Python Programming for Data Science

Week 41, Monday

External modules:

- Numpy
- Matplotlib

Python external modules

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups
- `sudokumaker`: Create sudoku puzzles

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups
- `sudokumaker`: Create sudoku puzzles
- `IMDb`: Query the internet movie database

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups
- `sudokumaker`: Create sudoku puzzles
- `IMDb`: Query the internet movie database
- ...

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups
- `sudokumaker`: Create sudoku puzzles
- `IMDb`: Query the internet movie database
- ...

They can be found in the Python Package Index (PyPI)

Lots, and lots, and lots of modules...

There are Python modules to solve almost any task that you can imagine...

- `grmaster`: Module for dividing students into groups
- `sudokumaker`: Create sudoku puzzles
- `IMDb`: Query the internet movie database
- ...

They can be found in the Python Package Index (PyPI)

In addition to the many small modules (like the ones above) - there are some very well developed packages which are extremely useful for Scientific applications.

Python modules for data science

Python modules for data science

Pandas

A Python module for data manipulation similar to what you can do in `R` (i.e. arrays with labels).

Python modules for data science

Pandas

A Python module for data manipulation similar to what you can do in `R` (i.e. arrays with labels).

Scikit-learn

A data mining and machine learning toolkit: classification, clustering, regression, etc.

Python modules for data science

Pandas

A Python module for data manipulation similar to what you can do in `R` (i.e. arrays with labels).

Scikit-learn

A data mining and machine learning toolkit: classification, clustering, regression, etc.

Tensorflow/Pytorch

A Machine Learning libraries - with particular focus on neural networks and deep learning.

Matlab-like functionality in Python

Matlab-like functionality in Python

numpy, matplotlib and scipy bring a lot of Matlab-like functionality to python:

Matlab-like functionality in Python

numpy, matplotlib and scipy bring a lot of Matlab-like functionality to python:

Numpy

vector and matrix operations

Matlab-like functionality in Python

numpy, matplotlib and scipy bring a lot of Matlab-like functionality to python:

Numpy
vector and matrix operations

Matplotlib
Visualization

Matlab-like functionality in Python

numpy, matplotlib and scipy bring a lot of Matlab-like functionality to python:

Numpy
vector and matrix operations

Matplotlib
Visualization

Scipy
numerical integration, optimization, special functions

Python modules for science: Biology

Biopython

A package containing various tools for bioinformatics and computational biology

- Sequence class for manipulation biological sequences
- Parsing and iterating over various formats: fasta, blast, genbank, pubmed, etc
- Interfaces to external programs: multiple sequence alignment, blast
- Tools for structural biology: PDB
- Phylogenetic tools
- ...

Similar packages exist for many other scientific fields

Python package managers

In the old days, installing Python packages was a bit of a hassle

Python package managers

In the old days, installing Python packages was a bit of a hassle

Python packages managers make this job much easier

- easy_install
- pip
- anaconda (or miniconda)

Python package managers

In the old days, installing Python packages was a bit of a hassle

Python packages managers make this job much easier

- easy_install
- pip
- anaconda (or miniconda)

In this class we'll cover the last two.

pip

pip is a package management system used to install and manage external modules written in Python:

pip

pip is a package management system used to install and manage external modules written in Python:

Basic ideas:

pip

pip is a package management system used to install and manage external modules written in Python:

Basic ideas:

- Installing external modules by issuing one command (from the terminal).

  ```
  $ pip install package-name
  ```

  or (if you get permission errors)

  ```
  $ pip install --user package-name
  ```

pip

pip is a package management system used to install and manage external modules written in Python:

Basic ideas:

- Installing external modules by issuing one command (from the terminal).

  ```
  $ pip install package-name
  ```

  or (if you get permission errors)

  ```
  $ pip install --user package-name
  ```

- And uninstalling packages is equally easy

  ```
  $ pip uninstall package-name
  ```

Anaconda (miniconda)

Anaconda is a Python distribution for Scientific computing.

You can install packages using the conda command:

```
$ conda install some-package-name
```

Anaconda (miniconda)

Anaconda is a Python distribution for Scientific computing.

You can install packages using the conda command:

```
$ conda install some-package-name
```

Differences to pip:

- It can install non-Python dependencies if necessary
- Makes it easy to create different environments, containing different versions of modules (and different versions of Python)
- Has a more sophisticated dependency resolution method.

Anaconda (miniconda)

Anaconda is a Python distribution for Scientific computing.

You can install packages using the conda command:

```
$ conda install some-package-name
```

Differences to pip:

- It can install non-Python dependencies if necessary
- Makes it easy to create different environments, containing different versions of modules (and different versions of Python)
- Has a more sophisticated dependency resolution method.

Anaconda also includes pip.

Typical strategy: Try conda first, then pip.

Installing from PyCharm

You can use both pip and conda directly from within PyCharm.

See the Pycharm documentation for details: https://www.jetbrains.com/help/pycharm/installing-uninstalling-and-upgrading-packages.html

Numpy

numpy

Basic ideas:

numpy

Basic ideas:

- Make it easy to perform operations on a collection of numbers.

numpy

Basic ideas:

- Make it easy to perform operations on a collection of numbers.
- Make it easy to work with tables (matrices) of numbers

numpy

Basic ideas:

- Make it easy to perform operations on a collection of numbers.
- Make it easy to work with tables (matrices) of numbers
- Much faster than writing loops in Python

numpy

Basic ideas:

- Make it easy to perform operations on a collection of numbers.
- Make it easy to work with tables (matrices) of numbers
- Much faster than writing loops in Python

In short: whenever you have a list of numbers, consider using numpy.

numpy - importing

```
import numpy
```

numpy - importing

```
import numpy
```

or:

```
import numpy as np
```

numpy - array

The main ingredient in numpy is a new type: the numpy array.

Creating an array from a list or tuple:

```
a = np.array([1,2,3,4])
a = np.array((1,2,3,4))
```

numpy - array

The main ingredient in numpy is a new type: the numpy array.

Creating an array from a list or tuple:

```
a = np.array([1,2,3,4])
a = np.array((1,2,3,4))
```

In contrast to lists, numpy arrays are primarily designed to contain elements of the same type.

