Welcome to Data Science

Congratulations on taking a big step toward becoming a data scientist!
In addition to working through this course, be sure to take advantage of all of the learning support available to you on SoloLearn, including the daily tips, Try it Yourself practices, code coach challenges, code playground, and engagement with our amazing learner community. We love to hear from you, so please leave comments and feedback as you learn with us.

Let's get started!

What is Data Science?

There are many use cases in business for data science including finding a better housing price prediction algorithm for Zillow, finding key attributes associated with wine quality, and building a recommendation system to increase the click-through-rate for Amazon.

Extracting insights from seemingly random data, data science normally involves collecting data, cleaning data, performing **exploratory data analysis,** building and evaluating **machine learning** models, and communicating insights to stakeholders.

Data science is a multidisciplinary field that unifies statistics, data analysis, machine learning and their related methods to extract knowledge and insights.

Why Python?

In this **Introduction to Data Science** course we're learning data science with Python. As a **general-purpose programming language**, Python is now the most popular programming language in data science. It's easy to use, has great community support, and integrates well with other frameworks (e.g., web applications) in an engineering environment.

This course focuses on **exploratory data analysis** with three fundamental Python libraries: **numpy, pandas** and **matplotlib**. The machine learning library **scikit-learn** will be covered as well.

In the later modules, we will be predicting home values using linear regression, identifying classes of iris with classification algorithms, and finding clusters within wines, just a few examples of what we can do in data science.

In data science, there are other popular programming languages, such as R, which has an edge in statistical modeling.

Numerical Data

Datasets come from a wide range of sources and formats: it could be collections of numerical measurements, text corpus, images, audio clips, or basically anything. No matter the format, the first step in data science is to transform it into arrays of numbers.

We collected 45 U.S. president heights in centimeters in chronological order and stored them in a **list**, a built-in data type in python.

```
heights = [189, 170, 189, 163, 183, 171, 185, 168,
173, 183, 173, 173, 175, 178, 183, 193, 178, 173, 174,
183, 183, 180, 168, 180, 170, 178, 182, 180, 183, 178,
182, 188, 175, 179, 183, 193, 182, 183, 177, 185, 188,
188, 182, 185, 191]
```

In this example, George Washington was the first president, and his height was 189 cm.

If we wanted to know how many presidents are taller than 188cm, we could iterate through the list, compare each element against 188, and increase the count by 1 as the criteria is met.

```
cnt = 0
for height in heights:
  if height > 188:
    cnt +=1
print(cnt)
```

Try it Yourself

This shows that there are five presidents who are taller than 188 cm.

No matter the format of the data, the first step in data science is to transform it into arrays of numbers.

Introduction to Numpy

Numpy (short for **Num**erical **Py**thon) allows us to find the answer to how many presidents are taller than 188cm with ease. Below we show how to use the library and start with the basic object in numpy.

```
import numpy as np
heights_arr = np.array(heights)
print((heights_arr > 188).sum())
```

Try it Yourself

The **import statement** allows us to access the functions and modules inside the numpy library. The library will be used frequently, so by convention numpy is imported under a shorter name, np. The second line is to convert the list into a numpy array object, via np.array(), that tools provided in numpy can work with. The last line provides a simple and natural solution, enabled by numpy, to the original question.

As our datasets grow larger and more complicated, numpy allows us the use of a more efficient and for-loop-free method to manipulate and analyze our data. Our dataset example in this module will include the US Presidents' height, age and party.

Python modules can get access to code from another module by importing the file/function using the **import** statement.

Size and Shape

An array class in Numpy is called an **ndarray** or n-dimensional array. We can use this to count the number of presidents in heights_arr, use attribute **numpy.ndarray.size**

```
heights_arr.size
```

PY

Try it Yourself

Note that once an array is created in numpy, its size cannot be changed.

