Data science

NumPy

Chapter 4 – Numpy

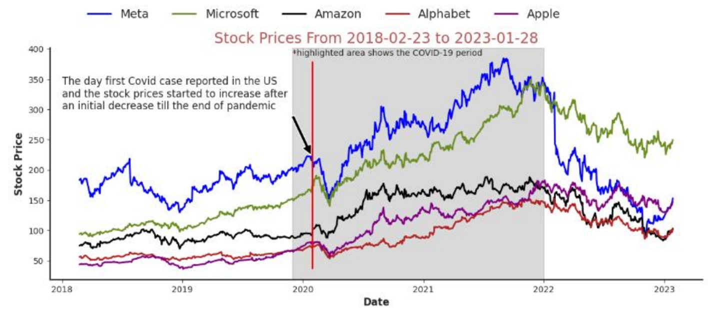
2025-2026

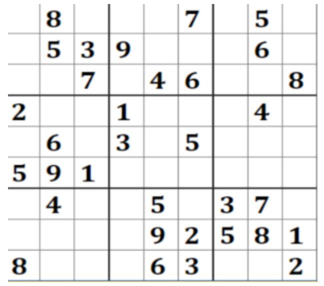
Overview

1. Creating a NumPy array
2. Properties of a NumPy array
3. Arithmetic operations
4. Extracting specific items from an array
5. Creating a new array from an existing array
6. Reshaping and flattening multidimensional arrays
7. Filtering arrays
8. Aggregations
9. Read csv file into NumPy array

NumPy

- Numeric Python
- Open-source extension for working with arrays in Python
  - Remember: basic Python only uses lists (or tuples or sets or dicts), not arrays
- Arrays are of fixed size and contain only 1 datatype
- An array can be multidimensional
- Some examples of 1D/2D/3D arrays:





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

What’s wrong with lists?

- When you’ve used lists you’ve gotten used to very flexible programming
  - Add items, remove items, multiple datatypes, …
- You’ll feel reluctant to let this flexibility go and when doing ‘normal’ Python programming the difference between a list and a np-array isn’t that big
- But when you start doing AI you will feel the difference:
  - Dijkstra algorithm is a way finding the path through a maze
  - For this you need to try every single path there is
  - As a result, you go over the maze *a lot*
- But NumPy won’t solve everything:
  - Arrays have one data type: you can’t store name, age and sex together in a np-array
  - The fix: Pandas dataframes
    - These are actually combined np-arrays

Install and import NumPy

- Run the install command in a terminal window:

```
pip install numpy
```

- In Jupyter you must use a preceding ! to force the code cell to be executed in a terminal window:

```
!pip install numpy
```

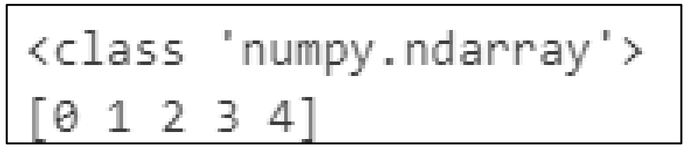
- To import numpy we usually import it with a shorter name since it's used so much:

```
import numpy as np
```

Creating a NumPy array

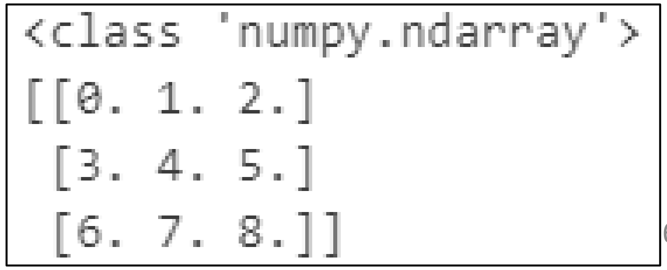
- Via the function **array()** you create an array object.
- The first argument is a list or tuple. The number of elements in this list or tuple determines the **size** of the array.
- The second **optional argument** specifies the **type** of the elements.
    - If this argument is not given, Python determines the type itself.

```
import numpy as np
# create a 1d array from a list
list1 = [0, 1, 2, 3, 4]
arr1d = np.array(list1)
```



```
<class 'numpy.ndarray'>
[0 1 2 3 4]
```

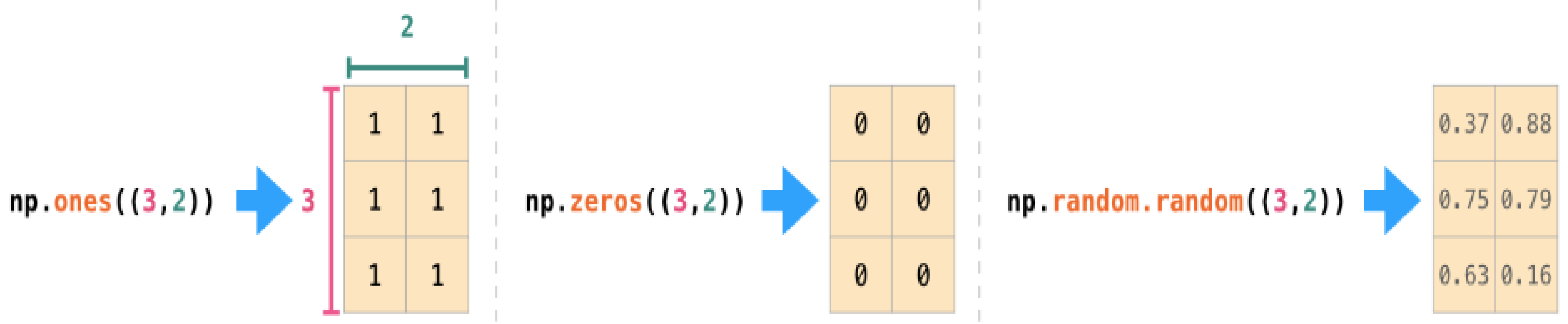
```
# create a 2d array from a list of lists and datatype float
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr2d_float = np.array(list2, dtype='float')
```

