5	4	$5,000.00
2	Toyota	Blue	16274.5	3	$7,000.00
3	BMW	Black	5589.5	5	$22,000.00
4	Nissan	White	106547.5	4	$3,500.00
5	Toyota	Green	49606.5	4	$4,500.00
6	Honda	Blue	22849.0	4	$7,500.00
7	Honda	Blue	27369.0	4	$7,000.00
8	Toyota	White	30000.0	4	$6,250.00
9	Nissan	White	15800.0	4	$9,700.00

Introduction to Matplotlib

Get straight into plotting data, that's what we're focused on.

Video 0 will be concepts and contain details like anatomy of a figure. The rest of the videos will be pure code based.

0. Concepts in Matplotlib
1. 2 ways of creating plots (pyplot & OO) - use the OO method
2. Plotting data (NumPy arrays), line, scatter, bar, hist, subplots
3. Plotting data directly with Pandas (using the pandas matplotlib wrapper)
4. Plotting data (pandas DataFrames) with the OO method, line, scatter, bar, hist, subplots
5. Cutomizing your plots, `limits`, `colors`, `styles`, `legends`
6. Saving plots

0. Concepts in Matplotlib

- What is Matplotlib?
- Why Matplotlib?
- Anatomy of a figure
- Where does Matplotlib fit into the ecosystem?
    - A Matplotlib workflow

1. 2 ways of creating plots

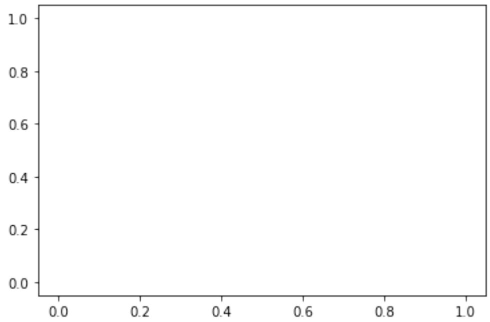
- `pyplot()`
- `OO` - https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html
- Matplotlib recommends the OO API
    - https://matplotlib.org/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py
    - https://matplotlib.org/3.1.1/tutorials/introductory/lifecycle.html

Start by importing `Matplotlib` and setting up the `%matplotlib inline` magic command.

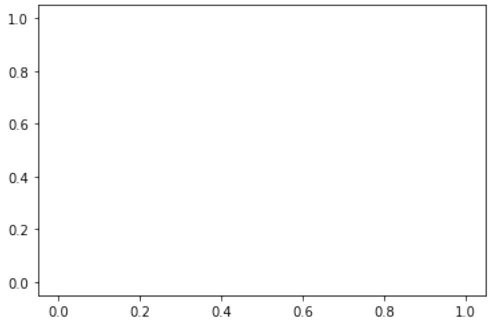
```
# Import matplotlib and setup the figures to display within the notebook
%matplotlib inline
import matplotlib.pyplot as plt
```

```
# Create a simple plot, without the semi-colon
plt.plot()
```

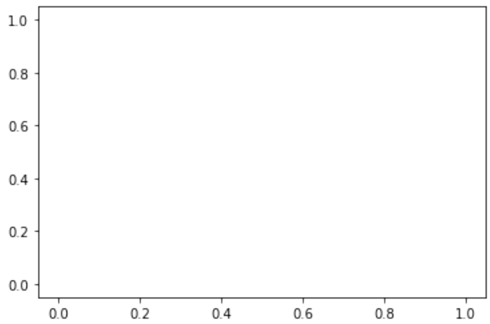
```
[]
```




```
# With the semi-colon
plt.plot();
```

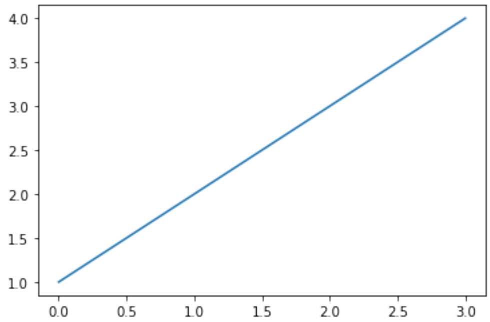


```
# You could use plt.show() if you want
plt.plot()
plt.show()
```



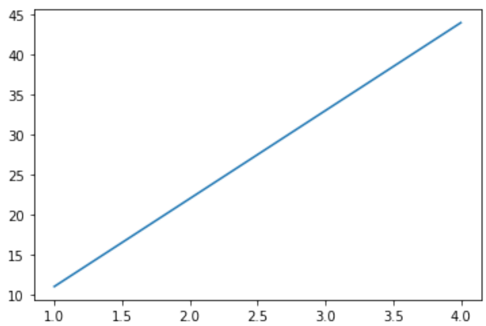
```
# Let's add some data
plt.plot([1, 2, 3, 4])
```

```
[<matplotlib.lines.Line2D at 0x11366a490>]
```

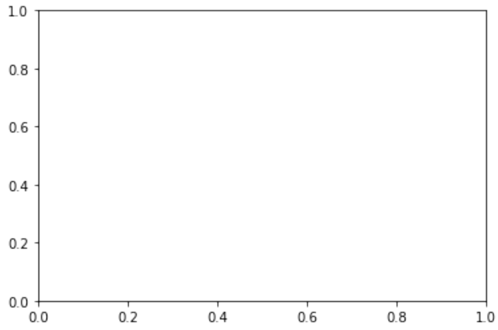


```
# Create some data
x = [1, 2, 3, 4]
y = [11, 22, 33, 44]
```

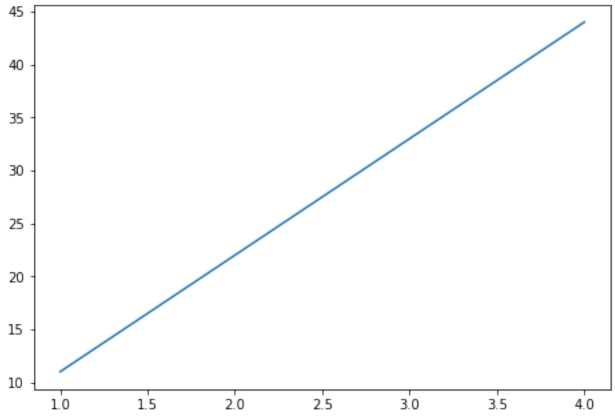
```
# With a semi-colon and now a y value
plt.plot(x, y);
```



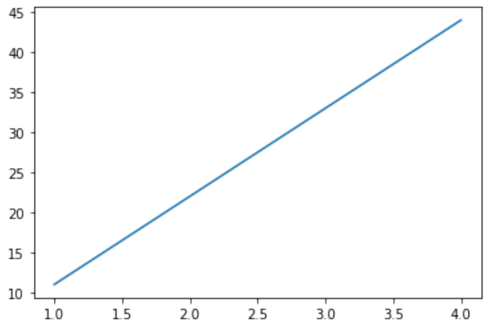
```
# Creating a plot with the OO verison, confusing way first
fig = plt.figure()
ax = fig.add_subplot()
plt.show()
```



```
# Confusing #2
fig = plt.figure()
ax = fig.add_axes([1, 1, 1, 1])
ax.plot(x, y)
plt.show()
```



```
# Easier and more robust going forward (what we're going to use)
fig, ax = plt.subplots()
ax.plot(x, y);
```

