DATA SOCIETY®

Day 2 - Data wrangling in Python

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Last session's material recap

- Did you have any problems working through the materials of the last session?
- Are there any concepts from last session that are still unclear?
- The topics we covered in the last session included:
  - Programming across industries and core functions of data scientists
  - Data science use cases for Python
  - Functions in Python

- In this module, we will explore **functionalities in numpy library and understand arrays**
- We will also introduce another important library in python, called pandas~~

Knowledge checks and exercises

- **Knowledge Checks:**
  - At the end of each mini-lesson, the instructor will launch a poll to check your knowledge
  - Read the questions carefully and answer each question

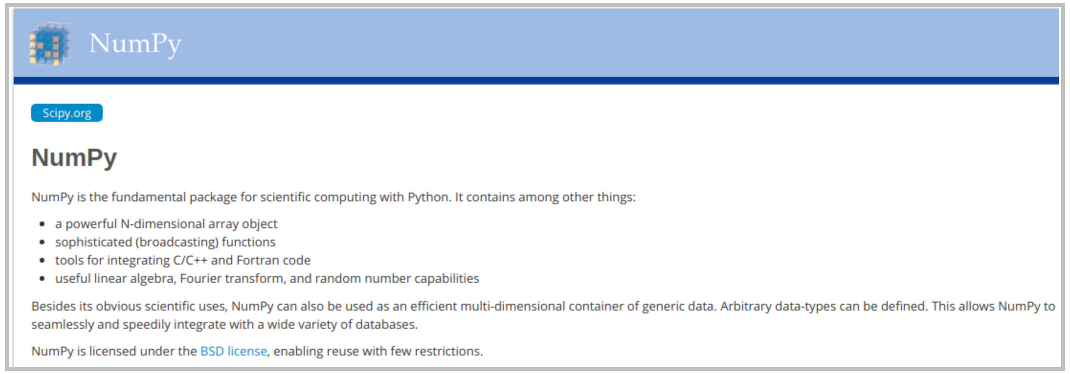
- **Exercises:**
  - To complete the exercises, open the file named "Exercises Day 2" provided at the time of enrollment

Module completion checklist

Objective	Complete
Illustrate NumPy objects	
Discuss filtering and reshaping arrays	
Summarize use cases of pandas and update directory settings	

Introduction to NumPy

- NumPy is widely used in machine learning and scientific computing due to its basic core data structure: **array**
- It is also widely used in combination with `matplotlib` and other plotting libraries to create graphs
- NumPy's array functions are similar to those available for vectors in Matlab and R


NumPy

Scipy.org

NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the BSD license, enabling reuse with few restrictions.

Creating arrays

- There are multiple ways to create a numpy array
- One of the easiest is to make it from a `list` and using NumPy's `array()` function
- To use the `array()` function, we need to import `numpy`
- Once again, when writing code, we usually want to **import all packages needed for the program at the beginning**
- However, **since we are learning as we go, we import them as we learn in class**

```
# Import numpy as 'np' sets 'np' as the shortcut/alias.
import numpy as np

# Create an array from a list.
arr = np.array([17, -10, 16.8, 11])
print(arr)

# Check the type of the object.
```

```
[ 17.  -10.   16.8  11. ]
```

```
print(type(arr))
```

```
<class 'numpy.ndarray'>
```

Dtype in arrays

- NumPy arrays have a property of `dtype` which records the data type of the array's members
- NumPy arrays are **required to have the same data type**, that is why they are called `atomic` data structures (i.e. structures that allow a single data type)!

```
# Check the data type stored in the array.
print(arr.dtype)
```

```
float64
```

Using ndarray

- The most important data type that NumPy provides is the "N-dimensional array," `ndarray`
- An `ndarray` is similar to a Python list in which all members have the same data type
- We create it using `np.array()`

```
x = np.array([3, 19, 7, 11])
print(x)
```

```
[ 3 19  7 11]
```

Documentation for ndarray

- Each package in Python has ***documentation*** for each function within

array

A homogeneous container of numerical elements. Each element in the array occupies a fixed amount of memory (hence homogeneous), and can be a numerical element of a single type (such as float, int or complex) or a combination (such as `(float, int, float)`). Each array has an associated data-type (or `dtype`), which describes the numerical type of its elements:

```
>>> x = np.array([1, 2, 3], float)

>>> x
array([ 1.,  2.,  3.])

>>> x.dtype # floating point number, 64 bits of memory per element
dtype('float64')


# More complicated data type: each array element is a combination of
# and integer and a floating point number
>>> np.array([(1, 2.0), (3, 4.0)], dtype=[('x', int), ('y', float)])
array([(1, 2.0), (3, 4.0)],
      dtype=[('x', '<i4'), ('y', '<f8')])
```

Fast element-wise operations, called a ufunc, operate on arrays.

