Week 2

Data Science Laboratory 1

Dr John Evans
`j.evans8@herts.ac.uk`

University of Hertfordshire UH

Plan for today

Binary Data

ASCII Data

Structured and Unstructured Data

Images

Fundamental component of data science: data

- In general, we have some data stored in a file (or other resource) and we want to load this data into a Python script so that we can clean up the data, manipulate the data, analyse the data, and so on. We then need to store the result. For this reason, reading and writing data (input and output, or I/O) is key to being able to work with data science.

Fundamental component of data science: data

- In general, we have some data stored in a file (or other resource) and we want to load this data into a Python script so that we can clean up the data, manipulate the data, analyse the data, and so on. We then need to store the result. For this reason, reading and writing data (input and output, or I/O) is key to being able to work with data science.
- Binary data is essentially any data that is not text.
- Data represented in a file points to the memory address on the system where the data actually resides, byte by byte.
- For this reason, using binary formats are more efficient (though less user friendly): they take a smaller memory footprint and have the benefit that to access a particular part of the data does not require reading the entire file.

Fundamental component of data science: data

- In general, we have some data stored in a file (or other resource) and we want to load this data into a Python script so that we can clean up the data, manipulate the data, analyse the data, and so on. We then need to store the result. For this reason, reading and writing data (input and output, or I/O) is key to being able to work with data science.
- Binary data is essentially any data that is not text.
- Data represented in a file points to the memory address on the system where the data actually resides, byte by byte.
- For this reason, using binary formats are more efficient (though less user friendly): they take a smaller memory footprint and have the benefit that to access a particular part of the data does not require reading the entire file.
- The NumPy `save` and `load` helper functions will write and read NumPy objects such as arrays in binary format for us. Without worrying too much about NumPy just yet, we demonstrate how these work.

NumPy = numerical python

Example

```
import numpy as np
arr = np.arange(10)  # create our array
np.save('some_array', arr)  # save our array as a string
```

If the file path does not end with *.npy* then the extension will be appended. An .npy file is simply a NumPy array file created by the Python with the NumPy. It contains an array saved in the NumPy (.npy) file format and stores all the information required to reconstruct an array on any computer, which includes dtype and shape information.

Reading data

- To read the data, we need to unpack it. We can do this with NumPy's `load` function: `np.load('some_array.npy')`

Reading data

- To read the data, we need to unpack it. We can do this with NumPy's `load` function: `np.load('some_array.npy')`
- If we want to save multiple arrays in an uncompressed archive, we use `np.savez` and then pass arrays as keyword arguments: `np.savez('array_archive.npz',a=arr, b=arr)`
- To load a *.npz* file, we again use `np.load`.
- Finally, if we want to compress our data as well, then we use `np.savez_compressed`.

ASCII data

- ASCII stands for American Standard Code for Information Interchange.
- It is an established encoding scheme for representing alphanumeric characters on computers.
- For all intents and purposes, data stored as human-readable text or numbers can be classed as ASCII data, but note that in practice ASCII represents a special encoding on the computer. Actually, most text is now encoded using UTF-8.[1]

[1] `https://en.wikipedia.org/wiki/UTF-8`

ASCII data

- ASCII stands for American Standard Code for Information Interchange.
- It is an established encoding scheme for representing alphanumeric characters on computers.
- For all intents and purposes, data stored as human-readable text or numbers can be classed as ASCII data, but note that in practice ASCII represents a special encoding on the computer. Actually, most text is now encoded using UTF-8.[1]
- In Python, to read a file containing text we can simply open a file object:

```
# Use the open() built-in function.  'r' means 'read'
myfile = open('my_file.txt','r')
```

- We have not actually read the data yet, simply created the object.

[1]https://en.wikipedia.org/wiki/UTF-8

Objects and Python

- An important characteristic of Python is the consistency of its *object model*.
- Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own ‘box’, which is referred to as a *Python object*.
- Each object has an associated *type*, such as a *string* or *function*, and internal data.
- In practice, this makes the language very flexible, as even functions can be treated like any other object.

Back to reading data

- We can read all the data in one go: `data = my_file.read()`.

Back to reading data

- We can read all the data in one go: `data = my_file.read()`.
- Normally, it is useful to read the data into a list where each item is a single line: `lines = myfile.readlines()`
- After we have read the data, we should always close the file: `myfile.close()`

Back to reading data

- We can read all the data in one go: `data = my_file.read()`.
- Normally, it is useful to read the data into a list where each item is a single line: `lines = myfile.readlines()`
- After we have read the data, we should always close the file: `myfile.close()`
- If we are reading in data this way, it is likely that we will be dealing with some form of columnar data. In order to get it into a usable format, we need to **parse** the data. At the moment each item in `lines` is a long string. So for example, if each line contained, say a date and an average temperature, the first item might look like:

```
lines = myfile.readlines()
print(lines[0])
# 2022-12-17 0.2
```

University of Hertfordshire UH

Getting the data in a format we want

- In the above example, we can see that there is whitespace **delimiting** the two columns. We can use string operations to get the data into the format we want.

