ENGG1811 Computing for Engineers

Week 8 : numpy 2

- elementwise operations
- numpy (Broadcasting, Slicing, Boolean indexing)
- Mutable and immutable data types

Arithmetic operators

- You can use +, -, *, /, ** on two numpy arrays
  - They perform elementwise operations
  - See the next two slides for illustration
- The shapes of these arrays are required to be **compatible**.
- We will first consider the case where both arrays have the same shape
  - Code in numpy_arith_1.py

Elementwise multiplication

```
array1 = np.array([ [-3.2,  0,  0.5, 5.8],
                    [   6, -4,  6.2, 7.1],
                    [ 3.8,  5,  2.7, 3.7]])

array2 = np.array([ [-1.2,  2, -3.1, 0.0],
                    [   4, -5,  3.5, 7.1],
                    [ 2.7,  2,  1.7, 3.4]])
```

```
array_mul = array1 * array2 # NOT matrix multiplication

□array([[ 3.84,  0.   , -1.55,  0.   ],
        [24.   , 20.   , 21.7 , 50.41],
        [10.26, 10.   ,  4.59, 12.58]])
```

Elementwise division

```
array1 = np.array([ [-3.2,  0,  0.5, 5.8],
                    [   6, -4,  6.2, 7.1],
                    [ 3.8,  5,  2.7, 3.7]])

array2 = np.array([ [-1.2,  2, -3.1, 0.0],
                    [   4, -5,  3.5, 7.1],
                    [ 2.7,  2,  1.7, 3.4]])
```

```
array_div = array1 / array2


array([[ 2.667,  0.        , -0.161,        inf],
       [ 1.5  ,  0.8       ,  1.771,         1.],
       [ 1.407,  2.5       ,    1.8,      1.088]])
```

Exercise: A simple survey

- You have conducted a survey.
  - The survey has 3 questions.
  - Each question has only two possible choices: Yes and No.
  - Each respondent can answer any number of questions.
- The results are in the table below. You want to determine the fraction of Yes votes for each question.

	Y	N
Q1	21	15
Q2	34	23
Q3	17	31

- Define the following two numpy arrays

```
yes_votes = np.array([21,34,17])

no_votes = np.array([15,23,31])
```

- Use these two arrays and numpy elementwise computation to compute the fraction of Yes votes. The expected answer is:

$$\left[\frac{21}{21+15}, \frac{34}{34+23}, \frac{17}{17+31}\right]$$

File:

numpy_arith_1_prelim.py

Exercise: A simple survey (Discussion)

- Lesson learnt: If you put the data in the right way, then you can use elementwise computation to simplify the code

	Y	N
Q1	21	15
Q2	34	23
Q3	17	31

Good way:

```
yes_votes = np.array([21,34,17])

no_votes = np.array([15,23,31])
```

Need only one line of code!

Bad way:

□✗ [21,15],[34,23],[17,31]

More on numpy arithmetic operators

- You have seen that you can use the numpy arithmetic operators on two arrays of the same shape

- You can also use the numpy arithmetic operators on two arrays when
  - One array is a scalar
  - The other is a numpy array of any shape

- Let us look at the examples in numpy_arith_2.py

Elementwise division: an array and a scalar

```
array1 = np.array([ [-3.2,  1,  0.5, 5.8],
                    [   6, -4,  6.2, 7.1],
                    [ 3.8,  5,  2.7, 3.7]])
```

```
array_div_1 = array1 / 2.0
array([[-1.6 ,  0.5 ,  0.25,  2.9 ],
       [ 3.  , -2.  ,  3.1 ,  3.55],
       [ 1.9 ,  2.5 ,  1.35,  1.85]])
```

```
array_div_2 = 2.0 / array1
□array([[-0.625,  2.,       4.,  0.345],
       [  0.333, -0.5,  0.322,  0.282],
       [  0.526,  0.4,  0.741,  0.541]])
```

Exercise

- If you drop an object from a height of *h0* and if the air resistance is small, then the height of the object at time *t* is

$$h0 - 0.5 * g * t^2$$

where g is the acceleration due to gravity

- For given h0 and g, you want to compute the height of the object at t = 0, 2, 4, 6, 8

Exercise: Hint

- Numpy array
  - □time_array = np.array([0,2,4,6,8])

- The following hint for array [0,2,4]

[0 , 2 , 4]

$$[\ h0 - 0.5 * g * 0^2 \quad , h0 - 0.5 * g * 2^2 \quad , h0 - 0.5 * g * 4^2]$$

⇧ final result wanted

⇩ Work backwards.

$$= h0 - [\ 0.5 * g * 0^2 \quad , 0.5 * g * 2^2 \ ', 0.5 * g * 4^2]$$

Until you use [0,2,4]

- Complete the exercise in numpy_arith_2_prelim.py

Mathematical functions

- The numpy mathematical functions are documented here:
  - https://docs.scipy.org/doc/numpy/reference/routines.math.html
- Example: sin, cos, asin, log, exp, sqrt, absolute
- Notes:
  - You need to append the library name, say you import numpy as np, then np.cos etc.
  - They are different to those in the math library
  - They are **elementwise operation**. The output is an array of the same size as input and the operation is applied to each element (illustrated on the next slide)
- Code in numpy_math_func.py

Elementwise operation

```
array2 = np.array([[-1.2,  2. , -3.1,  4.5],
                   [ 4. , -5. ,  3.5,  7.1],
                   [ 2.7,  9. ,  1.7,  3.4]])

array2_sin = np.sin(array2)
array([[-0.93203909,  0.90929743, -0.04158066, -0.97753012],
       [-0.7568025 ,  0.95892427, -0.35078323,  0.72896904],
       [ 0.42737988,  0.41211849,  0.99166481, -0.2555411 ]])
```

sin(4.5)

sin(2.7)

sin(1.7)

Recap: numpy

- numpy has a lot of useful functions for data analysis
  - E.g., mean(), sum() etc.

