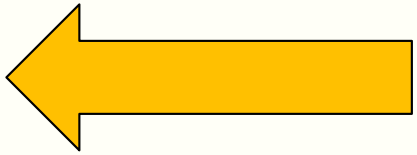
ENGG1811 Computing for Engineers

Week 7C: numpy (Broadcasting, Slicing, Boolean indexing)

Key topics

- Broadcasting



- Slicing
- Boolean indexing

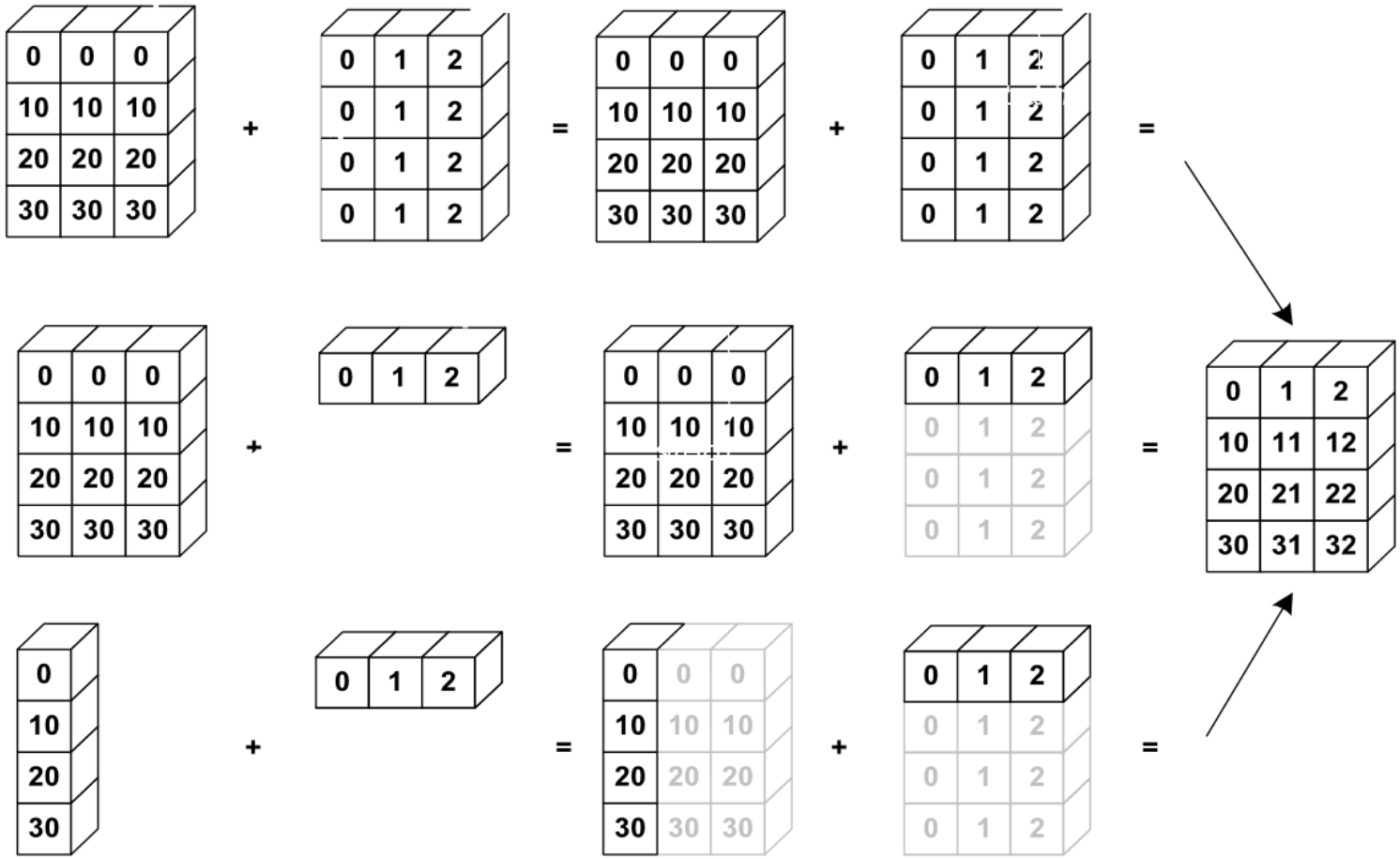
Broadcasting rules

- 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 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