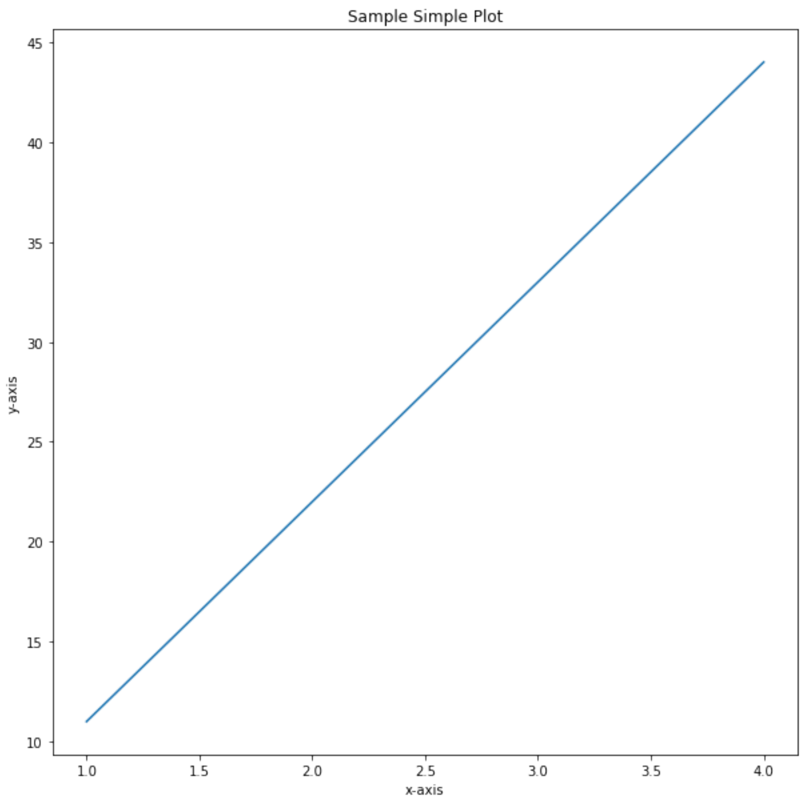


-> Show figure/plot anatomy here <-

```
# This is where the object orientated name comes from
type(fig), type(ax)
```

```
(matplotlib.figure.Figure, matplotlib.axes._subplots.AxesSubplot)
```

```
# A matplotlib workflow

# 0. Import and get matplotlib ready
%matplotlib inline
import matplotlib.pyplot as plt

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

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

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

# 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 see what suits it best.

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

- `line`
- `scatter`
- `bar`
- `hist`
- `subplots()`

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

```
import numpy as np
```

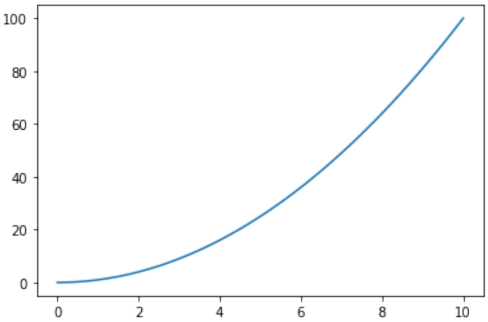
Line

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

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

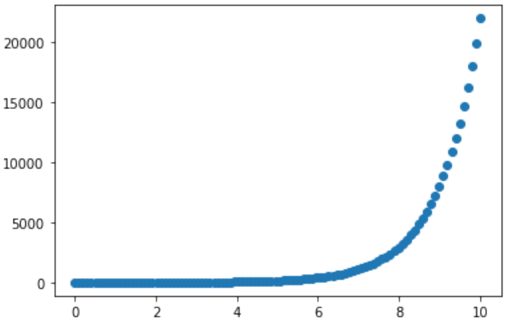
```
array([0.        , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,
       0.50505051, 0.60606061, 0.70707071, 0.80808081, 0.90909091])
```

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

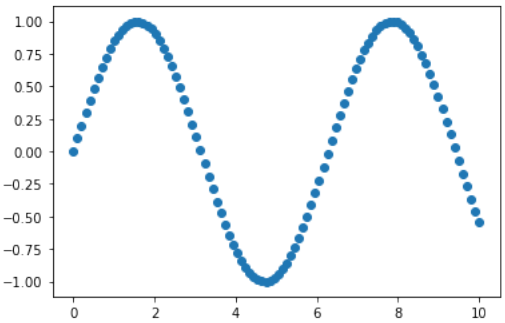


Scatter

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



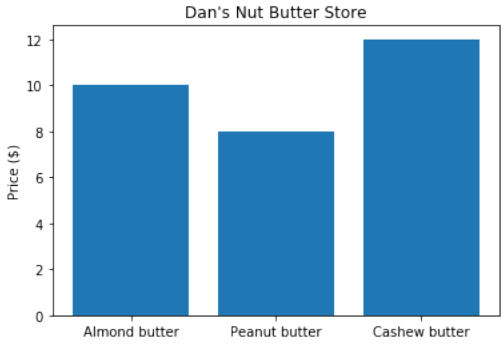

```
fig, ax = plt.subplots()
ax.scatter(x, np.sin(x));
```



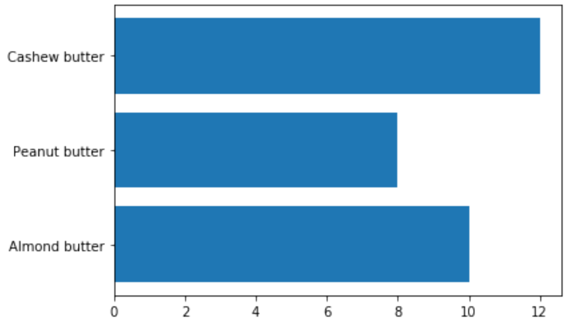
Bar

- Vertical
- Horizontal

```
# 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 ($)");
```



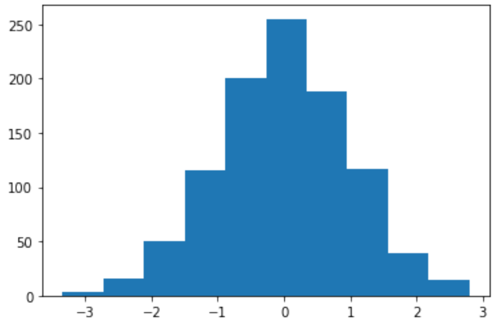

```
fig, ax = plt.subplots()
ax.barh(list(nut_butter_prices.keys()), list(nut_butter_prices.values()));
```



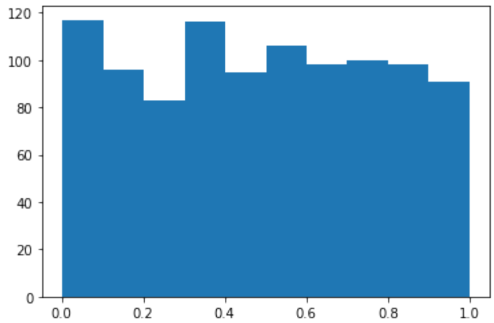
Histogram (hist)

- Could show image of normal distribution here

```
# 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);
```



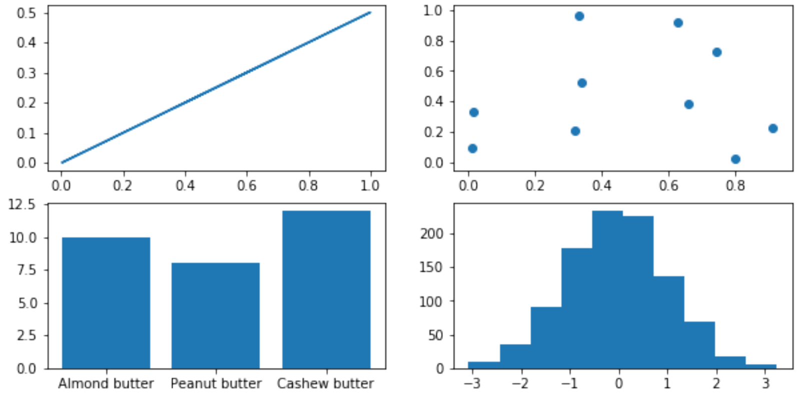
```
x = np.random.random(1000) # random data from random distribution