- Many numpy functions allow you to do computation without using loops
  - Reason: numpy functions are implemented with speed in mind so they are often faster than the equivalent Python code that you can write to do the same task
  - The maxim: Use numpy function as much as possible

Key topics

- Broadcasting
- Slicing
- Boolean indexing

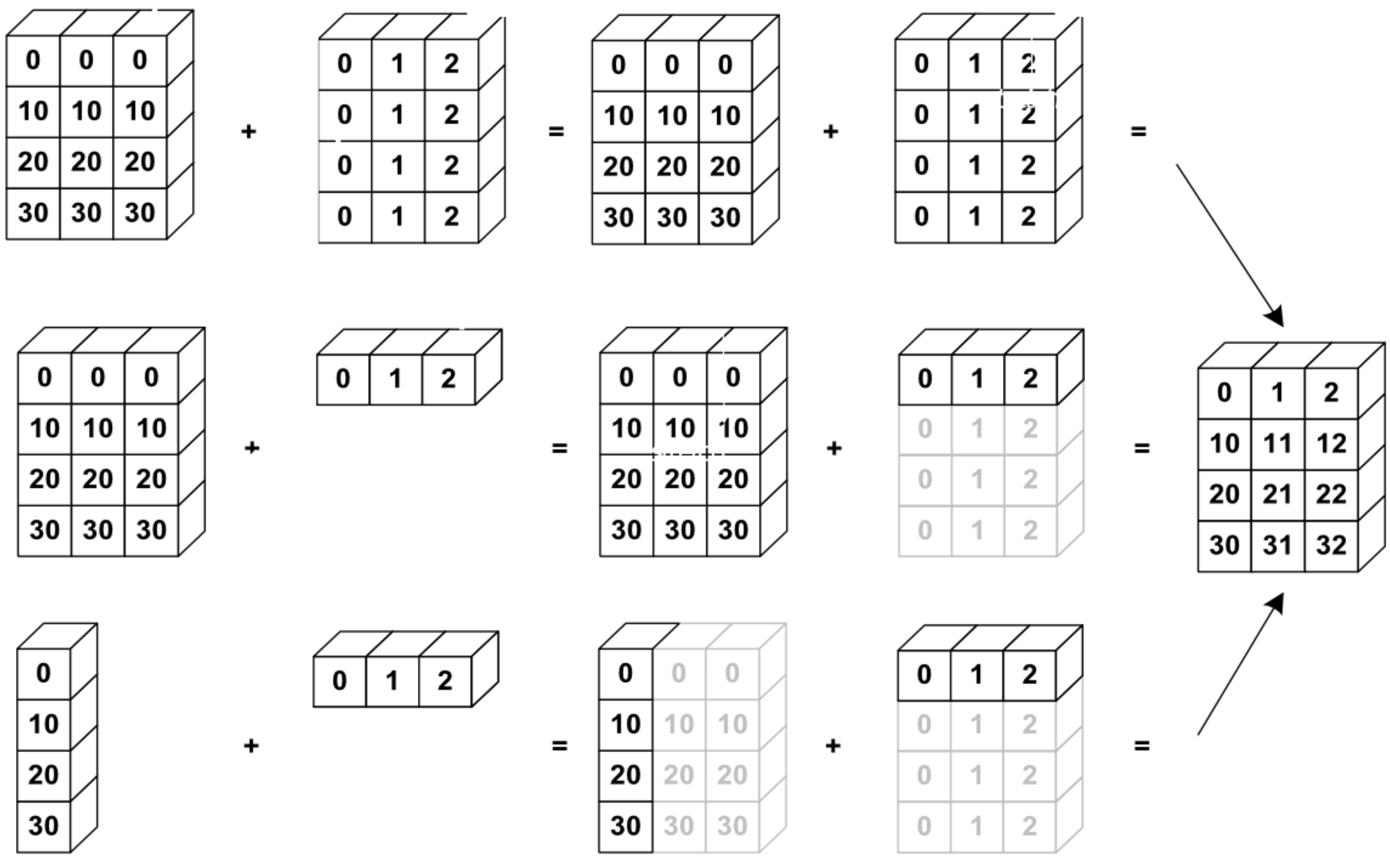
Broadcasting rules

- You have seen that you can use numpy elementwise arithmetic operators +, -, *, / and ** for
  - Two arrays of the same shape
  - An array and a scalar

- In general, numpy arithmetic operators can be used on two arrays as long as their shapes are compatible
  - Informal view: Next slide
  - Formally, compatibility is defined according to the numpy **broadcasting rules**

- The broadcasting rules were modified from:
  - https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html
    - You may wish to read the examples in this document to further understand the broadcasting rules

Broadcast: informal view



Source: https://scipy-lectures.org/intro/numpy/operations.html#broadcasting

Broadcasting Rule 1

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

```
In [32]: a1
Out[32]:
array([[ 1.1,  2.2,  3.3],
       [ 3.1,  3.2,  3.3]])

In [33]: a1.shape
Out[33]: (2, 3)
```

- Dimension of a1 is 2
  - a1.ndim shows the dimension

```
In [34]: b1
Out[34]: array([10, 20, 30])

In [35]: b1.shape
Out[35]: (3,)
```

- Dimension of b1 is 1
- After Rule 1, the shape of b1 goes from (3,) to (1,3)

Broadcasting Rule 2

- Rule 2: If the shape of the two arrays does not match in any dimension, the axes whose shape is equal to 1 are stretched to match the shape of the other array.

```
In [32]: a1
Out[32]:
array([[ 1.1,  2.2,  3.3],
       [ 3.1,  3.2,  3.3]])

In [33]: a1.shape
Out[33]: (2, 3)
```

```
In [34]: b1
Out[34]: array([10, 20, 30])

In [35]: b1.shape
Out[35]: (3,)
```

- Shape of a1 is (2,3)
- Shape of b1 after Rule 1 is (1,3)
- Axis 0 of b1 is 1, it is stretched to 2 to match a1
- After Rule 2, the shape of b1 becomes (2,3)

Broadcasting Rule 3

- Rule 3: If the two arrays have the same shape, then they are compatible; otherwise they are not.

```
In [32]: a1
Out[32]:
array([[ 1.1,  2.2,  3.3],
       [ 3.1,  3.2,  3.3]])

In [33]: a1.shape
Out[33]: (2, 3)
```

```
In [34]: b1
Out[34]: array([10, 20, 30])

In [35]: b1.shape
Out[35]: (3,)
```

- Example:
- Shape of a1 is (2,3)
- Shape of b1 after Rule 2 is (2,3)