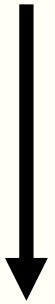
Operating on broadcast compatible arrays (1)

a1 is

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

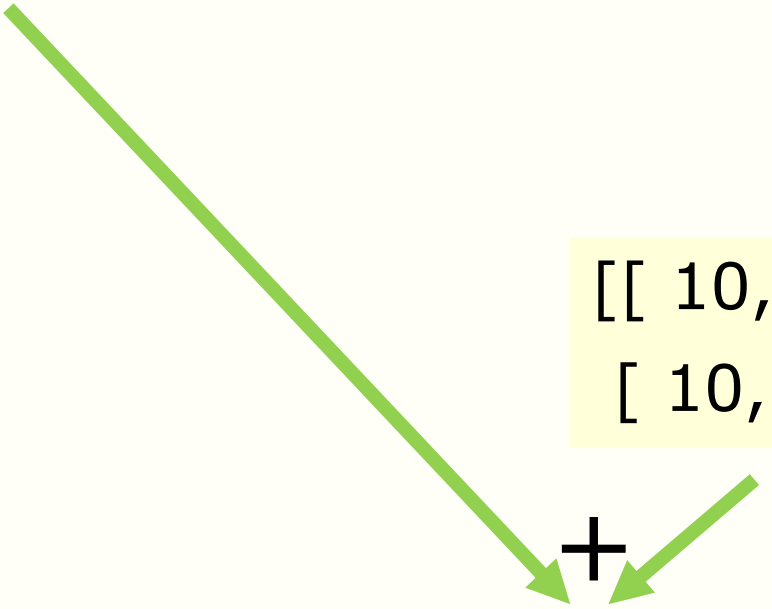
b1 is

```
[ 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

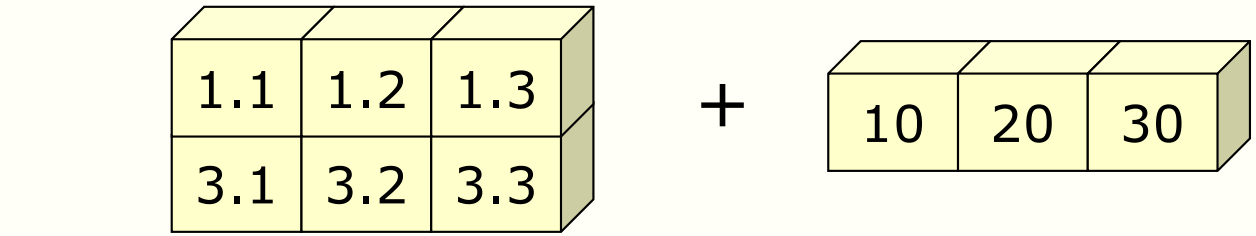
a1 is

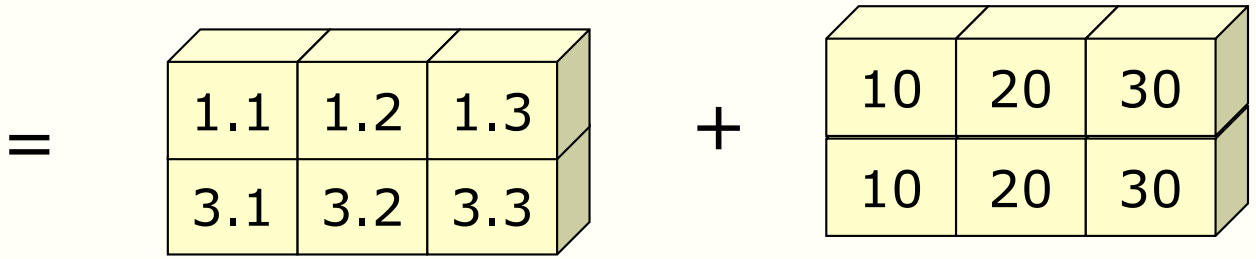
[[1.1, 2.2, 3.3],
 [3.1, 3.2, 3.3]]

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

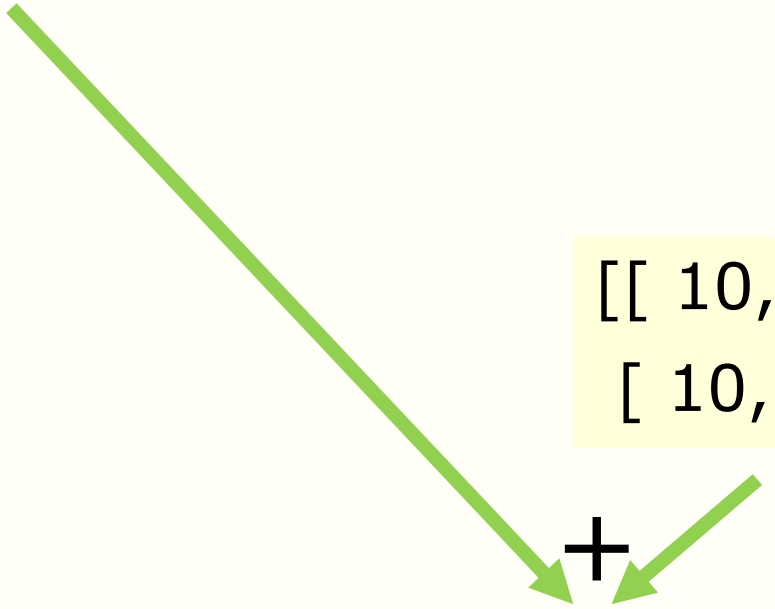
```
[[ 1.1,  2.2,  3.3],
 [ 3.1,  3.2,  3.3]]
```

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]])
```

