Introduction to Numpy

Introduction

In this section, we'll take a more formal look at *NumPy*. Besides being ubiquitous in Data Science, NumPy also provides us with blistering fast and efficient, list-like, data types called N-Dimensional Arrays or **ndarrays** or more simply arrays. This list-like data type is effectively a lighter weight version of a Python **list**, as it uses less of your computer's memory, which makes it more efficient, especially when dealing with large datasets. Don't worry if that seems a little vague. We will take a closer look at NumPy and how its arrays work in this lesson.

An important note: *Pandas* was actually built on top of *NumPy*! So many of the functionalities that *NumPy* has, are also part of *Pandas*. It is still important to cover *NumPy* separately as *NumPy* arrays are very important building blocks in many Data Science applications!

Objectives

You will be able to:

- Describe why NumPy is used at times over standard Python
- Instantiate a NumPy array with specified values
- Use broadcasting to perform a math operation on an entire NumPy array

Getting Started With NumPy

Just like with any other library, we need to first install the library with a package manager like `pip` . We have already installed this library in the background, so, no need to worry about this step. Next, we need to import the dependency into our code.

The conventional method to import NumPy is by aliasing it as `np` , like this:

In [5]:
```
import numpy as np
```

That was easy! Now we can use any functions from the NumPy library by simply typing `np.function_name()` . For example, if we wanted to create a NumPy array containing the values 1, 2, 3, and 4, we would write `np.array([1,2,3,4])` . Let's try it out below:

In [3]:
```
numpy_arr = np.array([1, 2, 3, 4])
print('Here is a NumPy array:', numpy_arr)
print('You know it is a NumPy array because its type is:', type(numpy_arr))
```

```
Here is a NumPy array: [1 2 3 4]
You know it is a NumPy array because its type is: <class 'numpy.ndarray'>
```

While we'll focus on one-dimensional arrays in this lesson, it is important to mention that NumPy is very famous for its multi-dimensional arrays, like:

```
In [6]: numpy_ndarr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
        numpy_ndarr

Out[6]: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])
```

The benefits of using NumPy here will become more obvious in our math-heavier, linear algebra / matrices sections!

Performing array operations

So why would you want to use a NumPy array instead of a list? Because compared to a list, NumPy makes it very easy to perform array operations, like adding, multiplying, and otherwise operating on each element of the array.

First, let's take a look at a simple example. We have a list of integers, and we want to add 3 to each element in the list. One might try the following:

```
In [1]: list_of_integers = [0, 1, 2, 3]
```

```
In [2]: # Add 3 to each element
        list_of_integers + 3

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        /tmp/ipykernel_117/569701397.py in <module>
              1 # Add 3 to each element
        ----> 2 list_of_integers + 3

        TypeError: can only concatenate list (not "int") to list
```

You'll see that this doesn't work, because Python expects a list-like object. And even if you convert the integer 3 to a list-like element, you won't exactly get the desired result.

```
In [10]: # Add 3 to each element
         list_of_integers + [3]

Out[10]: [0, 1, 2, 3, 3]
```

Let's see what happens if we convert our list to a *NumPy* array!

```
In [12]: # Convert to NumPy Array
         array_of_integers = np.array(list_of_integers)
         # Add 3
         array_of_integers + 3

Out[12]: array([3, 4, 5, 6])
```

It worked this time! So what actually happens behind the scenes here, is referred to as *broadcasting*. The term broadcasting describes how NumPy treats objects with different shapes during arithmetic operations: So, what this means in this context is that the value "3" when performing the addition is actually being *reused* throughout the entire array. This might seem trivial, but lists don't support this behavior!