- Identical shape, hence compatible

Operating on broadcast compatible arrays (1)

a1 is

[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

b1 is

[10, 20, 30]

Broadcast b1 to shape (2,3)

[[10, 20, 30],
 [10, 20, 30]]

+

The result of a1 + b1 is

[[11.1, 22.2, 33.3],
 [13.1, 23.2, 33.3]]

See numpy_broadcast.py

Informal view

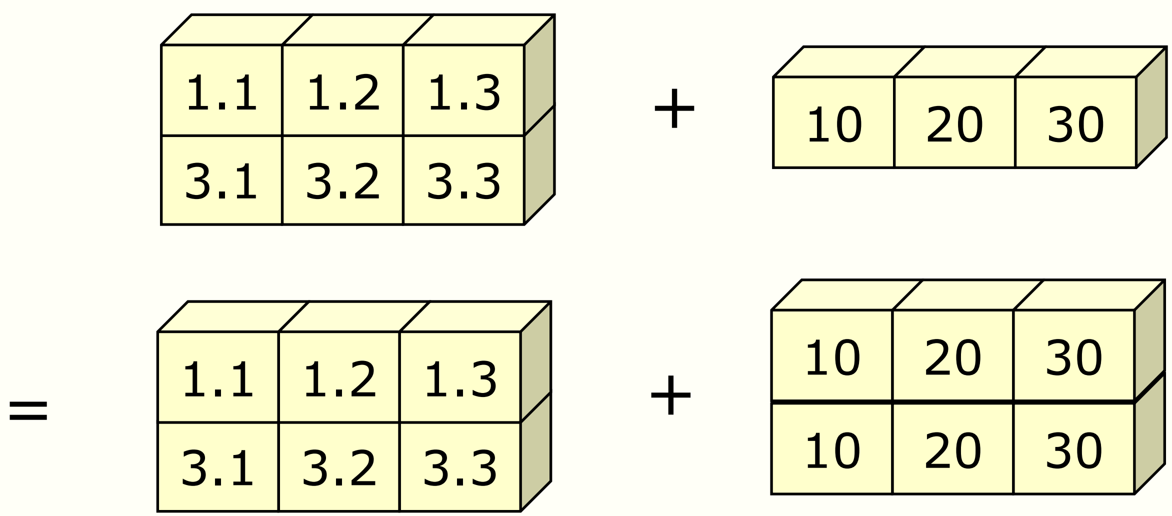
a1 is

[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

b1 is

[10, 20, 30]

Broadcast rule 1 makes b1 goes from (3,) to (1,3). Intuitively, for the purpose of broadcasting, a 1-d array should be thought of a 2-d array with one row



Operating on broadcast compatible arrays (2)

a1 is
[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

c1 is
10

Broadcast c1 to shape (2,3)

[[10, 10, 10],
 [10, 10, 10]]

+

The result of a1 + c1 is

[[11.1, 12.2, 13.3],
 [13.1, 13.2, 13.3]]

See numpy_broadcast.py

Broadcasting rules

- You can generalise the example in the previous slide to show that a scalar is compatible to numpy array of any shape

- Broadcast rules are general and they cover the two special cases we mentioned earlier
  - Two arrays of identical shape
  - A scalar and an array of any shape

Exercise 1

- Given

```
a1 = np.array([[1.1, 2.2, 3.3],[3.1, 3.2, 3.3]])
d1 = np.array([[100], [200]])
```

Predict what a1 + d1 should be without running the code in numpy_broadcast.py

We will run the cell in numpy_broadcast.py later so you can check your prediction

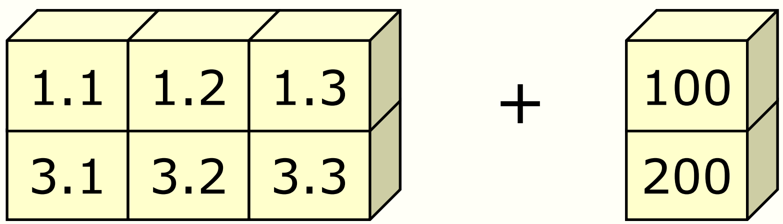
Informal view

a1 is

```
[[ 1.1,  2.2,  3.3],
 [ 3.1,  3.2,  3.3]]
```

d1 is

```
np.array([[100], [200]])
```



Compatible arrays

d1 is np.array([[100], [200]])

Its shape is (2,1)

a1 is

[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

Shape (2,3)

Step 1: No change

Shape (2,1)

Rule 2: Stretching

Shape (2,3)

[[100, 100, 100],
 [200, 200, 200]]

Compatible

Exercise 2

- Given

```
a1 = np.array([[1.1, 2.2, 3.3],[3.1, 3.2, 3.3]])
e1 = np.array([100, 200])
```

Are the arrays a1 and e1 compatible?

We will run the cell in numpy_broadcast.py later so you can check your prediction

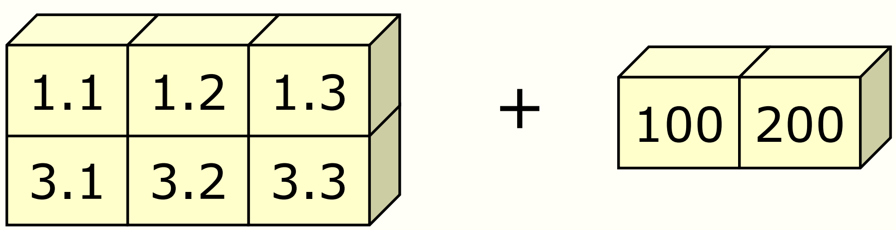
Informal view

a1 is

```
[[ 1.1,  2.2,  3.3],
 [ 3.1,  3.2,  3.3]]
```

e1 is

```
np.array([100, 200])
```



Incompatible arrays

a1 is

[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

Shape (2,3)

e1 has shape (2,)

Rule 1: Padding on the left

Shape (1,2)

Rule 2: Stretching

Shape (2,2)