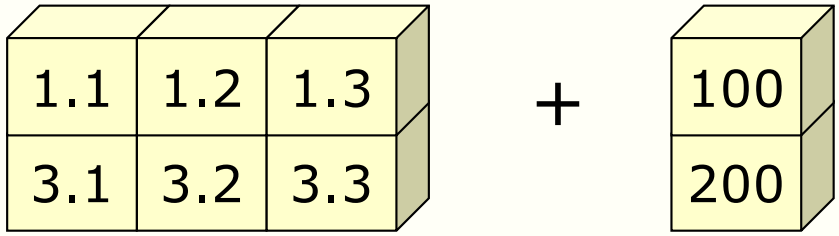
Predict what a1 + d1 should be without running the code in numpy_broadcast.py. The informal view is on the next page.

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

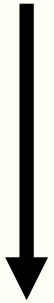
d1 is np.array([[100], [200]])

Its shape is (2,1)

a1 is

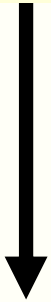
```
[[ 1.1,  2.2,  3.3],
 [ 3.1,  3.2,  3.3]]
```

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?

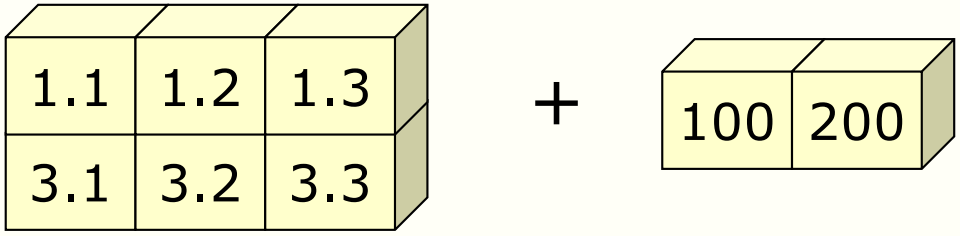
Informal view on the next page.

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])
```



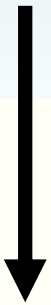
Incompatible arrays

a1 is

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

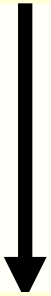
Shape (2,3)

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 – exercise

numpy_broadcast_prelim.py

Data:

Tensile force (pound force)	0	1650	3400	5200
Length (inches)	2.000	2.002	2.004	2.006

Stored in a numpy array:

```
load_length =  np.array([[    0,  1650,  3400, 5200 ],
                         [2.000, 2.002, 2.004, 2.006]])
```

You want to use SI unit instead:

1 pound force = 4.45 N

1 inch = 2.54 cm

Long winded method:

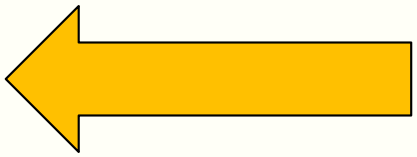
```
load_length_SI = np.array([[    4.45*0,  4.45*1650,  4.45*3400, 4.45*5200 ],
                           [2.54*2.000, 2.54*2.002, 2.54*2.004, 2.54*2.006]])
```

Exercise:

Use load_length and broadcast to obtain load_length_SI

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 learnt to use the : notation to slicing a list (Week 3B) and to slice numpy arrays (Week 5B)
- We will look at some addition methods for numpy
- Examples in:
  - numpy_slicing_1.py for one dimensional arrays
  - numpy_slicing_2.py for two dimensional arrays

numpy :: notation

numpy_slicing_1.py

- You can slice numpy arrays in a way similar to using the Python range function with 3 inputs
  - Ex: range(1,10,2) generates the integers 1, 3, 5, 7 and 9

```
Indices      0   1   2   3   4    5    6    7    8    9   10   11
a = np.array([ 1,  2,  3,  5,  7,  11,  13,  17,  19,  23,  29,  31])
```

```
In [13]: a[1:10:2]
Out[13]: array([ 2,  5, 11, 17, 23])
```

Same as extracting

a[1], a[3], a[5], a[7], a[9]

---

numpy *start*:*stop*:*step* ⇔ range(*start*, *stop*, *step*)

Numpy ::	Default value
start	0
stop	Array length
step	1

Examples:

```
a[:5:]   # a[0:5:1], a[:5]
a[4::2] # a[4:len(a):2]
```

1-D array: select specific elements

- You can use:
  - the : notation to slice out a continuous section
  - the :: notation to select regularly spaced elements

- How about specific or non-regularly spaced elements?

indices
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: rectangular block or regularly spaced slicing

numpy_slicing_2.py