Size tells us how big the array is, shape tells us the dimension. To get current shape of an array use attribute shape:

```
heights_arr.shape
```

PY

Try it Yourself

The output is a **tuple**, recall that the built-in data type tuple is immutable whereas a list is mutable, containing a single value, indicating that there is only one dimension, i.e., axis 0. Along axis 0, there are 45 elements (one for each president) Here, heights_arr is a 1d array.

Attribute **size** in numpy is similar to the built-in method len in python that is used to compute the length of iterable python objects like str, list, dict, etc.

Reshape

Other data we have collected includes the ages of the presidents:

```
ages = [57, 61, 57, 57, 58, 57, 61, 54, 68, 51, 49,
64, 50, 48, 65, 52, 56, 46, 54, 49, 51, 47, 55, 55,
54, 42, 51, 56, 55, 51, 54, 51, 60, 62, 43, 55, 56,
61, 52, 69, 64, 46, 54, 47, 70]
```

Since both heights and ages are all about the same presidents, we can combine them:

```
heights_and_ages = heights + ages
# convert a list to a numpy array
heights_and_ages_arr = np.array(heights_and_ages)
heights_and_ages_arr.shape
```

Try it Yourself

This produces one long array. It would be clearer if we could align height and age for each president and reorganize the data into a 2 by 45 matrix where the first row contains all heights and the second row contains ages. To achieve this, a new array can be created by calling **numpy.ndarray.reshape** with new dimensions specified in a tuple:

```
heights_and_ages_arr.reshape((2,45))
```

Try it Yourself

The reshaped array is now a 2darray, yet note that the original array is not changed. We can reshape an array in multiple ways, as long as the size of the reshaped array matches that of the original.

Numpy can calculate the shape (dimension) for us if we indicate the unknown dimension as -1. For example, given a 2darray `arr` of shape (3,4), arr.reshape(-1) would output a 1darray of shape (12,), while arr.reshape((-1,2)) would generate a 2darray of shape (6,2).

Data Type

Another characteristic about numpy array is that it is **homogeneous**, meaning each element must be of the same data type.

For example, in heights_arr, we recorded all heights in whole numbers; thus each element is stored as an integer in the array. To check the data type, use numpy.ndarray.dtype

```
heights_arr.dtype
```

PY

Try it Yourself

If we mixed a float number in, say, the first element is 189.0 instead of 189:

```
heights_float = [189.0, 170, 189, 163, 183, 171, 185,
168, 173, 183, 173, 173, 175, 178, 183, 193, 178, 173,
174, 183, 183, 180, 168, 180, 170, 178, 182, 180, 183,
178, 182, 188, 175, 179, 183, 193, 182, 183, 177, 185,
188, 188, 182, 185, 191]
```

Then after converting the list into an array, we'd see all other numbers are coerced into floats:

```
heights_float_arr = np.array(heights_float)
heights_float_arr
heights_float_arr.dtype
```

PY

Try it Yourself

Numpy supports several data types such as int (integer), float (numeric floating point), and bool (boolean values, True and False). The number after the data type, ex. int64, represents the bitsize of the data type.

Indexing

We can use array indexing to select individual elements from arrays. Like Python lists, numpy index starts from 0.

To access the height of the 3rd president Thomas Jefferson in the 1darray 'heights_arr':

```py
heights_arr[2]
```

Try it Yourself

In a 2darray, there are two axes, axis 0 and 1. Axis 0 runs downward down the rows whereas axis 1 runs horizontally across the columns.

In the 2darrary heights_and_ages_arr, recall that its dimensions are (2, 45). To find Thomas Jefferson's age at the beginning of his presidency you would need to access the second row where ages are stored:

```py
heights_and_ages_arr[1,2]
```

Try it Yourself

In 2darray, the row is axis 0 and the column is axis 1, therefore, to access a 2darray, numpy first looks for the position in rows, then in columns. So in our example heights_and_ages_arr[1,2], we are accessing row 2 (ages), column 3 (third president) to find Thomas Jefferson's age.