You can specify a type explicitly:

```
a = np.array([1,2,3,4], np.float)   # np.int, np.bool, ...
```

If not specified, numpy will take a guess.

numpy - array initialization

There are various other ways to initialize numpy arrays

numpy - array initialization

There are various other ways to initialize numpy arrays

Range of numbers:

```
print(np.arange(2.0, 2.4, 0.1))
```

```
[ 2. ,  2.1,  2.2,  2.3]
```

output

numpy - array initialization

There are various other ways to initialize numpy arrays

Range of numbers:

```
print(np.arange(2.0, 2.4, 0.1))
```

```
[ 2. ,   2.1,   2.2,   2.3]
```

output

Zeros

```
print(np.zeros(4))
```

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

output

numpy - array initialization

There are various other ways to initialize numpy arrays

Range of numbers:

```
print(np.arange(2.0, 2.4, 0.1))
```

```
[ 2. ,  2.1,  2.2,  2.3]
```

output

Zeros

```
print(np.zeros(4))
```

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

output

Ones:

```
print(np.ones(4))
```

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

output

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
```

output

```
[1 2 3]
```

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
```

```
output

[1 2 3]
[3 6 9]
```

numpy - operations on arrays

The common mathematical operators are available

```python
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
```

output

```
[1 2 3]
[3 6 9]
[2 4 6]
```

numpy - operations on arrays

The common mathematical operators are available

```python
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
```

output

```
[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
```

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
print(a/a)
```

output

```
[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
[1. 1. 1.]
```

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
print(a/a)
print(np.cos(a))
```

```
output

[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
[1. 1. 1.]
[ 0.54030231 -0.41614684 -0.9899925 ]
```

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
print(a/a)
print(np.cos(a))
```

output

```
[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
[1. 1. 1.]
[ 0.54030231 -0.41614684 -0.9899925 ]
```

Note how they automatically apply on all elements.

numpy - operations on arrays

The common mathematical operators are available

```
a = np.arange(1,4)

print(a)
print(3*a)
print(a+a)
print(a*a)
print(a/a)
print(np.cos(a))
```

```
output

[1 2 3]
[3 6 9]
[2 4 6]
[1 4 9]
[1. 1. 1.]
[ 0.54030231 -0.41614684 -0.9899925 ]
```

Note how they automatically apply on all elements.

What would happen if we did the same with normal lists?

numpy - random arrays

You can create arrays with random numbers

numpy - random arrays

You can create arrays with random numbers

Random floating point numbers between 0 and 1

```
print(np.random.rand(3))
```

```
output
[ 0.53066947  0.03155089  0.39243265]
```

numpy - random arrays

You can create arrays with random numbers

Random floating point numbers between 0 and 1

```
print(np.random.rand(3))
```

```
output
[ 0.53066947  0.03155089  0.39243265]
```

Random integers:

```
# 5 numbers between 1 and 3 (3 not included)
print(np.random.randint(1,3,5))
```

```
output
[2 1 2 2 2]
```

numpy - random arrays

You can create arrays with random numbers

Random floating point numbers between 0 and 1

```
print(np.random.rand(3))
```

```
output
[ 0.53066947  0.03155089  0.39243265]
```

Random integers:

```
# 5 numbers between 1 and 3 (3 not included)
print(np.random.randint(1,3,5))
```

```
output
[2 1 2 2 2]
```

Different distributions:

```
# normal distr: mean=0, stddev=1
print(np.random.standard_normal(2))
```

```
output
[ 1.70067518  0.63932443]
```

numpy - Exercise 1

The previous slide provides us with a very simple way to simulate throwing 2 dice (like we did earlier in the course).

1. Create a one-line python statement that calculates the average of 10,000 throws of the sum of two dice.
   Hint: `np.average()` calculates the average over an array.

numpy - Exercise 1 - solution

1. *Create a one-line python statement that calculates the average of 10,000 throws of the sum of two dice. Hint:* `np.average()` *calculates the average over an array*.

numpy - Exercise 1 - solution

1. *Create a one-line python statement that calculates the average of 10,000 throws of the sum of two dice. Hint: `np.average()` calculates the average over an array.*

```
print(np.average(np.random.randint(1,7,10000) + np.random.randint(1,7,10000)))
```

```
7.041100000000001
```
output

numpy - multidimensional arrays

Common scenario: data set with multiple columns

numpy - multidimensional arrays

Common scenario: data set with multiple columns

Numpy is ideal for handling such data - using 2D arrays.

numpy - multidimensional arrays

Common scenario: data set with multiple columns

Numpy is ideal for handling such data - using 2D arrays.

Initializing from a list of lists:

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

```
output
[[1 2 3 4]
 [5 6 7 8]]
```

numpy - multidimensional arrays: shape

In general arrays can be of any dimension. This is encoded in the *shape* of an array.

numpy - multidimensional arrays: shape

In general arrays can be of any dimension. This is encoded in the *shape* of an array.

The shape tuple specifies how many values there are in each dimension

numpy - multidimensional arrays: shape

In general arrays can be of any dimension. This is encoded in the *shape* of an array.

The shape tuple specifies how many values there are in each dimension

```
# 3 values in one dimension
a = np.arange(1,4)

print(a)
print(a.shape)
```

output

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

numpy - multidimensional arrays: shape

In general arrays can be of any dimension. This is encoded in the *shape* of an array.

The shape tuple specifies how many values there are in each dimension

```python
# 3 values in one dimension
a = np.arange(1,4)

print(a)
print(a.shape)

# 2 rows and 4 columns
a = np.array([[1,2,3,4],[5,6,7,8]])

print(a)

print(a.shape)
```

output

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

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

numpy - multidimensional arrays: initializing

Many of numpys initializer functions take a shape as argument

```
print(np.zeros(shape=(2,3)))
```

output

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

```
print(np.random.random((2,3)))
```

output

```
[[ 0.44923652  0.15528157  0.66292896]
 [ 0.5421247   0.44336697  0.27264862]]
```

numpy - arrays: initializing from file

Numpy has very convenient functionality for reading in data from file

numpy - arrays: initializing from file

Numpy has very convenient functionality for reading in data from file

data.txt

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

```python
a = np.genfromtxt('data.txt')
print(a)
```

output

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

numpy - array indexing and slicing (1)

You can index into a 1D numpy array just as a list

```
a = np.arange(1,4)
print(a[1])
```

output

```
2
```

numpy - array indexing and slicing (1)

You can index into a 1D numpy array just as a list

```
a = np.arange(1,4)
print(a[1])
```

```
output
2
```

For multidimensional arrays, you just specify an index for each dimension

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

print(a[0,2])
```

```
output
[[1 2 3 4]
 [5 6 7 8]]
3
```

numpy - array indexing and slicing (1)

You can index into a 1D numpy array just as a list

```
a = np.arange(1,4)
print(a[1])
```

output

```
2
```

For multidimensional arrays, you just specify an index for each dimension

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

print(a[0,2])
```

output

```
[[1 2 3 4]
 [5 6 7 8]]
3
```

Note the difference to how you would index into a list of lists

numpy - array indexing and slicing (2)

As for lists, use `:` to indicate that you want a range of values in a given dimension

```
a = np.array([[1,2,3,4],[5,6,7,8],[0,0,0,0]])
print(a)
```

output

```
[[1 2 3 4]
 [5 6 7 8]
 [0 0 0 0]]
```

numpy - array indexing and slicing (2)

As for lists, use `:` to indicate that you want a range of values in a given dimension

```
a = np.array([[1,2,3,4],[5,6,7,8],[0,0,0,0]])
print(a)



print(a[0,1:3])
```

output

```
[[1 2 3 4]
 [5 6 7 8]
 [0 0 0 0]]

[2 3]
```

numpy - array indexing and slicing (2)

As for lists, use `:` to indicate that you want a range of values in a given dimension

```
a = np.array([[1,2,3,4],[5,6,7,8],[0,0,0,0]])
print(a)



print(a[0,1:3])

print(a[0:2,1])
```

```
output

[[1 2 3 4]
 [5 6 7 8]
 [0 0 0 0]]

[2 3]

