3. Data Science Tools I: NumPy and Matplotlib

Overview of today

1. Quick recap

2. NumPy

- The need for NumPy
- 1D arrays
- 2D arrays
- Generating random numbers

3. Matplotlib

- Creating a figure
- Line plots
- Scatter plots
- Histograms

Recap: `input()`

- The `input()` function is used to gather text input from a user running your code.

```
answer = input('What is your favourite colour? ')

print('Your favourite colour is', answer)
```

```
What is your favourite colour? Green
Your favourite colour is Green
```

- Here, whatever the user types in will be saved as a string to the variable `answer`.

Recap: `if-else` conditions

- You can control the flow of your algorithm by using `if-else` conditions.

```
true_password = 'qwerty1234'
entered_password = input('Please enter your password: ')

if entered_password == true_password:
    print('Welcome')
else:
    print('Incorrect, try again')
```

```
Please enter your password: qwerty1234
Welcome
```

- There are many different possible conditions you could use. In general, any question you can ask that is either `True` or `False` should be possible to create.

Recap: Loops

- When you want to repeat some code many times, `for`-loops are often useful.

```
shopping_list = ['apples', 'pears', 'bananas', 'oranges']

for shopping_item in shopping_list:
    print(shopping_item)
```

```
apples
pears
bananas
oranges
```

Recap: functions

- In Python you can design your own functions

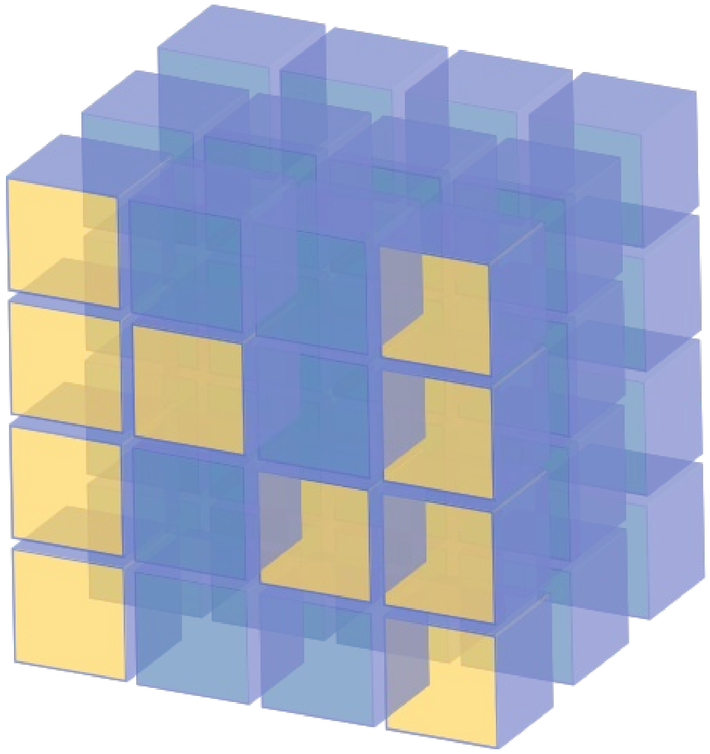
```
def my_function(number):
    return 5 + number ** 2

my_numbers = [1, 2, 3, 4, 5]

for number in my_numbers:
    print(my_function(number))
```

```
6
9
14
21
30
```

NumPy



NumPy

Definition and installation

- NumPy, which comes from *Numerical Python*, is a very widely used tool for mathematical programming in Python.
- Its an example of a *library*, also sometimes called a *package*. This means it does not come pre-installed when we download Python, but needs to be installed separately.

- In general, the way to install a python package is to go to the command line and type

```
pip install <package_name>
```

With numpy, this would be

```
pip install numpy
```

If you are using Python on your own computer, you need to make sure you have installed it, however it is already installed on the server (more information here).

How to begin using NumPy

- When we are using any external library in Python, it must be "imported". This is done with the special `import` statement

```
import numpy

print(numpy.__version__)
```

```
1.19.1
```

- This code simply imports numpy and then prints the version number.
- We now will have access to all the special functions and variables that numpy provides by writing `numpy.whatever()`.

What does it do?

- The core capability of NumPy is performing fast operations on lots of numbers at once. This is essential for many applications such as data analysis and mathematical/scientific computing.

- We do this via a new data type: the NumPy array

```
import numpy

my_array = numpy.array([1, 2, 3, 4])
print(my_array)
```

```
[1 2 3 4]
```

- Congratulations: you've just created a NumPy array!

NumPy arrays

- NumPy arrays are somewhat similar to a list of numbers. For example, they have a length

```
print(len(my_array))
```

```
4
```

- Just like lists, you can iterate through a NumPy array

```
for num in my_array:
    print(num)
```

```
1
2
3
4
```

NumPy arrays

- You can also index NumPy arrays just like lists

```
print(my_array[0])
```

```
1
```

- However, they do have some different behaviours. For example, you cannot use the `.append()` method on an array.

```
my_array.append(5) # this will cause an error!
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

NumPy arrays

- The real power comes from being able to perform operations on the whole array all at once. Say we have a list of numbers and we want to multiply every number in the list by two.

