Data wrangling in Python - Data wrangling with NumPy - 2

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Perform operations on NumPy arrays	
Manipulating arrays using set operations	

Helper functions: min and max

- Arrays have many useful functions available
- For instance you can check the **minimum** and **maximum** values of a numeric array

```
import numpy as np

# Generate 5 numbers between 15 and 19.
x = np.linspace(15, 19, 5)

# Find the min of x.
np.amin(x)
```

```
15.0
```

```
# Find the max of x.
np.amax(x)
```

```
19.0
```

numpy.amin¶

numpy.amin(*a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>*) [source]

Return the minimum of an array or minimum along an axis.

numpy.amax

numpy.amax(*a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>*) [source]

Return the maximum of an array or maximum along an axis.

Helper functions: argmin and argmax

- You can obtain the index of the maximum and minimum values in the array using the `argmax` and `argmin` functions respectively

```
# Obtain index of max value
np.argmax(x)
```

```
4
```

```
# Obtain index of min value
np.argmin(x)
```

```
0
```

numpy.argmin

numpy.argmin(*a*, *axis=None*, *out=None*, ***, *keepdims=<no value>*) [source]

Returns the indices of the minimum values along an axis.

numpy.argmax

numpy.argmax(*a*, *axis=None*, *out=None*, ***, *keepdims=<no value>*) [source]

Returns the indices of the maximum values along an axis.

Helper functions: sum

- Or the **sum** of its elements

```
# Find the max of x.
np.sum(x)
```

```
85.0
```

numpy.sum

numpy.**sum**(*a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>*)

Sum of array elements over a given axis.

Helper functions: mean and median

- We can check the mean and median of the elements present in the numeric array

```
# Find the mean of x.
np.mean(x)
```

```
17.0
```

```
# Find the median of x.
np.median(x)
```

```
17.0
```

numpy.mean

numpy.mean(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, ***, *where=<no value>*) [source]

Compute the arithmetic mean along the specified axis.

numpy.median

numpy.median(*a*, *axis=None*, *out=None*, *overwrite_input=False*, *keepdims=False*)

Compute the median along the specified axis. [source]

Returns the median of the array elements.

Helper functions: digitize

- `digitize` is the function used to filter the elements of an array into respective bins
- Let's define an array with the bins and filter the elements of array `x`

```
# Filter array using digitize
bins = np.array([15, 17])
np.digitize(x,bins)
```

```
array([1, 1, 2, 2, 2])
```

numpy.digitize

numpy.digitize(*x*, *bins*, *right=False*) [source]

Return the indices of the bins to which each value in input array belongs.

right	order of bins	returned index *i* satisfies
False	increasing	bins[i-1] <= x < bins[i]
True	increasing	bins[i-1] < x <= bins[i]
False	decreasing	bins[i-1] > x >= bins[i]
True	decreasing	bins[i-1] >= x > bins[i]

If values in *x* are beyond the bounds of *bins*, 0 or `len(bins)` is returned as appropriate.

Helper functions: expand and squeeze

- You can also choose to change the dimensions of the array using the `expand_dims` and `squeeze` functions

```
# Expand dimensions of the array horizontally
np.expand_dims(x,axis=0)
```

```
array([[15., 16., 17., 18., 19.]])
```

```
# Expand dimensions of the array vertically
np.expand_dims(x,axis=1)
```

```
array([[15.],
       [16.],
       [17.],
       [18.],
       [19.]])
```

numpy.expand_dims

numpy.expand_dims(*a*, *axis*) [source]

Expand the shape of an array.

Insert a new axis that will appear at the *axis* position in the expanded array shape.

Helper functions: expand and squeeze (cont'd)

```
# Define an array in the format of lists
y = np.array([[10],[25],[28],[30]])
```

```
# Reduce dimensions of the array
np.squeeze(y)
```

```
array([10, 25, 28, 30])
```

numpy.squeeze

numpy.squeeze(*a*, *axis=None*) [source]

Remove axes of length one from *a*.

