Numpy Exercises

Most of your answers to this notebook will be in markdown, so we won't use the autograder. When completed, please export to HTML, print to PDF, and upload the resulting PDF.

Because most answers will be written in markdown, formatting will be especially important, so be sure to review the guidelines for writing good notebooks!

Exercise 1

First, lets make a common array to work with.

In [ ]:
```
import numpy as np

# Seed insures results are stable.
np.random.seed(21)
random_integers = np.random.randint(1, high=500000, size=(20, 5))
random_integers
```

Out[ ]:
```
array([[ 80842, 333008, 202553, 140037,  81969],
       [ 63857,  42105, 261540, 481981, 176739],
       [489984, 326386, 110795, 394863,  25024],
       [ 38317,  49982, 408830, 485118,  16119],
       [407675, 231729, 265455, 109413, 103399],
       [174677, 343356, 301717, 224120, 401101],
       [140473, 254634, 112262,  25063, 108262],
       [375059, 406983, 208947, 115641, 296685],
       [444899, 129585, 171318, 313094, 425041],
       [188411, 335140, 141681,  59641, 211420],
       [287650,   8973, 477425, 382803, 465168],
       [  3975,  32213, 160603, 275485, 388234],
       [246225,  56174, 244097,   9350, 496966],
       [225516, 273338,  73335, 283013, 212813],
       [ 38175, 282399, 318413, 337639, 379802],
       [198049, 101115, 419547, 260219, 325793],
       [148593, 425024, 348570, 117968, 107007],
       [ 52547, 180346, 178760, 305186, 262153],
       [ 11835, 449971, 494184, 472031, 353049],
       [476442,  35455, 191553, 384154,  29917]])
```

Exercise 2

What is the average value of the second column (to one decimal place).

In [ ]:
```
avg_2 = random_integers[:,1].mean()
```

```
avg_2
```

Out[ ]: 214895.8

Exercise 3

What is the average value of the first 5 rows of the third and fourth columns (to one decimal place)?

In [ ]:
```
avg_5 = random_integers[:5,2:4].mean()
avg_5
```

Out[ ]: 286058.5

Exercise 4

Without using Python, read the following code and predict the result of the last print statement (`print(first_matrix + second_matrix)`):

In [ ]:
```
first_matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(first_matrix)
```

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

In [ ]:
```
second_matrix = np.array([1, 2, 3])
print(second_matrix)
```

```
[1 2 3]
```

```
print(first_matrix + second_matrix)
```

$$\begin{bmatrix} 2 & 4 & 6 \\ 5 & 7 & 9 \end{bmatrix}$$

Please write your result in LaTeX in a markdown cell. You can write matrices in Markdown in Jupyter Notebooks using the following syntax, where the `&` symbols separate columns and `\\` is used to end a row, and you have an empty row above and below this block:

```
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
```

This will be rendered as:

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

Execise 5

Still without Python! Evaluate the following code. Write your prediction of the final print statement in markdown:

```
my_vector = np.array([1, 2, 3, 4, 5, 6])
selection = my_vector % 2 == 0
print(my_vector[selection])
```

$$\begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$$

Exercise 6

Now open python and check your answers to Exercises 4 and 5. How did your answers compare with what Python generates? What errors did you make?

In [ ]:
```
#exercise 4
first_matrix = np.array([[1, 2, 3], [4, 5, 6]])
#print(first_matrix)
second_matrix = np.array([1, 2, 3])
#print(second_matrix)
print(first_matrix + second_matrix)
```

```
[[2 4 6]
 [5 7 9]]
```

In [ ]:
```
#exercise 5
my_vector = np.array([1, 2, 3, 4, 5, 6])
selection = my_vector % 2 == 0
print(my_vector[selection])
```

```
[2 4 6]
```

Working with Views

One of the nuances of `numpy` can can easily lead to problems is that when one takes a slice of an array, one does not actually get a new array; rather, one is given a "view" on the original array, meaning they are sharing the same underlying data.

Views exist because they are more memory efficient (a view doesn't require making a new copy of data) and faster (again, no copying required). And if you're doing super-computer simulations where every milisecond counts, or working with truely huge datasets, this is important. But for most data scientists, I tend to see it as a a trap waiting to get you in trouble.

This is especially since there's no reliable way to check if two arrays are views of one another except by modifying one and seeing if the other changes. (You may find people saying otherwise; don't trust them!). The way we use `is` in regular Python to see if two variables point at the same object doesn't work for numpy arrays. Thus its on you to remember the rules.