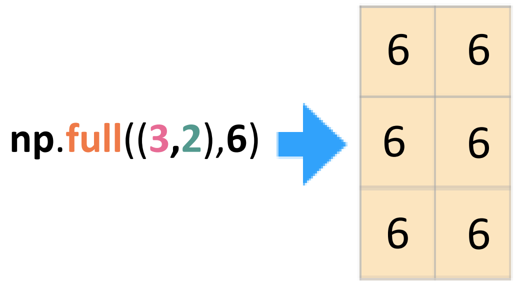


```
<class 'numpy.ndarray'>
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

Creating a NumPy array

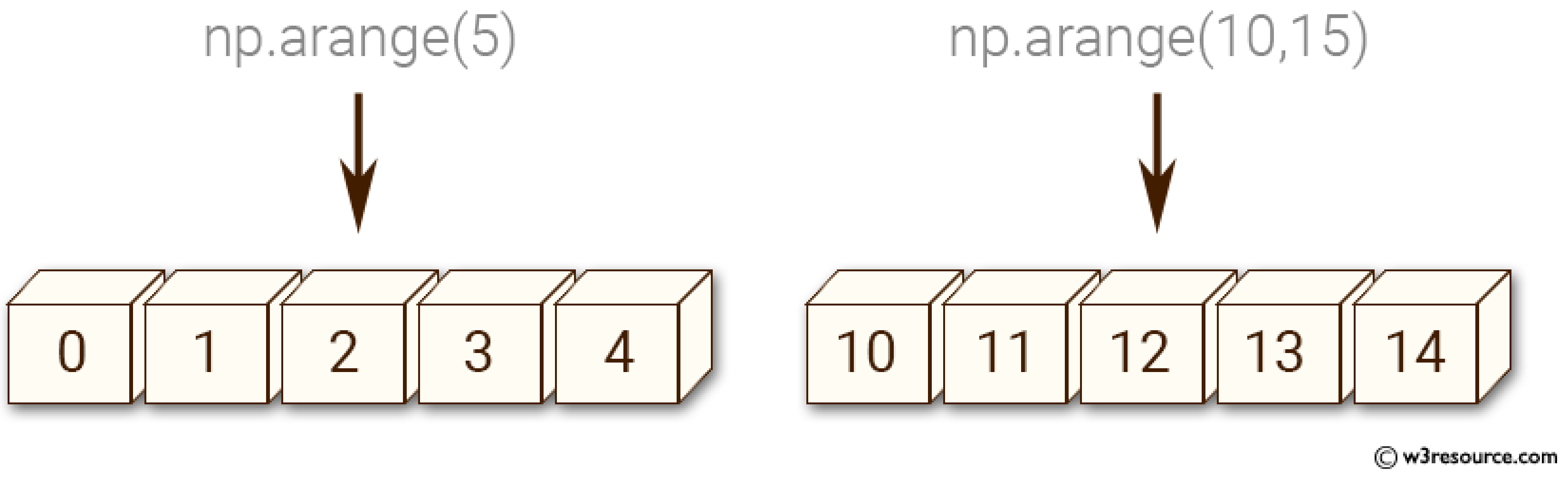
- With the np methods ones(), zeros(), full(), random.random() an array of desired shape with initialized values can be created





Creating a NumPy array

- With the np method arrange() a sequence of numbers can be handled as a 1-dimensional array



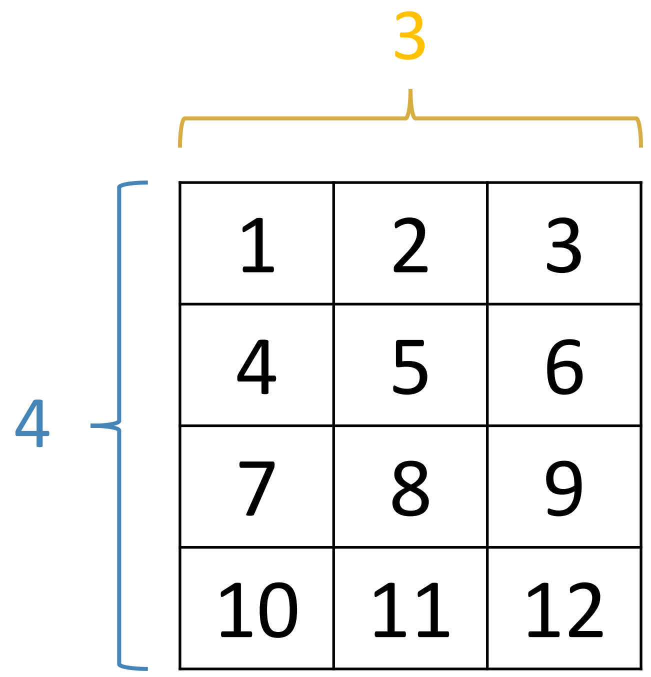
Properties of a NumPy array

.ndim = 2

.shape = (4,3)

.shape[0] = 4

.shape[1] = 3



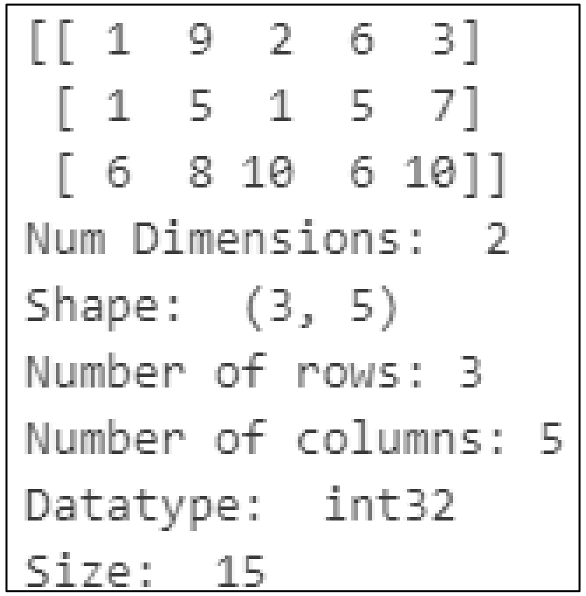
1	2	3
4	5	6
7	8	9
10	11	12

.dtype = int32

.size = 12 → 4*3

Properties of a NumPy array

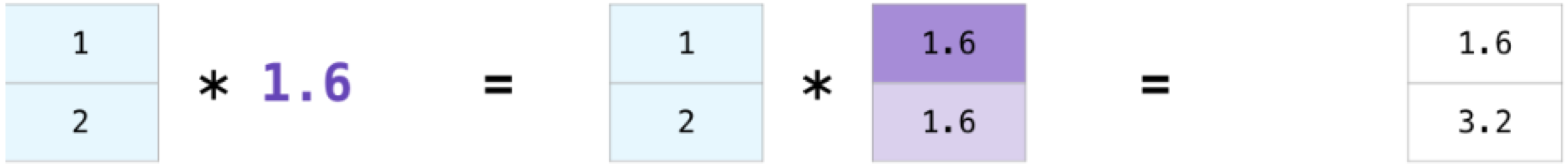
```python
# create a 2d array with 3 rows and 5 columns
# initialized with random integers from 1 to 10
arr = np.random.randint(1,11,(3,5))
print(arr)

# ndim
print('Num Dimensions: ', arr.ndim)
# shape
print('Shape: ', arr.shape)
# number of rows
print('Number of rows:',arr.shape[0])
# number of columns
print('Number of columns:',arr.shape[1])
# dtype
print('Datatype: ', arr.dtype)
# size
print('Size: ', arr.size)
```