```
my_list = [1, 2, 3, 4]
new_list = []

for num in my_list:
    new_list.append(num * 2)

print(new_list)
```

```
[2, 4, 6, 8]
```

- This is slow and cumbersome!

NumPy arrays

- By contrast, with NumPy arrays, we can multiply the *whole array by 2*. This means each number will get multiplied by 2.

```
my_array = numpy.array([1, 2, 3, 4])
new_array = my_array * 2

print(new_array)
```

```
[2 4 6 8]
```

- Not only is this faster to write out in code, but the computation is *much faster*. When we are dealing with tiny arrays like this, the difference will be hard to notice. But when dealing with very large arrays (think millions or billions of numbers) the difference is *very noticeable*.

Operations on arrays

- NumPy allows all sorts of mathematical operations to be performed on arrays.

```
my_array = numpy.array([1, 2, 3, 4])

print(my_array * 2)     # multiply array by 2
print(my_array / 5)     # divide array by 5
print(my_array + 10)    # add 10 to array
print(my_array ** 2)    # square array
```

```
[2 4 6 8]
[0.2 0.4 0.6 0.8]
[11 12 13 14]
[ 1  4  9 16]
```

Operations on arrays

- Similarly, when two NumPy arrays of the same length are combined via a maths operation, the effect is to perform the operation on each element at once.

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 4, 6, 8])

print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
[ 6  6  9 12]
[-4 -2 -3 -4]
[ 5  8 18 32]
[0.2 0.5 0.5 0.5]
```

NumPy functions

- NumPy also comes packed with many functions which can be applied to arrays.

```
my_array = numpy.array([1, 2, 3, 4])

print(numpy.sqrt(my_array))    # take square root of array
print(numpy.sin(my_array))     # apply sin(x) to the array
print(numpy.exp(my_array))     # apply e^x to the array
print(numpy.log(my_array))     # apply log(x) to the array
```

```
[ 1.00000000  1.41421356  1.73205081  2.000000000 ]
[ 0.84147098  0.90929743  0.141120010 -0.75680250 ]
[ 2.71828183  7.38905610  20.08553692 54.59815003 ]
[ 0.00000000  0.69314718  1.098612290 1.386294360 ]
```

- All these functions can also be applied on single numbers, as well as arrays.

Operations on arrays: NumPy functions

- There are also a number of functions that collapse arrays into a single number

```
my_array = numpy.array([1, 2, 3, 4])

print(numpy.min(my_array))     # find the smallest number in the array
print(numpy.max(my_array))     # find the largest number in the array
print(numpy.sum(my_array))     # add all numbers in the array
print(numpy.mean(my_array))    # find the average of the array
```

```
1
4
10
2.5
```

- These operations can only be applied to NumPy arrays and not individual numbers.

`zeros()` and `ones()`

- Two useful NumPy functions are `numpy.zeros()` and `numpy.ones()`.

```
a = numpy.zeros(10)
b = numpy.ones(8)

print(a)
print(b)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 1.]
```

- They give you an array full of zeros and ones respectively.

`linspace()`

- Another very useful NumPy function is `numpy.linspace()`. It is used to create a sequence of evenly spaced numbers in some interval.

- The function expects 3 arguments: the interval start, the interval end, and the number of points in the interval.

```
x = numpy.linspace(0, 1, 11)  # start=0, stop=1, num_points=11

print(x)
```

```
[0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0]
```

2D Arrays

- So far we have seen only 1D arrays, that is, arrays that are just a sequence/line of numbers. However, NumPy also allows for 2D arrays, or rectangles of numbers.

- We can create 2D arrays using a list of lists.

```
my_2d_array = np.array([[1, 2, 3], [4, 5, 6]])
print(my_2d_array)
```

```
[[1 2 3]
 [4 5 6]]
```