```
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]])
```

---

array_name[,]

: for a continuous section
:: for regularly spaced

Examples:

c[1::2, ::2]

c[::2, -3:]

2-D array: Slicing with np.ix_

```
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]])
```

Column index
2 3 6

Row index
1
3

```
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

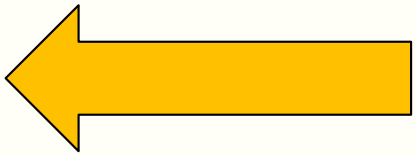
```
In [37]: c
Out[37]:
array([[11, 23,  7,  5, 29, 37, 43],
       [13, 57, 71, 26, 31, 47, 53],
       [17, 67, 73,  3,  2, 19, 31],
       [41, 53, 59, 61, 91, 79, 83]])

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

Key topics

- Broadcasting
- Slicing
- Boolean indexing



Boolean indexing

- This indexing method which select elements with some specific property in an array
  - The property is specified by a Boolean expression
- 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]
bool_array1         [False, True,  True,  False,   True]

array1[bool_array1]         [0.4, 1.4, 0.1]
# Note: array1 and bool_array1 have the same shape
```

```
array1              [0.3,    0.4,   1.4,    1.7,    0.1]
bool_array2         [True, False, False,  False,   True]

array1[bool_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)

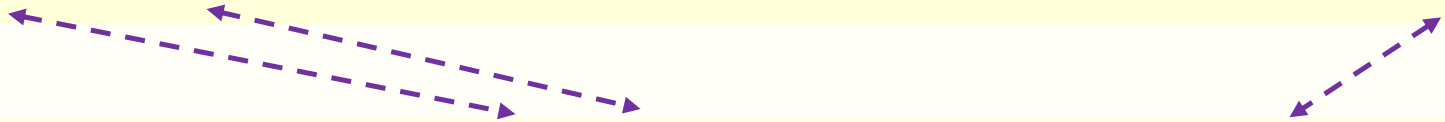
This quiz is in numpy_boolean_indexing_1.py

```
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 8 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-d arrays
  - The Boolean expression being used for indexing can contain: &, |, ~ (which are logical and, or, not in numpy)
  - Using assignment with Boolean indexing

- numpy_boolean_indexing_3.py for 2-d 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

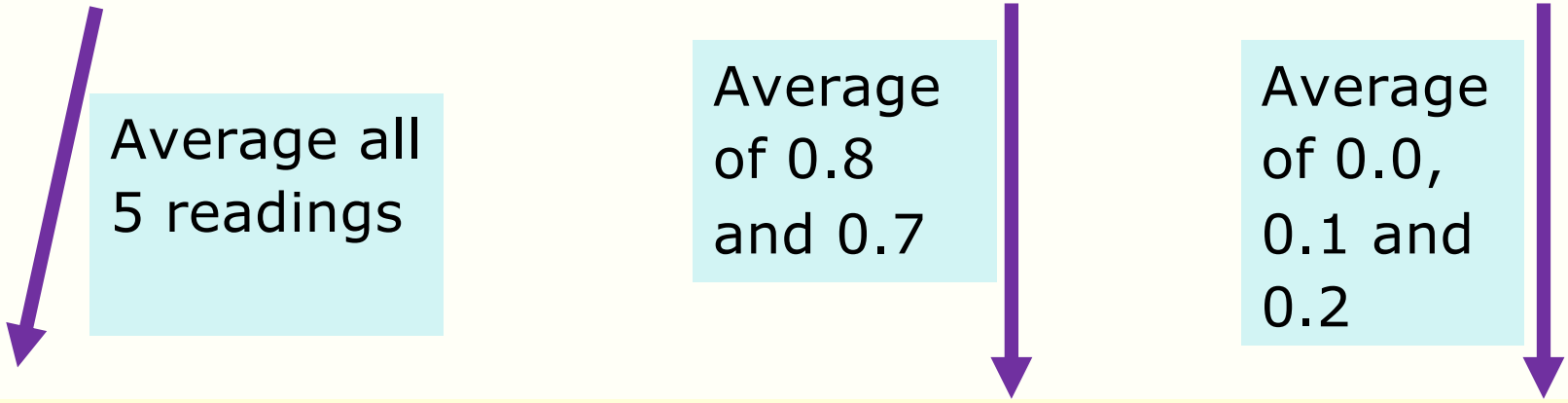
Average of 0.0, 0.1 and 0.2

[**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) but some students needed only 1 line of code

```
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]]
)
```



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

Summary

- Broadcasting
  - Elementwise computation of arrays of compatible dimensions
- Element selection with
  - A continuous section with :
  - Regularly spaced elements with ::
  - Specific elements
  - Boolean indexing