NumPy Basics

From the NumPy homepage:

> 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

We will focus mostly on the array object, with some coverage of other functions NumPy provides.

In [ ]:
```
# import numpy, alias as np for convenience.
import numpy as np
```

The Humble NumPy Array

The NumPy array is an N-dimensional grid of items of the same type (in our case, usually floats or integers).

We can create NumPy arrays from python lists, or in any number of ways described in the documentation.

In [ ]:
```
# creating NumPy array from python list using np.array() function
a = np.array([1, 2, 3, 4, 5, 6])

# these can be multiple dimensions -- just use nested lists!
b = np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])

print(a)
print(b)
```

```
[1 2 3 4 5 6]
[[1.1 2.2 3.3]
 [4.4 5.5 6.6]]
```

In [ ]:
```
# Instantiate a 1D array containing every value in a specified range
# Here, array c contains every whole number up to 5
c = np.arange(5)

print(c)
```

```
[0 1 2 3 4]
```

In [ ]:
```
# Instantiate an empty array with a specified shape, in this case (2,3)
# The array is not truly "empty" - it just contains uninitialized values!
```

```
shape = (2, 3)
d = np.empty(shape)

print(d)
```

```
[[1.1 2.2 3.3]
 [4.4 5.5 6.6]]
```

In [ ]:
```
# Instantiate an array with a specified shape, and fill every cell with a spec
# Here, array e is a 2x3 array filled with 7s
e = np.full((2,3), 7)

# Similar to np.full, np.zeros creates an array filled with 0s and np.ones crea
f = np.zeros((2,3))
g = np.ones((2,3))

print(e)
print(f)
print(g)
```

```
[[7 7 7]
 [7 7 7]]
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

Properties:

Homogeneous

All the elements of a NumPy array are of the same type. You can see what type our values are stored as with `dtype`

In [ ]:
```
[a.dtype, b.dtype]
```

Out[ ]:
```
[dtype('int64'), dtype('float64')]
```

N-dimensional

NumPy arrays don't need to be 1-dimensional. They can be 2-dimensional (like a matrix), or even more. Each dimension is called an `axis` and the `shape` is a tuple of sizes along each axis.

Remember these terms: `axis` and `shape`. These are essential in using NumPy.

In [ ]:
```
b
```

Out[ ]:
```
array([[1.1, 2.2, 3.3],
       [4.4, 5.5, 6.6]])
```

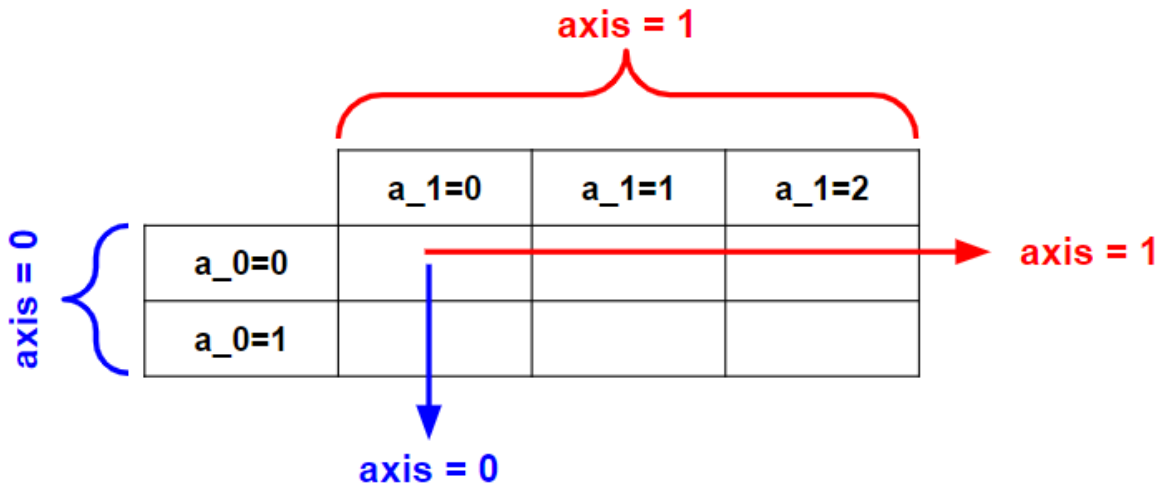
In [ ]:
```
[a.shape, b.shape]
```

Out[ ]:
```
[(6,), (2, 3)]
```

The shapes printed encode all the relevant information for interpreting the dimensionality and shape of NumPy arrays:

- `a` has shape `(6,)` with a single element = `a` has 1 axis with 6 elements
- `b` has shape `(2,3)` with two elements = `b` has 2 axes, axis 0 contains 2 elements, and axis 1 has 3 elements.

The following diagram depicts array `b`'s axes, visually showing the size of each axis:



Note: Despite all having 3 elements, be careful to distinguish the shapes (N,), (N, 1), and (1, N)

In [ ]:
```
print(np.ones(3)) #shape (3,)
print(np.zeros((3,1))) #shape (3,1)
print(np.ones((1,3))) #shape (1,3)
```