See numpy_broadcast.py

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

Broadcast – round up

- There is one additional example in the last cell of numpy_broadcast.py

- There is an exercise in numpy_broadcast_prelim.py

Key topics

- Broadcasting
- Slicing
- Boolean indexing

numpy slicing

- Slicing is a very useful method to select a portion of data
  - E.g. You have a 2-dimension array where each column contains the data for a day of the week. You may want to study the data over the weekdays. This means you need a way to extract 5 columns of the data
- You have learn about slicing a list
  - You can use the list slicing methods on numpy array too
- numpy has some additional methods to select elements
- Examples in:
  - numpy_slicing_1.py for one dimensional arrays
  - numpy_slicing_2.py for two dimensional arrays

1-D array: select specific elements

```
index
0      1      2     3      4      5      6
In [11]: b = np.array([11, 23, 7, 5, 29, 37, 43])

In [11]:

In [12]: b[ [3, 6, 2] ]
Out[12]: array([ 5, 43,  7])
```

numpy_slicing_1.py

2-D array: Slicing out a rectangular block (1)

```
In [34]: c
Out[34]:
array([[11, 23,  7,  5, 29, 37, 43],
       [13, 57, 71, 26, 31, 47, 53],
       [17, 67, 73,  3,  2, 19, 31],
       [41, 53, 59, 61, 91, 79, 83]])

In [35]: c[:,2:4] # columns with indices 2 and 3
Out[35]:
array([[ 7,  5],
       [71, 26],
       [73,  3],
       [59, 61]])
```

numpy_slicing_2.py

2-D array: Slicing out a rectangular block (2)

```
In [25]: c
Out[25]:
array([[11, 23,  7,  5, 29, 37, 43],
       [13, 57, 71, 26, 31, 47, 53],
       [17, 67, 73,  3,  2, 19, 31],
       [41, 53, 59, 61, 91, 79, 83]])

In [26]: c[-2:,-3:]  # Last 2 rows and last 3 columns
Out[26]:
array([[ 2, 19, 31],
       [91, 79, 83]])
```

You can use :: notation too

E.g. Try c[1::2,0::2]

numpy_slicing_2.py

2-D array: Slicing with np.ix_

Column index
2 3 6

Row index
1
3

```
In [7]: c
Out[7]:
array([[11, 23,  7,  5, 29, 37, 43],
       [13, 57, 71, 26, 31, 47, 53],
       [17, 67, 73,  3,  2, 19, 31],
       [41, 53, 59, 61, 91, 79, 83]])

In [8]: c[ np.ix_([1,3],[3, 6, 2])]
Out[8]:
array([[26, 53, 71],
       [61, 83, 59]])
```

From row:
1
3

From column: 3 6 2

[[c[1,3], c[1,6], c[1,2]] ,
[c[3,3], c[3,6], c[3,2]]]

numpy_slicing_2.py

Put specific elements in a 1-D array

```
In [37]: c
Out[37]:
array([[11, 23,  7,  5, 29, 37, 43],
       [13, 57, 71, 26, 31, 47, 53],
       [17, 67, 73,  3,  2, 19, 31],
       [41, 53, 59, 61, 91, 79, 83]])

In [38]: c[[3,2,0],[-2,2,3]] # array([c[3,-2], c[2,2], c[0,3]])
Out[38]: array([79, 73,  5])
```

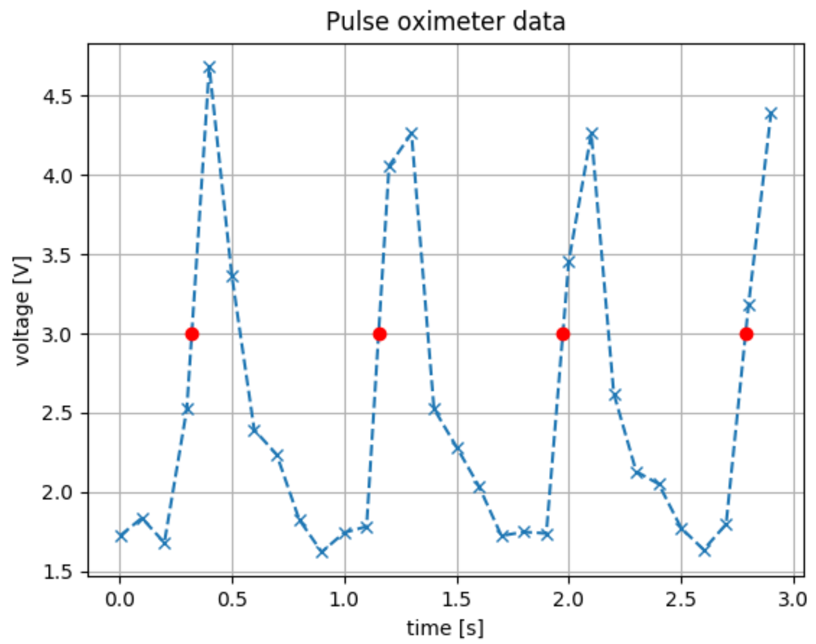
[c[3,-2], c[2,2], c[0,3]]

numpy_slicing_2.py

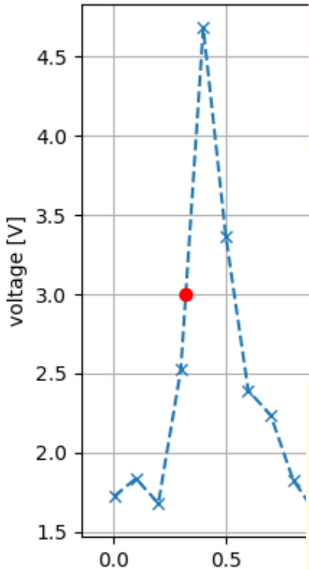
Exercise: Counting heart beats

- In the lab in Week 5, you counted the number of heart beats by counting the number of times the voltage crosses the 3V threshold and is increasing
- How can you do this in numpy without using for?