```
[[ 1  9  2  6  3]
 [ 1  5  1  5  7]
 [ 6  8 10  6 10]]
Num Dimensions:  2
Shape:  (3, 5)
Number of rows: 3
Number of columns: 5
Datatype:  int32
Size:  15
```

Arithmetic operations

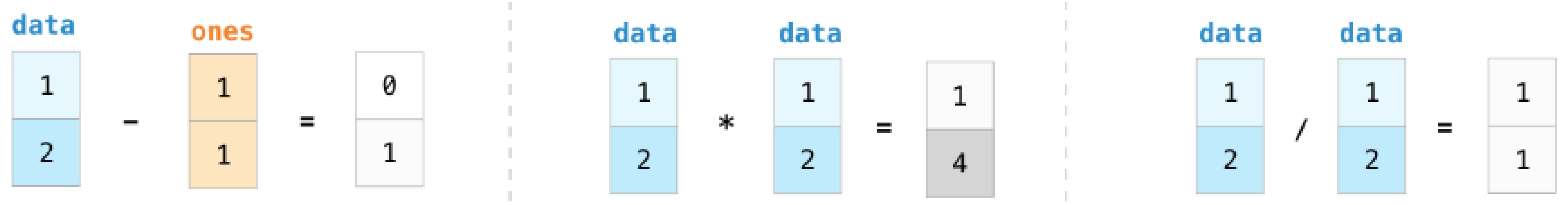
- There are often cases when you want to carry out an operation between an array and a single number (you can also call this an operation between a vector and a scalar).
- Say, for example, an array represents the distance in miles, and you want to convert it to kilometers. You can simply say data * 1.6:

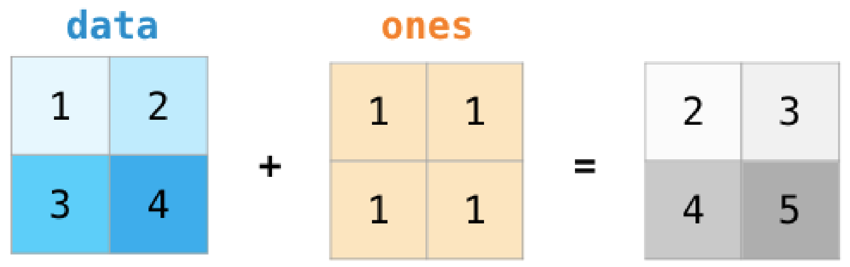


- See how NumPy understood that operation to mean that the multiplication should happen with each cell. That concept is called **broadcasting.**

Arithmetic operations

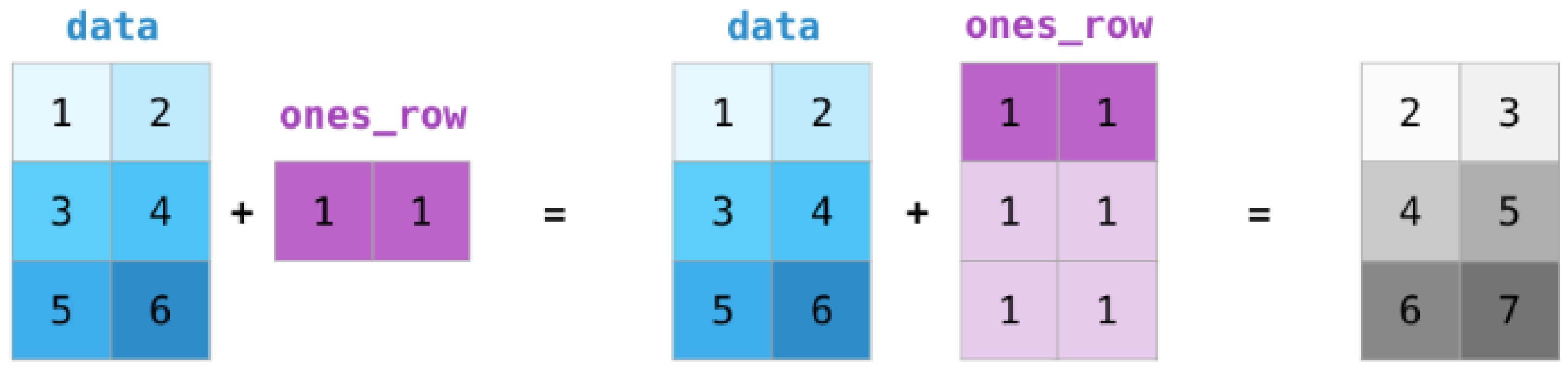
- You can add and multiply matrices using arithmetic operators (+-*/) if the two matrices are the same size. NumPy handles those as position-wise operations:





Arithmetic operations

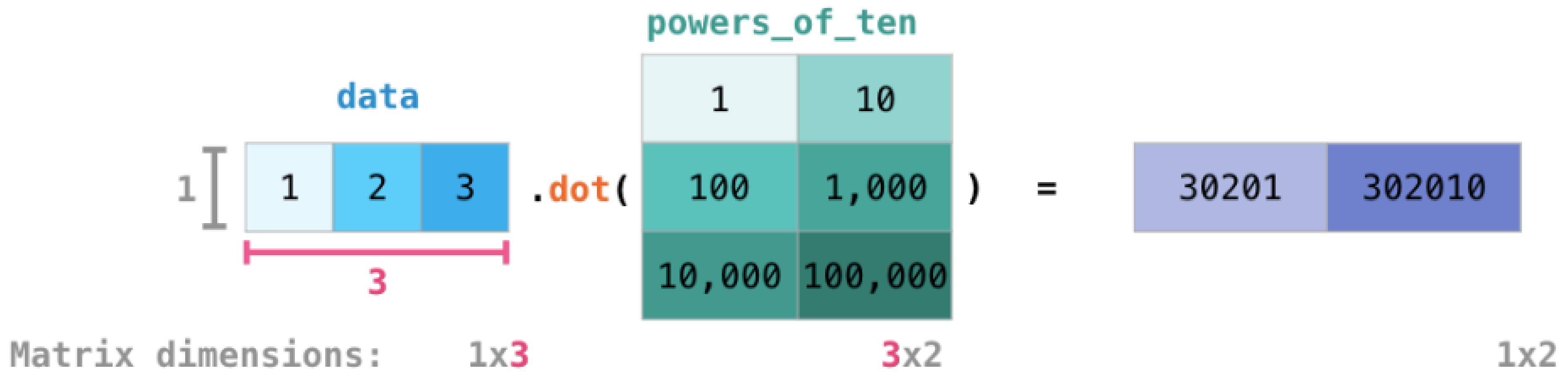
- NumPy broadcasting is a mechanism that enables element-wise operations on arrays of different shapes by automatically "stretching" the smaller array along the appropriate dimensions to match the larger array's shape, all without creating unnecessary copies of the data.
- This process requires that the array dimensions are compatible, meaning they are either equal or one of them is 1 (i.e. has just one row or one column)



$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix} + \begin{bmatrix}1 & 1\end{bmatrix} = \begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 1\\1 & 1\end{bmatrix} = \begin{bmatrix}2 & 3\\4 & 5\\6 & 7\end{bmatrix}$$

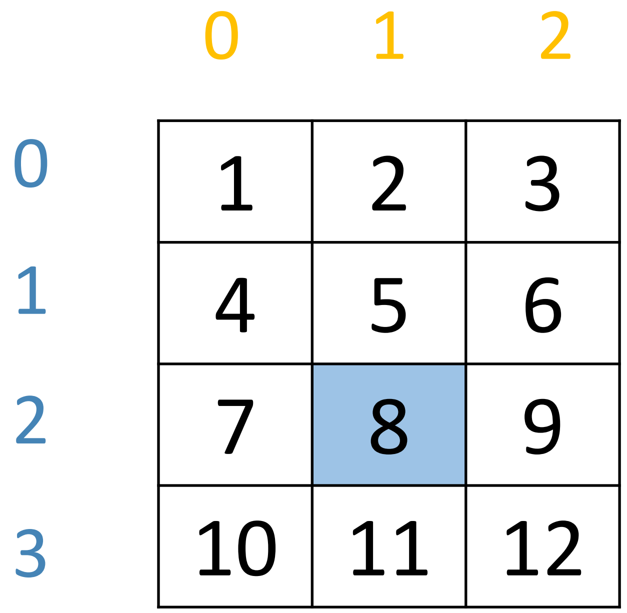
Arithmetic operations

- A key distinction to make with position-wise arithmetic is the case of matrix multiplication using the dot product.
- NumPy gives every matrix a dot() method to carry-out dot product operations with other matrices

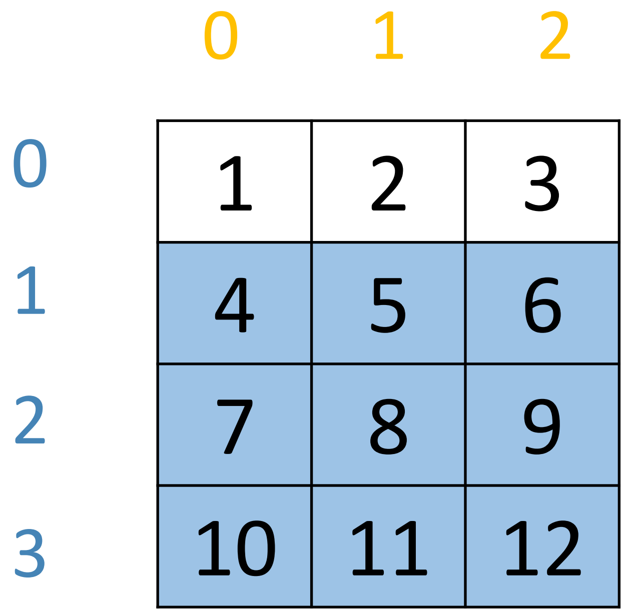


Extracting specific items from an array

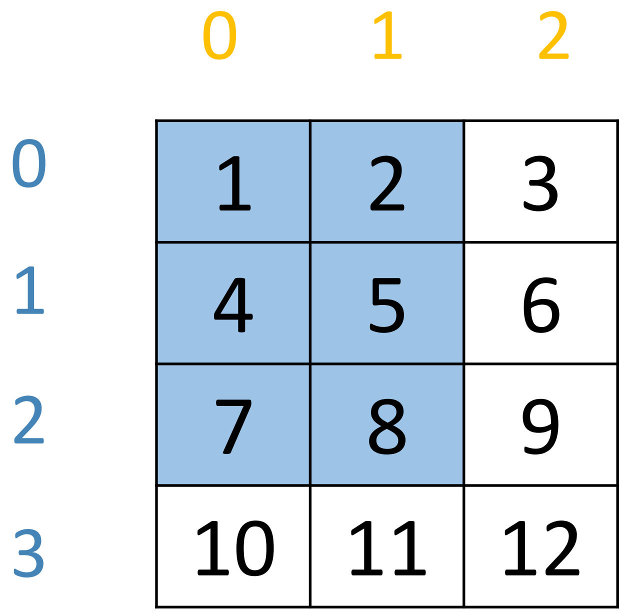
arr[2 , 1]



arr[1:4]



arr[:3 , :2]



Creating a new array from an existing array

- When assigning a part of an existing array to a new array, in memory you just created a reference to the parent array.
- This means that every action taken in the new array reflects in the parent array.

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

arr

```
arr_new = arr[ :3 , :2]
arr_new[0,0]= 10
```

10	2
4	5
7	8

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

arr

Creating a new array from an existing array

- To create an entire new array object, use the method copy()
- Both arrays now live independently from each other

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

arr