2D arrays: `ones()` and `zeros()`

- We can also create 2D arrays using `numpy.ones()` and `numpy.zeros()`.

```
A = numpy.zeros((3, 4))
B = numpy.ones((2, 5))

print(A)
print(B)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Indexing 2D arrays

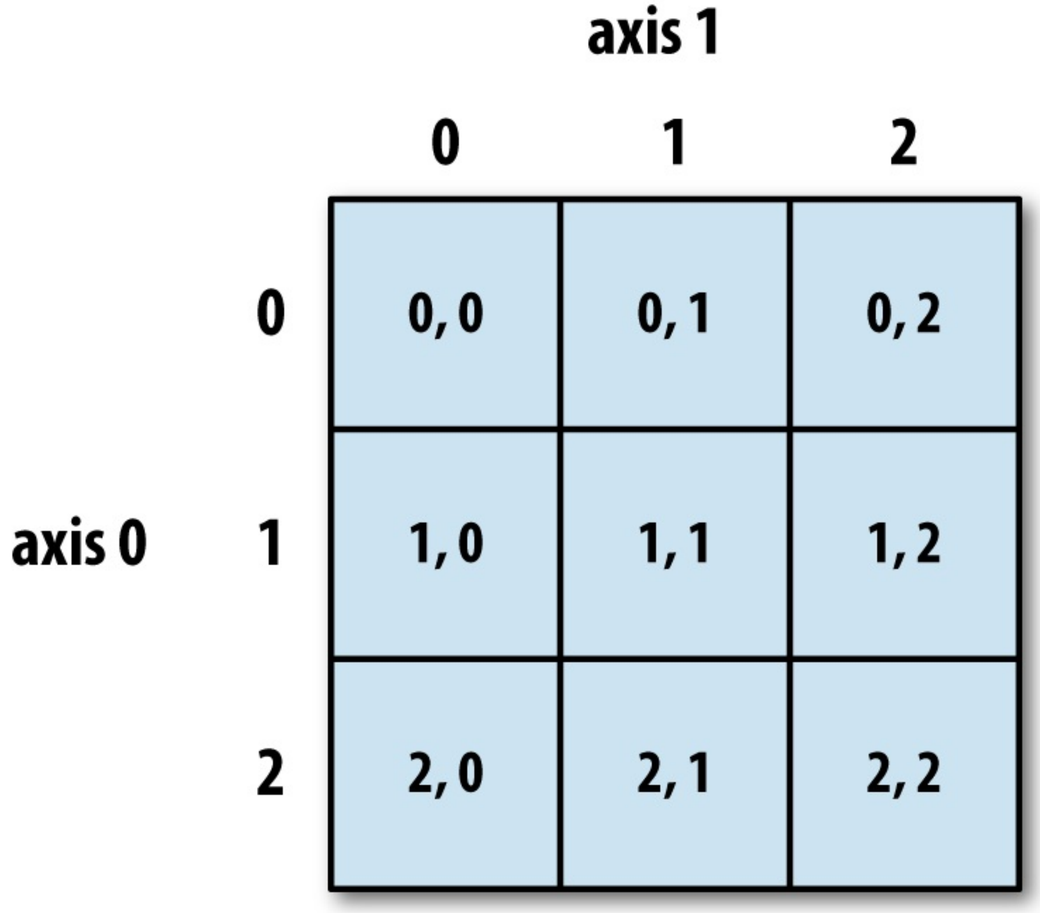
- Indexing 2D arrays is similar to indexing 1D arrays, expect now we have to worry about both row-index and the column-index.

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

print(A[0, 0])
print(A[0, 1])
print(A[2, 0])
print(A[2, 2])
```

```
1
2
4
9
```

Indexing 2D arrays



Indexing 2D arrays

- We can select individual columns...

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

print(A[:, 0])
```

```
[1 4 7]
```

- Or individual rows ...

```
print(A[1, :])
```

```
[4 5 6]
```

Indexing 2D arrays

- In the same way, individual elements, columns, rows etc can be set in place.

```
A = numpy.zeros((4, 4))
A[:, 1] = 1
A[:, 2] = 2
A[:, 3] = 3
print(A)
```

```
[[0. 1. 2. 3.]
 [0. 1. 2. 3.]
 [0. 1. 2. 3.]
 [0. 1. 2. 3.]]
```

Generating random numbers

- One sub-module within NumPy is called `random` where random numbers can be generated quickly.

- The code below creates a random number between 0 and 1 with uniform probability.

```
rand_num = numpy.random.uniform()
print(rand_num)
```

```
0.2896624971278092
```

Generating random numbers

- This function also has three optional arguments, `start`, `stop` and `size`.

```
rand_nums = numpy.random.uniform(4, 5, (5, 5))
print(rand_nums)
```

```
[[4.45772478 4.3738132  4.02164613 4.64031678 4.15628723]
 [4.71902929 4.2951666  4.20125751 4.49981043 4.99134346]
 [4.53251653 4.02612465 4.92644378 4.09518438 4.67688315]
 [4.48668533 4.99436289 4.88419433 4.66758867 4.59234259]
 [4.17023685 4.46759662 4.18416802 4.89479636 4.99490762]]
```

Generating random numbers

- There are also a number of other random functions that behave similarly. This one creates random whole numbers (integers)

```
rand_ints = numpy.random.randint(0, 3, 10)
print(rand_ints)
```

```
[0 1 1 2 2 0 0 1 2 2]
```

- Let's use this to create a function called `roll_dice` which simulates rolling a dice.