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



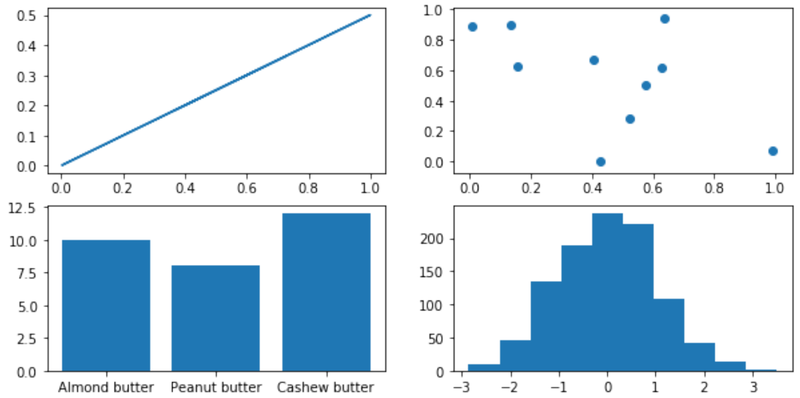
Subplots

- Multiple plots on one figure https://matplotlib.org/3.1.1/gallery/recipes/create_subplots.html

```
# Option 1: Create multiple subplots
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));
```




```
# Option 2: Create multiple subplots
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))

# Index 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

This section uses the pandas `pd.plot()` method on a DataFrame to plot columns directly.

- https://datatofish.com/plot-dataframe-pandas/
- https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html
- `line`
- `scatter`
- `bar`
- `hist`
- `df.plot(subplots=True, figsize=(6, 6))`

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

```
import pandas as pd
```

Now we need some data to check out.

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

	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

- Concept
- DataFrame

Often, reading things won't make sense. Practice writing code for yourself, get it out of the docs and into your workspace. See what happens when you run it.

Let's start with trying to replicate the pandas visualization documents.

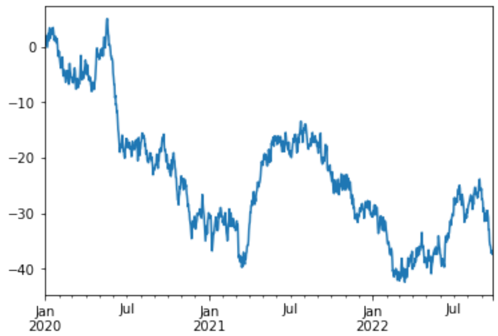
```
# Start with some dummy data
ts = pd.Series(np.random.randn(1000),
               index=pd.date_range('1/1/2020', periods=1000))
ts
```

```
2020-01-01    0.738301
2020-01-02   -0.436335
2020-01-03    1.552973
2020-01-04   -0.721055
2020-01-05   -0.522301
                ...
2022-09-22   -0.529207
2022-09-23   -0.760224
2022-09-24    0.399311
2022-09-25   -0.669529
2022-09-26    0.238585
Freq: D, Length: 1000, dtype: float64
```

```
# What does cumsum() do?
ts.cumsum()
```

```
2020-01-01     0.738301
2020-01-02     0.301966
2020-01-03     1.854938
2020-01-04     1.133883
2020-01-05     0.611582
                ...
2022-09-22   -36.324290
2022-09-23   -37.084515
2022-09-24   -36.685204
2022-09-25   -37.354733
2022-09-26   -37.116148
Freq: D, Length: 1000, dtype: float64
```

```
ts.cumsum().plot();
```



Working with actual data

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

```
# Remove price column symbols
car_sales["Price"] = car_sales["Price"].str.replace('[\$\,\.]', '')
car_sales
```

	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

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

	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

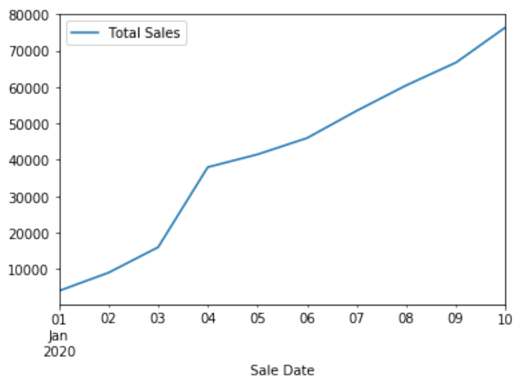
```
# Add a date column
car_sales["Sale Date"] = pd.date_range("1/1/2020", periods=len(car_sales))
car_sales
```

	Make	Colour	Odometer (KM)	Doors	Price	Sale Date
0	Toyota	White	150043	4	4000	2020-01-01
1	Honda	Red	87899	4	5000	2020-01-02
2	Toyota	Blue	32549	3	7000	2020-01-03
3	BMW	Black	11179	5	22000	2020-01-04
4	Nissan	White	213095	4	3500	2020-01-05
5	Toyota	Green	99213	4	4500	2020-01-06
6	Honda	Blue	45698	4	7500	2020-01-07
7	Honda	Blue	54738	4	7000	2020-01-08
8	Toyota	White	60000	4	6250	2020-01-09
9	Nissan	White	31600	4	9700	2020-01-10

```
# 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
```

	Make	Colour	Odometer (KM)	Doors	Price	Sale Date	Total Sales
0	Toyota	White	150043	4	4000	2020-01-01	4000
1	Honda	Red	87899	4	5000	2020-01-02	9000
2	Toyota	Blue	32549	3	7000	2020-01-03	16000
3	BMW	Black	11179	5	22000	2020-01-04	38000
4	Nissan	White	213095	4	3500	2020-01-05	41500
5	Toyota	Green	99213	4	4500	2020-01-06	46000
6	Honda	Blue	45698	4	7500	2020-01-07	53500
7	Honda	Blue	54738	4	7000	2020-01-08	60500
8	Toyota	White	60000	4	6250	2020-01-09	66750
9	Nissan	White	31600	4	9700	2020-01-10	76450

```
car_sales.plot(x='Sale Date', y='Total Sales');
```



Scatter

- Concept
- DataFrame

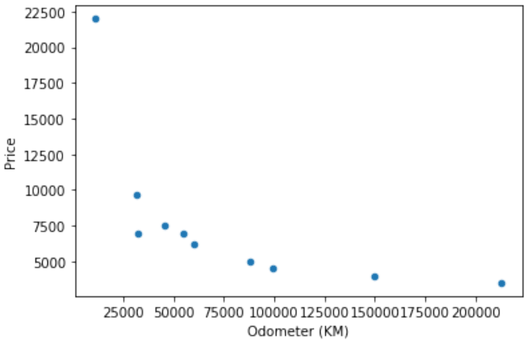
```
# Doesn't work
car_sales.plot(x="Odometer (KM)", y="Price", kind="scatter")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-34-540f318a89d0> in <module>
      1 # Doesn't work
----> 2 car_sales.plot(x="Odometer (KM)", y="Price", kind="scatter")
```

3 frames

```
~/Desktop/ml-course/work-in-progress/env/lib/python3.7/site-packages/pandas/plotting/_matplotlib/core.py in __init__(self, data, x, y, **kwargs)
    870             raise ValueError(self._kind + " requires x column to be numeric")
    871         if len(self.data[y]._get_numeric_data()) == 0:
--> 872             raise ValueError(self._kind + " requires y column to be numeric")
    873 
    874         self.x = x