```
arr_new = arr[ :3 , :2].copy()
arr_new[0,0]= 10
```

10	2
4	5
7	8

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

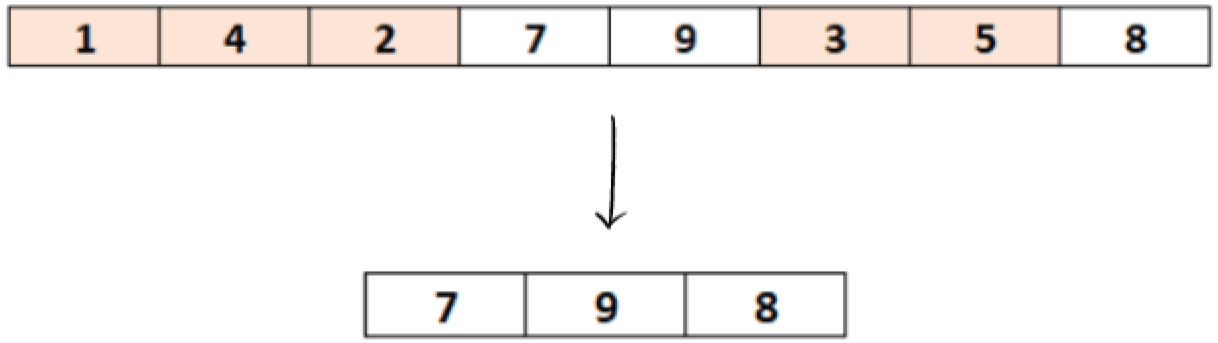
arr

Creating a new array from an existing array

- NumPy where() function is used to create a new array from the existing array based on conditions.
- It returns the indices of the array for which each condition is True. So, you get an indication of whether or not to keep the element in the corresponding array.
- You can use the result for boolean mask slicing.
- For example, if you filter the array [1, 2, 3] with the boolean list [True, False, True], the filtered array would be [1, 3].

Creating a new array from an existing array

```
# Create a numpy array
arr = np.array([1, 4, 2, 7, 9, 3, 5, 8])

# indexes of elements to keep
indexes_to_keep = np.where(arr > 5)
print(indexes_to_keep)

# filter the array
arr_filtered = arr[indexes_to_keep]

# show the filtered array
print(arr_filtered)
```



```
indexes_to_keep: (array([3, 4, 7], dtype=int64),)
arr_filtered: [7 9 8]
```

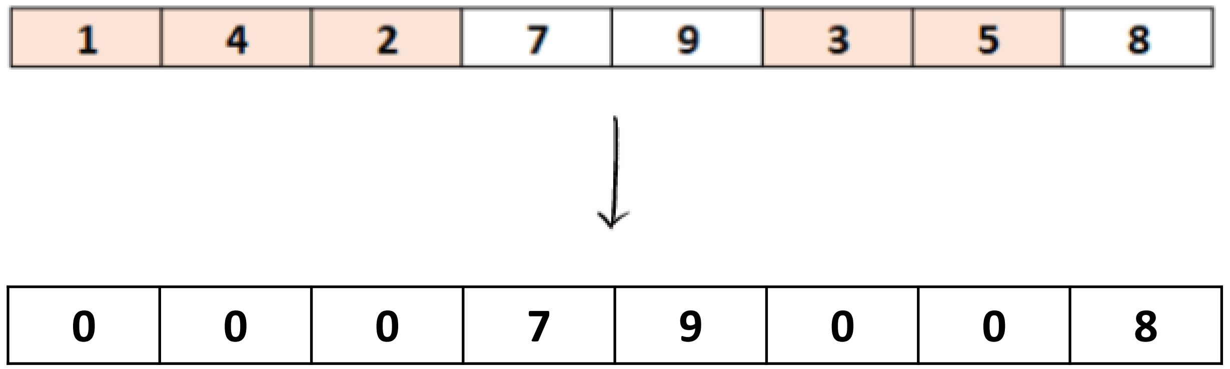
Creating a new array from an existing array

np.where (conditions, x, y)

- The where() function has 2 optional extra parameters x and y.
  - If condition holds true, the new array will choose elements from x.
  - Otherwise, if it’s false, elements from y will be taken.
- With that, our final output array will be an array with elements from x wherever condition = True, and elements from y whenever condition = False.
- Although x and y are optional, if you specify x, you MUST also specify y.
  - The output array shape must be the same as the input array.

Creating a new array from an existing array

```python
# Create a numpy array
arr = np.array([1, 4, 2, 7, 9, 3, 5, 8])

# filter the array
arr_filtered = np.where(arr > 5, arr,0)

# show the filtered array
print("arr:",arr)
print("arr_filtered:",arr_filtered)
```

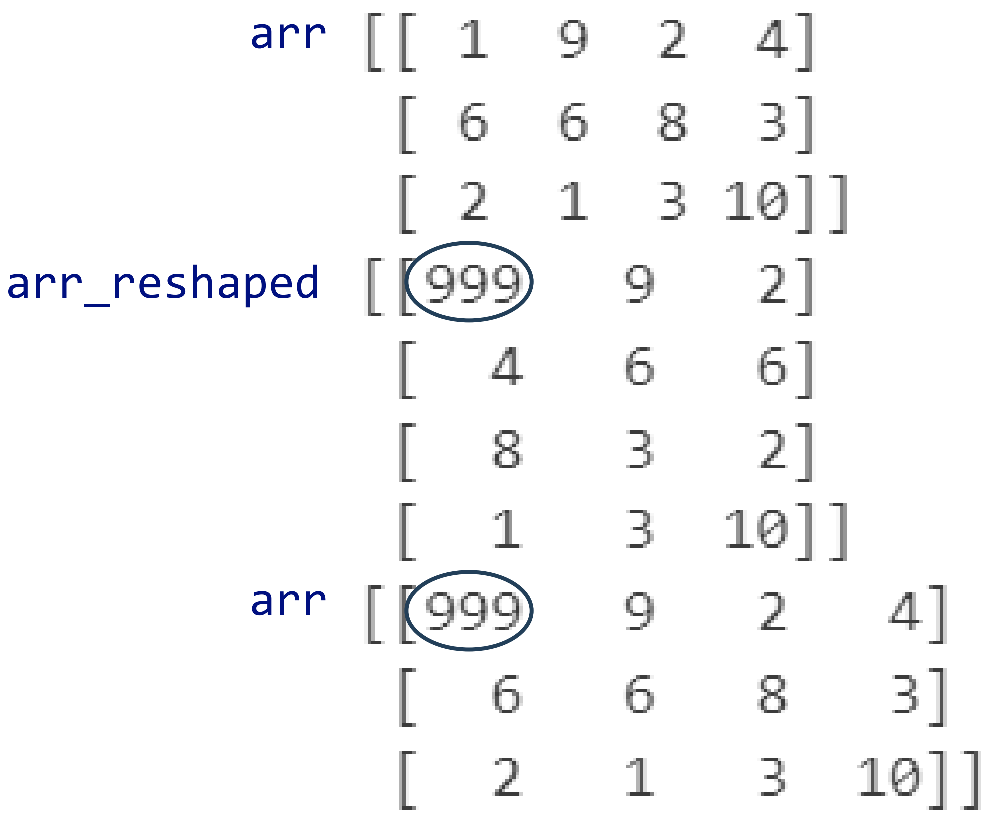


```
arr: [1 4 2 7 9 3 5 8]
arr_filtered: [0 0 0 7 9 0 0 8]
```

Reshaping and flattening multidimensional arrays

- Reshaping means changing the shape of an array

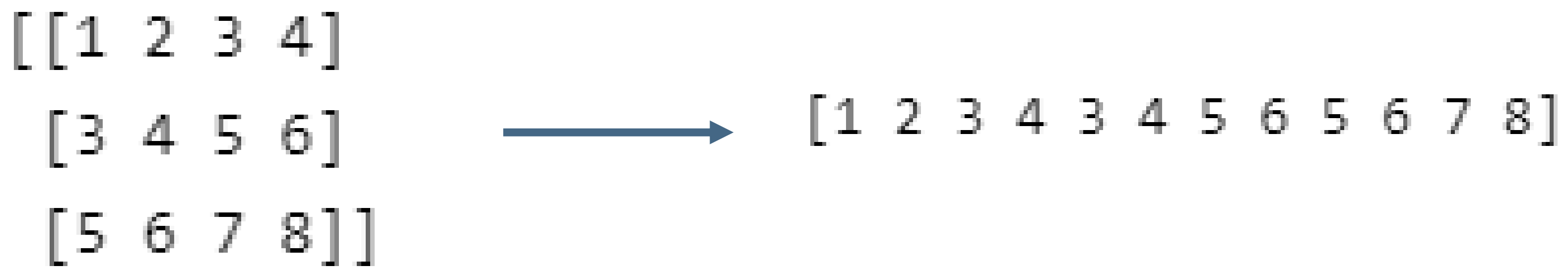
```python
# create a 2d array with 3 rows and 4 columns
arr = np.random.randint(1,11,(3,4))
print(arr)

