ENGG1811: Computing For Engineers

2025 Term 3

(even more on!) NumPy

Week 8: Monday 3^{rd} November, 2025

Monday 14:00 - 16:00 | HarpM15570

Today

Housekeeping

File-Handling

(even more on) Numpy

Lab Tips

Appendix

Housekeeping

Reminders

- Assignment 2 is out!
  - Due 5pm Friday Week 10 (21/11/2025)
  - Check the course website for details
  - Same advice as last time: start early and seek help when needed
- Self-directed lab 2 on Matlab
  - Due 5pm Monday Week 11 (24/11/2025)
  - Very similar to the first self-directed lab: a sequence of videos followed by exercises on MATLAB.
  - No need to submit; just complete the exercises on MATLAB Grader; your mark will be the final completion status after the due date.
- Note these are due around the same time, so don't leave them both to the last minute.

File-Handling

File-Handling: Introduction

- File-handling is probably the most confusing concept in ENGG1811
  - Your fundamentals need to be strong
  - Lists, indexing, for-loops, strings, will all be leveraged when dealing with files

File-Handling: Introduction (Cont)

- With that being said, files are just really big strings!
  - Suppose we had a text file with the following riveting content:

```
Hello world 1
Hello world 2
```

  - Then this is nothing more than: `"Hello world 1\nHello world 2"`
- So what *are* the differences?
  - A file makes the newline character, `\n`, invisible (we are really inserting this every time we press "enter" on our keyboards!)
  - Note very carefully, that there is no space between `1`, `\n` and `Hello`.

File-Handling: Files to Strings Examples

- Lets go through a few examples of this to make sure it's clear:
  - `"Hello Hello Hello 3\n\nHello 1"`

    ```
    Hello Hello Hello 3

    Hello 1
    ```

  - `"Hello"`

    ```
    Hello
    ```

  - `"\n\n"`

    ```


    ```

File-Handling: Getting Python to Open Files

- Structure:

  ```
  with open("filename", mode = "r") as f:
      ## some sort of code here
  ```

  - `with` is a key-word that does some house-keeping in the background: safely closes the file after you are done using it
  - `open` is a key-word that finds the file named `"filename"` and converts it into a stream of data for us to use
  - `as f` is a shorthand, just like how we write `import numpy as np`
- What is `mode` doing?
  - There are different modes we can set when accessing a file:
    - `"r"` - read only
    - `"w"` - write only
    - `"a"` - append only
    - `"r+"` - read and write

File-Handling Function Summary

To summarise, these are the only functions you'll ever need when working with files:

- `open(file, mode='r')`: opens a file.
- `np.loadtxt("filename.txt", dtype=str)`: loads the file into a 2D NumPy array (each row is a line, each column is data separated by spaces).
- `split()`: splits a string (by default on spaces) into a list of words.
- `readline()`: reads a single line from a file.
- `readlines()`: reads all lines from a file and returns them as a list of strings.

Examples on the next slides!

File-Handling: Example

- Suppose we have the following file called `text_file.txt`:

```
This is line 1
This is line 2
```

- We access this in Python using:

```python
with open("text_file.txt", mode = "r") as f:
    lines = f.readlines()
```

  - Then `lines` will be the list `["This is line 1", "This is line 2"]`.
  - The `realines()` function converts the file into a string, and then will append each line (it knows where a line begins/ends because of the `\n` character).

File-Handling: Example (Cont I)

- Remember that `lines` is the list `["This is line 1", "This is line 2"]`.
  - Sometimes getting a list of all the lines might be enough, but what if we want to separate each word from every line?
- We will access each word using the following for-loop:

```
all_words = []
for i in range(len(lines)):
    words_from_line = lines[i].split()
    all_words.append(words_from_line)
```

  - `lines[0].split() = ["This", "is", "line", "1"]`.
  - `lines[1].split() = ["This", "is", "line", "2"]`.
- **Question:** How do we think `.split()` knows when a word ends?

File-Handling: Example (Cont II)

- We now have
  `all_words = [["This", "is", "line", "1"],`
  `             ["This", "is", "line", "2"]]` We could have also gotten this from
  `all_words = np.loadtxt("text_file.txt", dtype=str)`

  - **Problem**: Python will assume *everything* is a string/character, even if it's something else!
  - it is on us to convert data-types where needed — python *will not* do it for us automatically
- We can do this by the following method:

```
for word_line in all_words:
    word_line[-1] = int(world_line[-1])
```

- This gives us:

```
all_words = [["This", "is", "line", 1],
             ["This", "is", "line", 2]]
```

File-Handling: Example (Cont III)

- Again, suppose we have the following file:

```
This is line 1
This is line 2
```

- Using `readline()`, we get one line at a time:

```python
line1 = f.readline()   # "This is line 1"
line2 = f.readline()   # "This is line 2"
```

- We can then use `split` to get words:

```python
# ["This", "is", "line", "1"]
words_line1 = line1.split()
```

File-Handling: Closing Remarks

- It is one thing to understand all of this as I go through it, and very different doing this yourself
  - Difficulty is not in any individual step, but doing/remembering everything in the right order and knowing what to do if (and likely when) you get an error
- **Tips**:
  - If you understand what each operation is doing, less likely for you to forget it or to have a nasty surprise from the output
  - Do 'all' the file-handling *first*
    - Process it all, put them nicely into well-labelled lists, and convert data-types
    - You are much more likely to make an error if you try processing and 'analysing' at the same time!