```
[1. 1. 1.]
[[0.]
 [0.]
 [0.]]
[[1. 1. 1.]]
```

Check Your Understanding 1

Use the cell below to create the following NumPy arrays:

1. `arr1` - a one dimensional NumPy array containing all integers from 0 through 11.
2. `arr2` - a two dimensional NumPy array with 2 rows and 3 columns. The first row contains the integers `1,2,3` and the second row contains the integers `4,5,6`
3. `arr3` - a two dimensional NumPy array with 4 elements in axis 0 and 5 elements in axis 1, filled entirely with zeros
4. `arr4` (Challenge) - a 3x5x3 array full of the floating point number `22.7`

Tip: Each of these arrays can be created using a single line of code

In [ ]:
```
arr1 = None # TODO: replace NONE with your definition for arr1
arr2 = None # TODO: replace NONE with your definition for arr2
arr3 = None # TODO: replace NONE with your definition for arr3
arr4 = None # TODO: replace NONE with your definition for arr4
```

Indexing

Like vanilla Python lists, we can access data stored in a NumPy array by either:

- indexing elements with integers
- slicing subarrays using the `start:end` syntax

In [ ]:
```
# gives us the data at index 0
print(a[0])
print(b[0, 0])
```

Out[ ]:
```
1.1
```

In [ ]:
```
print(a)
# gives us data from index 2 inclusive to the end.
print(a[2:])
# gives us data from start of array to index 2 exclusive.
print(a[:2])
```

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

We can use negative indexes to start at the back of the array and move left

In [ ]:
```
# gives us the data at the second to last index
a[-2]
```

Out[ ]:
```
5
```

Unlike Python lists, we have some more options when indexing into NumPy arrays.

Multiple Indices

You can pass a list (or a NumPy array) as the index to get data from multiple indices.

In [ ]:
```
# gives us data from index 2, 3,and 5
list1 = [2, 3, 5]
a[list1]
```

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

Indexing across dimensions

`b[0]` gives us the first row: `[1.1, 2.2, 3.3]`

What if, instead of the first row, we want just the second value of the first row? We can use commas to separate along axes.

In [ ]:
```
# index 0 of axis 0 gives us the first row: [1.1, 2.2, 3.3].
# index 1 of axis 1 gives us the second column: [[2.2], [5.5]]
# together, we get the value at the first row, second column.
b[0, 1]
```

Out[ ]:
```
2.2
```

We can also use slices to get, for instance, the entire second column.

In [ ]:
```
# slice : of axis 0 gives us all rows.
# index 1 of axis 1 gives us the second column.
# together, we get the second column of all rows.
b[:2, 1]
```

Out[ ]:
```
array([2.2, 5.5])
```

Check Your Understanding 2

You are given the following 2D array:

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

Perform the following operations on this array:

1. Set the element in row 0 and column 4 to be equal to 42 (use 0-indexing)
2. Set the element in row 1 and column 0 to be equal to 42 (use 0-indexing)
3. Set the entire second column to be equal to -5
4. Set the all elements in the last 3 columns and last 2 rows to be equal to 10

The final mutated array should look like this