```
def roll_dice(n_times):
    return numpy.random.randint(1, 7, n_times)

print(roll_dice(5))
print(roll_dice(10))
```

```
[3 2 3 1 6]
[6 1 3 4 2 5 6 3 5 4]
```

Generating random numbers

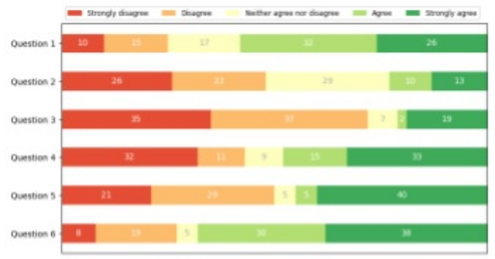
- Here's another one which creates random numbers drawn from a normal (Gaussian) distribution.

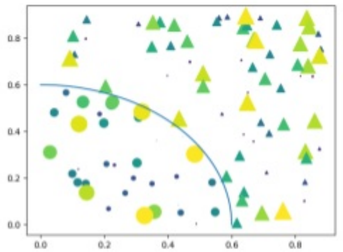
- It takes three parameters, the mean, the standard deviation and the size.

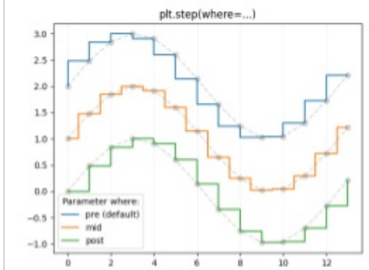
```
normal_nums = numpy.random.normal(0, 5, 10)
print(normal_nums)
```

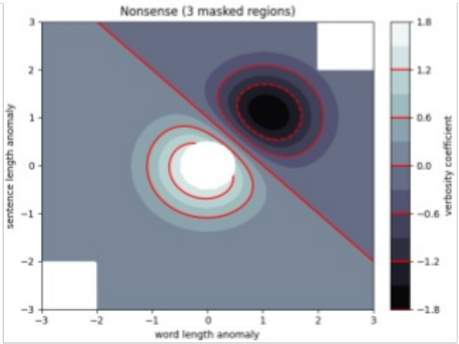
```
[ 1.38368018  0.73685003 -6.77452176  2.19092933 13.2942945  -3.83038345  0.72156331
-0.54531725  8.57530468  4.33849341]
```

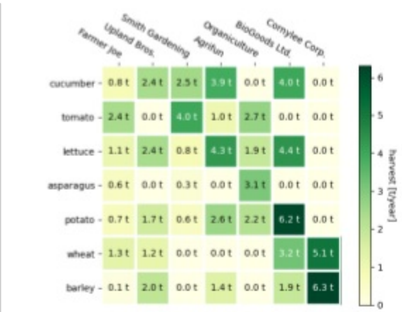
New topic: Matplotlib

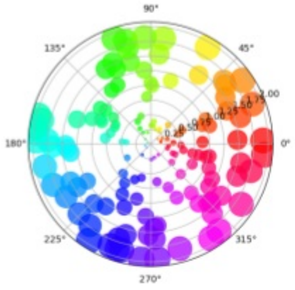


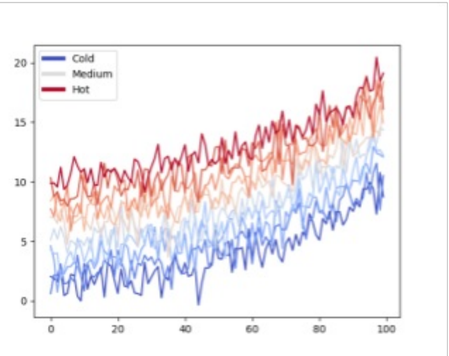


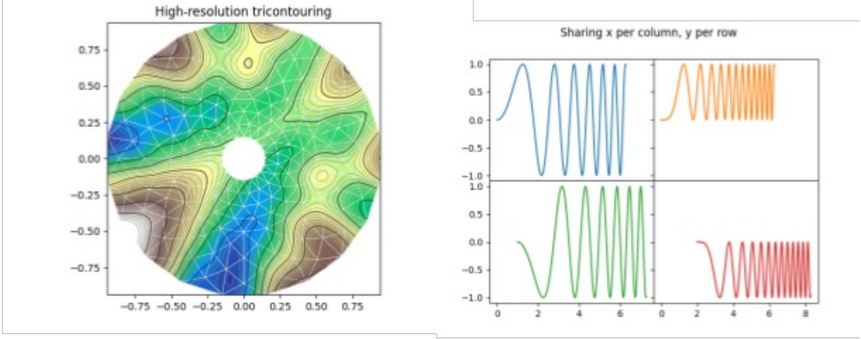












Matplotlib

- Matplotlib, like NumPy is a Python library/package. It is used to create plots and data visualisations.

- The typical way to get access to all the Matplotlib functions is to run the following `import` statement.

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

- All of the matplotlib magic will then be accessible through `plt.whatever`

- In Jupyter, we can also make all plots interactive (pan, zoom etc) by running

```
%matplotlib notebook
```

- Note that the above line isn't typical Python code - this is a Jupyter-specific command.

Matplotlib

- When writing code for Matplotlib, the philosophy is a bit different to NumPy and what we've seen so far.

- The idea is to write "commands" which make certain things appear on the plot. For example "plot this line", "add this label" etc.

- When creating a new graph, the first command is almost always:

```
plt.figure()
```

- This creates a new figure, which is a blank canvas for us to plot onto.

Line plots

- One of the most useful graphs we can create with Matpllotlib is the line plot function: `plt.plot()`.

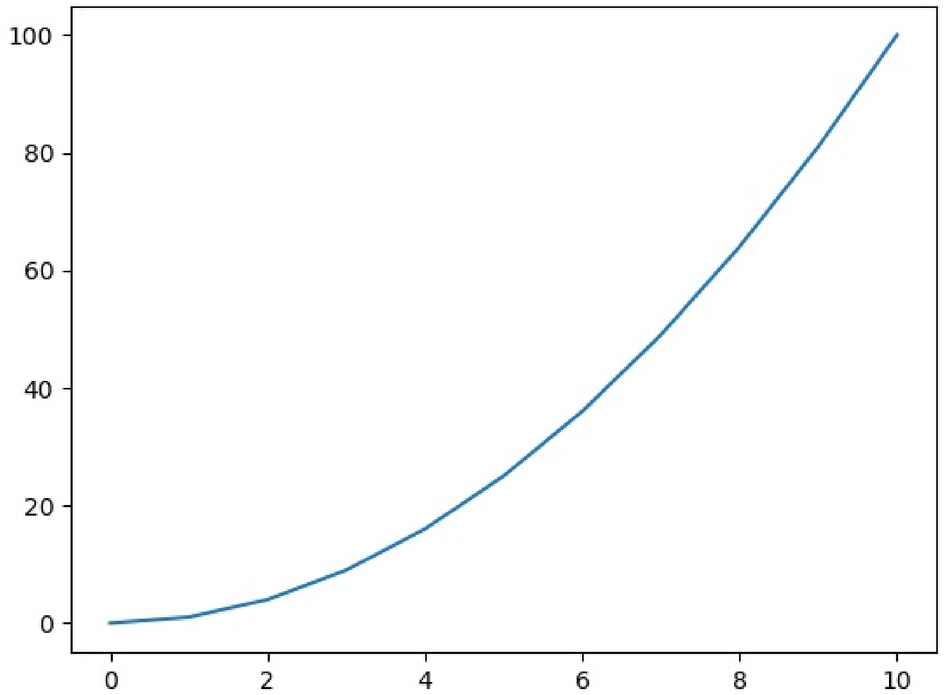
- This function needs at least two inputs, the x-coordinates and the y-coordinates. Let's start by creating a NumPy array of x-points, and a NumPy array of y-points.

```
x = numpy.linspace(0, 10, 11)
y = x ** 2    # y = x^2
print(x)
print(y)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 0.   1.   4.   9.  16.  25.  36.  49.  64.  81. 100.]
```

Line plots

```
plt.figure()
plt.plot(x, y)
```



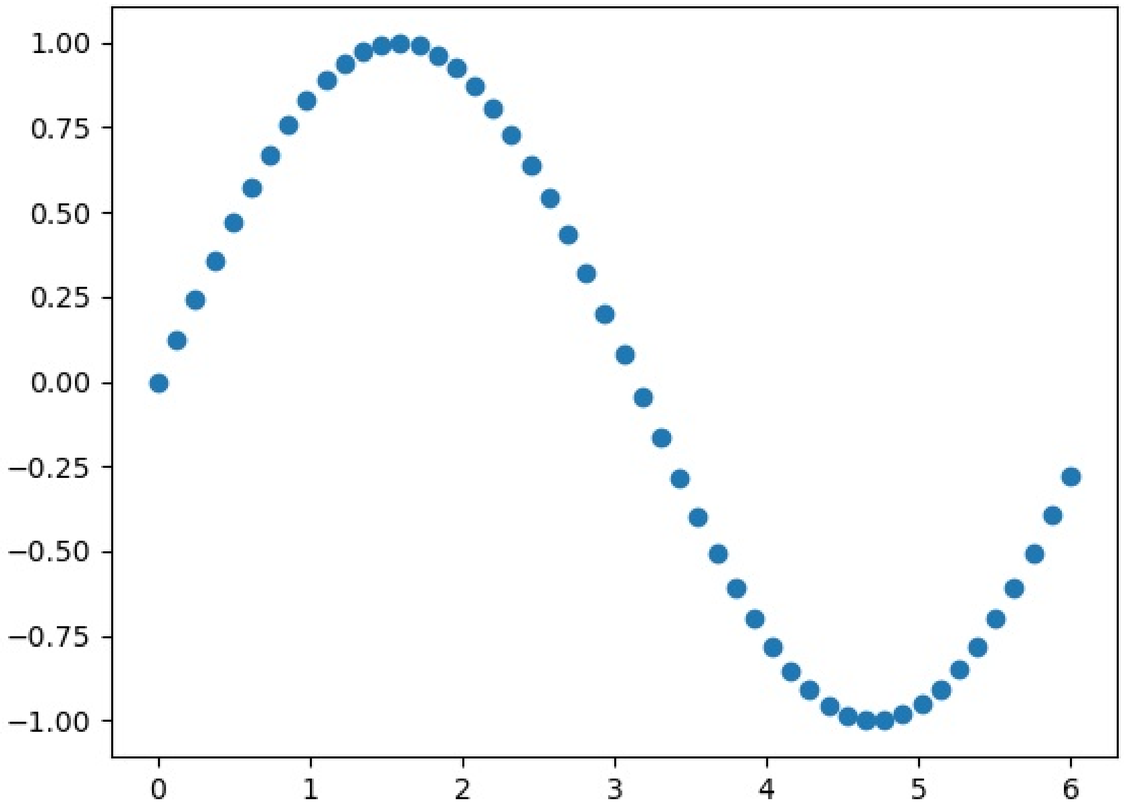
Scatter plots

- A very similar function is `plt.scatter()`. This accepts the same arguments but scatters each point, rather than "joining the dots".