# reshape a 3x4 array to 4x3 array
arr_reshaped = arr.reshape(4, 3)

# every change in the reshaped array is
# reflected in the parent (base)
arr_reshaped[0,0]=999
print(arr_reshaped)
print(arr)
```



```
arr          [[ 1  9  2  4]
              [ 6  6  8  3]
              [ 2  1  3 10]]
arr_reshaped [[999   9   2]
              [  4   6   6]
              [  8   3   2]
              [  1   3  10]]
arr          [[999   9   2   4]
              [  6   6   8   3]
              [  2   1   3  10]]
```

Reshaping and flattening multidimensional arrays

- Flattening an array means converting a multidimensional array into a 1D array.
- 3 methods available:
  - reshape(-1): returned array is reference
  - flatten(): a new 1D array object is created
  - ravel(): the returned 1D array is just a reference to the parent array



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

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

Selecting unique values: unique()

- Suppose we have the data of 2 tests:

```
test_1 = np.array([3, 7, 1, 9, 0, 4, 6, 5, 8, 5, 3, 7, 1, 9, 0, 4, 6, 3, 8, 5])
test_2 = np.array([9, 10, 8, 4, 10, 6, 9, 0, 10, 7, 9, 6, 10, 8, 9, 7, 10, 6, 8, 9])
```

- What are the unique values in each of these datasets?

```
print(np.unique(test_1), np.unique(test_2), sep='\n’)
```

```
[0 1 3 4 5 6 7 8 9]
[ 0  4  6  7  8  9 10]
```

- How many times do these unique values occur in dataset test_1?

```
unique, counts = np.unique(test_1, return_counts=True)

print(unique, counts )
```

```
[0 1 3 4 5 6 7 8 9] [2 2 3 2 3 2 2 2 2]
```

Selecting shared values: intersect1d()

```
test_1 = np.array([3, 7, 1, 9, 0, 4, 6, 5, 8, 5, 3, 7, 1, 9, 0, 4, 6, 3, 8, 5])
test_2 = np.array([9, 10, 8, 4, 10, 6, 9, 0, 10, 7, 9, 6, 10, 8, 9, 7, 10, 6, 8, 9])
```

- And which numbers do the two tests share?