```
[[1,-5,3,6,42]
 [42,-5,10,10,10]
 [7,-5,10,10,10]]
```

Your implementation should use 4 lines of code at maximum. For a challenge, try accomplishing this in 3 lines of code!

In [ ]:
```
checkpoint2_arr = np.array([[1,2,3,6,7],[4,5,6,7,8],[7,8,9,10,11]])
# TODO: Mutate the above array based on the operations above
```

Doing math with NumPy

NumPy also provides lots of built-in functions for us to use. We will cover just a few relevant ones here:

In [ ]:
```
np.random.seed(42)
# let's create two 3x3 matrices filled with random integers in the range [0, 1
x = np.random.randint(0, 10, size=(3, 3))
y = np.random.randint(0, 10, size=(3, 3))
```

In [ ]:
```
x
```

Out[ ]:
```
array([[6, 3, 7],
       [4, 6, 9],
       [2, 6, 7]])
```

In [ ]:
```
y
```

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

Basic Arithmetic

We can use the arithmetic operators we're already familiar with in Python to do element-wise math.

In [ ]:
```
x + y  # addition
```

Out[ ]:
```
array([[10,  6, 14],
       [11,  8, 14],
       [ 6,  7, 14]])
```

In [ ]:
```
x * y  # multiplication
```

Out[ ]:
```
array([[24,  9, 49],
       [28, 12, 45],
       [ 8,  6, 49]])
```

In [ ]:
```
x - y  # subtraction
```

Out[ ]:
```
array([[ 2,  0,  0],
       [-3,  4,  4],
       [-2,  5,  0]])
```

In [ ]:
```
x / y  # division
```

Out[ ]:
```
array([[1.5       , 1.        , 1.        ],
       [0.57142857, 3.        , 1.8       ],
       [0.5       , 6.        , 1.        ]])
```

In [ ]:
```
x // y  # integer division
```

Out[ ]:
```
array([[1, 1, 1],
       [0, 3, 1],
       [0, 6, 1]])
```

In [ ]:
```
x ** y  # raising to the power
```

Out[ ]:
```
array([[  1296,     27, 823543],
       [ 16384,     36,  59049],
       [    16,      6, 823543]])
```

Math within an array

We can also call NumPy functions to sum across an array or find the mean across an array (and much more!)

In [ ]:
```
np.sum(x)
```

Out[ ]:
```
50
```

In [ ]:
```
np.mean(x)
```

Out[ ]:
```
5.555555555555555
```

In [ ]:
```
np.std(x) #standard deviation
```

Out[ ]:
```
2.0608041101101566
```

We can even only sum across a single axis (e.g. if we want the sum of each row, we take the sum across the columns (axis 1))

In [ ]:
```
print(np.sum(x, axis=0))
print(np.sum(x, axis=1))
```

```
[12 15 23]
[16 19 15]
```

In [ ]:
```
# A negative axis lets us move backwards from the last dimension
# Because x is 2-dimensional, using axis=-1 has the same result as using axis=1
print(np.sum(x, axis=-1))
```

```
[16 19 15]
```

We can also sum across more than one axis at a time

In [ ]:
```
# Here, we are summing across both axis 0 and axis 1, which returns the sum of
# Notice that this output has a different shape than when we used only axis=0 o
print(np.sum(x, axis=(0,1)))
```

```
50
```

Check Your Understanding 3

Consider the following 3x2x2 array

```
[[[0,1],
  [2,3]],
 [[4,5],
  [6,7]],
 [[8,9],
  [10,11]]]
```

Find the sum across the rows and columns for each 2x2 slice of the array. The resultant sum array should have 3 elements, one value for the sum across each slice.

In [ ]:
```
arr_3d = np.array([[[0,1],[2,3]],[[4,5],[6,7]],[[8,9],[10,11]]])
# TODO - compute the sum across the rows and columns for each 2x2 slice of arr
```

Linear Algebra

There are also linear algebra functions you can use.

In [ ]:
```
# get the transpose of a matrix
x.T
```

Out[ ]:
```
array([[6, 4, 2],
       [3, 6, 6],
       [7, 9, 7]])
```

In [ ]:
```
# get the L2 norm of a vector
np.linalg.norm(x[0])
```

Out[ ]:
```
9.695359714832659
```

In [ ]:
```
# get the dot product of two vectors
np.dot(x[0], x[1])
```

Out[ ]:
```
105
```

In [ ]:
```
# multiply two matrices
np.matmul(x, y)
```

Out[ ]:
```
array([[ 73,  31, 106],
       [ 94,  33, 121],
       [ 78,  25,  93]])
```

argmax and argmin

In mathematical notation, $\arg\min_x f(x)$ returns the x that minimizes $f(x)$.

In the same vein, `np.argmin(a)` returns the index of the minimum element of `a`.

In [ ]:
```
x
```

Out[ ]:
```
array([[6, 3, 7],
       [4, 6, 9],
       [2, 6, 7]])
```

In [ ]:
```
# index of minimum value for each row
np.argmin(x, axis=1)
```

Out[ ]:
```
array([1, 0, 0])
```

In [ ]:
```
# find the actual minimum value for each row
np.min(x, axis=1)
```

Out[ ]:
```
array([3, 4, 2])
```

Adjusting Array Shape and Dimensionality

Sometimes our arrays are not in the format we want, so we need to restructure them.

We can use np.reshape() to redistribute the existing array values into a different shape. The new shape must contain the same number of values as the original shape, but it can have a different number of dimensions.

In [ ]:
```
y = np.arange(6)
print(y)