So, we see that NumPy can operate on each element just by giving an operation to a NumPy array. But NumPy can *also* use two arrays to operate on one another. This is useful in cases where you have two sets of data that are indirectly related, but commonly used to create statistics like population and area of a given city or state, which would give us population density (i.e. nyc_population_density = nyc_population / nyc_square_miles)

What if we had a friend who is trying to figure out the square footage of their apartment. They've measured out the lengths of each room and put those into a list for us, and then made another list for the widths of each room. Instead of trying to figure out this bizarre way our friend grouped their data, let's use NumPy to create a list with the area in square feet for each room.

In [13]:
```
lengths_of_each_room = np.array([10, 12, 20, 5])
widths_of_each_room = np.array([13, 15, 16, 4])
areas_of_each_room = lengths_of_each_room * widths_of_each_room
print ('Here is an array with the square footages for each room:', areas_of_each_
```

```
Here is an array with the square footages for each room: [130 180 320  20]
```

In [14]:
```
# Example
length_of_each_room = np.array([10, 20, 30, 40])
width_of_each_room = np.array([1, 2, 3, 4])
area_of_each_room = length_of_each_room * width_of_each_room
print(area_of_each_room)
```

```
[ 10  40  90 160]
```

In [16]:
```
# Example
print(area_of_each_room)
added_area_of_each_room = area_of_each_room + 3
print(added_area_of_each_room)
```

```
[ 10  40  90 160]
[ 13  43  93 163]
```

A Temperature Conversion Example

In [ ]:
```

```

Now, let's imagine we have a list of temperatures that represent the average high temperatures for each month of the year in NYC. Currently, this list has all the temperatures in Fahrenheit. However, since NYC has such a large international presence and population, it would be great to have these numbers in Celsius as well. Without NumPy, we would have to access each element individually, get its value, convert the value to Celsius, and add the new value to a new array. With NumPy, we can just multiply each element by the factor we need to convert Fahrenheit to Celsius.

The formula for converting Fahrenheit to Celsius is below:

```
T(°C) = (T(°F) - 32) × 5/9
```

Let's see an example of how we would perform this conversion with a Python list and a NumPy array.

In [18]:
```
# Average temps in NYC from January -> December (in fahrenheit)
nyc_avg_temps_f = [39, 42, 50, 62, 72, 80, 85, 84, 76, 65, 54, 44]

# ----- Without NumPy -----
nyc_avg_temps_c = list(range(0,12))
nyc_avg_temps_c[0] = (nyc_avg_temps_f[0] - 32) * (5/9)
nyc_avg_temps_c[1] = (nyc_avg_temps_f[1] - 32) * (5/9)
nyc_avg_temps_c[2] = (nyc_avg_temps_f[2] - 32) * (5/9)
nyc_avg_temps_c[3] = (nyc_avg_temps_f[3] - 32) * (5/9)
nyc_avg_temps_c[4] = (nyc_avg_temps_f[4] - 32) * (5/9)
nyc_avg_temps_c[5] = (nyc_avg_temps_f[5] - 32) * (5/9)
nyc_avg_temps_c[6] = (nyc_avg_temps_f[6] - 32) * (5/9)
nyc_avg_temps_c[7] = (nyc_avg_temps_f[7] - 32) * (5/9)
nyc_avg_temps_c[8] = (nyc_avg_temps_f[8] - 32) * (5/9)
nyc_avg_temps_c[9] = (nyc_avg_temps_f[9] - 32) * (5/9)
nyc_avg_temps_c[10] = (nyc_avg_temps_f[10] - 32) * (5/9)
nyc_avg_temps_c[11] = (nyc_avg_temps_f[11] - 32) * (5/9)
# -------------------------

# ------ With NumPy -------
np_nyc_avg_temps_f = np.array(nyc_avg_temps_f)
np_nyc_avg_temps_c = (np_nyc_avg_temps_f - 32) * (5/9)
# -------------------------

print('1. Without NumPy:', nyc_avg_temps_c)
print('2. With NumPy:', np_nyc_avg_temps_c)
```

```
1. Without NumPy: [3.8888888888888893, 5.555555555555555, 10.0, 16.666666666666
668, 22.22222222222222, 26.666666666666668, 29.444444444444446, 28.888888888888
89, 24.444444444444446, 18.333333333333336, 12.222222222222223, 6.6666666666666
67]
2. With NumPy: [ 3.88888889  5.55555556 10.         16.66666667 22.22222222 26.
66666667
 29.44444444 28.88888889 24.44444444 18.33333333 12.22222222  6.66666667]
```