Slicing

What if we want to inspect the first three elements from the first row in a 2darray? We use ":" to select all the elements from the index up to but not including the ending index. This is called **slicing**

```
heights_and_ages_arr[0, 0:3]
```

PY

Try it Yourself

When the starting index is 0, we can omit it as shown below:

```
heights_and_ages_arr[0, :3]
```

PY

Try it Yourself

What if we'd like to see the entire third column? Specify this by using a ":" as follows

```
heights_and_ages_arr[:, 3]
```

PY

Try it Yourself

Numpy slicing syntax follows that of a python list: **arr[start:stop:step]**. When any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1.

Assigning Single Values

Sometimes you need to change the values of particular elements in the array. For example, we noticed the fourth entry in the heights_arr was incorrect, it should be 165 instead of 163, we can re-assign the correct number by:

```python
heights_arr[3] = 165
```

Try it Yourself

In a 2darray, single values can be assigned easily. You can use indexing for one element. For example, change the fourth entry in heights_arr to 165:

```python
heights_and_ages_arr[0, 3] = 165
heights_and_ages_arr
```

Try it Yourself

Or we can use **slicing** for multiple elements. For example, to replace the first row by its mean 180 in heights_and_ages_arr:

```python
heights_and_ages_arr[0,:] = 180
heights_and_ages_arr
```

Try it Yourself

We can also combine slicing to change any subset of the array. For example, to reassign 0 to the left upper corner:

```python
heights_and_ages_arr[:2, :2] = 0
heights_and_ages_arr
```

Try it Yourself

It is easy to update values in a subarray when you combine arrays with slicing. For more on basic slicing and advanced indexing in numpy check out this link.

Assigning an Array to an Array

In addition, a 1darray or a 2darry can be assigned to a subset of another 2darray, as long as their shapes match. Recall the 2darray heights_and_ages_arr:

```python
heights_and_ages_arr
```

Try it Yourself

If we want to update both height and age of the first president with new data, we can supply the data in a list:

```python
heights_and_ages_arr[:, 0] = [190, 58]
heights_and_ages_arr
```

Try it Yourself

We can also update data in a subarray with a numpy array as such:

```python
new_record = np.array([[180, 183, 190], [54, 50, 69]])
heights_and_ages_arr[:, 42:] = new_record
heights_and_ages_arr
```

Try it Yourself

Note the last three columns' values have changed.

Updating a multidimensional array with a new record is straightforward in numpy as long as their shapes match.

Combining Two Arrays

Oftentime we obtain data stored in different arrays and we need to combine them into one to keep it in one place. For example, instead of having the ages stored in a list, it could be stored in a 2darray:

```
ages_arr.shape
ages_arr[:3,]
```

Try it Yourself

If we reshape the heights_arr to (45,1), the same as 'ages_arr', we can stack them horizontally (by column) to get a 2darray using '**hstack**':

```
heights_arr = heights_arr.reshape((45,1))
height_age_arr = np.hstack((heights_arr, ages_arr))
height_age_arr.shape
height_age_arr[:3,]
```

Try it Yourself

Now height_age_arr has both heights and ages for the presidents, each column corresponds to the height and age of one president.

Similarly, if we want to combine the arrays vertically (by row), we can use 'vstack'.

```
heights_arr = heights_arr.reshape((1,45))
ages_arr = ages_arr.reshape((1,45))

height_age_arr = np.vstack((heights_arr, ages_arr))
height_age_arr.shape
height_age_arr[:,:3]
```

Try it Yourself

! To combine more than two arrays horizontally, simply add the additional arrays into the tuple.

Concatenate

More generally, we can use the function numpy.concatenate. If we want to **concatenate**, link together, two arrays along rows, then pass 'axis = 1' to achieve the same result as using numpy.hstack; and pass 'axis = 0' if you want to combine arrays vertically.