#Change the shape of y from (6,) to (2,3)
np.reshape(y, (2, 3))
```

Out[ ]:
```
[0 1 2 3 4 5]
array([[0, 1, 2],
       [3, 4, 5]])
```

In [ ]:
```
#If you list the size of one dimension as -1, NumPy will determine the appropri
#Here, the new shape will be (3,2)
np.reshape(y, (-1,2))
```

Out[ ]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

To insert an additional axis/dimension, we can use np.expand_dims()

In [ ]:
```
y = np.ones((2,2))
print(y.shape)

#Insert an axis to expand the shape from (2,2) to (2,1,2). This changes y from
y = np.expand_dims(y, axis=-2)
print(y.shape)
```

```
(2, 2)
(2, 1, 2)
```

We can use np.transpose() to reorder the sizes of the dimensions

In [ ]:
```
y = np.ones((3,4,5))
print(y.shape)

#Permute the axes such that the shape of y changes from (3, 4, 5) to (4, 3, 5)
y = np.transpose(y, axes=(1,0,2))
print(y.shape)
```

```
(3, 4, 5)
(4, 3, 5)
```

To repeat or duplicate elements in an array, we can use np.repeat()

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

```
#Repeat every element twice along axis 1
z = np.repeat(y, repeats=2, axis=1)
print(z)
```

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

Some NumPy operations have a keepdims parameter that, when set to True, causes the output array to have the same dimensionality as the input array.

In [ ]:
```
y = np.ones((3,2))

#If unspecified, np.sum sets keepdims=False. Here, the output array is 1D
z = np.sum(y, axis = 1)
print(z.shape)

#When we set keepdims=True, the output array is 2D
z = np.sum(y, axis = 1, keepdims=True)
print(z.shape)
```