Getting the data in a format we want

- In the above example, we can see that there is whitespace **delimiting** the two columns. We can use string operations to get the data into the format we want.
- First, the `split()` method will split a string into a list of components. We could unpack these into two variables called `thedate` and `temperature`

```
lines = myfile.readlines()
thedate, temperature = lines[0].split()
print(thedate, temperature)
```

- By default it will split the string according to whitespace, but we could also supply an argument like `split(',')` that will split a string at every instance of (in this case) a comma. This is useful for when dealing with comma separated values (.csv).

Getting the data in a format we want

- In the above example, we can see that there is whitespace **delimiting** the two columns. We can use string operations to get the data into the format we want.
- First, the `split()` method will split a string into a list of components. We could unpack these into two variables called `thedate` and `temperature`

```
lines = myfile.readlines()
thedate, temperature = lines[0].split()
print(thedate, temperature)
```

- By default it will split the string according to whitespace, but we could also supply an argument like `split(',')` that will split a string at every instance of (in this case) a comma. This is useful for when dealing with comma separated values (.csv).
- At the moment `thedate` and `temperature` are still strings, so we need to cast them into an appropriate data type.

The `datetime` module

- We can use the `datetime` module to turn the date into a `date` object, and we can make the temperature a float:

  ```
  from datetime import date

  thedate = date.fromisoformat(thedate)
  temperature = float(temperature)
  ```

- The `fromisoformat()` method takes a string of the form YYYY-MM-DD and turns it into a `date` object.

Looping over all lines

- What we would like to do is loop over all lines and create two sequences storing all the dates and temperatures. One way to do this is through list appending:

```
# Declare two empty lists
dates, temperatures =  [], []

# Loop over all lines
for line in lines:
    thedate, temperature = line.split()
    dates.append(date.fromisoformat(thedate)) append = adding data
    temperatures.append(float(temperature))
```

This allows us to now manipulate the data using NumPy functionality.

Looping over all lines

- Now we have two sequences representing our two columns. We could cast them as NumPy arrays if we wanted to:

```
import numpy as np

dates = np.array(dates, dtype='datetime64')
temperatures = np.array(temperatures, dtype='float')
```

- This allows us to now manipulate the data using NumPy functionality.

Looping over all lines

- Now we have two sequences representing our two columns. We could cast them as NumPy arrays if we wanted to:

  ```
  import numpy as np

  dates = np.array(dates, dtype='datetime64')
  temperatures = np.array(temperatures, dtype='float')
  ```

- This allows us to now manipulate the data using NumPy functionality.

Question: What about writing data to a file?

Example

In this example we create some dummy data: a sequence of dates generated using the NumPy `arange` function, and a random number representing some ‘data’. The idea is that we want to write this data as two columns to an output file:

Example

In this example we create some dummy data: a sequence of dates generated using the NumPy `arange` function, and a random number representing some ‘data’. The idea is that we want to write this data as two columns to an output file:

```
import numpy as np
# Generate sequence of dates for all days in 2022
dates = np.arange('2022-01',
                  '2022-12',
                  dtype='datetime64[D]')

# Create output file for writing ('w')
outfile = open('out.txt', 'w')

# Loop over dates
for d in dates:
    #Generate random number
    v = np.random.uniform()
    # Use an f-string to write out a string to the file
    outfile.write(f'{d} {v:.5f}\n')
# Close the file
outfile.close()
```

Example continued

- The most important line in the code above is the write statement itself.
- It describes how the data is going to appear in the file. We use an ‘f-string’ which means ‘formatted string’.
- The string in the argument to the `write()` method is contained within quotation marks but preceded by ‘`f`’. Inside we have two sets of curly brackets.

Example continued

- The most important line in the code above is the write statement itself.
- It describes how the data is going to appear in the file. We use an ‘f-string’ which means ‘formatted string’.
- The string in the argument to the `write()` method is contained within quotation marks but preceded by ‘`f`’. Inside we have two sets of curly brackets.
- The first references `d`, which in the loop we have written is pointing to each date in the sequence. That date will be printed in the familiar YYYY-MM-DD format, so we do not need to do anything fancy to it.
- The next set of curly brackets contain a reference to `v` which is our dummy data – just a random number. We want to format this in a specific way, and so we supply some more information. The `.5f` bit after the colon says that we want to write out the number to five decimal places.
- We use whitespace between the columns just by putting a space, and finally a newline return at the end to make sure the next write goes on the following line.