My advice on copies: UNLESS YOU REALLY NEED A VIEW AND ARE BEING SUPER CAREFUL: don't use views for anything but *looking* at data. If you ever want to *modify* or *work with* a sub-array, just make a copy to be safe. Computers are fast enough and ram is plentiful enough that for most applications, it's almost never a problem.

Exercise 7

Without Python Let's try and work out a few problems in our heads to test our understanding of `numpy` views. Let's start with the following array:

In [ ]:
```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
print(my_array)
```

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

Now, in markdown write down the result of this code:

```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
print(my_slice)
```

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

Exercise 8

Now suppose we run the code:

```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
my_array[:, :] = my_array * 2
print(my_slice)
```

Now what does `my_slice` look like? Again, show your result in markdown.

$$\begin{bmatrix} 4 & 6 \\ 10 & 12 \end{bmatrix}$$

Exercise 9

Now suppose we run the following code:

```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
my_array = my_array * 2
print(my_slice)
```

What does `my_slice` look like?

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

Exercise 10

Using Python, run the preceding code. Were your predictions correct? If not, why not?

In [ ]:
```
# exercise 7
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
print(my_slice)
```

```
[[2 3]
 [5 6]]
```

In [ ]:
```
# exercise 8
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
my_array[:, :] = my_array * 2
print(my_slice)
```

```
[[ 4  6]
 [10 12]]
```

In [ ]:
```
# exercise 9
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3]
my_array = my_array * 2
print(my_slice)
```

```
[[2 3]
 [5 6]]
```

Exercise 11

OK, let's close Python again and go back to markdown. Let's also reset `my_array` and start over with the following code:

```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3].copy()
my_array[:, :] = my_array * 2
print(my_slice)
```

What does `my_slice` look like?

$$\begin{bmatrix} 4 & 6 \\ 10 & 12 \end{bmatrix}$$

Exercise 12

Arrays aren't the only data structure in Python with the slice syntax (i.e., the ability to subset like this: `v[1:3]`). Close Python and, in markdown, try and predict the result of this code:

```
x = [1, 2, 3]
y = x[0:2]
y[0] = "a change"
print(y)
```

$$['a \quad change', \quad 2]$$

Exercise 13

Now what is the output if we subsequently run `print(x)`?

$$[1 \quad 2 \quad 3]$$

Exercise 14

Now open Python and check your answer. Were you correct? Why or why not?

In [ ]:
```
# exercise 11
my_array = np.array([[1, 2, 3], [4, 5, 6]])
my_slice = my_array[:, 1:3].copy()
my_array[:, :] = my_array * 2
print(my_slice)
```

```
[[2 3]
 [5 6]]
```

Explaination for exercise 11

I came out the wrong answer without running in the Python. I think I am confused by

copy(). And I go back to the readings realted to this part after got the correct answer.

In [ ]:
```
# exercise 12
x = [1, 2, 3]
y = x[0:2]
y[0] = "a change"
print(y)
```

```
['a change', 2]
```

In [ ]:
```
# exercise 13
x = [1, 2, 3]
y = x[0:2]
y[0] = "a change"
print(x)
```

```
[1, 2, 3]
```

Note: Don't trust my_array.base

Before we wrap up this section, and important note about views: You will find some tutorials online that suggest you can test if one array is a view of another with the code `my_slice.base is my_array`. The problem is... this doesn't always work. It does sometimes:

In [ ]:
```
my_array = np.array([1, 2, 3])
my_slice = my_array[1:3]
my_slice.base is my_array
```

Out[ ]:
```
True
```

But not always. Here's an example where `my_array` and `my_slice` point to the same data, but `my_slice.base is my_array` returns false.

In [ ]:
```
my_array = np.array([1, 2, 3])
my_array = my_array[1:4]
my_slice = my_array[1:3]
my_slice.base is my_array
```

Out[ ]:
```
False
```

In [ ]:
```
my_slice
```

Out[ ]:
```
array([3])
```

In [ ]:
```
my_array
```

Out[ ]:
```
array([2, 3])
```

In [ ]:
```
# But a change to `my_slice` still impacts `my_array`.
my_slice[0] = -1
my_array
```

Out[ ]:
```
array([ 2, -1])
```

(The reason is that the `.base` property can be defined recursively. In this case, the slicing of `my_array` made `my_array` a view on data you can no longer access, so they actually do both point to the same data, but that data is not `my_array`, it's `my_array.base`. So:

In [ ]:
```
my_slice.base is my_array.base
```

Out[ ]:
```
True
```

In practice, you can get infinite chains of `.base.base...`.

And yes, if this is making your head hurt, that's because you're doing it right. :)