Template is in numpy_heart_prelim.py



Exercise: Counting heart beats (Hint)




```
Voltage data
[1.72, 1.84,  1.68,  2.52,  4.68,  3.37, ...
```

1.72 < 3 and 1.84 > 3

1.84 < 3 and 1.68 > 3

```
Use slicing to get 2 arrays:
```

Is it less than 3?

```
[1.72,  1.84,  1.68,  2.52,  4.68,  3.37,  ...
```

Is it bigger than 3?

```
[1.84,  1.68,  2.52,  4.68,  3.37,  2.39,  ...
```

```
[False, False, False,  True,  False, False,...
```

Need numpy.logical_and()

Key topics

- Broadcasting
- Slicing
- Boolean indexing

Boolean indexing

- This indexing method uses Boolean expressions to select elements in an array

- Useful for data analysis

- Example:
  - numpy_boolean_indexing_1.py

Boolean indexing

This example is in numpy_boolean_indexing_1.py

```
array1               [0.3,    0.4,   1.4,    1.7,    0.1]
boo_array1           [False, True,  True,  False,   True]

array1[boo_array1]       [0.4, 1.4, 0.1]
# Note: array1 and boo_array1 have the same shape
```

```
array1               [0.3,    0.4,   1.4,    1.7,    0.1]
boo_array2           [True, False, False,  False,   True]

array1[boo_array2]       [0.3, 0.1]
```

```
If True, then the entry is selected.
Identical shape requirement.
```

Boolean indexing (Quiz 1)

This quiz is in numpy_boolean_indexing_1.py

```
array1          [0.3,   0.4,   1.4,   1.7,   0.1]
array1 >= 1     [False, False, True,  True,  False]


# Think about what the following would give before
# trying it out
array1[array1 >= 1]
```

Boolean indexing (Quiz 2)

This quiz is in numpy_boolean_indexing_1.py

```
array1            [0.3,   0.4,   1.4,   1.7,   0.1]
array1 >= 1       [False, False, True,  True,  False]


array2            [1.1,   0.1,    0.8,  0.3,  1.5]


# Think about what the following would give before
# trying it out
array2[array1 >= 1]
```

Boolean indexing (Quiz 3)

This quiz is in numpy_boolean_indexing_1.py

```
temp_array contains temperature measurements

[24.5, 31.5, 27.4, 34.1, 33.2, 28.9, 27.9, 34.8]
```

```
week_array        [1, 2, 3, 4, 5, 6, 7, 8]
# Temperature in Week 1 is 24.5
# Temperature in Week 2 is 31.5
```

```
Use Boolean indexing to find the week numbers that have
temperature >= 30
Expect: [2, 4, 5, 8]
```

Boolean indexing (Further examples)

- numpy_boolean_indexing_2.py for 1 dimensional arrays
  - This introduces Boolean operators:
    - &, |, ~ (for AND, OR and NOT respectively)
  - Using assignment with Boolean indexing

- numpy_boolean_indexing_3.py for 2 dimensional arrays
  - There is also a quiz
  - Quiz answer:

Forum exercise

- This is a forum exercise which puts together what you have learnt today
- Consider the following array which contains some sensor measurements

```
np.array(
[[ 0.4,  0.4,  0.6,  0.5,  0.7,  0.8,  0.8,  0.5,  0.0,  0.7],
 [ 0.4,  0.4,  0.8,  0.4,  0.8,  1.1,  0.9,  0.4,  1.1,  1.1],
 [ 0.4,  1.1,  0.8,  0.3,  0.7,  1.1,  0.9,  0.5,  1.1,  0.6],
 [ 0.4,  0.5,  0.6,  0.4,  0.9,  1.2,  0.8,  0.5,  0.1,  0.6],
 [ 0.3,  0.4,  0.8,  0.3,  0.8,  0.7,  0.7,  0.4,  0.2,  0.7]]
)
```

- Each row contain the readings from a sensor
- Each column contains the readings at a specific time
- (To be continued on the next page)

Forum exercise (cont'd)

- You want to compute the average at each time from the five sensor readings
- If you use all the data, you would use
  - numpy.mean(, axis = 0)

```
np.array(
[[ 0.4,  0.4,  0.6,  0.5,  0.7,  0.8,  0.8,  0.5,  0.0,  0.7],
 [ 0.4,  0.4,  0.8,  0.4,  0.8,  1.1,  0.9,  0.4,  1.1,  1.1],
 [ 0.4,  1.1,  0.8,  0.3,  0.7,  1.1,  0.9,  0.5,  1.1,  0.6],
 [ 0.4,  0.5,  0.6,  0.4,  0.9,  1.2,  0.8,  0.5,  0.1,  0.6],
 [ 0.3,  0.4,  0.8,  0.3,  0.8,  0.7,  0.7,  0.4,  0.2,  0.7]]
)
```

- However, you have reasons to believe the sensor readings which are >= 1 are due to faulty sensors and you want to exclude them when you compute the average
- (To be continued on the next page)

Forum exercise (cont'd)

- The array on yellow background shows the final result that you want

```
np.array(
[[ 0.4,  0.4,  0.6,  0.5,  0.7,  0.8,  0.8,  0.5,  0.0,  0.7],
 [ 0.4,  0.4,  0.8,  0.4,  0.8,  1.1,  0.9,  0.4,  1.1,  1.1],
 [ 0.4,  1.1,  0.8,  0.3,  0.7,  1.1,  0.9,  0.5,  1.1,  0.6],
 [ 0.4,  0.5,  0.6,  0.4,  0.9,  1.2,  0.8,  0.5,  0.1,  0.6],
 [ 0.3,  0.4,  0.8,  0.3,  0.8,  0.7,  0.7,  0.4,  0.2,  0.7]]
)
```

Average all 5 readings

Average of 0.8 and 0.7

Average of 0.0, 0.1 and 0.2

[**0.38**, 0.425, 0.72, 0.38, 0.78, **0.75**, 0.82, 0.46, **0.1**, 0.65]

Forum exercise (Hint)

- Hint: For each column, sum only entries that are less 1
- I used 5 lines of code to do that (no loops)

```
np.array(
[[ 0.4,  0.4,  0.6,  0.5,  0.7,  0.8,  0.8,  0.5,  0.0,  0.7],
 [ 0.4,  0.4,  0.8,  0.4,  0.8,  1.1,  0.9,  0.4,  1.1,  1.1],
 [ 0.4,  1.1,  0.8,  0.3,  0.7,  1.1,  0.9,  0.5,  1.1,  0.6],
 [ 0.4,  0.5,  0.6,  0.4,  0.9,  1.2,  0.8,  0.5,  0.1,  0.6],
 [ 0.3,  0.4,  0.8,  0.3,  0.8,  0.7,  0.7,  0.4,  0.2,  0.7]]
)
```