Building an array with linspace

- Another function we can use to build an array is `np.linspace`

```
y = np.linspace(-2, -1, 25)
print(y)
```

```
[-2.         -1.95833333 -1.91666667
-1.875      -1.83333333 -1.79166667
 -1.75       -1.70833333 -1.66666667
-1.625      -1.58333333 -1.54166667
 -1.5        -1.45833333 -1.41666667
-1.375      -1.33333333 -1.29166667
 -1.25       -1.20833333 -1.16666667
-1.125      -1.08333333 -1.04166667
 -1.        ]
```

- This function will return 25 evenly-spaced numbers between -2 and -1

numpy.linspace

numpy.linspace(*start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0*)

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

Alternative ways of accessing functions

- Another way, which can be useful if you are only going to use a handful of functions from a library, is as follows:

```
from numpy import array, linspace
x = array([0.01, 0.45, -0.3])
y = linspace(0, 1, 50)
```

- With this syntax, we can use array or linspace without the np. prefix

NumPy array data types

Data type	Description
“bool_”	Boolean (True or False) stored as a byte
“int_”	Default integer type (same as C “long”; normally either “int64” or “int32”)
“intc”	Identical to C “int” (normally “int32” or “int64”)
“intp”	Integer used for indexing (same as C “ssize_t”; normally either “int32” or “int64”)
“int8”	Byte (-128 to 127)
“int16”	Integer (-32768 to 32767)
“int32”	Integer (-2147483648 to 2147483647)

NumPy array data types (cont'd)

Data type	Description
"uint8"	Unsigned integer (0 to 255)
"uint16"	Unsigned integer (0 to 65535)
"uint32"	Unsigned integer (0 to 4294967295)
"uint64"	Unsigned integer (0 to 18446744073709551615)
"float_"	Byte (-128 to 127)
Shorthand for "float64"	Integer (-32768 to 32767)
"float16"	Integer (-2147483648 to 2147483647)
"int64"	Integer (-9223372036854775808 to 9223372036854775807)

Arrays vs Lists

- Unlike lists, NumPy arrays can hold values of only a single data type
- This makes arrays much more powerful for vectorized complex manipulations
- Let's see what happens if we try to create an array from a list of mixed data types

```
mixed_array = np.array([1, 2, "apple", "XYZ",
5.5])
print(mixed_array)
```

```
['1' '2' 'apple' 'XYZ' '5.5']
```

```
print(mixed_array.dtype)
```

```
<U21
```

- `<U11` is a data type for Unicode strings

This means that all values in the initial list are **cast** into string data type to maintain homogeneity

- Similarly, creating an array from a list of integer and float values, changes all elements to float data type

```
mixed_array = np.array([3, 12, 5.56])
print(mixed_array)
```

```
[ 3.    12.     5.56]
```

```
print(mixed_array.dtype)
```

```
float64
```

You can read more about NumPy data types ***here***

Arrays from sequences

- We can also create an array that contains a sequence of numbers
- To create the range of numbers of 0 to 50, use the `arange` command

numpy.arange

numpy.arange([*start*,]*stop*, [*step*,]*dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use numpy.linspace for these cases.