ValueError: scatter requires y column to be numeric
```

```
# Convert Price to int
car_sales["Price"] = car_sales["Price"].astype(int)
car_sales.plot(x="Odometer (KM)", y="Price", kind='scatter');
```



Bar

- Concept
- DataFrame

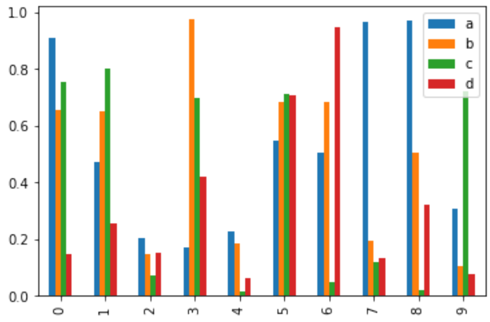
```
x = np.random.rand(10, 4)
x
```

```
array([[0.91054912, 0.65668407, 0.75347508, 0.1488774 ],
       [0.4739657 , 0.65199569, 0.80087623, 0.25613654],
       [0.20515965, 0.14991211, 0.07454593, 0.15030318],
       [0.17102306, 0.97405707, 0.69580935, 0.41898253],
       [0.22654692, 0.1848998 , 0.01482526, 0.0647843 ],
       [0.54732069, 0.68484856, 0.71222659, 0.70537797],
       [0.50304196, 0.68331734, 0.0471555 , 0.94868537],
       [0.96833686, 0.19313494, 0.11765464, 0.13561539],
       [0.96998806, 0.50634506, 0.02096006, 0.32375073],
       [0.30732541, 0.10588319, 0.72021475, 0.07767541]])
```

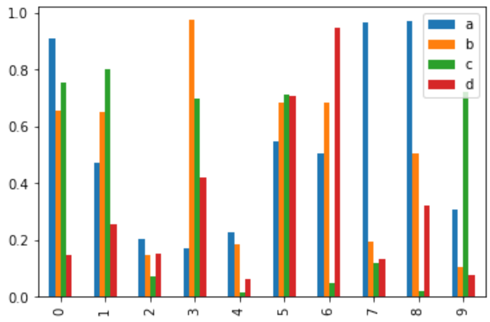
```
df = pd.DataFrame(x, columns=['a', 'b', 'c', 'd'])
df
```

	a	b	c	d
0	0.910549	0.656684	0.753475	0.148877
1	0.473966	0.651996	0.800876	0.256137
2	0.205160	0.149912	0.074546	0.150303
3	0.171023	0.974057	0.695809	0.418983
4	0.226547	0.184900	0.014825	0.064784
5	0.547321	0.684849	0.712227	0.705378
6	0.503042	0.683317	0.047155	0.948685
7	0.968337	0.193135	0.117655	0.135615
8	0.969988	0.506345	0.020960	0.323751
9	0.307325	0.105883	0.720215	0.077675

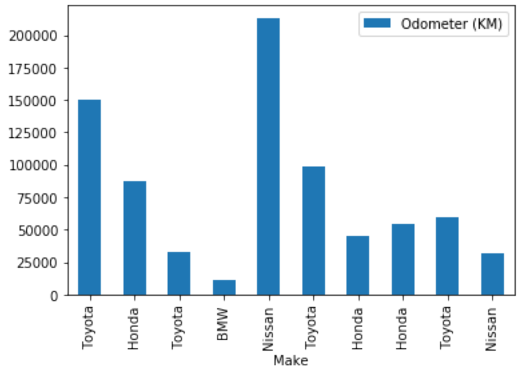
```
df.plot.bar();
```




```
# Can do the same thing with 'kind' keyword
df.plot(kind='bar');
```

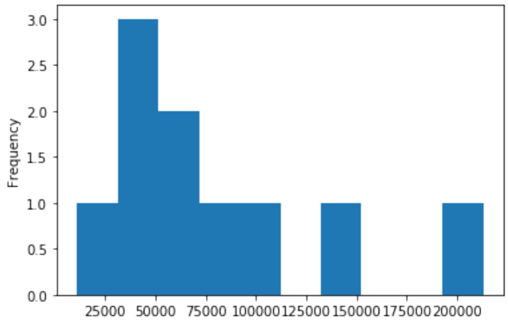



```
car_sales.plot(x='Make', y='Odometer (KM)', kind='bar');
```

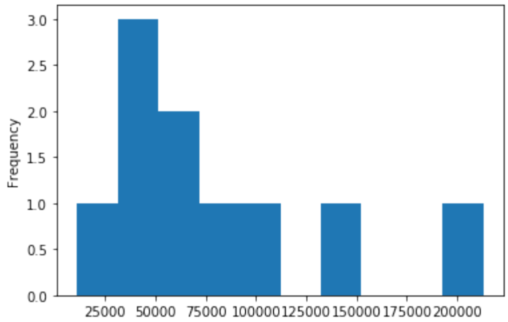


Histograms

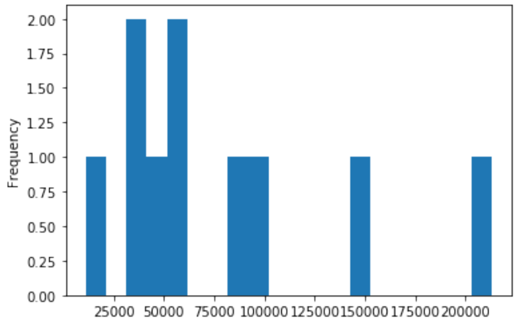
```
car_sales["Odometer (KM)"].plot.hist();
```



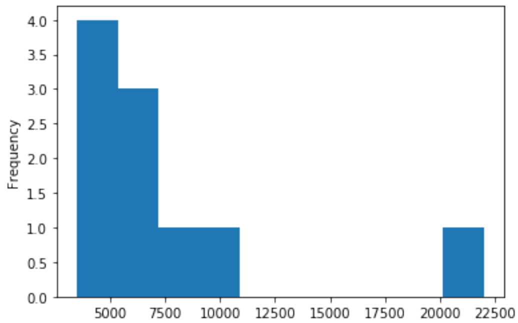

```
car_sales["Odometer (KM)"].plot(kind="hist");
```




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



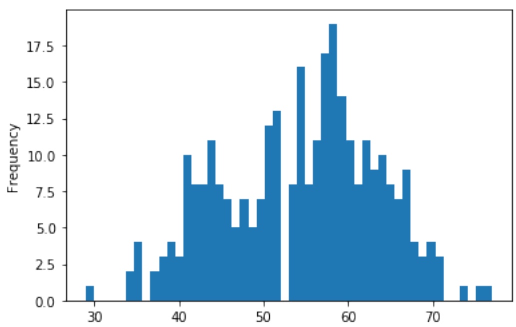

```
car_sales["Price"].plot.hist(bins=10);
```




```
# Let's try with another dataset
heart_disease = pd.read_csv("../data/heart-disease.csv")
heart_disease.head()
```

	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