```
(3,)
(3, 1)
```

Broadcasting

Broadcasting is the system by which NumPy handles arrays with different shapes during arithmetic operations. Subject to some constraints, the smaller array is broadcasted or *stretched* across the larger array so that they have compatible shapes.

For example, consider adding the constant integer 5 to every element in the vector $\overline{x} = (1, 2, 3)^T$. Formally, you would have to create another vector like $(5, 5, 5)^T$ in order to add the constant 5 to each entry of the vector $\overline{x}$.

With NumPy however, we need only add the integer 5 to $\overline{x}$ to achieve the same output:

In [ ]:
```
x = np.array([1,2,3])
x + 5
```

Out[ ]:
```
array([6, 7, 8])
```

This behavior is caused by broadcasting - the integer value 5 is implicitly stretched out into a vector the same size as the vector $\overline{x}$ before the operation is performed.

Broadcasting generally improves efficency by avoiding the creation of a new array for a single computation. The creation of a new array is internally replaced by a loop through the array to perform the mathematical operation.

General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, the exception:

`ValueError: operands could not be broadcast together`

is thrown, indicating that the arrays have incompatible shapes.

The size of the resulting array is the size that is not 1 along each axis of the input (indicating that the smaller array along each dimension has been stretched out to match the larger array).

For example, consider two NumPy arrays:

- `a` which is of size (3,1)
- `b` which is of size (1,4)

As `a` and `b` are compatible along both of their dimensions as per broadcasting rules, they can be combined via some operation to create an output array of size (3,4).

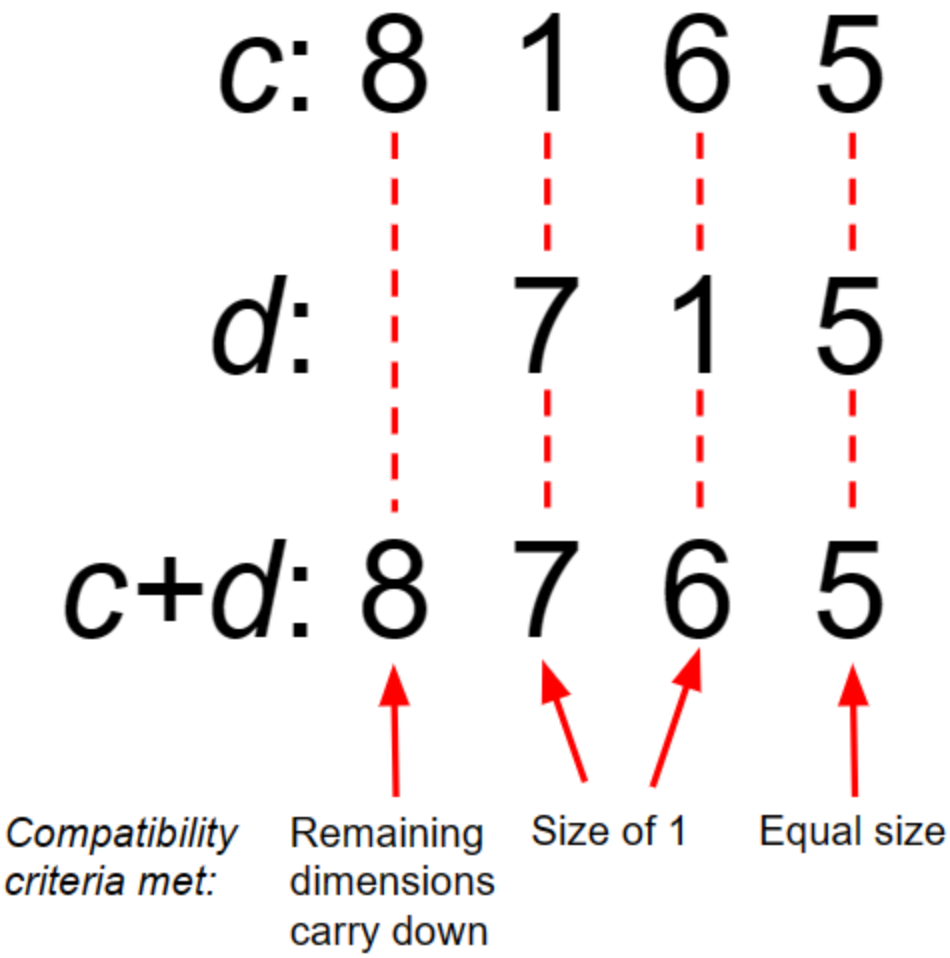
In [ ]:
```
a = np.array([[3],[2],[1]])
b = np.array([[-1,1,-2,2]])
a * b
```

Out[ ]:
```
array([[-3,  3, -6,  6],
       [-2,  2, -4,  4],
       [-1,  1, -2,  2]])
```

As another example, consider two other NumPy arrays:

- `c` which is of size (8, 1, 6, 5)
- `d` which is of size (7, 1, 5)

As depicted below, `c` and `d` have compatible dimensions for broadcasting. Thus, we can use broadcasting to compute `c+d`, which will have size (8, 7, 6, 5)



Check Your Understanding 4

Question 1

Consider the following array of temperatures in Fahrenheit

```
[98.7, 26.5, -10.23, 56.93, 129.32, 83.52, 92.53]
```

Convert this array to an equivalent array of temperatures in Celsius. Recall that the formula for converting a temperature in Fahrenheit F to a temperature in Celsius C is given by

$$C = \frac{5}{9}(F - 32)$$

You should complete this conversion in a single line of code.

In [ ]:
```
temperatures_f = np.array([98.7, 26.5, -10.23, 56.93, 129.32, 83.52, 92.53])
temperatures_c = None # TODO: Convert the fahrenheit temperatures to celsius te
```

Question 2 (Challenge)

Create the following 2D array using only 1D arrays. You may not directly instantiate a 2D array (i.e., you may not use a nested list to create an array).