[2 6]
```

numpy - array indexing and slicing (2)

As for lists, use `:` to indicate that you want a range of values in a given dimension

```
a = np.array([[1,2,3,4],[5,6,7,8],[0,0,0,0]])
print(a)



print(a[0,1:3])

print(a[0:2,1])
```

```
output

[[1 2 3 4]
 [5 6 7 8]
 [0 0 0 0]]

[2 3]

[2 6]
```

If you want all values in a given dimension, just use `:` by itself

```
# Extract second column
print(a[:,1])
```

```
output

[2 6 0]
```

numpy - array indexing and assignment

Just like lists, you can also using indexing to change part of an array, by *assigning* to it:

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

# Setting part of array to zero
a[:,3] = 0
print(a)
```

output

```
[[1 2 3 4]
[5 6 7 8]]

# What is a now?
```

numpy - array indexing and assignment

Just like lists, you can also using indexing to change part of an array, by *assigning* to it:

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

# Setting part of array to zero
a[:,3] = 0
print(a)
```

output

```
[[1 2 3 4]
[5 6 7 8]]

# What is a now?
[[1 2 3 0]
 [5 6 7 0]]
```

numpy - boolean arrays

A condition on an array provides a Boolean array:

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a < 4)
```

output

```
[[ True  True  True False]
 [False False False False]]
```

numpy - boolean arrays

A condition on an array provides a Boolean array:

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a < 4)
```

```
output
[[ True  True  True False]
 [False False False False]]
```

You use such an array to select values in the original array

```
a = np.array([[1,2,3,4],[5,6,7,8]])
mask = a < 4      # This is a boolean array
print(a[mask])    # Use the mask as index
```

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

numpy - boolean arrays

A condition on an array provides a Boolean array:

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a < 4)
```

```
output
[[ True  True  True False]
 [False False False False]]
```

You use such an array to select values in the original array

```
a = np.array([[1,2,3,4],[5,6,7,8]])
mask = a < 4      # This is a boolean array
print(a[mask])    # Use the mask as index
```

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

Note that we lost the 2D structure of the array, because the relevant values not necessarily line up in rows and columns.

numpy - Exercise 2

Remember the hand-in where we calculated the average of two columns of numbers?

1. If you don't already have it, get experimental_results.txt: https://wouterboomsma.github.io/ppds2021/data/experimental_resu
2. Read this file into a numpy array called `data`.
3. Calculate the average of each of the two columns
4. Create a numpy array called `data_subset`, and set it to contain the first 500 values of the second column of the array.
5. Set all entries in the `data_subset` variable to zero.
6. Now calculate the average of each of the two columns in `data` array again.

numpy - Exercise 2 - solution

1. If you don't already have it, get experimental_results.txt:
   https://wouterboomsma.github.io/ppds2021/data/experimental_results.
2. Read this file into a numpy array called `data`.

```
import numpy as np
data = np.genfromtxt("experimental_results.txt")
```

3. Calculate the average of each of the two columns

```
print(np.average(data[:,0]),
      np.average(data[:,1]))
```

```
outpu
0.495052368 0.498952201
```

numpy - Exercise 2 - solution

1. If you don't already have it, get experimental_results.txt: https://wouterboomsma.github.io/ppds2021/data/experimental_results.
2. Read this file into a numpy array called `data`.

```
import numpy as np
data = np.genfromtxt("experimental_results.txt")
```

3. Calculate the average of each of the two columns

```
print(np.average(data[:,0]),
      np.average(data[:,1]))
```

```
outpu
0.495052368 0.498952201
```

or...

```
print(np.average(data, axis=0))
```

```
outpu
[ 0.49505237  0.4989522 ]
```

numpy - Exercise 2 - solution (2)

4. Create a numpy array called `data_subset`, and set it to contain the first 500 values of the second column of the `data` array.

```
data_subset = data[:500, 1]
```

5. Set all entries in the `data_subset` variable to zero.

```
data_subset[:] = 0
```

6. Now calculate the average of each of the two columns in the `data` array again.

```
print(np.average(data, axis=0))
```

```
[ 0.49505237  0.25336498]
```

output

numpy - Exercise 2 - solution (2)

4. Create a numpy array called `data_subset`, and set it to contain the first 500 values of the second column of the `data` array.

```
data_subset = data[:500, 1]
```

5. Set all entries in the `data_subset` variable to zero.

```
data_subset[:] = 0
```

6. Now calculate the average of each of the two columns in the `data` array again.

```
print(np.average(data, axis=0))
```

```
[ 0.49505237  0.25336498]
```
output

What happened here?

numpy - slices are references!

When you make a slice of an array, numpy doesn't make a new copy of the data

```
a = np.array(((1,2,3), (4,5,6), (7,8,9)))
b = a[0:2, 0:2]
b[0,0] = 8
print(a)
```

output

```
[[8 2 3]
 [4 5 6]
 [7 8 9]]
```

numpy - slices are references!

When you make a slice of an array, numpy doesn't make a new copy of the data

```
a = np.array(((1,2,3), (4,5,6), (7,8,9)))
b = a[0:2, 0:2]
b[0,0] = 8
print(a)
```

output

```
[[8 2 3]
 [4 5 6]
 [7 8 9]]
```

This is important when working with large data sets

If you want a copy, use np.copy

```
b = np.copy(a[0:2, 0:2])
```

numpy - conclusions

- Numpy has functionality for handling large amounts of data efficiently
- It handles vectors and matrices in a similar way as Matlab
- It is one of the success-stories of Python: https://www.nature.com/articles/s41586-020-2649-2

Matplotlib

matplotlib

Very powerful, general purpose plotting functionality

matplotlib

Very powerful, general purpose plotting functionality

There are thousands of things you can do with it. We'll just show a few examples here...

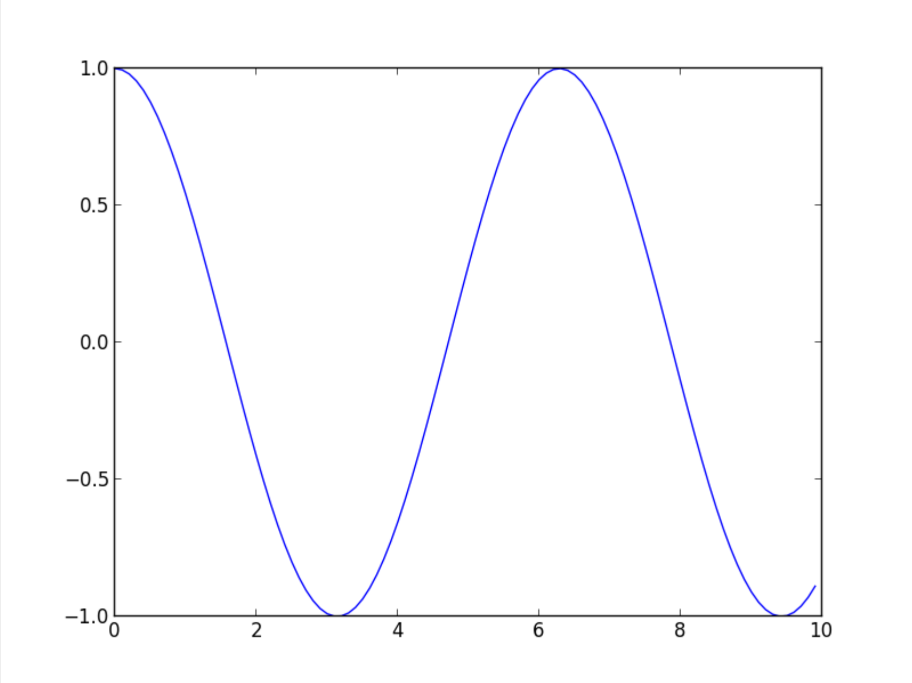
matplotlib - importing

The standard way to import matplotlib:

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

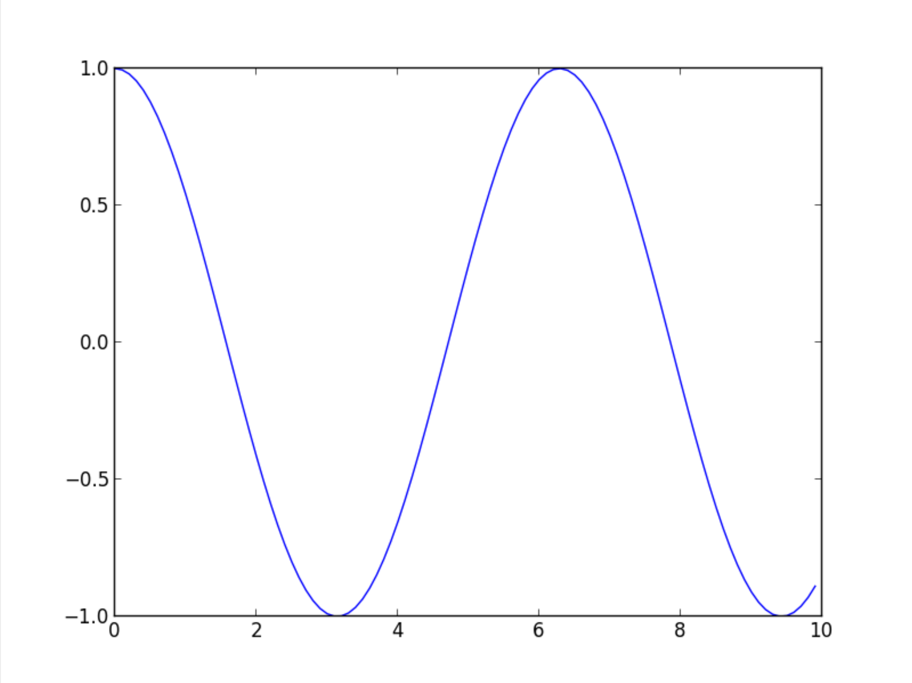
matplotlib - simple plot

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.1)
y = np.cos(x)
plt.plot(x,y)
plt.show()
```



matplotlib - simple plot

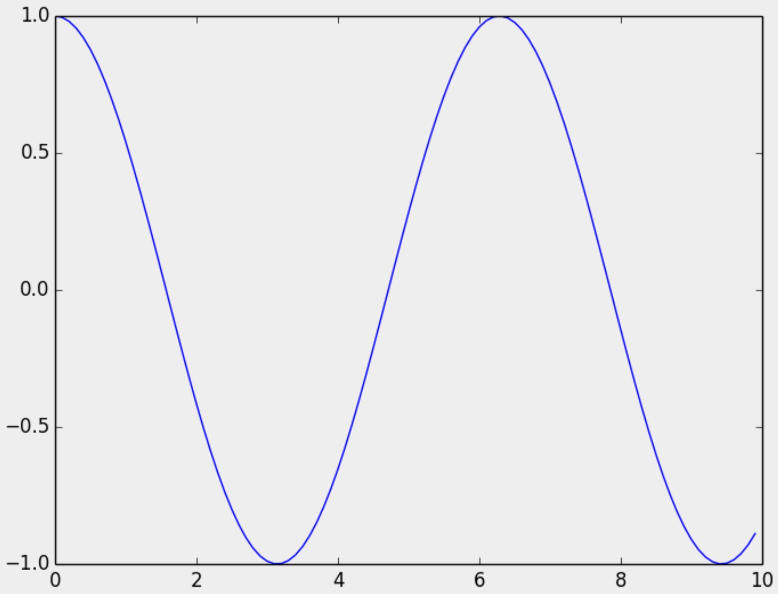
```python
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.1)
y = np.cos(x)
plt.plot(x,y)
plt.show()
```



Try it out!

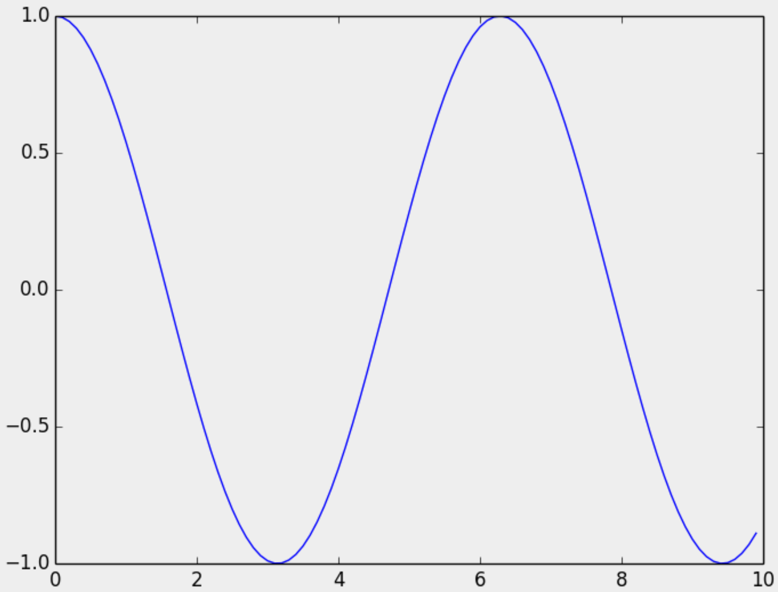
matplotlib - saving to a file

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.1)
y = np.cos(x)
plt.plot(x,y)
plt.savefig('plot.png',
            transparent=True)
```



matplotlib - saving to a file

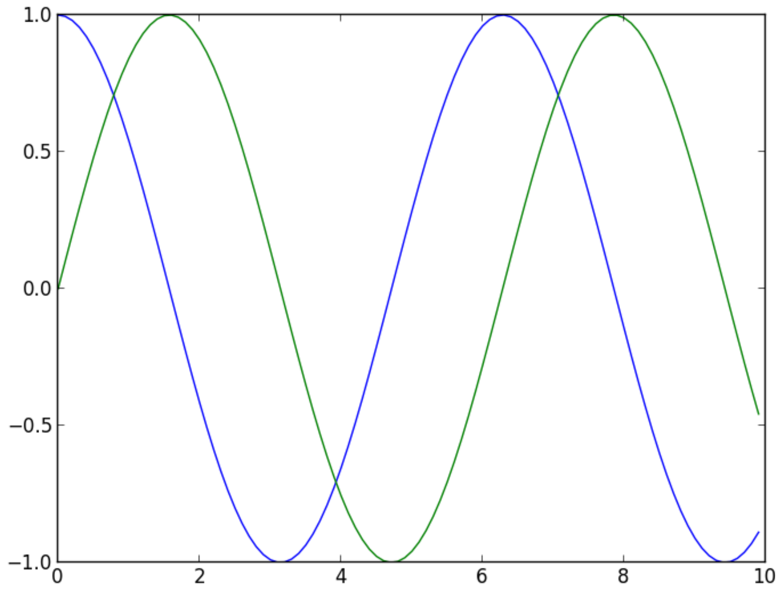
```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.1)
y = np.cos(x)
plt.plot(x,y)
plt.savefig('plot.png',
            transparent=True)
```



Note that you can use `transparent=True` for transparent backgrounds.