```
heart_disease["age"].plot.hist(bins=50);
```



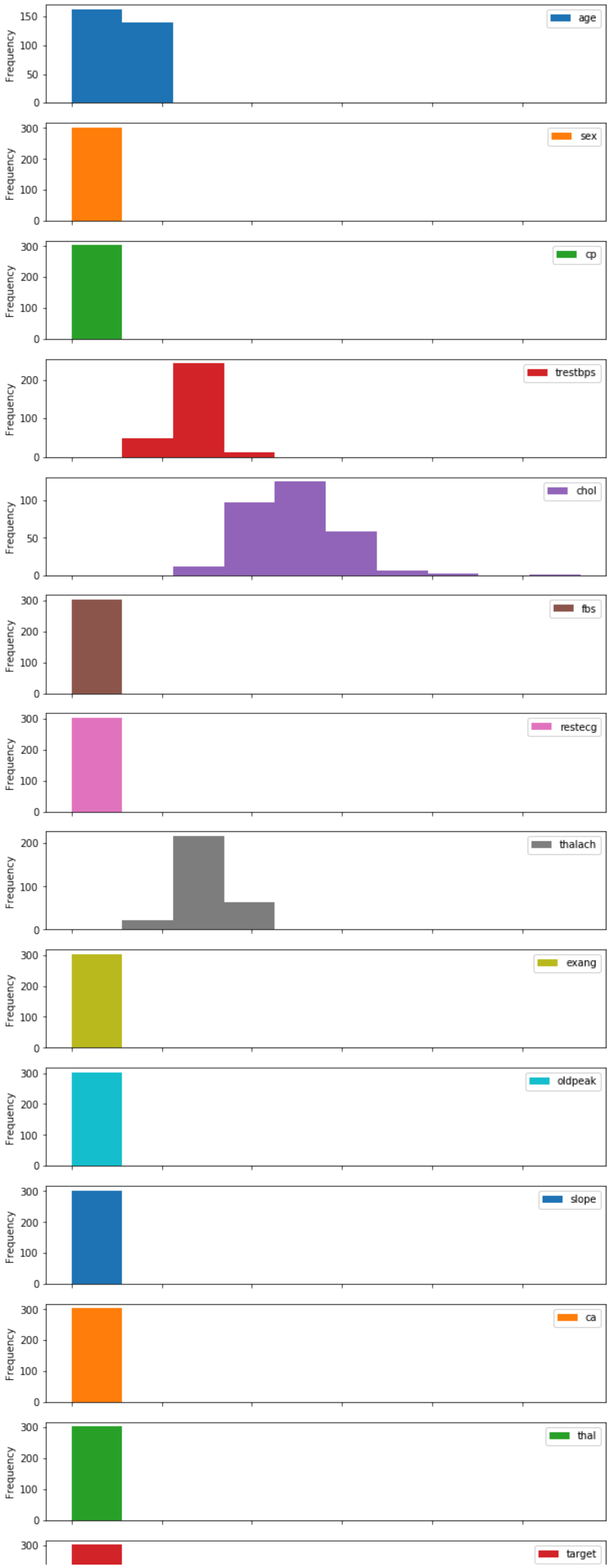
Subplots

- Concept
- DataFrame

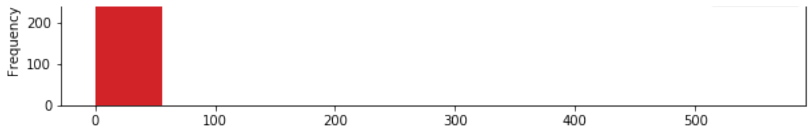
```
heart_disease.head()
```

	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

```
heart_disease.plot.hist(figsize=(10, 30), subplots=True);
```



Frequency
age
sex
cp
trestbps
chol
fbs
restecg
thalach
exang
oldpeak
slope
ca
thal
target
0
50
100
150
200
300



4. Plotting with pandas using the OO method

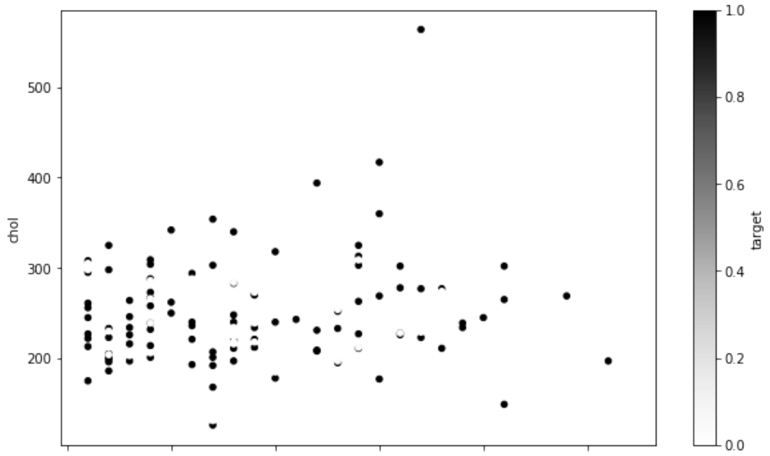
For more complicated plots, you'll want to use the OO method.

```
# Perform data analysis on patients over 50
over_50 = heart_disease[heart_disease["age"] > 50]
over_50
```

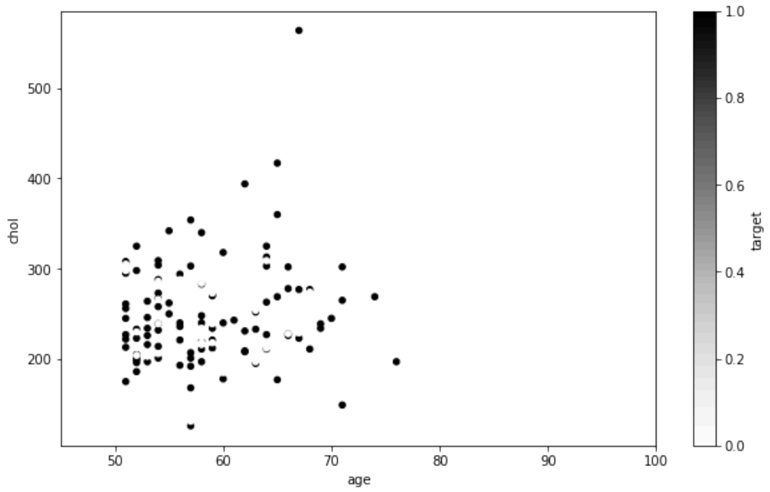
	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
5	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1
6	56	0	1	140	294	0	0	153	0	1.3	1	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
297	59	1	0	164	176	1	0	90	0	1.0	1	2	1	0
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

208 rows × 14 columns

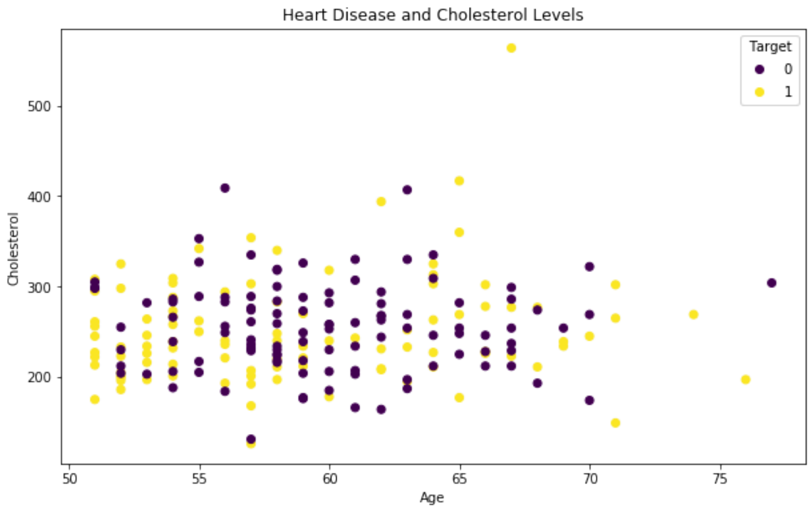
```
over_50.plot(kind='scatter',
             x='age',
             y='chol',
             c='target',
             figsize=(10, 6));
```