Arrays from sequences (cont'd)

```
rng = np.arange(0, 51)
print(rng)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50]
```

- The last number in the range is one less than the value you provided, so we provide 51 to ensure that the last value is 50

Arrays from sequences - using a step size

- We can also have the numbers increase by a step size other than 1

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

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

```
quarters = np.arange(0, 1, .25)  #<- contains 0 to 0.75
print(quarters)
```

```
[0.   0.25 0.5  0.75]
```

Helper functions: min, max, and sum

- Arrays have many useful functions available
- For instance, for numeric arrays, you can check their maximum, minimum, or sum value

```
# 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>*)

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>*)

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

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.

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

```
85.0
```

Convert an array to a list

- We can convert an array to a normal `list` with the `list` function
- We will demonstrate that with the array we created earlier, `evens`

```
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

Mathematical functions on lists

- You might be wondering if we can perform operations on lists
  - The answer is **no**!
- 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 1

Exercise 1

Module completion checklist

Objective	Complete
Illustrate NumPy objects	✔
Discuss filtering and reshaping arrays	
Summarize use cases of pandas and update directory settings	

Accessing array values

- Just like with lists, we can grab individual elements or a range of elements from an array using square bracket notation

```
# Import numpy as 'np' sets 'np' as the shortcut/alias.
import numpy as np

nums = np.arange(20, 30, .5) #<- Create array
print(len(nums)) #<- get the length of array
```

```
20
```

```
print(nums[1])  #<- get the second element
```

```
20.5
```

```
print(nums[0:3]) #<- get the first three elements
```

```
[20.  20.5 21. ]
```

Logical filtering

- You can't filter lists by a logical condition, however **you can filter arrays by a logical condition**
- If the corresponding condition is met, then it retains the value from the array, otherwise it excludes it

```
print(nums)
```

```
[20.  20.5 21.  21.5 22.  22.5 23.  23.5 24.  24.5 25.  25.5 26.  26.5
 27.  27.5 28.  28.5 29.  29.5]
```

```
large_nums = nums[nums > 26]
print(large_nums)
```

```
[26.5 27.  27.5 28.  28.5 29.  29.5]
```

Logical filtering (cont'd)

```
print(nums)
```

```
[20.  20.5 21.  21.5 22.  22.5 23.  23.5 24.  24.5 25.  25.5 26.  26.5
 27.  27.5 28.  28.5 29.  29.5]
```

```
large_nums = nums[nums > 26]
print(large_nums)
```

```
[26.5 27.  27.5 28.  28.5 29.  29.5]
```

- It is important to remember that there are a few steps happening here:
    - The expression within the brackets produces a so-called **Boolean mask**: an array of `True/False` values
    - The logical statement `> 26`, is applied to each value of `nums`, so the result is an array of `True`/`False` values
    - Our `nums` array and the mask array are then lined up, and the values out of `nums` are filtered based on the corresponding mask value

Two-dimensional arrays

- As the name suggests, `ndarray` (i.e. n-dimensional array) can have more than one dimension!
- Multiple dimensions are created by nesting lists within each other
- To create a 2D array (a matrix), we can write the following:

```
mat = np.array([
        [8, 2, 6, 8],
        [4, 5, 7, 2],
        [3, 9, 7, 1]
       ])
print(mat)
```

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

Two-dimensional arrays - shape

- The `shape` property of an array tells us the size of each of its dimensions

numpy.ndarray.shape

ndarray.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with numpy.reshape, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

numpy.reshape similar function
ndarray.reshape similar method

Two-dimensional arrays - shape (cont'd)

```
print(mat.shape) #<- 3 rows and 4 columns -- returned as a tuple
```

```
(3, 4)
```

```
nrows, ncols = mat.shape
print(nrows)
```

```
3
```

Two-dimensional arrays - extracting elements

- To extract a value from the matrix, we use 2-dimensional bracket notation:
  - **1st number is the row position**
  - **2nd is the column position**

```
print(mat[1, 3]) #<- 2nd row 4th column - remember that indexing starts at 0!
```

```
2
```

Two-dimensional arrays - rows

- To extract an entire row of a matrix, replace the column ID with colon
- The colon indicates that you would like to include all of the columns
- Alternatively, you can specify a range of column positions, which uses normal Python list slicing notation

```
print(mat[0, :]) #<- first row
```

```
[8 2 6 8]
```

```
print(mat[0, 0:2]) #<- first row and just first 2 columns
```

```
[8 2]
```

Two-dimensional arrays - columns

- Similarly, to extract a single column, replace the row argument with a colon or leave it blank

```
print(mat[:, 2]) #<- 3rd column
```

```
[6 7 7]
```

```
print(mat[1:3, 2]) #<- 3rd column but skipping over the first row
```

```
[7 7]
```

```
print(mat[1:3, 2:3]) #<- same as previous, but maintains the vertical structure of the column
```

```
[[7]
 [7]]
```

Reshaping arrays

Sometimes we may need to reshape an array according to our needs

- We can do so by calling `.reshape()` function on an array and passing the **new shape** as an argument

```
arr = np.arange(1,13)
print(arr)
```

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

```
print(arr.reshape(3, 4))
```

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

numpy.reshape

numpy.reshape(*a*, *newshape*, *order='C'*) [source]

Gives a new shape to an array without changing its data.