In the example from the previous part, we were using hstack to combine two arrays horizontally, instead:

```
height_age_arr = np.concatenate((heights_arr,
ages_arr), axis=1)
```

PY

Try it Yourself

Also you can get the same result as using vstack:

```
height_age_arr = np.concatenate((heights_arr,
ages_arr), axis=0)
```

PY

Try it Yourself

You can use np.hstack to concatenate arrays ONLY if they have the **same number** of **rows**.

Mathematical Operations on Arrays

Performing mathematical operations on arrays is straightforward. For instance, to convert the heights from centimeters to feet, knowing that 1 centimeter is equal to 0.0328084 feet, we can use multiplication:

```
height_age_arr[:,0]*0.0328084
```

PY

Try it Yourself

Now we have all heights in feet. Note that this operation won't change the original array, it returns a new 1darray where 0.0328084 has been multiplied to each element in the first column of 'heights_age_arr'.

Other mathematical operations for addition, subtraction, division and power (+, -, /, **) work the same way on arrays.

Numpy Array Method

In addition, there are several methods in numpy to perform more complex calculations on arrays. For example, the **sum()** method finds the sum of all the elements in an array:

```
height_age_arr.sum()
```

PY

Try it Yourself

The sum of all heights and ages is 10575. In order to sum all heights and sum all ages separately, we can specify axis=0 to calculate the sum across the rows, that is, it computes the sum for each column, or column sum. On the other hand, to obtain the row sums specify axis=1. In this example, we want to calculate the total sum of heights and ages, respectively:

```
height_age_arr.sum(axis=0)
```

PY

Try it Yourself

The output is the row sums: heights of all presidents (i.e., the first row) add up to 8100, and the sum of ages (i.e., the second row) is 2475.

Other operations, such as .min(), .max(), .mean(), work in a similar way to .sum().

Comparisons

In practicing data science, we often encounter comparisons to identify rows that match certain values. We can use **operations** including "<", ">", ">=", "<=", and "==" to do so. For example, in the height_age_arr dataset, we might be interested in only those presidents who started their presidency younger than 55 years old.

```
height_age_arr[:, 1] < 55
```

PY

Try it Yourself

The output is a 1darray with boolean values that indicates which presidents meet the criteria. If we are only interested in which presidents started their presidency at 51 years of age, we can use "==" instead.

```
height_age_arr[:, 1] == 51
```

PY

Try it Yourself

To find out how many rows satisfy the condition, use .sum() on the resultant 1d boolean array, e.g., (height_age_arr[:, 1] == 51).sum(), to see that there were exactly five presidents who started the presidency at age 51. True is treated as 1 and False as 0 in the sum.

Mask & Subsetting

Now that rows matching certain criteria can be identified, a subset of the data can be found. For example, instead of the entire dataset, we want only tall presidents, that is, those presidents whose height is greater than or equal to 182 cm. We first create a **mask**, 1darray with boolean values:

```python
mask = height_age_arr[:, 0] >= 182
mask.sum()
```

Try it Yourself

Then pass it to the first axis of `height_age_arr` to filter presidents who don't meet the criteria:

```python
tall_presidents = height_age_arr[mask, ]
tall_presidents.shape
```

Try it Yourself

This is a subarray of height_age_arr, and all presidents in tall_presidents were at least 182cm tall.

! Masking is used to extract, modify, count, or otherwise manipulate values in an array based on some criterion. In our example, the criteria was height of 182cm or taller.

Multiple Criteria

We can create a **mask** satisfying more than one criteria. For example, in addition to height, we want to find those presidents that were 50 years old or younger at the start of their presidency. To achieve this, we use **&** to separate the conditions and each condition is encapsulated with parentheses "()" as shown below:

```
mask = (height_age_arr[:, 0]>=182) &
(height_age_arr[:,1]<=50)
height_age_arr[mask,]
```

Try it Yourself

The results show us that there are four presidents who satisfy both conditions.

Data manipulation in Python is nearly synonymous with Numpy array manipulation. Operations shown here are the building blocks of many other examples used throughout this course. It is important to master them!