Average all 5 readings

Average of 0.8 and 0.7

Average of 0.0, 0.1 and 0.2

[**0.38**, 0.425, 0.72, 0.38, 0.78, **0.75**, 0.82, 0.46, **0.1**, 0.65]

Mutable and immutable data types

You can modify part of a list

- You can modify the elements in a list by assigning new values to them

```
In [11]: x = [11,22,33,43,55]

In [12]: x[3] = 44

In [13]: x
Out[13]: [11, 22, 33, 44, 55]

In [14]: x[2:4] = [37,47]

In [15]: x
Out[15]: [11, 22, 37, 47, 55]
```

String as a sequence of characters

```
In [12]: word = 'silly'

In [13]: word[0]
Out[13]: 's'

In [14]: word[1]
Out[14]: 'i'

In [15]: word[2]
Out[15]: 'l'

In [16]: word[3]
Out[16]: 'l'

In [17]: word[4]
Out[17]: 'y'
```

But you can’t modify part of a string

```
In [16]: word = 'silly'

In [17]: word[0] = 'b'
Traceback (most recent call last):

  File "<ipython-input-17-3b299587d77e>", line 1, in <module>
    word[0] = 'b'

TypeError: 'str' object does not support item assignment


In [18]: word = 'billy'
```

← Error

← You can’t change part of a string but you can assign an entirely new string

Tuples

- A tuple is a sequence of elements enclosed in ( )
- For example, the numpy where () function returns a tuple, the shape of a numpy function is given in a tuple
- Tuples are in many ways similar to lists
- But you can’t modify tuples

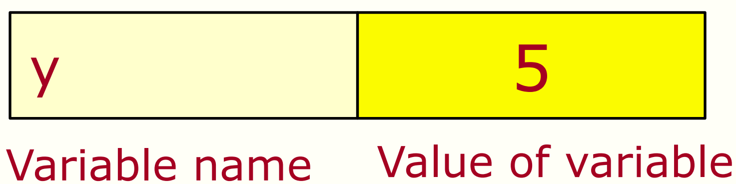
```
In [16]: t = (3,7,21) # A tuple with 3 elements

In [17]: t[1]
Out[17]: 7

In [18]: t[0:2]
Out[18]: (3, 7)

In [19]: t[1] = 10
Traceback (most recent call last):

  File "<ipython-input-19-5a9388635924>", line 1, in <module>
    t[1] = 10

TypeError: 'tuple' object does not support item assignment
```

Mutable and immutable data types

- The data types in Python are divided into 2 kinds
  - Mutable
  - Immutable
- Lists, numpy arrays (and dictionaries) are mutable
  - You can change the individual elements
- Strings are immutable
  - So are int, float, bool, tuples
- Note: dictionaries is a datatype in Python
  - E.g. We won’t be covering dictionaries in this course

Simplified mental picture on variables [From Week 1]

- Variables are stored in computer memory
- A variable has a name and a value
- A mental picture is:



A program manipulates variables to achieve its goal

Note: This is a simplified view. We will introduce the more accurate view later in the course.

How Python really stores variables

- In order to understand mutability, we need to understand how Python stores variables

```
In [100]: x = 5.5

In [101]: type(x)
Out[101]: float

In [102]: id(x)
Out[102]: 4728505688
```

Variable x is associated with an identifier

x	4728505688

The identifier is associated with the data type and a value.

For a list, a sequence of values

4728505688
float
5.5

Indirect association

The most important concept that you need to know is that a variable name is associated with its value via an identifier

Variable x is associated with an identifier

x	4728505688

The identifier is associated with the data type and a value.

For a list, a sequence of values

4728505688
float
5.5

Copying a mutable type

- We will look at and run the code in mut_1.py

```
list1 = [10,11,12,13]
list2 = list1
```

```
id of list1 =  4728419656
id of list2 =  4728419656
```

list1	4728419656
list2	4728419656

4728419656
list
...

Note: You will **not** get the same id shown above when you run the program. The essence is whether list1 or list2 have the same or different id

Lessons learnt

- The key lessons learnt from mut_1.py are
  - There are two different ways to copy lists

```
list2 = list1
```