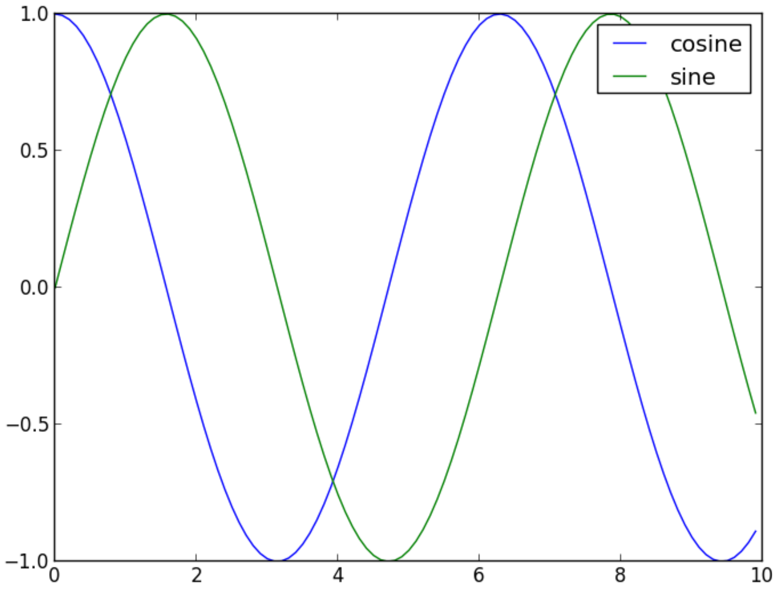
matplotlib - multiple plots

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

x = np.arange(0, 10, 0.1)

y1 = np.cos(x)
plt.plot(x,y1)

y2 = np.sin(x)
plt.plot(x,y2)

plt.show()
```



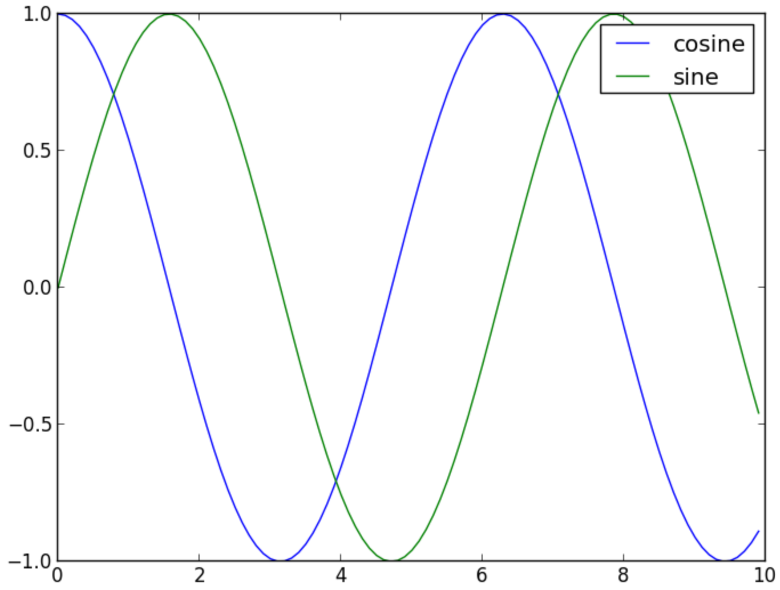
matplotlib - legends

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

x = np.arange(0, 10, 0.1)

y1 = np.cos(x)
plt.plot(x,y1, label='cosine')

y2 = np.sin(x)
plt.plot(x,y2, label='sine')

plt.legend()

plt.show()
```



matplotlib - legends

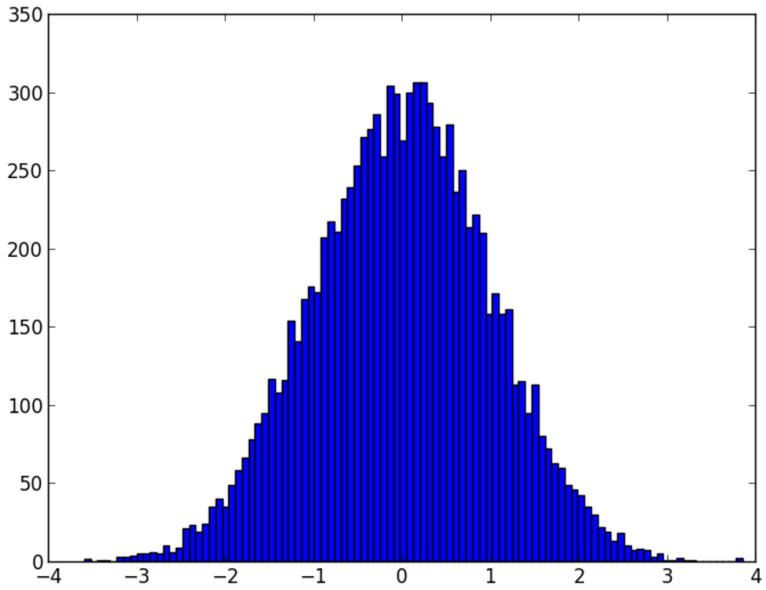
```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.1)

y1 = np.cos(x)
plt.plot(x,y1, label='cosine')

y2 = np.sin(x)
plt.plot(x,y2, label='sine')

plt.legend()

plt.show()
```



Note that the `legend()` automatically reads the labels from the individual plot commands.