Header information

It is good practice to have some form of header information in the files we write. This can take the form of some ‘comment’-like indicator, such as a hash symbol which we can ignore when the file is read. Modifying the read example above yields the following:

Header information

It is good practice to have some form of header information in the files we write. This can take the form of some 'comment'-like indicator, such as a hash symbol which we can ignore when the file is read. Modifying the read example above yields the following:

```
# Declare two empty lists
dates, temperatures =  [], []

# What symbol identifies a line to ignore in the input?
comment = '#'

# Loop over all lines
for line in lines:
    # Check if first character matches the comment symbol.
    # Read the data if it doesn't.
    if not line.startswith(comment):
        thedate, temperature = line.split()
        dates.append(date.fromisoformat(thedate))
        temperatures.append(float(temperature))
```

if statement

```
EXAMPLE:
if statement_1:
    action1
elif statement_2:
    action2
else:
    action3
```

if statement = when statement is true

if not statement = when statement is false

Saving using NumPy

- Knowing how to read and write data files in this way will give us maximum flexibility when it comes to interacting with data stored on a file system.
- There also exist helper functions in NumPy that allow us to achieve similar results.

Saving using NumPy

- Knowing how to read and write data files in this way will give us maximum flexibility when it comes to interacting with data stored on a file system.
- There also exist helper functions in NumPy that allow us to achieve similar results. For example, if `my_data` represents a 2d array containing three rows of data that we want to save as three columns in .csv format then we could use the following:

3-zero padded integer

integer

```
np.savetxt(‘my_data.csv’,my_data,delimiter=’,’,fmt=[’%d’,’%.1f’,’%03d’])
```

floating point

- Here we specify the output format of each column using the `fmt` keyword argument. In this case, we want column one to be an integer, column two to be a float to one decimal place and column three to be a 3-zero padded integer.

Loading using NumPy

- Similarly, we can load data, say columns 1, 3 and 5 of a .csv file with # comments:

  d = np.loadtxt('my_data.csv', dtype='float', comments='', delimiter=',', usecols=(0,2,4))

- Note, column numbers are zero-indexed.
- The resulting `d` array will have a shape corresponding to the number of rows and columns to be read.

Structured and unstructured data

- Most data in the world is unstructured – that is, it does not follow a single set of rules governing formatting.
- The text of a book is an example: its contents could represent anything, and we do not know in advance how many words it contains, what language it is in, and so-on.
- A website is another.

Structured and unstructured data

- Most data in the world is unstructured – that is, it does not follow a single set of rules governing formatting.
- The text of a book is an example: its contents could represent anything, and we do not know in advance how many words it contains, what language it is in, and so-on.
- A website is another.
- Structured data on the other hand does follow a fixed format.
- An example would be a table of data with a set number of columns and a description of what each column contains.

Unstructured data

- First let us see how we could deal with some unstructured data: the contents of the BBC news website.
- For this, we will use the requests module which provides methods to ‘get’ or ‘post’ data via HTTP[2].
- We can use requests to interact with web content.

[2]HTTP = Hypertext Transfer Protocol

Example

```
import requests
# We use requests to perform a get request on the URL.
r = requests.get('https://www.bbc.co.uk/news')

# The actual data is the content attribute. We want to decode it as UTF-8 formatted
# text representing the HTML of the page. We can use the text() method to do this,
# equivalent to data = (r.content).decode('utf-8')
data = r.text

# Define a counter.
n_coronavirus = 0

# Split the data based on whitespace and loop over it.
for d in data.split():
    # Look for instances of 'coronavirus' or 'Covid-19' strings in each d.
    if ('coronavirus' in d) or ('Covid-19' in d):
            # Add to counter.
            n_coronavirus += 1

print(f'There are {n_coronavirus} mentions of coronavirus or Covid-19 today.')
```

Example

- In this simple example we got the HTML of the current version of `https://www.bbc.co.uk/news` as text.
- Then, after splitting the contents based on whitespace, we simply looped over each item in the HTML and checked for instances of mentions of ‘coronavirus’ or ‘Covid-19’, adding to the counter when we found one.
- Finally, the information is reported back.

Structured data

- Now let us see an example of dealing with structured data on the web.
- We will use the record of weather data measured at the Heathrow station, accessed via the MET office.