```
x = numpy.linspace(0, 6, 100)
y = numpy.sin(x)

plt.figure()
plt.scatter(x, y)
```

Scatter plots



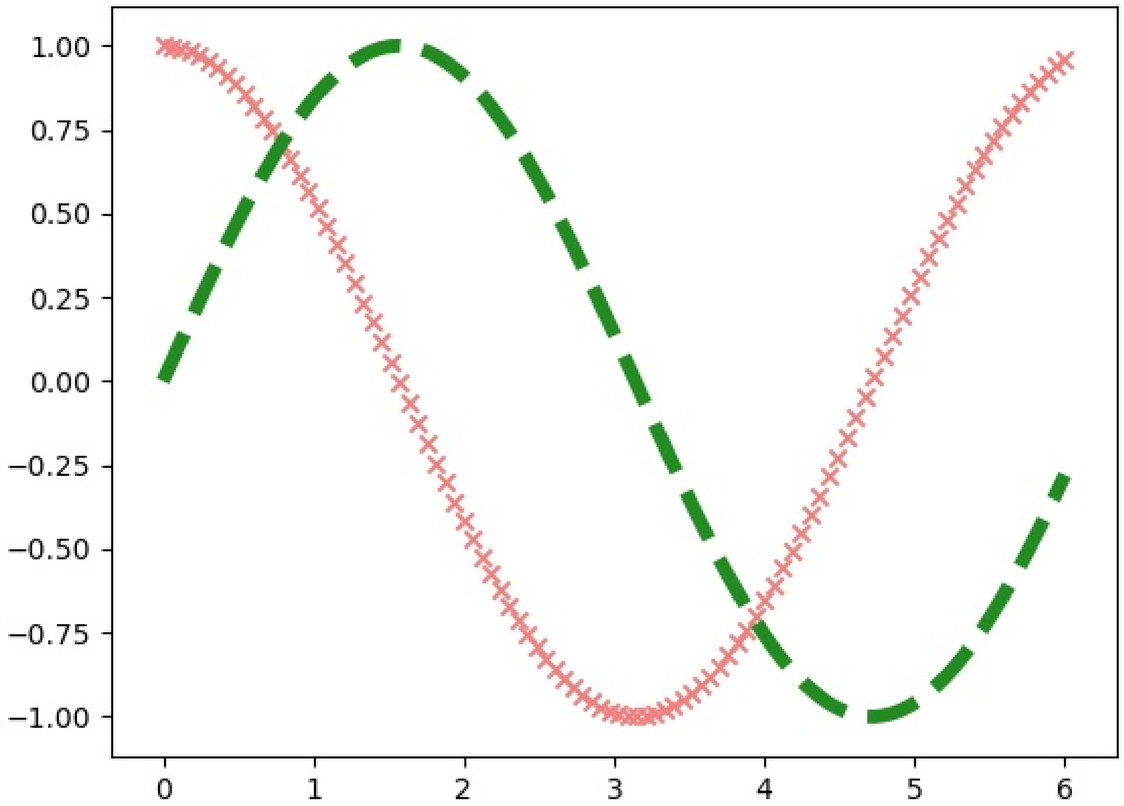
Plotting: optional arguments

- When creating line plots or scatter plots there are also a number of optional arguments to specify things like colour, transparency, size, etc .

```
x = numpy.linspace(0, 6, 100)
y1 = numpy.sin(x)
y2 = numpy.cos(x)