```
print(np.intersect1d(test_1, test_2))
```

```
[0 4 6 7 8 9]
```

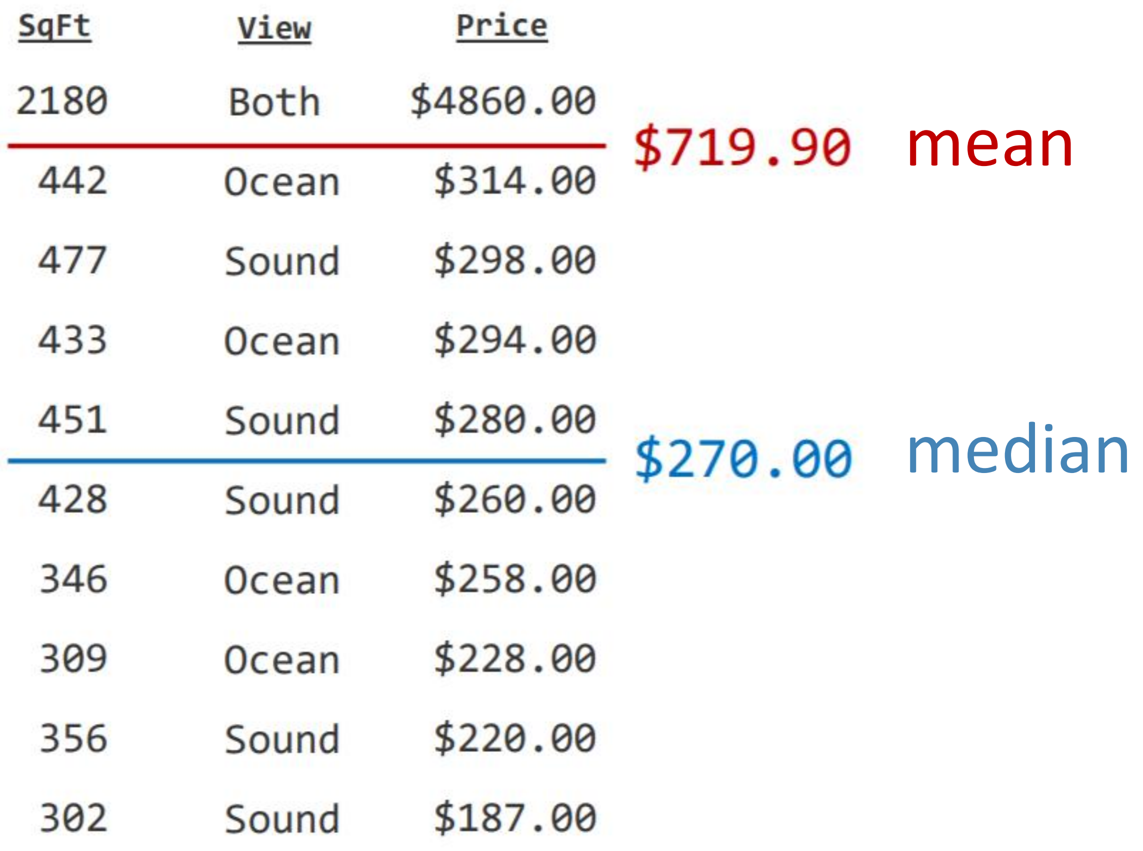
Aggregations

- When faced with a large amount of data, a first step is to compute summary statistics for the data in question. This can be done with aggregate functions that perform a calculation on a set of values and return a single value.
- The most common summary statistics are the **mean**, the **median**, the **minimum** and the **maximum** value in a dataset.
- A data scientist also wants to know how spread-out the numbers in the dataset are by computing **percentiles** and the **standard deviation**.
- Other aggregates like **sum** and **product** could be useful as well.

Aggregations: mean versus median

- E.g. hotel room prices

- Due to the extremely high price of the suite mean and median are far apart
  - In this case median is a far better tendency indicator than average.
  - The average is pulled upwards by the one high rental price.
  - We say: an **outlier skews** the data.



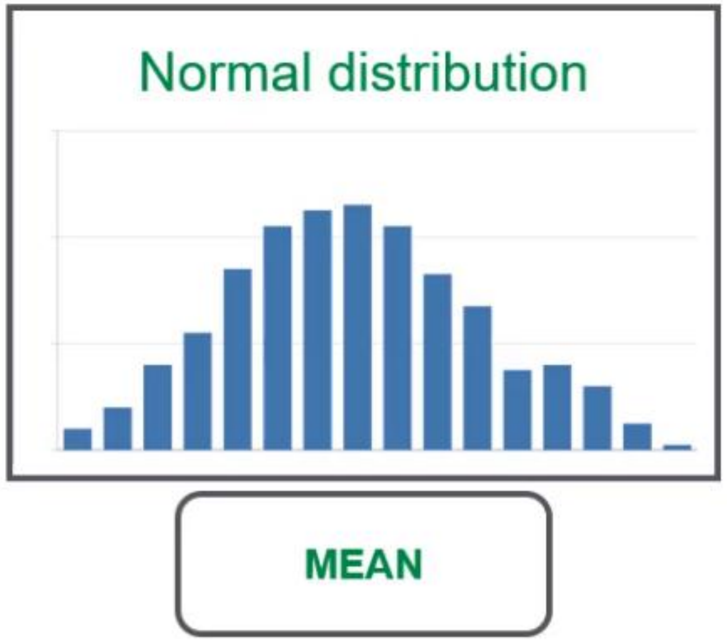
SqFt	View	Price
2180	Both	$4860.00
442	Ocean	$314.00
477	Sound	$298.00
433	Ocean	$294.00
451	Sound	$280.00
428	Sound	$260.00
346	Ocean	$258.00
309	Ocean	$228.00
356	Sound	$220.00
302	Sound	$187.00

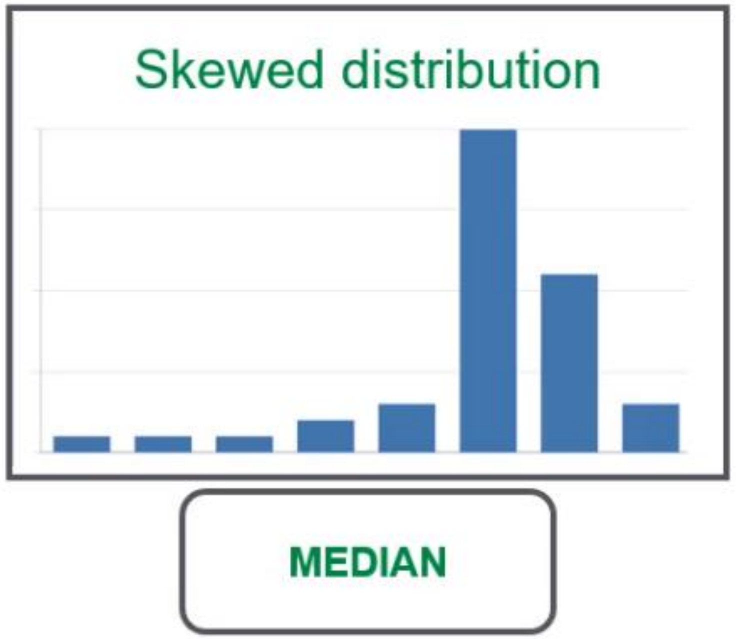
$719.90 mean

$270.00 median

Aggregations: mean versus median

In general, outliers have a major effect on the mean but they don't have such an effect on the median. Therefore, in a skewed distribution the median gives a more realistic picture of the data.





Aggregations: percentiles and fractiles

- Percentile is the value below which a stated percentage of the observations lie when all the values are sorted from least to greatest.

- Example:
  If 64 out of 80 students score less than 70 points on a test, then 70 points is the 80% percentile, i.e. 80% of the students (64/80*100) have grades below 70 points.
  The 80% percentile is written as P80.

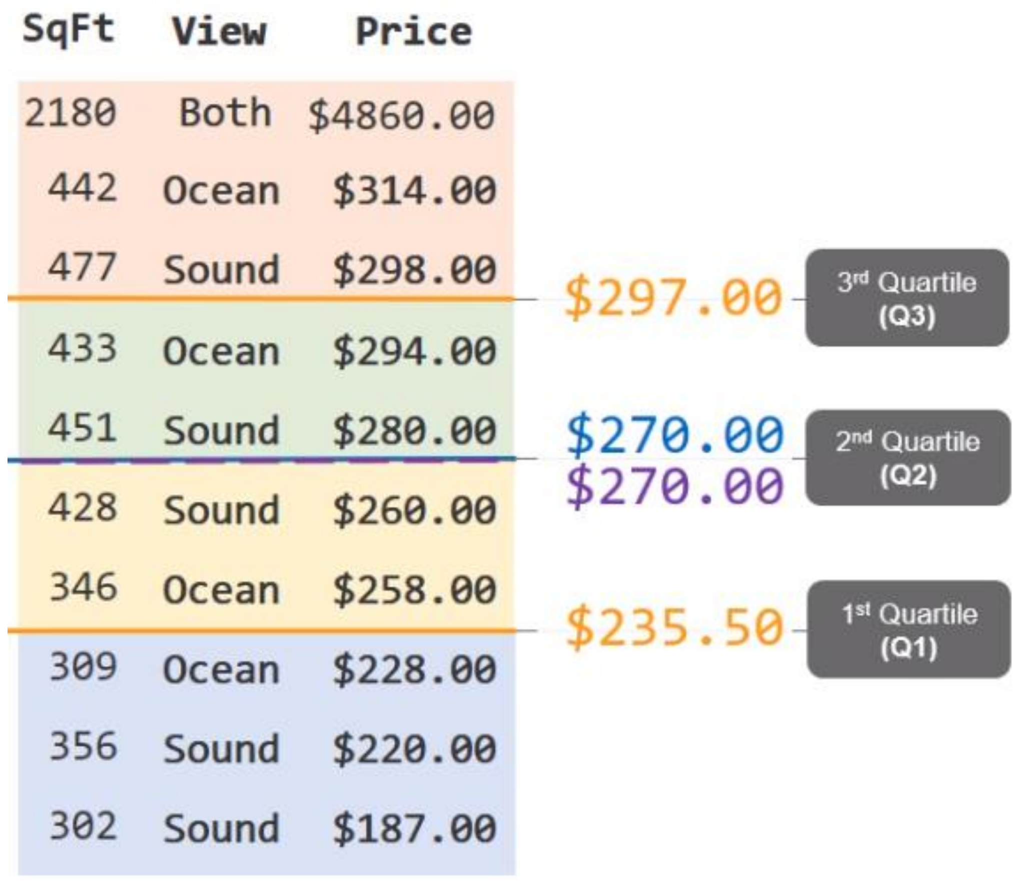
Aggregations: percentiles and fractiles

- There are 3 special percentiles called the quartiles, which divide the data into four groups of equal size.
- The **first quartile (Q1)** is the same as the 25th percentile or 0.25-fractile. This number indicates that 25% of the observations are below this number.
- The **second quartile (Q2)** is the 50th percentile or the 0.50-fractile and thus equal to the median.
- The **third quartile (Q3)** equals the 75th percentile or 0.75-fractile.

- Be aware that there is no fourth quartile!

Aggregations: percentiles and fractiles

The dataset is split into 4 groups



SqFt	View	Price
2180	Both	$4860.00
442	Ocean	$314.00
477	Sound	$298.00
433	Ocean	$294.00
451	Sound	$280.00
428	Sound	$260.00
346	Ocean	$258.00
309	Ocean	$228.00
356	Sound	$220.00