```
fig, ax = plt.subplots(figsize=(10, 6))
over_50.plot(kind='scatter',
             x="age",
             y="chol",
             c='target',
             ax=ax);
ax.set_xlim([45, 100]);
```



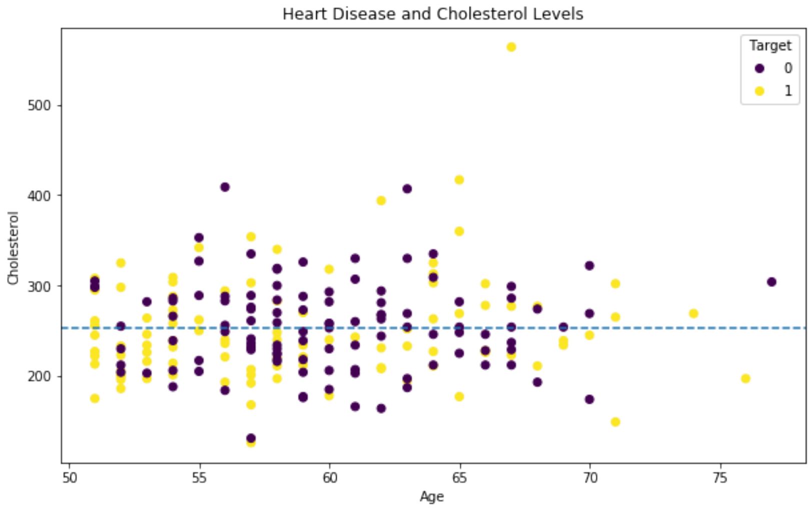

```
# Make a bit more of a complicated plot

# Create the plot
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the data
scatter = ax.scatter(over_50["age"],
                     over_50["chol"],
                     c=over_50["target"])

# Customize the plot
ax.set(title="Heart Disease and Cholesterol Levels",
       xlabel="Age",
       ylabel="Cholesterol");
ax.legend(*scatter.legend_elements(), title="Target");
```



What if we wanted a horizontal line going across with the mean of `heart_disease["chol"]`?

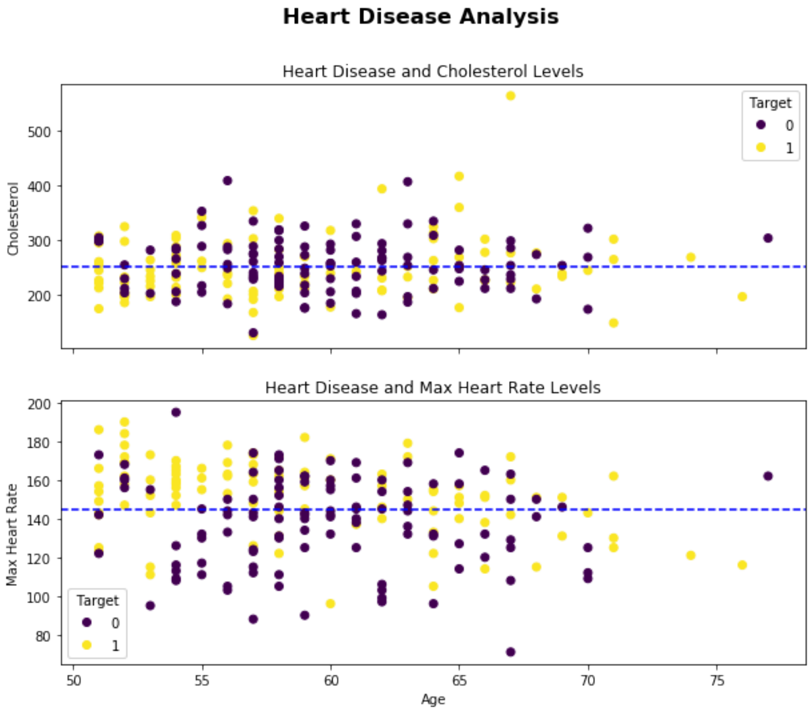
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.axhline.html

```
# Make a bit more of a complicated plot

# Create the plot
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the data
scatter = ax.scatter(over_50["age"],
                     over_50["chol"],
                     c=over_50["target"])

# Customize the plot
ax.set(title="Heart Disease and Cholesterol Levels",
       xlabel="Age",
       ylabel="Cholesterol");
ax.legend(*scatter.legend_elements(), title="Target")

# Add a meanline
ax.axhline(over_50["chol"].mean(),
           linestyle="--");
```



Adding another plot to existing styled one

```
# Setup plot (2 rows, 1 column)
fig, (ax0, ax1) = plt.subplots(nrows=2, # 2 rows
                               ncols=1,
                               sharex=True,
                               figsize=(10, 8))

# Add data for ax0
scatter = ax0.scatter(over_50["age"],
                      over_50["chol"],
                      c=over_50["target"])
# Customize ax0
ax0.set(title="Heart Disease and Cholesterol Levels",
        ylabel="Cholesterol")
ax0.legend(*scatter.legend_elements(), title="Target")

# Setup a mean line
ax0.axhline(y=over_50["chol"].mean(),
            color='b',
            linestyle='--',
            label="Average")


# Add data for ax1
scatter = ax1.scatter(over_50["age"],
                      over_50["thalach"],
                      c=over_50["target"])

# Customize ax1
ax1.set(title="Heart Disease and Max Heart Rate Levels",
        xlabel="Age",
        ylabel="Max Heart Rate")
ax1.legend(*scatter.legend_elements(), title="Target")

# Setup a mean line
ax1.axhline(y=over_50["thalach"].mean(),
            color='b',
            linestyle='--',
            label="Average")

# Title the figure
fig.suptitle('Heart Disease Analysis', fontsize=16, fontweight='bold');
```



5. Customizing your plots

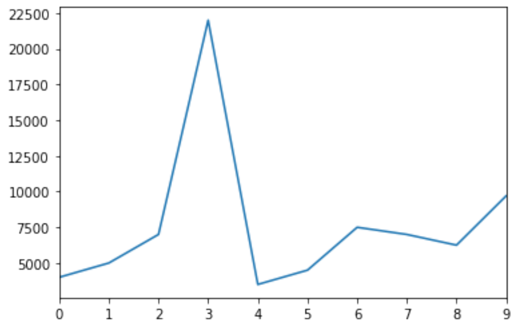
- `limits (xlim, ylim)`, `colors`, `styles`, `legends`

Style

```
plt.style.available
```

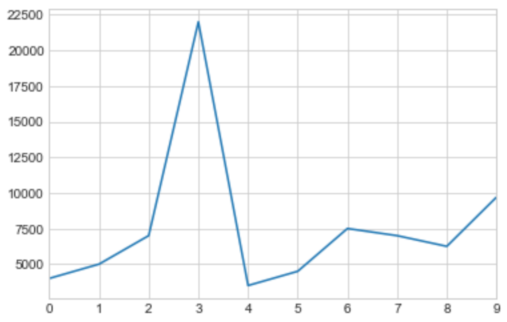
```
['seaborn-dark',
 'seaborn-darkgrid',
 'seaborn-ticks',
 'fivethirtyeight',
 'seaborn-whitegrid',
 'classic',
 '_classic_test',
 'fast',
 'seaborn-talk',
 'seaborn-dark-palette',
 'seaborn-bright',
 'seaborn-pastel',
 'grayscale',
 'seaborn-notebook',
 'ggplot',
 'seaborn-colorblind',
 'seaborn-muted',
 'seaborn',
 'Solarize_Light2',
 'seaborn-paper',
 'bmh',
 'tableau-colorblind10',
 'seaborn-white',
 'dark_background',
 'seaborn-poster',
 'seaborn-deep']
```