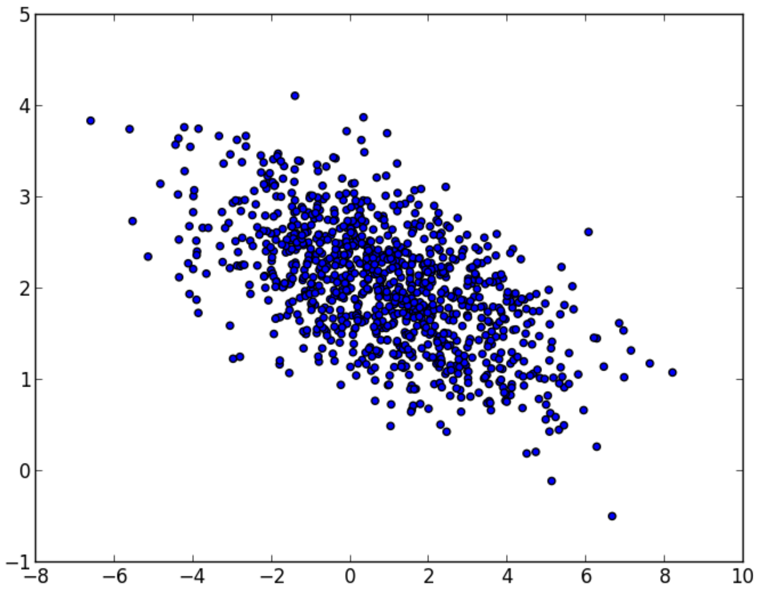
matplotlib - histograms

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

x = np.random.standard_normal(10000)
plt.hist(x, bins=100)

plt.show()
```



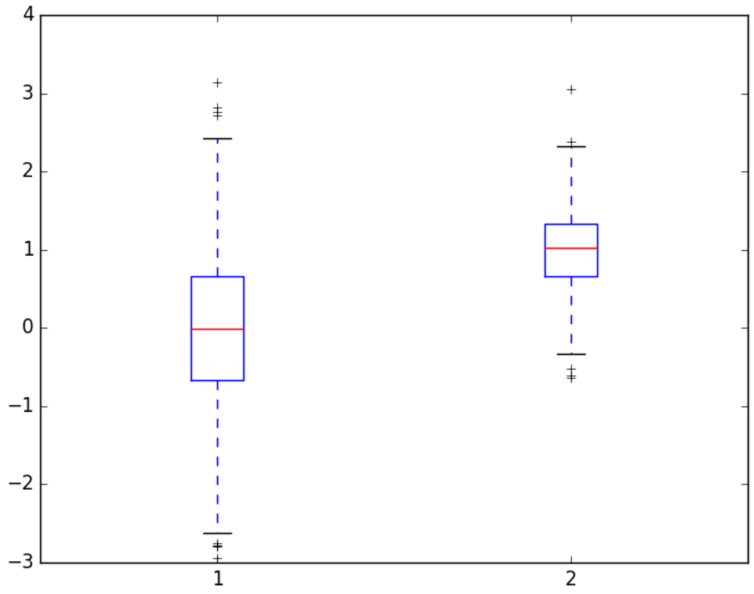
matplotlib - scatter plot

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

# 2D-Gauss distribution
x = np.random.multivariate_normal(
      mean=(1,2),
      cov=[[1,-0.5],[-0.5,1]],
      size=1000)
plt.scatter(x[:,0],x[:,1])

plt.show()
```



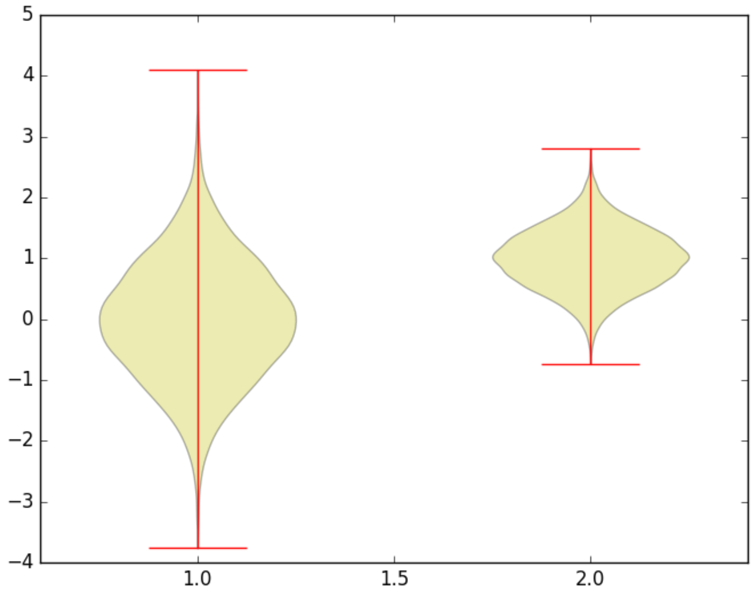
matplotlib - box plot

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

# Two different distributions
data = [np.random.normal(0, 1,
                         1000),
        np.random.normal(1, 0.5,
                         1000)]
plt.boxplot(data)

plt.show()
```



matplotlib - violin plot

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

# Two different distributions
data = [np.random.normal(0, 1,
                         10000),
        np.random.normal(1, 0.5,
                         10000)]
plt.violinplot(data)

plt.show()
```



matplotlib - Exercise 1

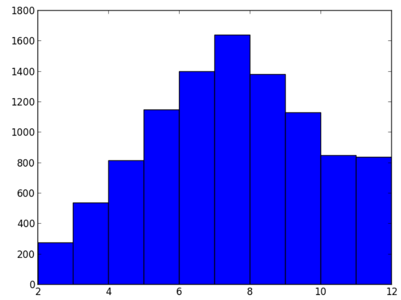
Remember the die example?

```
np.average(np.random.randint(1,7,10000) + np.random.randint(1,7,10000))
```

1. Instead of taking the average, use the `plt.hist` function to plot the distribution of outcomes from the sum of two dice. That is: plot the frequency of observing all the different outcomes.
2. Does the result look like what you would expect?

matplotlib - Exercise 1 - solution

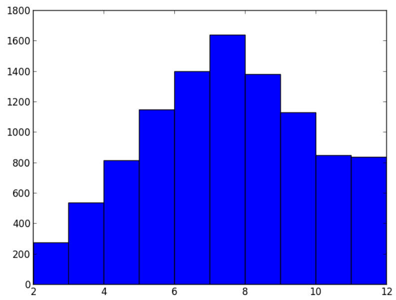
```
import matplotlib.pyplot as plt
import numpy as np

dist = (np.random.randint(1,7,10000) +
        np.random.randint(1,7,10000))
plt.hist(dist)

plt.show()
```



Is this what we expect?

matplotlib - Exercise 1 - solution

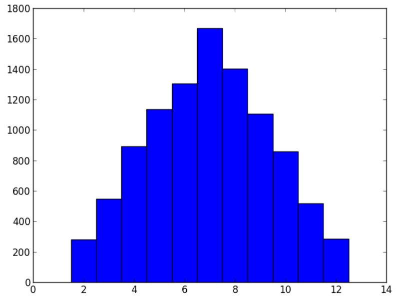
```
import matplotlib.pyplot as plt
import numpy as np

dist = (np.random.randint(1,7,10000) +
        np.random.randint(1,7,10000))
plt.hist(dist)

plt.show()
```



Is this what we expect? No.