302	Sound	$187.00

$297.00
3rd Quartile (Q3)
$270.00
$270.00
2nd Quartile (Q2)
$235.50
1st Quartile (Q1)

Q3 = P75

Q2 = P50 = Median

Q1 = P25

Wrong?!?

- The quartiles as calculated on the previous slide aren't entirely correct according to the formula you might have learned earlier

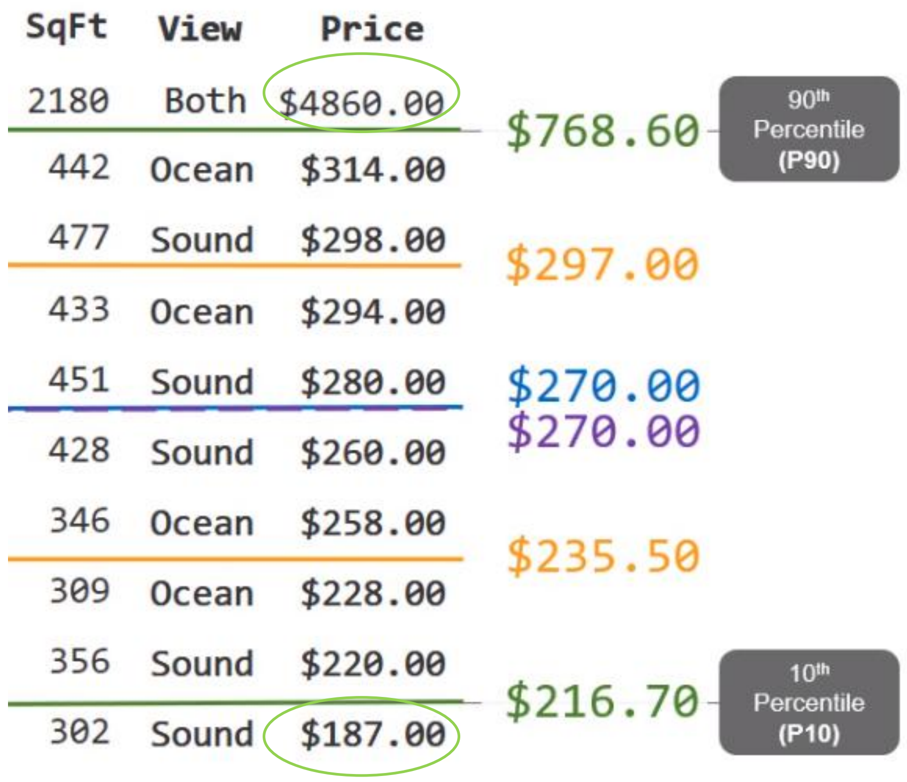
$$Q1 = \frac{1}{4}(n+1)^{th}\text{term}$$

- That's because there are 9 ways to calculate a quartile:
  - https://robjhyndman.com/publications/quantiles/
  - Quantiles and Percentiles, Clearly Explained!!! - YouTube

- If you ever take a master course in statistics: dive deep into all 9 of them
- By default np.percentile() uses the method 'linear'
- In this class: whatever Python tells you is correct

Aggregations: percentiles and fractiles

- Next to the quartiles other percentiles like P10 and P90 can be calculated.
- Values not in the 0,90-fractile or in the 0,10-fractile could be seen as outliers.



SqFt	View	Price
2180	Both	$4860.00
442	Ocean	$314.00
477	Sound	$298.00
433	Ocean	$294.00
451	Sound	$280.00
428	Sound	$260.00
346	Ocean	$258.00
309	Ocean	$228.00
356	Sound	$220.00
302	Sound	$187.00

$768.60
90th Percentile (P90)
$297.00
$270.00
$270.00
$235.50
$216.70
10th Percentile (P10)

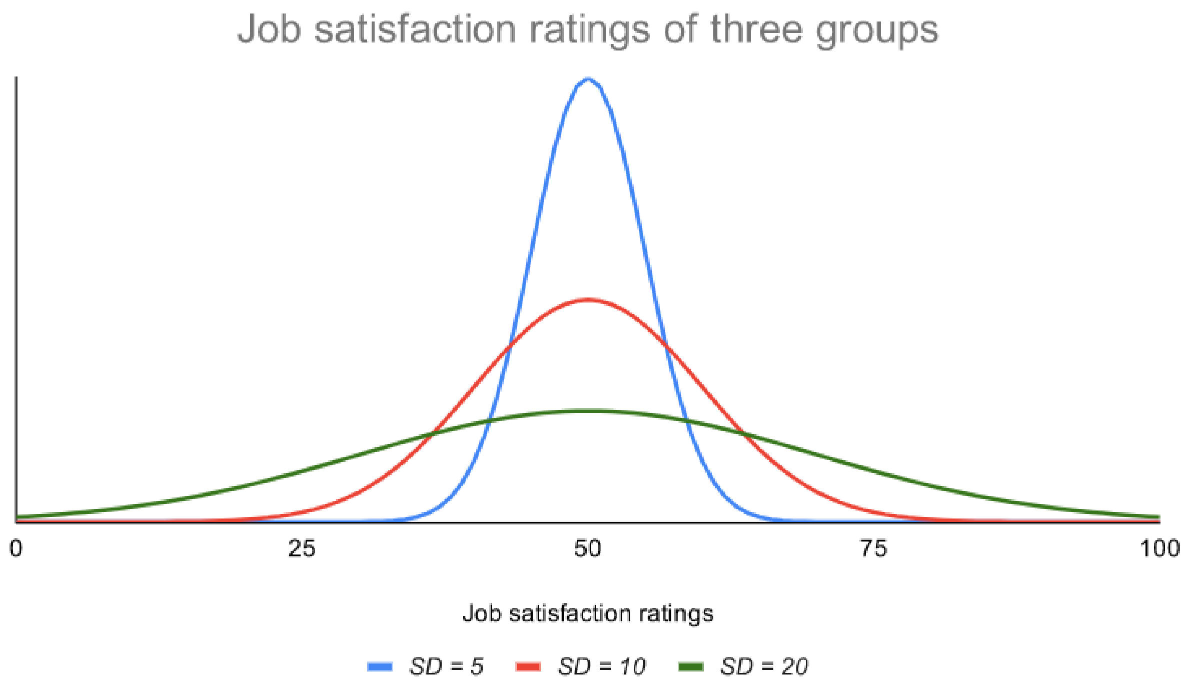
Aggregations: standard deviation

Standard deviation measures the dispersion of a data population in a normal distribution.

↓	*A low standard deviation can show low dispersion*
↑	*A high standard deviation can show high dispersion*
σ	*= Standard Deviation*
μ	*= Mean*

Aggregations: standard deviation

The curve with the lowest standard deviation has a high peak and a small spread, while the curve with the highest standard deviation is more flat and widespread.



Aggregations: standard deviation

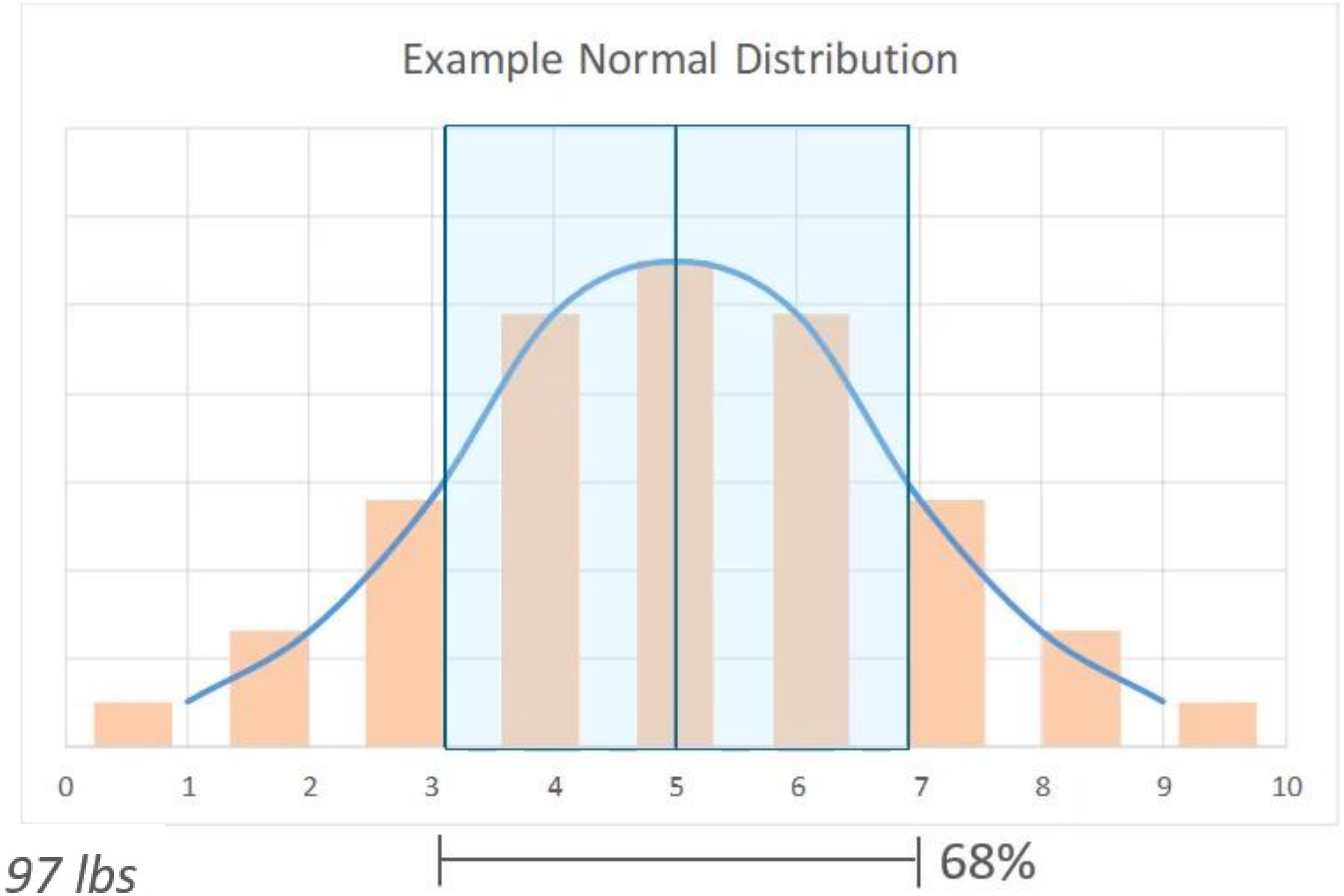
The **empirical rule,** or the 68-95-99.7 rule, tells you where your values lie:

- Around 68% of scores are within +/-1 standard deviation of the mean,
- Around 95% of scores are within +/- 2 standard deviations of the mean,
- Around 99.7% of scores are within +/- 3 standard deviations of the mean.

Aggregations: standard deviation

μ = *185 lbs*

σ = *12 lbs*



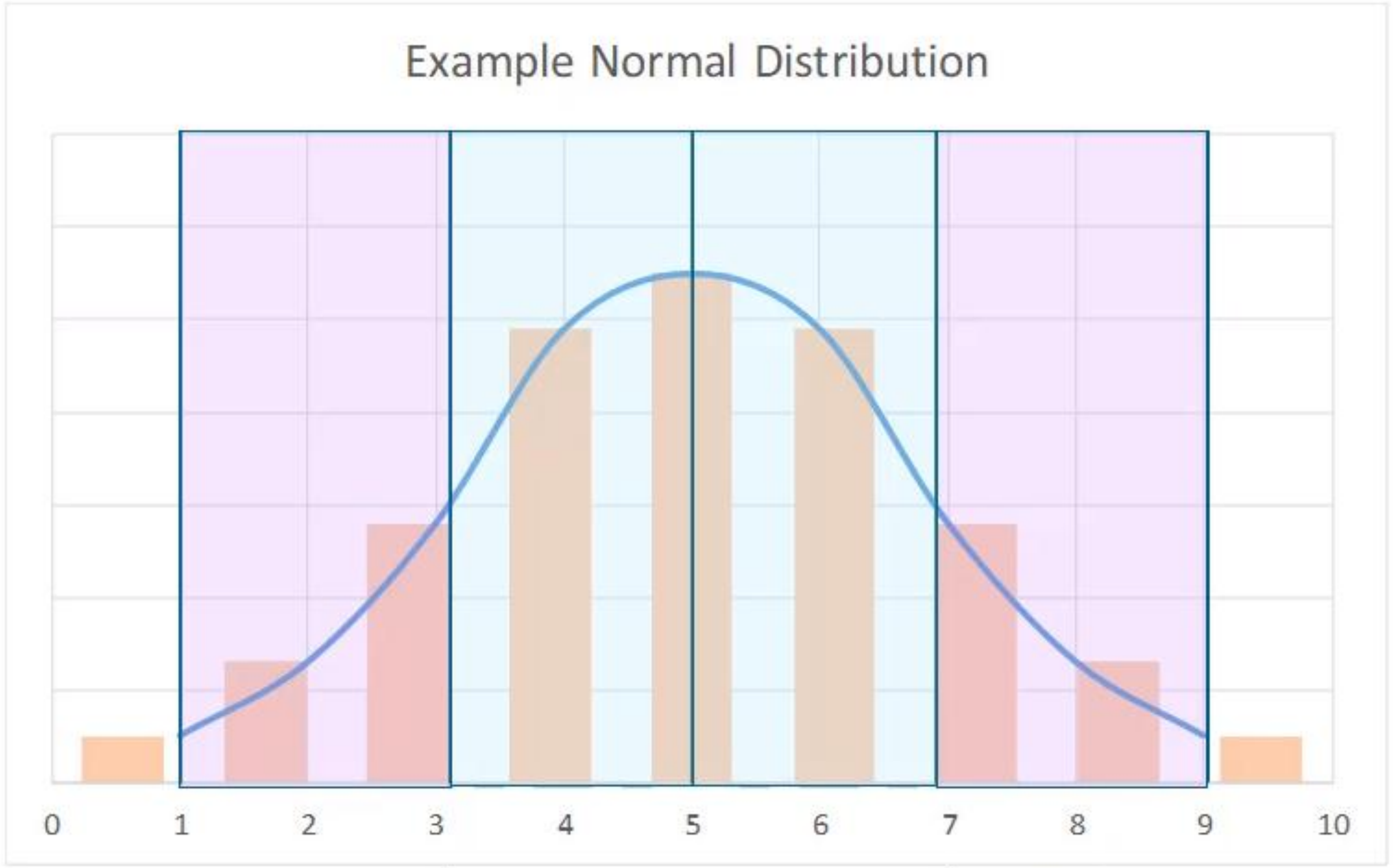
+/- 1σ = *173 – 197 lbs*

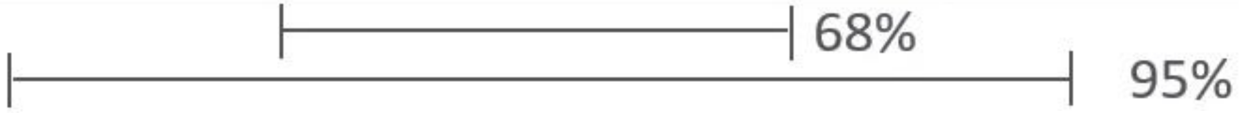
of all data is covered in this range

Aggregations: standard deviation

μ = 185 lbs

σ = 12 lbs



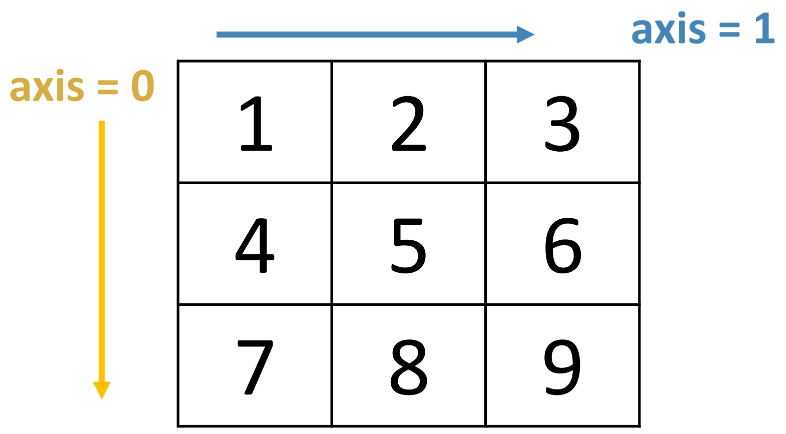


+/- 2σ = 161 – 209 lbs

of all data is covered in this range

Aggregations: min, max, mean, median, std

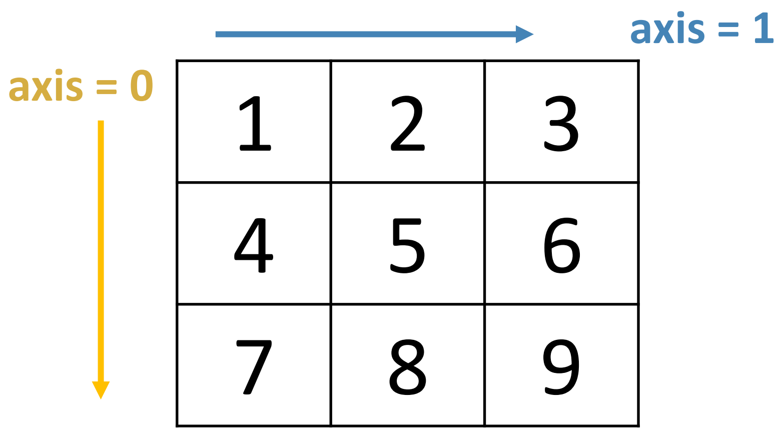
```python
# mean, median, max and min
print("Mean value is: ", arr.mean())
print("Median value is: ", np.median(arr))
print("Standard deviation is: ", arr.std())
print("Max value is: ", arr.max())
print("Min value is: ", arr.min())