```
# Plot before changing style
car_sales["Price"].plot();
```



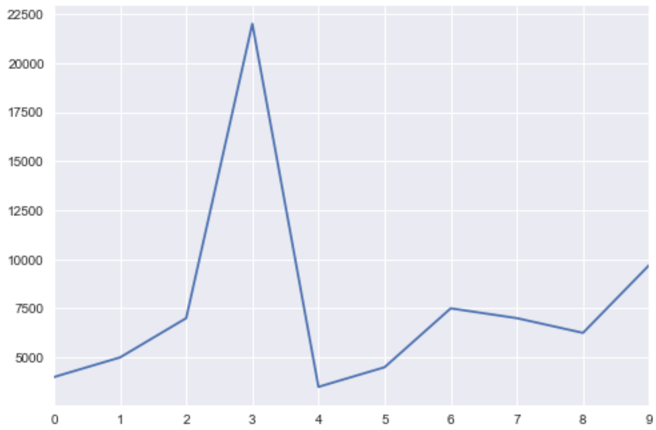

```
# Change the style...
plt.style.use('seaborn-whitegrid')
```

```
car_sales["Price"].plot();
```

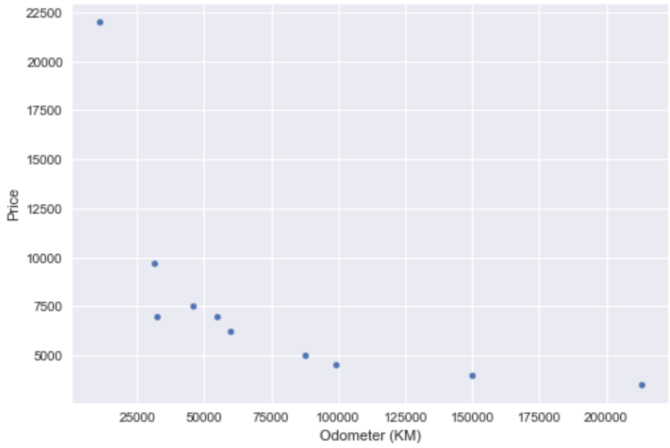



```
plt.style.use('seaborn')
```

```
car_sales["Price"].plot();
```

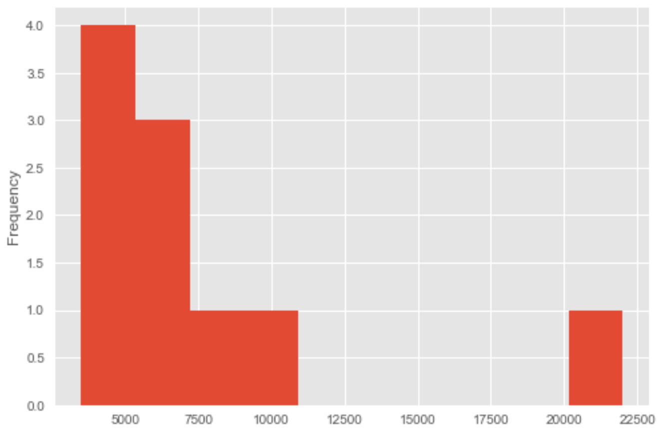



```
car_sales.plot(x="Odometer (KM)", y="Price", kind="scatter");
```




```
plt.style.use('ggplot')
```

```
car_sales["Price"].plot.hist();
```



Changing the title, legend, axes

```
x = np.random.randn(10, 4)
x
```

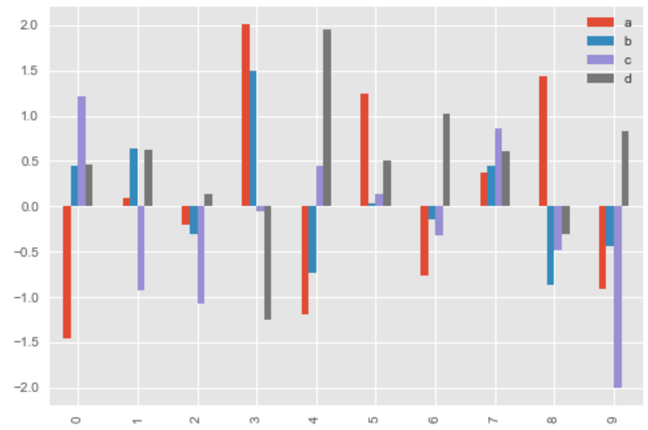
```
array([[-1.45604975,  0.44398039,  1.21617191,  0.46778121],
       [ 0.09043707,  0.64565222, -0.92772261,  0.63044677],
       [-0.20260212, -0.30685306, -1.07970088,  0.13664664],
       [ 2.01577535,  1.49857223, -0.05013591, -1.24773112],
       [-1.1872596 , -0.73286209,  0.45447678,  1.9601397 ],
       [ 1.24279567,  0.03839697,  0.1417006 ,  0.50332953],
       [-0.76513327, -0.14311738, -0.32238378,  1.02932238],
       [ 0.37193522,  0.44785763,  0.85386339,  0.60622919],
       [ 1.44272823, -0.86638843, -0.48638364, -0.30357948],
       [-0.90626553, -0.44139532, -1.99812976,  0.83367383]])
```

```
df = pd.DataFrame(x, columns=['a', 'b', 'c', 'd'])
df
```

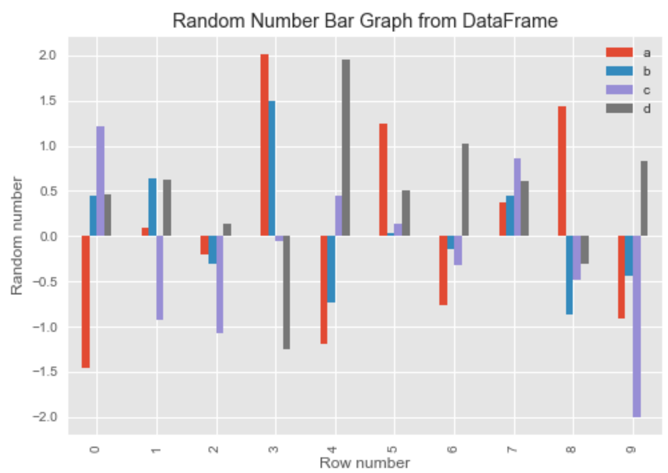
	a	b	c	d
0	-1.456050	0.443980	1.216172	0.467781
1	0.090437	0.645652	-0.927723	0.630447
2	-0.202602	-0.306853	-1.079701	0.136647
3	2.015775	1.498572	-0.050136	-1.247731
4	-1.187260	-0.732862	0.454477	1.960140
5	1.242796	0.038397	0.141701	0.503330
6	-0.765133	-0.143117	-0.322384	1.029322
7	0.371935	0.447858	0.853863	0.606229
8	1.442728	-0.866388	-0.486384	-0.303579
9	-0.906266	-0.441395	-1.998130	0.833674

```
ax = df.plot(kind='bar')
type(ax)
```

```
matplotlib.axes._subplots.AxesSubplot
```



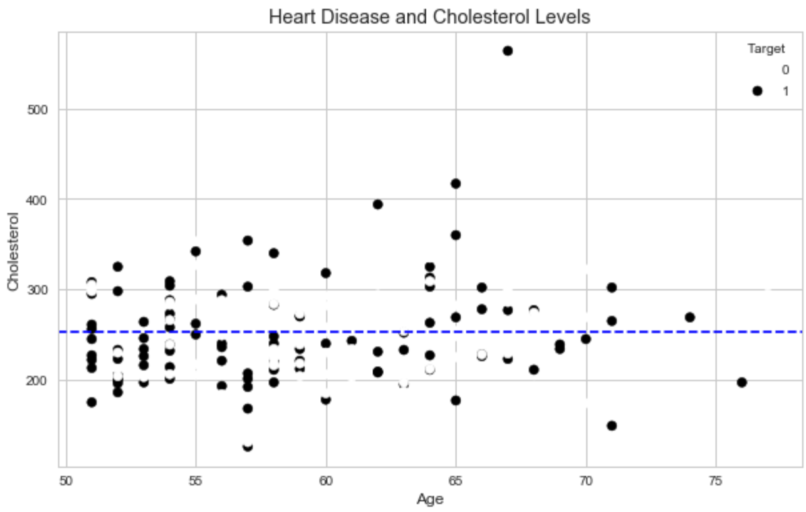