(even more on) Numpy

NumPy: Shape & Size

1	2	3	4
5	6	7	8
9	10	11	12

All examples are using the top-right array

- `np.shape(array)`
  - Gives the shape — the number of rows and columns — of the given array
  - `np.shape(array) = (3, 4)`
- `np.size(array)`
  - Gives the area (number of elements) of the given array
  - `np.size(array) = 12`

NumPy: Ravel

1	2	3	4
5	6	7	8
9	10	11	12

- `np.ravel(array)`
  Using the array at the top-right
  - Unravels the data into a single horizontal array (list) — converts an n dimensional array into a 1 dimensional array
  - `np.ravel(array) = [1, 2, 3, 4, 5, 6,`
    `7, 8, 9, 10, 11, 12]`
- **Question:** Can we achieve the same result using the reshape function? Why or why not?

NumPy: Slicing

1	2	3	4
5	6	7	8
9	10	11	12

We can slice out a subarray using the format:

```
array[row_start:row_end:row_step,
      column_start:column_end:column_step]
```

Here are some examples:

- ```
  array[1:, :2] = array([[ 5,  6],
                         [ 9, 10]])
  ```
- ```
  array[::-1, :] = array([[ 9, 10, 11, 12],
                          [ 5,  6,  7,  8],
                          [ 1,  2,  3,  4]])
  ```
- ```
  array[::3, :2:] = array([[1, 2]])
  ```
  ```

NumPy: ix_

1	2	3	4
5	6	7	8
9	10	11	12

- **Recall:** When we wish to use boolean indexing, we must create an array of the same shape as the array we wish to index, and then fill each entry with either a `True` or `False` value
- For the array at the top right, we might have to make (from scratch) the following array to index it:

```
mask = np.array([[False, False, False, False],
                 [False, True,  True,  True],
                 [False, True,  True,  True]])
```

- **Question:** What is the output of `array[mask]`?

NumPy: ix_ (Cont I)

1	2	3	4
5	6	7	8
9	10	11	12

- This is a slightly burdensome procedure — can we be a bit lazier?
  - Imagine having to create your own boolean array every time you want to do some simple indexing
- Fortunately, we can!
  - We have two options: we can select all the rows, and then boolean index on the columns **OR** we can select all the columns, and boolean index on the rows
  - **Question I**: What is the output of `array[[False, True, True], :]` ?
  - **Question II**: What is the output of `array[:, [False, True, True, True]]`?

NumPy: ix_ (Cont II)

1	2	3	4
5	6	7	8
9	10	11	12

Limitation

What if we want to boolean index both the rows and columns at the same time?

- What's actually stopping us? Let's try

  ```
  array[[False, True, True],
        [False, True, True, True]]
  ```

  Question: What should this *intuitively* give us?

NumPy: ix_ (Cont III)

False	False	False	False
True	True	True	True
True	True	True	True

&

False	True	True	True
False	True	True	True
False	True	True	True

=

False	False	False	False
False	True	True	True
False	True	True	True

Figure: Intuitive results

- Annoyingly, we get this back instead:

Error Message From Above

```
IndexError: shape mismatch: indexing arrays could
not be broadcast together with shapes (2,) (3,)
```

NumPy: ix_ (Cont IV)

1	2	3	4
5	6	7	8
9	10	11	12

- `np.ix_(rows_boolean_array,`
  `columns_boolean_array)`
  - `rows_boolean_array` is the
    boolean array which selects which rows you want to keep
  - `columns_boolean_array` is the
    boolean array which selects which columns you want to keep
  - creates a boolean array with desired rows and columns
- Use case:

```
mask = np.ix_([False, True, True],
              [False, True, True, True])
array[mask] = np.array([[6, 7, 8],
                        [10, 11, 12]])
```

This solves our problem, and we can now be slightly lazier when it comes to boolean indexing (so long as we remember the `ix_` function!)

NumPy: Diff

- `numpy.diff(array)`
  - If we have an array $[x_0, x_1, \ldots, x_{n-1}]$ then this will return us back the list

$$[x_1 - x_0, x_2 - x_1, \ldots, x_{i+1} - x_i, \ldots, x_{n-1} - x_{n-2}].$$

  - In plain English, it is calculating for us the (forward) difference between consecutive elements of a given `array`
- Example:

```
array = np.array([3, 7, 4, 9, 4, -1])
np.diff(array) = array([ 4, -3, 5, -5, -5])
```

NumPy: Unique

1	1	3	2
2	2	1	3
4	3	4	3

- `np.unique(array, return_counts = False)`
  - Returns an array of all the unique values of a given `array`
  - `return_counts = False` is an *optional* argument — if it is set to `True`, it will also return a list which gives back the amount of times each unique element occurs
- Example:

```
unique_vals, counts = np.unique(array,
                                return_counts = True)
unique_vals = np.array([1, 2, 3, 4])
counts      = np.array([3, 3, 4, 2])
```

Lab Tips

Lab Tips

- Exercise 1:
  - Convert your numbers from string to float.
  - Don't average all 5 numbers each time; the second number in the first line tells you how many to use.
  - The first 10 files are named `temp0x.txt`, not `tempx.txt`.
- Exercise 2:
  - Check your answers on the course website or in the Part A starter code before getting marked off.
  - No loops allowed in Exercise 2.
  - Watch for questions that specify a method; you must use that method.
  - For Question 4, use a different approach to Question 8 from Exercise 2 in Week 7 (i.e. if you used slicing before, use *reshape* here).

Appendix

Appendix Explanation

- The next slides will explain *broadcasting*, a very elegant NumPy technique.
- Instead of file handling, there used to be a broadcasting exercise in this week's lab.

NumPy: Broadcasting

- One of the many neat features of NumPy is that it allows arrays of different sizes to work together
- Intuitively, what should be the answer of adding these two arrays together?
  - Example 1:

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} \quad + \quad \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} \quad = \quad ?$$

  - Example 2:

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} \quad + \quad \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline \end{array} \quad = \quad ?$$

NumPy: Broadcasting (Cont I)

Here is what NumPy is *really* doing for the above examples

- Example 1:

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} + \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} + \begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 1 \\ \hline 2 & 2 & 2 & 2 \\ \hline 3 & 3 & 3 & 3 \\ \hline \end{array}$$

- Example 2:

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} + \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline \end{array} =$$

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} + \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ \hline \end{array}$$

NumPy: Broadcasting (Cont II)

Let's do a few more examples:

- Example 3:

$$\begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} \quad - \quad \begin{array}{|c|} \hline 1 \\ \hline \end{array} \quad = \quad ?$$

- Example 4 (perhaps this one won't be intuitive):

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} \quad \times \quad \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} \quad = \quad ?$$

NumPy: Broadcasting (Cont III)

- Example 3:

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array} - \begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 0 & 1 & 2 & 3 \\ \hline 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ \hline \end{array}$$

- Example 4:

$$\underset{(3,1)}{\begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array}} \times \underset{(1,3)}{\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}} = \underset{(3,\underline{3})}{\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline 3 & 3 & 3 \\ \hline \end{array}} \times \underset{(\underline{3},3)}{\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline \end{array}} = \underset{(3,3)}{\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 2 & 4 & 6 \\ \hline 3 & 6 & 9 \\ \hline \end{array}}$$

NumPy: Broadcasting (Cont IV)

- Hopefully from these examples, we can see that broadcasting is only going to work if the arrays are **compatible**:
  1. The two arrays share a dimension of the same size and *,
  2. One of the dimensions — for at least one of the arrays — is one
- **Examples**:
  - 2×3 and 1×3 are compatible — they satisfy both conditions
  - 5×89 and 5×1 are compatible — they satisfy both conditions
  - 2×3 and 3×4 are *not* compatible — why?
  - 5×6 and 5×2 are *not compatible* — why?
- **Question:** Why did I put an asterisk over the 'and'? What's the exception?

NumPy: Broadcasting Exercise

If you were curious, this was the exercise that was left out:

- Two NumPy arrays, `pos` (which has a shape of $(6, 2)$) and `ref`. One interpretation of the `pos` array is as a list of positions: `pos[0, 0]` and `pos[0, 1]` could store the x and $y$coordinates of an object, respectively.
- We ask you to compute the distance between each of the 6 positions in `pos` and the reference position. As a reminder, if we have an arbitrary position (x, y) and reference position (a, b), then the distance between them is:

$$\sqrt{(x-a)^2 + (y-b)^2}$$

- Complete this task using functions from the `numpy` library. You are not allowed to use loops. The expected answer and starter code is given on the next slide.

NumPy: Broadcasting Exercise

The expected answer and starter code:

```
import numpy as np

pos = np.array([[   1.72,   2.56],
                [   0.24,   5.67],
                [  -1.24,   5.45],
                [  -3.17,  -0.23],
                [   1.17,  -1.23],
                [   1.12,   1.08]])
ref = np.array([1.22, 1.18])

# The expected answer is approximately:
# [ 1.468,  4.596,  4.928,  4.611,  2.410,  0.141 ]
```

If you attempt it, feel free to get feedback on it!

Feedback

Feel free to provide anonymous feedback about the lab!

Feedback Form