# row wise and column wise min
# row wise and column wise min
print("Min per column", arr.min(axis=0))
print("Min per row", arr.min(axis=1))
```



```
Mean value is:  5.0
Median value is:  5.0
Standard deviation is:  2.581988897471611
Max value is:  9
Min value is:  1
Min per column (i.e. over all rows):  [1 2 3]
Min per row (i.e. over all columns)  [1 4 7]
```

Aggregations: sum

```python
#sum
print("Sum all items is: ", arr.sum())
print("Sum all items per column (i.e. over all rows): ", arr.sum(axis=0))
print("Sum all items per row (i.e. over all columns): ", arr.sum(axis=1))
```



```
Sum all items is:  45
Sum all items per column (i.e. over all rows):  [12 15 18]
Sum all items per row (i.e. over all columns):  [ 6 15 24]
```

Aggregations: percentiles and fractiles

```python
#array with hotel room prices
prices = np.array([187.00,220.00,228.00,258.00,260.00,280.00,294.00,298.00,314.00,4860.00])

#Find the quartiles (25th, 50th, and 75th percentiles) of the array
print(np.percentile(prices, [25, 50, 75]))

#find the 90th percentile of the array
print('90% of the room prices are lower than',round(np.percentile(prices, 90),2), '$')
```

```
[235.5 270.  297. ]
90% of the room prices are lower than 768.6 $
```

Read csv file into NumPy array

To load data in a csv into an array, numpy has several built-in functions:

- **no** missing values: numpy.**loadtxt**(fname, delimiter, …)
  more info:
  https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html

- missing values: numpy.**genfromtxt**(fname, delimiter, filling_values,…)
  more info:
  https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html#

Summary

- We’ve covered a lot of ground, but much of it was mainly new ways to code stuff
  - Creating numpy arrays, getting the properties, performing mathematical operations, extracting items, creating new arrays, reshaping and flattening them, read into a file
  - These you can look up easily if you know what you want to achieve
  - Practice using them!
- We’ve also covered some theoretical concepts
  - Mean, average, skewing, standard deviation
  - No amount of AI is going to help you on an exam if you don’t grasp these concepts