```
ax = df.plot(kind='bar')
ax.set(title="Random Number Bar Graph from DataFrame",
       xlabel="Row number",
       ylabel="Random number")
ax.legend().set_visible(True)
```



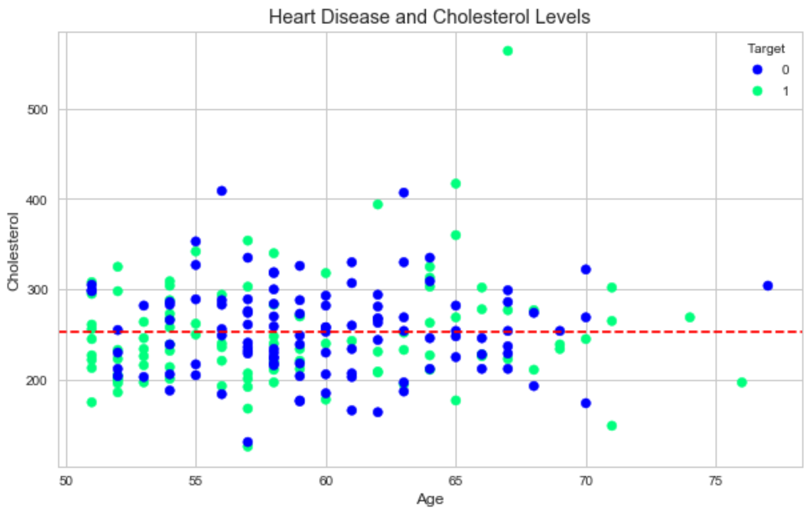
Changing the cmap

```
plt.style.use('seaborn-whitegrid')
```

```
# No cmap change
fig, ax = plt.subplots(figsize=(10, 6))
scatter = ax.scatter(over_50["age"],
                     over_50["chol"],
                     c=over_50["target"])
ax.set(title="Heart Disease and Cholesterol Levels",
       xlabel="Age",
       ylabel="Cholesterol");
ax.axhline(y=over_50["chol"].mean(),
           c='b',
           linestyle='--',
           label="Average");
ax.legend(*scatter.legend_elements(), title="Target");
```

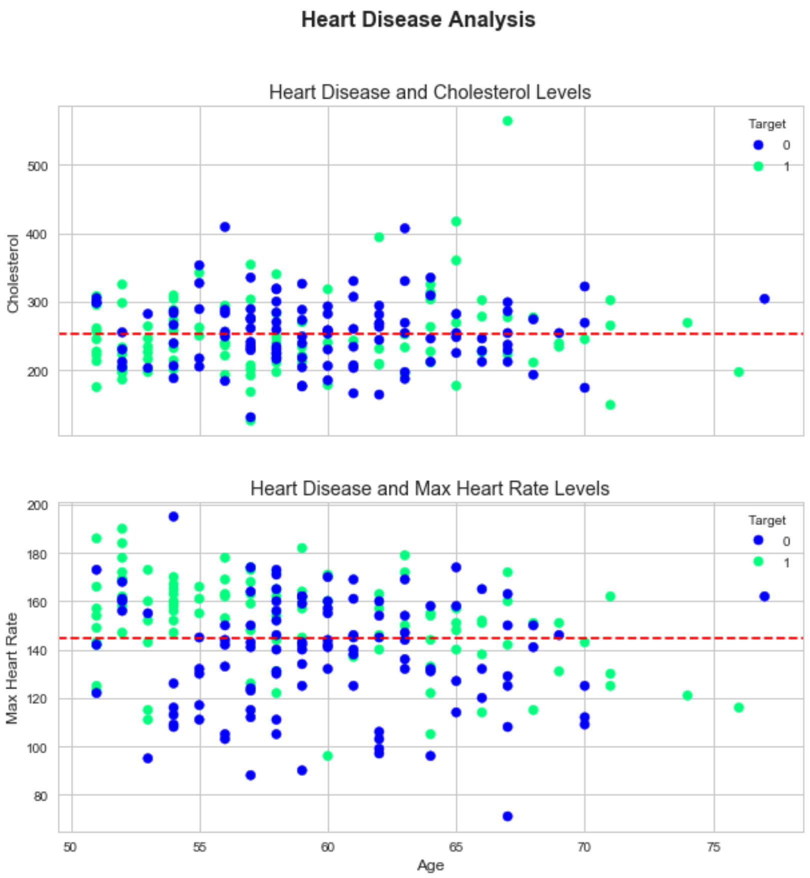



```
# Change cmap and horizontal line to be a different colour
fig, ax = plt.subplots(figsize=(10, 6))
scatter = ax.scatter(over_50["age"],
                     over_50["chol"],
                     c=over_50["target"],
                     cmap="winter")
ax.set(title="Heart Disease and Cholesterol Levels",
       xlabel="Age",
       ylabel="Cholesterol")
ax.axhline(y=over_50["chol"].mean(),
           color='r',
           linestyle='--',
           label="Average");
ax.legend(*scatter.legend_elements(), title="Target");
```



Changing the xlim & ylim

```
## Before the change (we've had color updates)

fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(10, 10))
scatter = ax0.scatter(over_50["age"],
                      over_50["chol"],
                      c=over_50["target"],
                      cmap='winter')
ax0.set(title="Heart Disease and Cholesterol Levels",
        ylabel="Cholesterol")

# Setup a mean line
ax0.axhline(y=over_50["chol"].mean(),
            color='r',
            linestyle='--',
            label="Average");
ax0.legend(*scatter.legend_elements(), title="Target")

# Axis 1, 1 (row 1, column 1)
scatter = ax1.scatter(over_50["age"],
                      over_50["thalach"],
                      c=over_50["target"],
                      cmap='winter')
ax1.set(title="Heart Disease and Max Heart Rate Levels",
        xlabel="Age",
        ylabel="Max Heart Rate")

# Setup a mean line
ax1.axhline(y=over_50["thalach"].mean(),
            color='r',
            linestyle='--',
            label="Average");
ax1.legend(*scatter.legend_elements(), title="Target")

# Title the figure
fig.suptitle('Heart Disease Analysis', fontsize=16, fontweight='bold');
```



```python
## After adding in different x & y limitations

fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(10, 10))
scatter = ax0.scatter(over_50["age"],
                      over_50["chol"],
                      c=over_50["target"],
                      cmap='winter')
ax0.set(title="Heart Disease and Cholesterol Levels",
        ylabel="Cholesterol")

# Set the x axis
ax0.set_xlim([50, 80])

# Setup a mean line
ax0.axhline(y=over_50["chol"].mean(),
            color='r',
            linestyle='--',
            label="Average");
ax0.legend(*scatter.legend_elements(), title="Target")

# Axis 1, 1 (row 1, column 1)
scatter = ax1.scatter(over_50["age"],
                      over_50["thalach"],
                      c=over_50["target"],
                      cmap='winter')
ax1.set(title="Heart Disease and Max Heart Rate Levels",
        xlabel="Age",
        ylabel="Max Heart Rate")

# Set the y axis
ax1.set_ylim([60, 200])

# Setup a mean line
ax1.axhline(y=over_50["thalach"].mean(),
            color='r',
            linestyle='--',
            label="Average");
ax1.legend(*scatter.legend_elements(), title="Target")

# Title the figure
fig.suptitle('Heart Disease Analysis', fontsize=16, fontweight='bold');
```