Note: Both variable names are associated with the SAME list

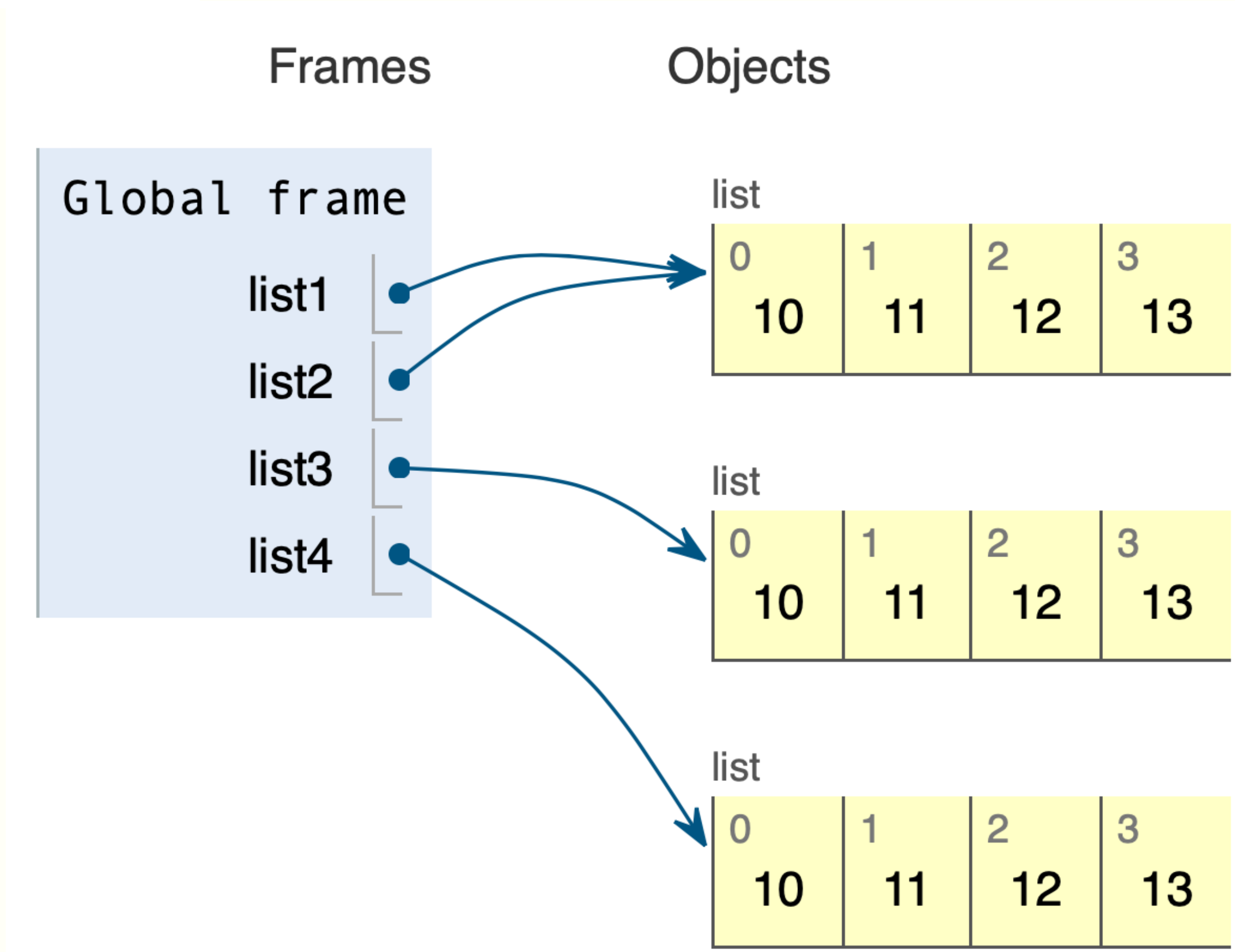
```
list4 = list3[:]
```

Note: The variable names are associated with different list

You can visualise the code on Python tutor.

See the screenshot from Python tutor on the next page.


```python
list1 = [10,11,12,13]
list2 = list1

list3 = [10,11,12,13]
list4 = list3[:]
```



Modifying list using functions

- We say in Week 2 that the scope of the variables in a function is local. This is true for immutable objects.
- For mutable data type, you can modify them by using functions
- Let us look at the examples in mut_2.py

How functions interact with parameters

- There are two ways that functions treat the parameters
  - Functions that do not modify the parameters
    - Pass by value
  - Functions that do modify the parameters
    - Pass by reference

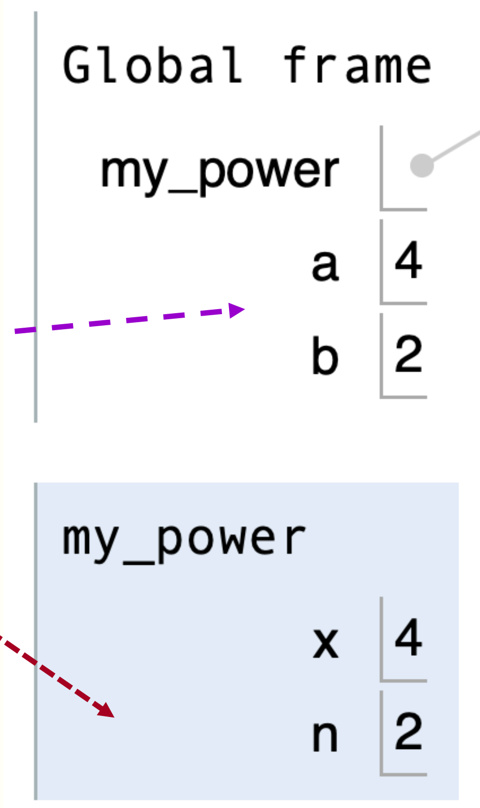
Pass by value

- In the example below, the values 4 and 2 are passed to the function
- The function does not modify the variables a and b
- Separate memory spaces for the variables within the function

```python
def my_power(x,n):
    y = x ** n
    return y

a = 4; b = 2
z = my_power(a, b)

print('y = ' ,y)
print('z = ' ,z)
```

$x \leftarrow 4$

$n \leftarrow 2$



Pass by reference

↓ From mut_2.py

```
def extend(input_list):
    input_list.append(-1)

list0 = [5, 11, 12, 13]
extend(list0)
```