Woah! Okay, we can see that in the first example, without NumPy, it took us **thirteen (13)** lines of code to accomplish the conversion from Fahrenheit to Celsius. With a NumPy array, we condensed that operation to **two (2)** lines of code.

Let's break this down. Essentially the problem was to operate on each number in the list of NYC average monthly temperatures. The operation was to convert the number in Fahrenheit to Celsius. To do this, without NumPy, we must access each value from the `nyc_avg_temps_f` list separately, use the value to convert it to Celsius, and assign the converted value to the `nyc_avg_temps_c` list. *With* NumPy, we just need to use the variable name for the list, as if it were a single element, within the operation. NumPy then quickly performs the operation on each element and returns a **new** array.

Don't worry too much about how this is implemented behind the scenes. The key takeaway is that when we have large datasets that we want to operate on, NumPy can usually greatly simplify our code as well as make it more performant, which we will learn about later!

Performance Benefits of NumPy Arrays

Another benefit to NumPy arrays, as we mentioned earlier, is that they use less memory and therefore make it easier for us to perform operations on them. However, this performance benefit is only really noticed when dealing with very large datasets. So, for now, the performance benefits of NumPy are purely educational, and we do not need to worry about them just yet.

Let's take a look at an example. We will perform a simple operation on two sets of data. One is a regular list and the other is a NumPy array. Don't worry about the code. We are only focusing on the time difference between how long it takes us to perform the same operation with and without NumPy.

In [20]:
```
import time

# Using 1 million integers
huge_list_of_integers = list(range(0, 1000000))
huge_np_array_of_integers = np.array(huge_list_of_integers)

def add_one(list_of_ints):
    return [num + 1 for num in list_of_ints]


start_time = time.perf_counter() # Time when operation starts
add_one(huge_list_of_integers) # Adds 1 to each number in the list of integers ab
end_time = time.perf_counter() # Time when operation finishes
total_time = (end_time - start_time) # Total time for operation


start_time_with_np = time.perf_counter() # Time when operation starts
huge_np_array_of_integers + 1 # Adds 1 to each number in the array of integers
end_time_with_np = time.perf_counter() # Time when operation finishes
total_time_with_np = (end_time_with_np - start_time_with_np) # Total time for ope

print('Time it takes to add 1 to each element in a list without NumPy:', total_ti
print('Time it takes to add 1 to each element in a list with NumPy:', total_time_

percent_faster = int((((total_time - total_time_with_np)/total_time)*100))
print('NumPy completes the operation', percent_faster, '% faster than a traditior
```

```
Time it takes to add 1 to each element in a list without NumPy: 0.0679420679807
663
Time it takes to add 1 to each element in a list with NumPy: 0.0012964084744453
43
NumPy completes the operation 98 % faster than a traditional list
```

Simulations with NumPy

To conclude this lesson, it is important to mention that NumPy is a very useful library to perform random sampling. What this means is that, given that we know what a certain population looks like, we can use `numpy.random` to essentially "produce" random samples given the population information.

Don't worry if this doesn't make sense right now. We'll explore this NumPy functionality later on.

Summary

In this lesson, we introduced using NumPy to create arrays in Python. NumPy is a library that is very commonly used in Python when performing scientific computing operations. We looked at how NumPy can greatly reduce the amount of code we write while keeping our code very clear and concise. Next, we briefly looked at an example of the performance benefits of NumPy compared to a traditional list in Python. Finally, we touched upon NumPy's random number generating capabilities.