Convert an array to a list

- We can convert an array to a normal `list` with the `list` function
- Let's create the `evens` array and convert it to a list

```
evens = np.arange(0, 23, 2)
print(list(evens))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

Operations on arrays

- Numeric arrays of the same length can be added, subtracted, multiplied, or divided

```
# Save two arrays as variables.
a = np.array([1,1,1,1])
b = np.array([2,2,2,2])

# Addition of arrays.
print(a + b)
```

```
[3 3 3 3]
```

```
# Subtraction of arrays.
print(a - b)
```

```
[-1 -1 -1 -1]
```

```
# Multiplication of arrays.
print(a * b)
```

```
[2 2 2 2]
```

```
# Division of arrays.
print(a / b)
```

```
[0.5 0.5 0.5 0.5]
```

- In NumPy, these operations are defined **element-wise**
- In other words, each pair of corresponding elements in the two arrays is operated on, and the result is a new array containing each result

Module completion checklist

Objective	Complete
Perform operations on NumPy arrays	✔
Manipulating arrays using set operations	

Set operations

- We can also perform set based operations like `intersect`, `union` and `difference` on numeric arrays
- Setting the `return_indices` parameter to `True` displays the indices of the common elements in both the arrays respectively

```
# Retrieving common elements present in the
arrays.
np.intersect1d(a, b, return_indices=True)
```

```
(array([], dtype=int64), array([],
dtype=int64), array([], dtype=int64))
```

```
# Retrieving elements present in both the arrays.
np.union1d(a, b)
```

```
array([1, 2])
```

```
# Retrives the elements that are present only in
the first array.
np.setdiff1d(a, b)
```

```
array([1])
```

Stacking arrays

- Functions like `hstack` and `vstack` allow us to combine two arrays and stack them

```
# Stacking arrays horizontally.
np.hstack((a,b))
```

```
array([1, 1, 1, 1, 2, 2, 2, 2])
```

```
# Stacking arrays vertically.
np.vstack((a,b))
```

```
array([[1, 1, 1, 1],
       [2, 2, 2, 2]])
```

Mathematical functions on lists

- We **cannot** perform operations on lists
- If we wanted an absolute value of a list of numbers, we **can't** do this:

```
abs([-2, -7, 1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-e2459d669344> in <module>()
----> 1 abs([-2, -7, 1])

TypeError: bad operand type for abs(): 'list'
```

- The `TypeError` tells us that `abs` is not set up to handle lists

Mathematical functions on arrays

- Remember when we transformed a list into a NumPy array?
- Many functions in NumPy are **vectorized** functions, meaning they can handle a single input or an array of inputs
- When we use the same function `abs()` on an `np.` object, we see different results

```
print(np.abs(-3))
```

```
3
```

```
print(np.abs([-2, -7, 1]))
```

```
[2 7 1]
```

```
nums = np.arange(20, 30, .5)
print(len(nums))
```

```
20
```

User-defined functions on arrays

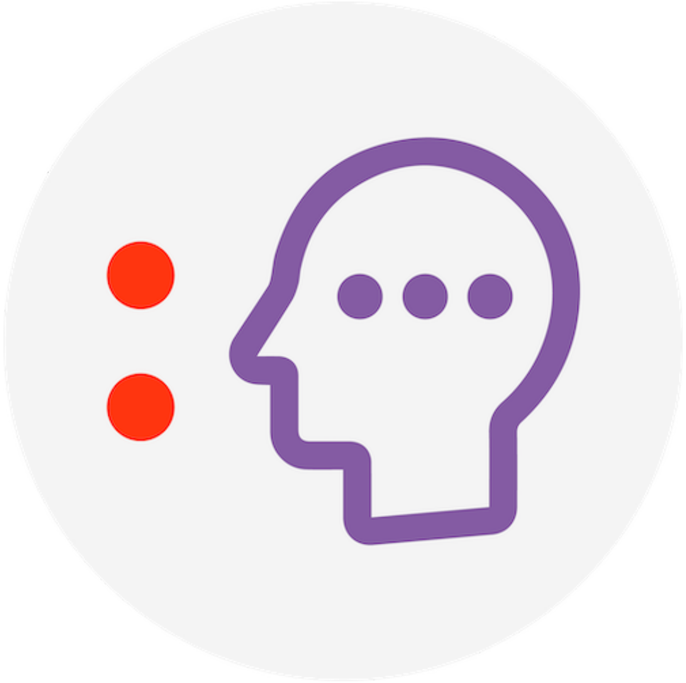
- We can also **write our own functions** to operate on arrays

```
# Define a function to multiply every element in array with 3 and add 1
def some_calculation(arr):
    return 3*arr+1

print(some_calculation(nums))
```

```
[61.  62.5 64.  65.5 67.  68.5 70.  71.5 73.  74.5 76.  77.5 79.  80.5
 82.  83.5 85.  86.5 88.  89.5]
```

Knowledge check



Link: ***https://forms.gle/DTzmnjH5yjUfhD9Q7***

Module completion checklist

Objective	Complete
Perform operations on NumPy arrays	✔
Manipulating arrays using set operations	✔

Congratulations on completing this module!

You are now ready to try Tasks 3-6 in the Exercise for this topic