Structured data

- Now let us see an example of dealing with structured data on the web.
- We will use the record of weather data measured at the Heathrow station, accessed via the MET office.
- This is just a simple record of key meteorological data on a month-by-month basis since 1948, represented by a table (the structure of which we can inspect): `https://www.metoffice.gov.uk/pub/data/weather/uk/climate/stationdata/heathrowdata.txt`.
- If we actually look at that URL in a browser, we will see a table of text, with columns for year, month, temperature etc., along with a short header describing the contents.
- We now write a script that pulls out the average rainfall for the month of August, and compares it to the latest value:

Example

```
import requests
import numpy as np

# Define URL.
heathrow = 'https://www.metoffice.gov.uk/pub/data/weather/uk/climate/
            stationdata/heathrowdata.txt'

# Use requests.
r = requests.get(heathrow)

# Get the text.
data = r.text

# Split it into lines based on newline character.
lines = data.split('\n')
# List to store rainfall.
mm_of_rain = []
```

Example

```
# Loop over lines - starting at index 7 since there are 7 header lines to ignore
for line in lines[7:]:
        # Split the line into columns.
        entries = line.split()
        # Column 2 is the month ID - we want August = 8.
        if entries[1]=='8':
                # The 6th column is the rainfall
                # which we cast as a float and append.
                mm_of_rain.append(float(entries[5]))

# Turn it into an array.
mm_of_rain = np.array(mm_of_rain)

# Calculate the mean
average_amount_of_rain = np.mean(mm_of_rain)
```

Example

```
# The latest value is at the [-1] index, which we compare to the mean
# Normalised by the standard deviation.
significance = (mm_of_rain[-1]-average_amount_of_rain) / np.std(mm_of_rain)

# Print the information to the screen
print(f'On average we expect {average_amount_of_rain:.1f} mm of rain in August
      at Heathrow.')
print(f'This August {mm_of_rain[-1]:.1f} mm of rain were recorded,
      equivalent to {significance:.2f} standard deviations from the mean.')
```

Images

- Having covered binary and text data, the other main data type we will encounter is imagery.
- Images are stored in various formats: Portable Network Graphics (PNG), Tagged Image File Format (TIF or TIFF), Joint Photographic Experts Group (JPEG) and Graphics Interchange Format (GIF) being the most common.

Images

- Having covered binary and text data, the other main data type we will encounter is imagery.
- Images are stored in various formats: Portable Network Graphics (PNG), Tagged Image File Format (TIF or TIFF), Joint Photographic Experts Group (JPEG) and Graphics Interchange Format (GIF) being the most common.
- Each format has its advantages and disadvantages.
- For example, JPEGs do not take up too much memory, but they use a compression algorithm which makes them ‘lossy’ – i.e. some of the information is thrown away when the file is written. This is why JPEGs often seem blocky or pixellated.
- TIFFs on the other hand can be lossless, which allows one to store high quality imaging at the expense of memory footprint.
- Which format we use really depends upon the application.

Reading an image

- An image file generally contains the data itself in some binary format and a header describing what sort of format the data is in.
- When we read an image, it is generally **rasterized**, which means that the data is represented by 2D grid, representing pixels.

[3] The Cyan Magenta Yellow Black or CMYK format is another system.

Reading an image

- An image file generally contains the data itself in some binary format and a header describing what sort of format the data is in.
- When we read an image, it is generally **rasterized**, which means that the data is represented by 2D grid, representing pixels.
- There is a synergy between images and NumPy arrays and there are well-established Python tools for reading and writing images.
- A greyscale image can be represented by one 2D array. A colour image can be represented by three 2D arrays, representing the red, green and blue channels.
- Sometimes there is a fourth channel called alpha, which can handle image transparency.
- Not all imaging formats support transparency.[3]
- We can interact with images in Python using the `skimage` package.

[3] The Cyan Magenta Yellow Black or CMYK format is another system.

Example

Take the example, `goodgirl.jpg`:

```
from skimage import io
import numpy as np

# Read an image
image = io.imread('goodgirl.jpg')

print(image.shape)
# (300, 200, 3)
```

Example

Whitespace and binary operators

- Note that `imread` automatically knows what format the image is in and applies the relevant decoding for us.
- It returns the data as a 3D NumPy array, with a shape (300, 200, 3).
- Note the ordering of the axes: the vertical direction is the *first* axis in the array.
- We could play around a bit, and just pull out the red channel, and then save that as a new image.

Example

```
from skimage import io
import numpy as np

# Read an image
image = io.imread('goodgirl.jpg')

# Get all the pixels, but just the red channel (RGB)
red = image[:,:,0]

# Write it out again
io.imsave('goodgirl_red.jpg', red)
```

The result is viewed as a single channel greyscale representing the luminance in the red channel of the original image:

Example

Manipulating images

- This functionality allows us to manipulate imaging data in the same way we would any NumPy array, which is quite powerful.
- For example, it now becomes trivial to flip, rotate or trim an image.
- Of course, if we have NumPy data that could be visualised by a raster map, we can create new images from scratch
- We will later use skimage to explore more image processing.

Summary

- We have seen how to save and load data.
- We have seen the difference between structured and unstructured data, and obtained data from webpages.
- We have manipulated images.