Adjust the bin vector to get the right result

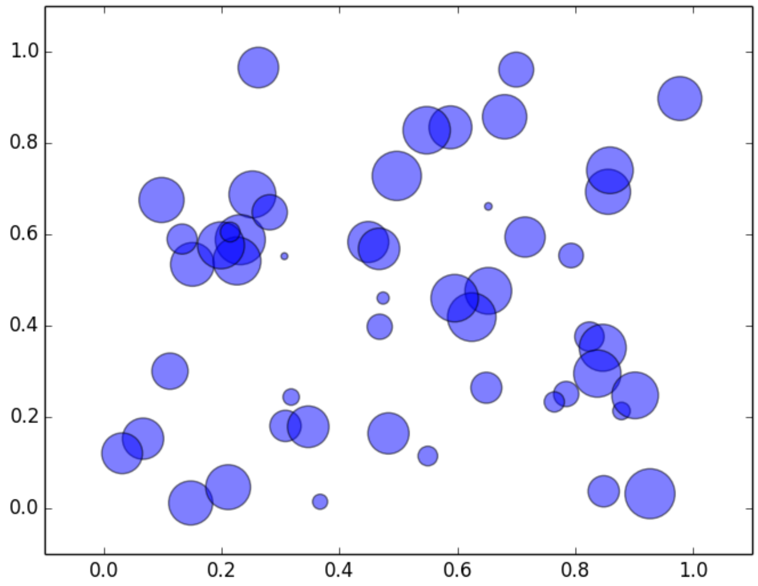
```
plt.hist(dist, bins=np.arange(1.5, 13.5, 1))
```



matplotlib - Opacity

Many of matplotlib functions support an alpha parameter, which sets the opacity

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

# Create distribution of points
dist = np.random.random((50,2))

# What does this line do?
sizes = np.random.randint(10, 1000,
                          dist.shape[0])


# Create scatter plot
plt.scatter(dist[:,0],dist[:,1],
            s=sizes,
            alpha=0.5) # set opacity

plt.show()
```



matplotlib - Exercise 2

1. Download the data file from:
   https://wouterboomsma.github.io/ppds2021/data/fern_data.t
2. Read the file into a numpy array
3. Create a scatter plot from this data (this might take a while)
4. Check out
   http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.sca
   - Figure out how to remove the edges around each poin
     (hint: edgecolor)
   - Figure out how to set the size of each point to 0.5
   - Figure out how to set the color of each point to green
   - Figure out how to remove the axes

matplotlib - Exercise 2 - Solution (1)

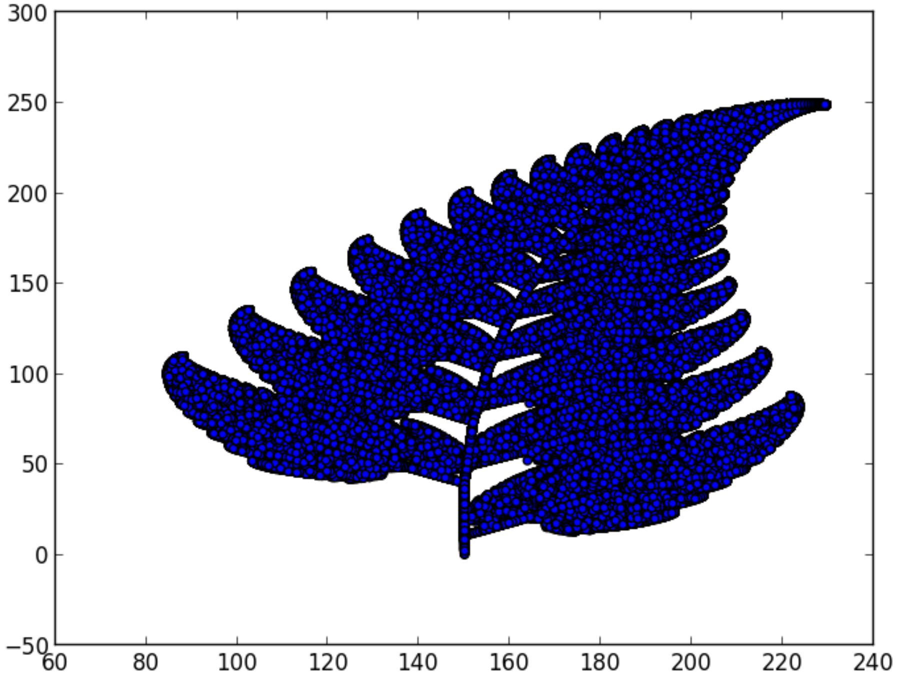
2. Read the file into a numpy array
3. Create a scatter plot from this data (this might take a while)

```
import numpy as np
import matplotlib.pyplot as plt
data = np.genfromtxt('fern_data.txt')

plt.scatter(data[:,0], data[:,1])

plt.savefig('matplotlib_fern1.png')
```

matplotlib - Exercise 2 - Solution (1)

2. Read the file into a numpy array
3. Create a scatter plot from this data (this might take a while)

```
import numpy as np
import matplotlib.pyplot as plt
data = np.genfromtxt('fern_data.txt')

plt.scatter(data[:,0], data[:,1])

plt.savefig('matplotlib_fern1.png')
```



matplotlib - Exercise 2 - Solution(2)

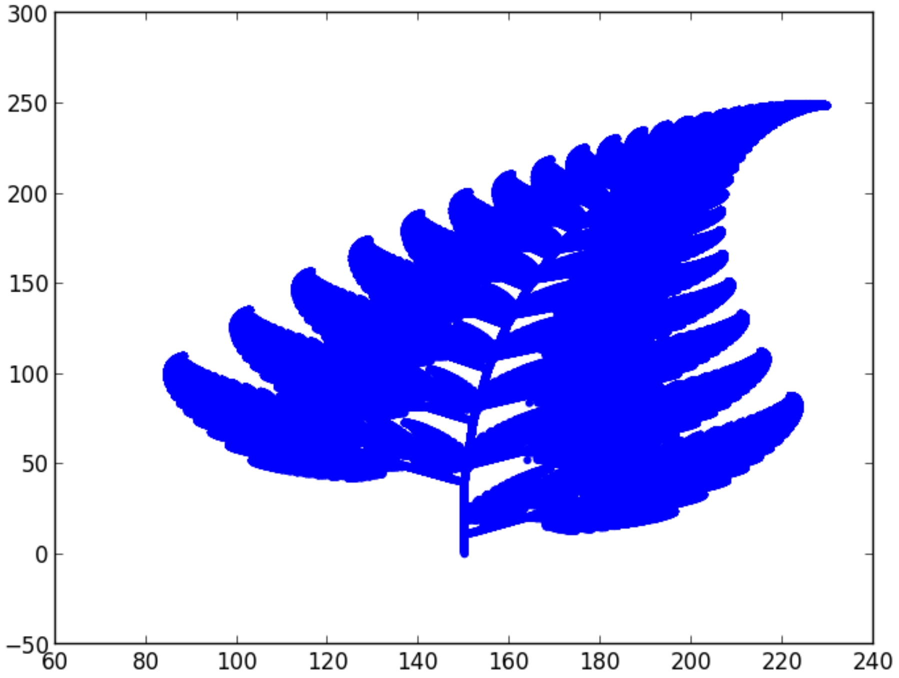
4. Figure out how to remove the edges around each point (hint: edgecolor)

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none')
```

matplotlib - Exercise 2 - Solution(2)