plt.figure()
plt.plot(x, y1, color='forestgreen', lw=5, ls='--')
plt.scatter(x, y2, color='lightcoral', marker='x')
```

Plotting: optional arguments



- For the full list of possible arguments, look at the documentation.

Available colours

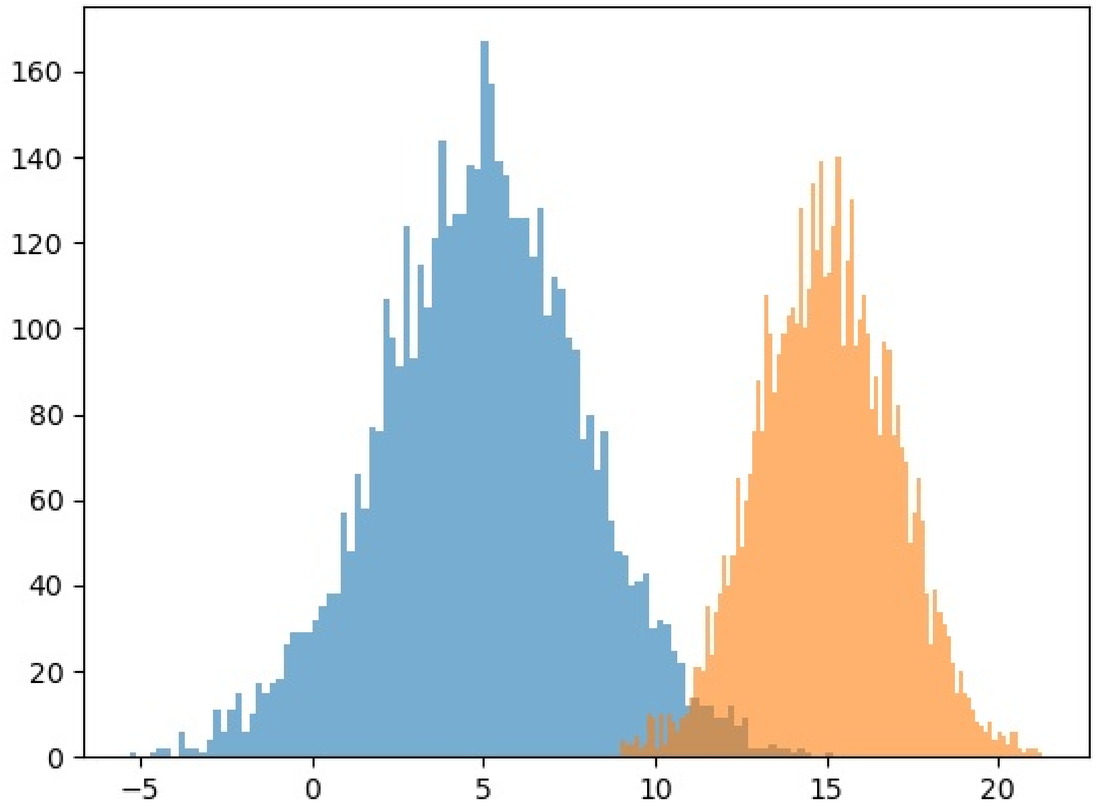
Histograms

- Another useful plot is the histogram. This requires only a single argument, which is just an array of values. Another useful optional argument in `bins` which tells MPL how many divisions to make.

```
values1 = numpy.random.normal(5, 3, 5000)
values2 = numpy.random.normal(15, 2, 5000)

plt.figure()
plt.hist(values1, bins=100, alpha=0.6)
plt.hist(values2, bins=100, alpha=0.6)
```

Histograms



Adding labels

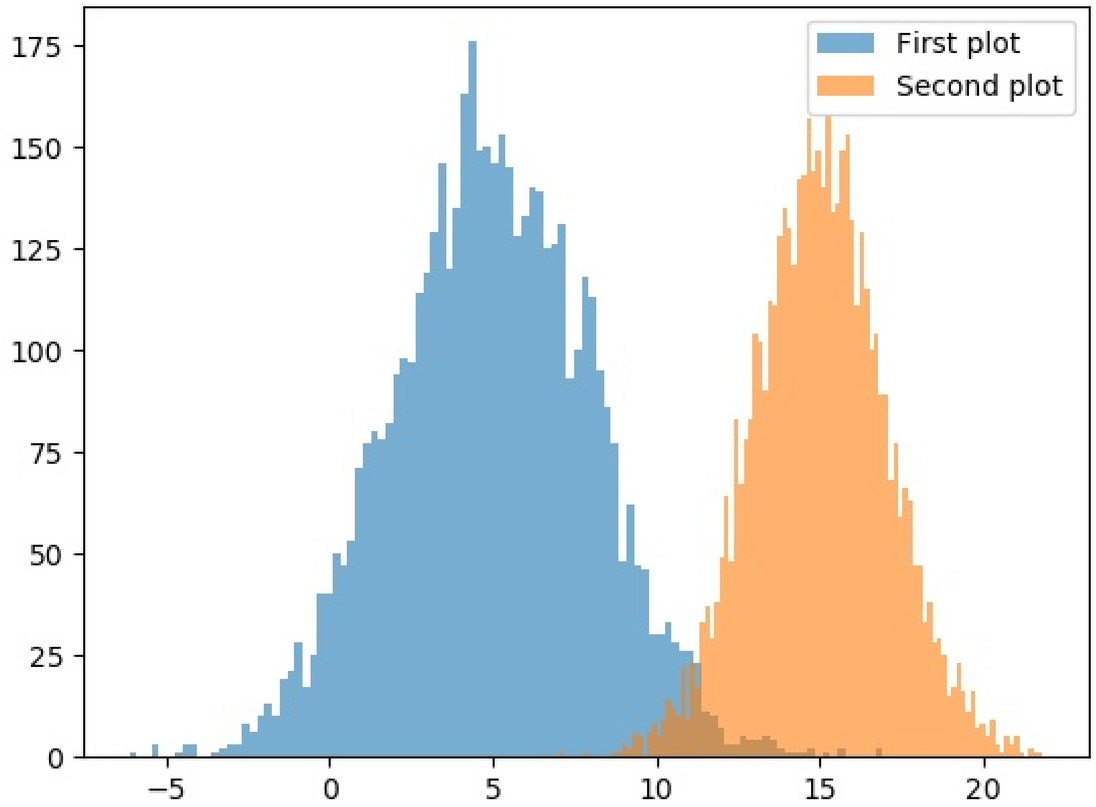
- When you have multiple things on one figure, it's useful to provide a label.

- This has to be done in conjunction with calling `plt.legend()`