```
[[1,2,3,4,5]
 [2,4,6,8,10],
 [3,6,9,12,15]]
```

Hint: notice that each of the rows are scalar multiples of the array `[1,2,3,4,5]`. How can this be achieved easily via broadcasting?

In [ ]:
```
arr_2d = None # TODO: Create the 2D array specified above using only 1D arrays
```

Practice Problems

In this section, we're going to apply what we've learned about NumPy to a famous dataset, MNIST, which contains images of handwritten digits. You may need to run `pip install keras` before running this part if you do not have keras installed already.

https://yann.lecun.com/exdb/mnist/

In [ ]:
```
# Load the data
from keras.datasets import mnist
import numpy as np
(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

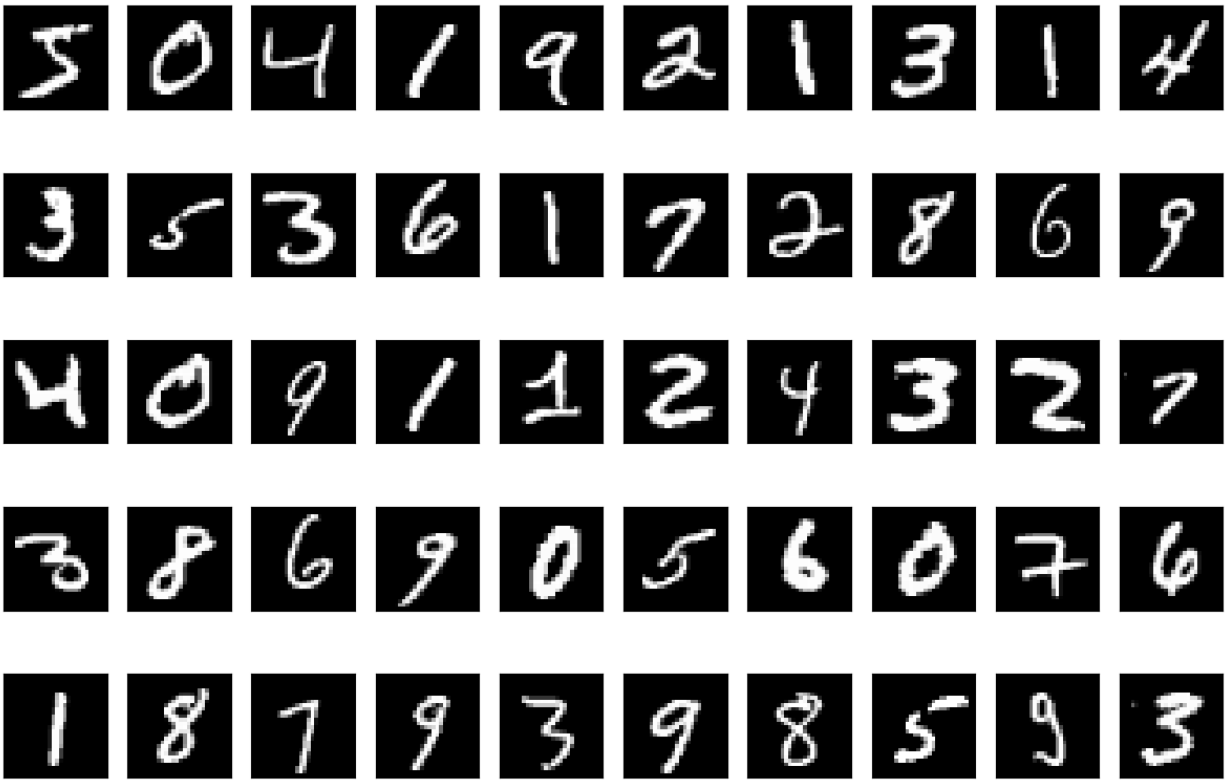
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datas
ets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

Plotting Data

Let's take a look at a few of the images

In [ ]:
```
import matplotlib.pyplot as plt
num_cols = 10
num_rows = 5
fig,axes = plt.subplots(num_rows,num_cols, figsize=(1.5*num_cols, 2*num_rows))

for i in range(50):
  ax = axes[i//10, i%10]
  ax.imshow(train_x[i], cmap='gray')
  ax.xaxis.set_visible(False)
  ax.yaxis.set_visible(False)
```



Shape of Data

Let's investigate some basic properties of the MNIST dataset

In [ ]:
```
#How many dimensions does the train_x data have?
train_x_ndim = None # TODO
print("Number of dimensions in train_x: " + str(train_x_ndim))

#How many dimensions does the train_y data have?
train_y_ndim = None # TODO
print("Number of dimensions in train_y: " + str(train_y_ndim))

#What is the shape of the train_x data?
train_x_shape = None # TODO
print("Shape of train_x: " + str(train_x_shape))

#What is the shape of the train_y data?
train_y_shape = None # TODO
print("Shape of train_y: " + str(train_y_shape))
```

```
Number of dimensions in train_x: None
Number of dimensions in train_y: None
Shape of train_x: None
Shape of train_y: None
```