4. Figure out how to remove the edges around each point (hint: edgecolor)

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none')
```



matplotlib - Exercise 2 - Solution(3)

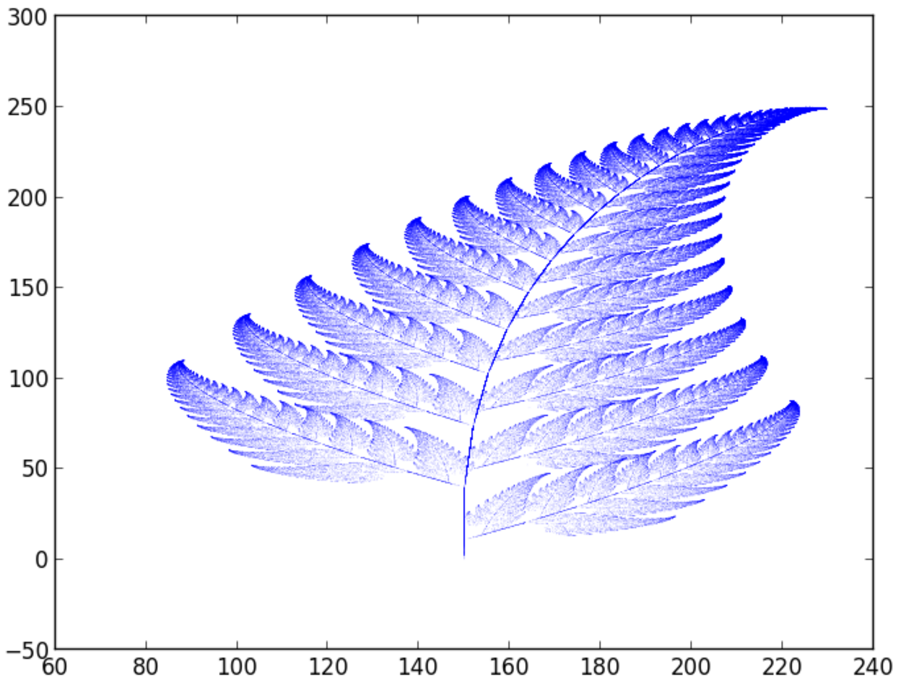
5. Figure out how to set the size of each point to 0.5

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none',
            s=0.5)
```

matplotlib - Exercise 2 - Solution(3)

5. Figure out how to set the size of each point to 0.5

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none',
            s=0.5)
```



matplotlib - Exercise 2 - Solution(4)

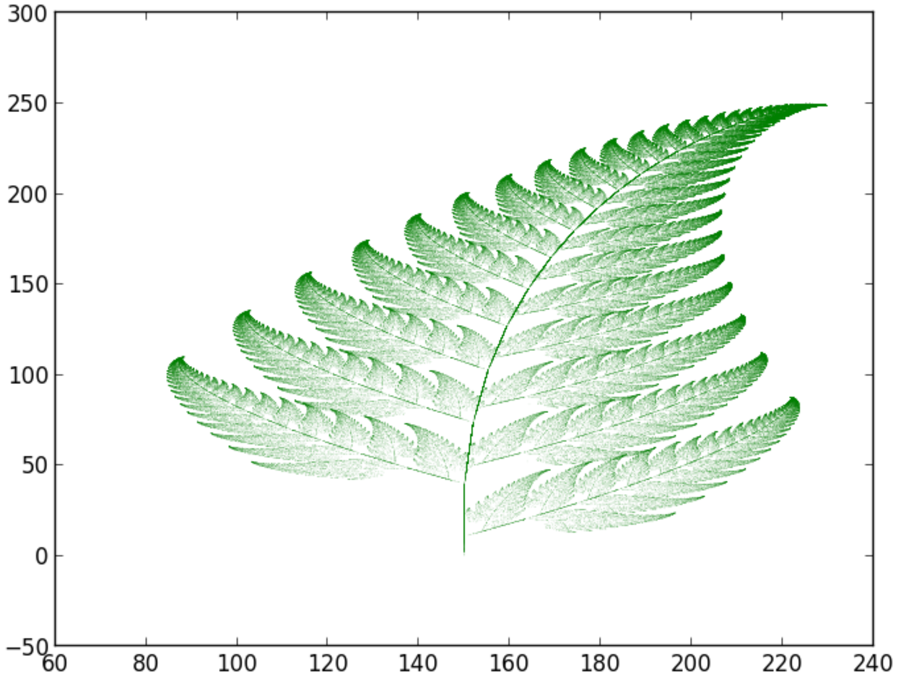
6. Figure out how to set the color of each point to green

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none',
            s=0.5,
            c='green')
```

matplotlib - Exercise 2 - Solution(4)

6. Figure out how to set the color of each point to green

```
plt.scatter(data[:,0], data[:,1],
            edgecolor='none',
            s=0.5,
            c='green')
```



matplotlib - Exercise 2 - Solution(5)

7. Figure out how to remove the axes

```
plt.axis('off')
```

matplotlib - Exercise 2 - Solution(5)

7. Figure out how to remove the axes

```
plt.axis('off')
```

matplotlib - Exercise 2 - Solution(6)

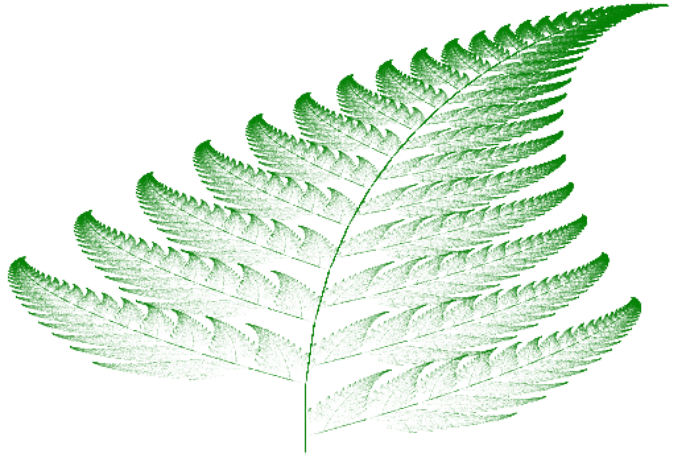
Making background transparent...

```
plt.savefig('matplotlib_fern6.png',
            transparent=True)
```

matplotlib - Exercise 2 - Solution(6)

Making background transparent...

```
plt.savefig('matplotlib_fern6.png',
            transparent=True)
```



For the curious: How the fern was created

```
import random
import matplotlib.pyplot as plt
import numpy as np

# Size of plot
size = (300, 300)

# Start with random coordinates
x, y = random.random(), random.random()

# Start with no points
points = []

# Repeat many times
for i in range(500000):

    # Random number decides action to take
    rand = random.random()

    # This recursion is described in detail
    # on http://en.wikipedia.org/wiki/Barnsley_fern
    if rand < 0.01:
        x, y = 0.0, 0.16 * y
    elif rand < 0.86:
        newx = (0.85 * x) + (0.04 * y)
        newy = (-0.04 * x) + (0.85 * y) + 1.6
        x, y = newx, newy
    elif rand < 0.93:
        newx = (0.2 * x) - (0.26 * y)
        newy = (0.23 * x) + (0.22 * y) + 1.6
        x, y = newx, newy
    else:
        newx = (-0.15 * x) + (0.28 * y)
        newy = (0.26 * x) + (0.24 * y) + 0.44
```