Parameters:

a : *array_like*

Array to be reshaped.

newshape : *int or tuple of ints*

The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

order : *{'C', 'F', 'A'}, optional*

Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

Returns:

reshaped_array : *ndarray*

This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the *memory layout* (C- or Fortran- contiguous) of the returned array.

Reshaping arrays (cont'd)

- We can also specify one of the new dimensions and let Python infer the other dimension, given the number of elements in the array if possible

```
print(arr.reshape(2,        #<- specify number of rows=2
              -1))    #<- number of columns=-1 lets Python infer it
```

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

If the array cannot be reshaped to the given dimensions, Python throws an error

```
print(arr.reshape(5,        #<- specify number of rows=5
              -2))    #<- number of columns=-2 lets Python infer it
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-c2ef230382ec> in <module>
----> 1 arr.reshape(5,-1)

ValueError: cannot reshape array of size 12 into shape (5,newaxis)
```

Module completion checklist

Objective	Complete
Illustrate NumPy objects	✔
Discuss filtering and reshaping arrays	✔
Summarize use cases of pandas and update directory settings	

Data wrangling and exploration

- As we learned earlier, a data scientist must be able to:

1. **Wrangle** the data (gather, clean, and sample data to get a suitable dataset)
2. **Manage** the data for easy access by the organization
3. **Explore** the data to generate a hypothesis

- Today, we will learn how to use one of the most powerful Python libraries, **Pandas**, that will help us achieve these goals!

Dataset manipulation with Pandas

- `Pandas` is a powerful library for **cleaning and analyzing datasets in Python**
- Dataset is a collection of data, usually in tabular format, where
  - Every column represents a particular **variable**
  - Every row represents a given **record**
- We learned about `numpy`, which helps us work with datasets, specifically arrays of numbers, to get ready for machine learning
- `Pandas` will help us with cleaning and analyzing datasets of all kinds
- For complete documentation, ***click here***

A little more about Pandas

- Pandas is an effective tool to **read, write and manipulate data**
- Pandas contains tools to perform high-performance **merging and joining datasets**
- Pandas is **highly optimized for performance**, with critical code paths written in C

Import Pandas and os

- Let's import the `pandas` library
- Note: it is not required that you also import `numpy` in order to use `pandas`
- However, you will often see both of them imported since many projects make use of both

```
import pandas as pd
```

- We now are going to introduce a package that allows you to set your working directory
- This will be the directory where your data lies, allowing you to import data directly from there

```
import os
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `skill-soft` folder

```
# Set `main_dir` to the location of your `skill-soft` folder (for Linux).
main_dir = "/home/[username]/Desktop/skill-soft"
```

```
# Set `main_dir` to the location of your `skill-soft` folder (for Mac).
main_dir = '/Users/[username]/Desktop/skill-soft'
```

```
# Set `main_dir` to the location of your `skill-soft` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\skill-soft"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"
```

Working directory

- Set working directory to the `data_dir` variable we set
- We do this using the `os.chdir` function, change directory
- We can then check the working directory using `.getcwd()`
- For complete documentation of the `os` package, ***click here***

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/skill-soft/data
```

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Illustrate NumPy objects	✔
Discuss filtering and reshaping arrays	✔
Summarize use cases of pandas and update directory settings	✔

Summary and next steps

In this module, we:

- Explored numpy and pandas packages
- Created, filtered and reshaped NumPy arrays
- Updated Directory settings for efficient workflow

In the next module, we will:

- Perform basic operations on Pandas series
- Learn to use data frames and load data sets using Pandas
- Summarize and reshape data using Pandas

This completes our module

Congratulations!