```
values1 = numpy.random.normal(5, 3, 5000)
values2 = numpy.random.normal(15, 2, 5000)

plt.figure()
plt.hist(values1, bins=100, alpha=0.6, label='First plot')
plt.hist(values2, bins=100, alpha=0.6, label='Second plot')
plt.legend()
```

Adding labels



Adding a title and axis labels

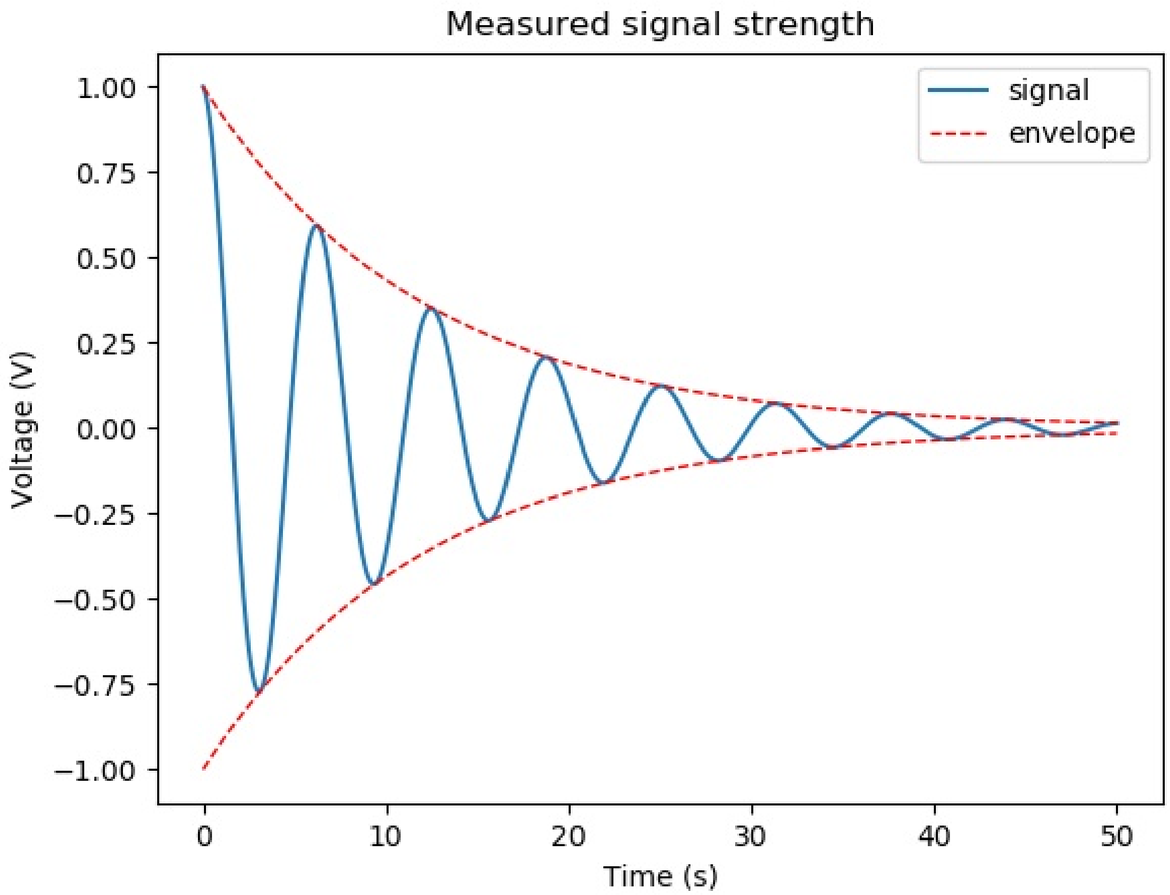
```
time = numpy.linspace(0, 50, 500)
envelope = numpy.exp(-time / 12)
signal = numpy.cos(time) * envelope

plt.figure()

plt.plot(time, signal, label='signal')
plt.plot(time, envelope, ls='--', lw=1, color='red', label='envelope')
plt.plot(time, -envelope, ls='--', lw=1, color='red')

plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.title('Measured signal strength')
plt.legend()
plt.show()
```

Adding a title and axis labels



Thank You