Memory space of the function extend

Input_list	4728419656

Memory space

list0	4728419656

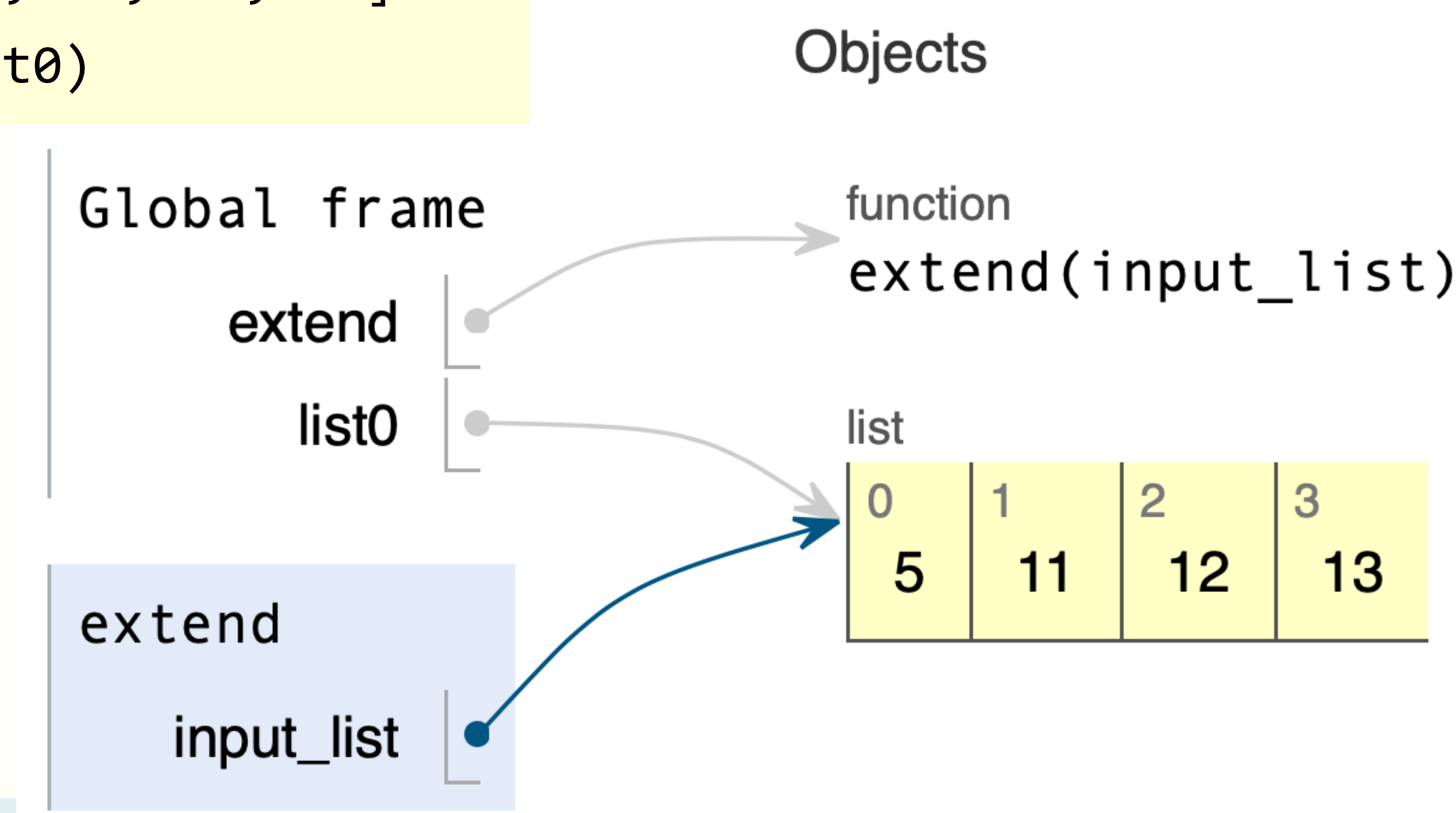
4728419656
list
...

When the function extend is called, this identifier is passed to the function. With the identifier, the function can locate the list. The identifier refers to the list, hence the name pass by reference.

Memory requirement: passing list by reference

```
def extend(input_list):
    input_list.append(-1)

list0 = [5, 11, 12, 13]
extend(list0)
```

← Need memory to store list0 only



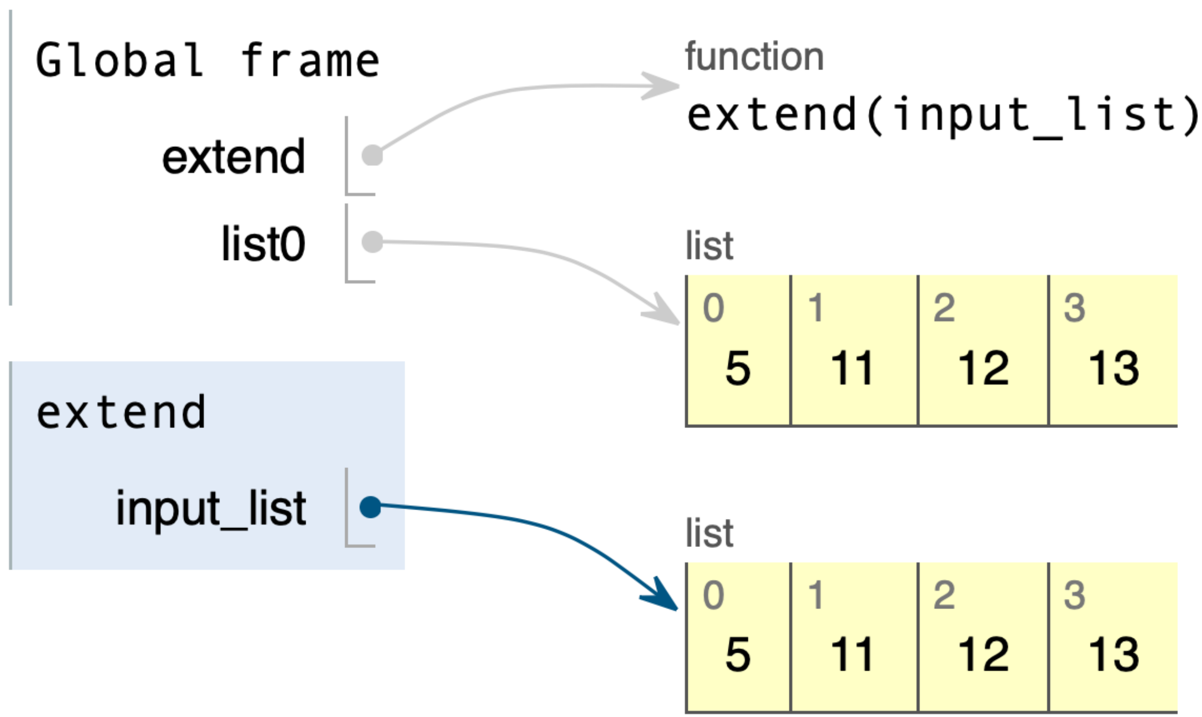
Memory requirement: passing list by value

```
def extend(input_list):
    input_list.append(-1)


list0 = [5, 11, 12, 13]
extend(list0[:])
```

Need memory to store list0 and memory for a copy of list0 in the function.

Double the memory requirement

The list is now passed by value.



Why mutable data types?

- Allow pass by reference
  - Lower memory requirement. Saves time to locate vacant memory and to duplicate the list.
  - Beneficial if the list is long
  - More data is collected than in the past, so large data sets become more prevalent

numpy arrays

- numpy arrays are mutable

- If you want to copy the contents of an array into another without associating them, you need to use the numpy function copy()
  - See mut_3.py

Summary

- Numpy elementwise operations allow you to do computation with arrays without using for-loops
  - Loops generally require more lines of code

- numpy topics covered
  - Broadcasting
  - Element selection with
    - Slicing
    - The :: notation
    - Boolean indexing

Summary

- Immutable: int, float, bool, str, tuple
- Mutable: list, numpy array

- Different ways copy mutable types

- Pass by value, pass by reference

- Passing by reference for list, numpy arrays
  - Beware that the function can modify the list/array
  - Memory requirement