Extracting information

Let's dig deeper into the contents of the images

In [ ]:
```
#What is the pixel value of the 4th image, row 18, column 7? Recall that we use
#0-based indexing in numpy, so the index of this value would be (3, 17, 6).
specific_pixel_val = None # TODO
print("Pixel val at (3,17,6): " + str(specific_pixel_val))

#What is the mean pixel value across the whole first image?
img_1_avg = None # TODO
print("Average pixel value of img_1: " + str(img_1_avg))

#What is the standard deviation of the pixel values across the whole first imag
img_1_std = None # TODO
print("Std of image 1 pixels: " + str(img_1_std))

#Examine the two lines of code below and observe their resulting shapes
#What do the shapes tell us about the nature of the operations?
#Can you infer the meaning of the data in avg1 and avg2?

avg1 = train_x.mean(axis = (1,2))
print("Shape of avg1: " + str(avg1.shape))

avg2 = train_x.mean(axis = 0)
print("Shape of avg2: " + str(avg2.shape))


#Which image has the highest average pixel value?
#Hint: use one of the two averages calculated above
highest_img_index = None #Todo
print("The image with the highest avg pixel value is image: " + str(highest_img
```

```
Pixel val at (3,17,6): None
Average pixel value of img_1: None
Std of image 1 pixels: None
Shape of avg1: (60000,)
Shape of avg2: (28, 28)
The image with the highest avg pixel value is image: None
```

Answers to Practice Problems

Number of dimensions in train_x: 3\ Number of dimensions in train_y: 1\ Shape of train_x: (60000,28,28)\ Shape of train_y: (60000,)

Pixel val at (3,17,6): 0\ Mean pixel value of img_1: 35.108418367346935\ Standard deviation of pixel values of img_1: 79.64882892760731\

Avg1 holds one value for each image, and represents the average pixel value of the image\ Avg2 holds one value for each pixel, and represents the average value of that pixel across all images

The image with the highest avg pixel value is image: 41358

Code Solutions to Practice Problems

In [ ]:
```
train_x_ndim = train_x.ndim
print(train_x_ndim)

train_y_ndim = train_y.ndim
print(train_y_ndim)

train_x_shape = train_x.shape
print(train_x_shape)

train_y_shape = train_y.shape
print(train_y_shape)

specific_pixel_val = train_x[3][17][6]
print(specific_pixel_val)

img_1_avg = np.mean(train_x[0])
print(img_1_avg)

img_1_std = np.std(train_x[0])
print(img_1_std)

highest_img_index = np.argmax(avg1)
print(highest_img_index)
```

```
3
1
(60000, 28, 28)
(60000,)
0
35.108418367346935
79.64882892760731
41358
```

Answers to Check Your Understanding

In [ ]:
```
# Check Your Understanding 1
arr1 = np.arange(0, 12)
arr2 = np.array([[1,2,3],[4,5,6]])
arr3 = np.zeros((4,5))
arr4 = np.ones((3,5,3))
```

In [ ]:
```
# Check Your Understanding 2
checkpoint2_arr = np.array([[1,2,3,6,7],[4,5,6,7,8],[7,8,9,10,11]])
checkpoint2_arr[(0,1),(4,0)] = 42
checkpoint2_arr[:, 1] = -5
checkpoint2_arr[1:3, 2:5] = 10 # Can equivalently set checkpoint2_arr[1:, 2:] =
```

In [ ]:
```
# Check Your Understanding 3
arr_3d.sum(axis=(1,2))
```

Out[ ]:
```
array([ 6, 22, 38])
```

In [ ]:
```
# Check Your Understanding 4
# Question 1
temperatures_c = 5 * (temperatures_f - 32) / 9
print(temperatures_c)
```

```
# Question 2
arr_2d = np.arange(1,6) * np.arange(1,4).reshape(3,1) # second array is a colu
print(arr_2d)
```

```
[ 37.05555556  -3.05555556 -23.46111111  13.85        54.06666667
  28.62222222  33.62777778]
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]]
```