Open in Colab

Python Refresher 3

In this notebook, we cover:

- Applying functions to all elements in a NumPy array
- Masking
- Pandas

First, import numpy

In [66]:
```
import numpy as np
```

Applying functions to all elements in a NumPy array

Different methods with speed comparisons

Numpy's vectorize function lets you take operations you wrote as functions and apply them to your data set.

In [67]:
```
def square_f(x):
  return x*x
square_v = np.vectorize(square_f)
```

In [68]:
```
size = 1000000

# declaring array
array = np.arange(size)
len(array)
```

Out[68]:
```
1000000
```

In [69]:
```
%%timeit
_ = square_v(array)
```

```
101 ms ± 2.94 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

However, list notation is sometimes even faster for simple functions.

In [70]:
```
%%timeit
_ = np.array([x*x for x in array])
```

```
96.9 ms ± 2.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

And **for the simplest functions** with numpy arrays such as **multiplication, division, and other simple math operators, you should just do those directly**. Notice how **multiplying an array by itself is the same as multiplying each item by itself but this is literally 100 times faster!** That's because libraries like numpy are built on languages like C and C++ which are much faster than python.

In [71]:
```
%%timeit
_ = array*array
```

```
796 µs ± 23.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops eac
h)
```

Masking

First let's get some "real" data. sklearn or **sci-kit learn is a great library for simple statistics and machine learning tools**. In this library, they also provide the **fetch_openml tool** which **allows you to download data** from the website openml.org.

Let's download the MNIST dataset, which contains 70,000 28x28 pixel images of handwritten numerical digits.

In [72]:
```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("aadeshkoirala/mnist-784")
```

In [73]:
```
path
```

Out[73]:
```
'/Users/juanpablosalas/.cache/kagglehub/datasets/aadeshkoirala/mnist-
784/versions/1'
```

In [74]:
```
#from sklearn.datasets import fetch_openml
#mnist = fetch_openml('mnist_784',version=1)
```

In [75]:
```
import pandas as pd
mnist = pd.read_csv(path+'/mnist_784.csv')
```

We can use .keys() to get the names of content within an sklearn openml file.

In [76]:
```
mnist.keys()
```

Out[76]:
```
Index(['pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6', 'pixel7',
       'pixel8', 'pixel9', 'pixel10',
       ...
       'pixel776', 'pixel777', 'pixel778', 'pixel779', 'pixel780', 'pixel781',
       'pixel782', 'pixel783', 'pixel784', 'class'],
      dtype='object', length=785)
```

In this case, our openml dataset is a sklearn.utils.Bunch file.

In [77]:
```
type(mnist)
```

Out[77]:
```
pandas.core.frame.DataFrame
```

Data and target are common labels for your input data and the output you would want to get from a model respectively.

In [78]:
```
#X, y = mnist["data"], mnist["target"]
X,y = mnist.drop(columns='class'), mnist['class']
```

mnist["data"] is a pandas DataFrame. We'll get to that more later.

In [79]:
```
type(X)
```

Out[79]:
```
pandas.core.frame.DataFrame
```

Our pandas dataframe also has keys. Notice how here, all of the **keys are labeled as pixels 1-784**. So **each item** in that dataframe is probably a **784 pixel image**.

In [80]:
```
print(X.keys())
```

```
Index(['pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6', 'pixel7',
       'pixel8', 'pixel9', 'pixel10',
       ...
       'pixel775', 'pixel776', 'pixel777', 'pixel778', 'pixel779', 'pixel780',
       'pixel781', 'pixel782', 'pixel783', 'pixel784'],
      dtype='object', length=784)
```

We can **use .values to extract numeric or other data type values** from our dataframe **as a numpy array**.

In [81]:
```
X_mat = X.values
y_arr = y.values.astype(np.uint8)
```

In [82]:
```
type(X_mat)
```

Out[82]:
```
numpy.ndarray
```

Our output/target data has a shape of 70,000 items and our input has a shape of 70,000 items by 784 pixels.

In [83]:
```
y_arr.shape
```

Out[83]:
```
(70000,)
```

In [84]:
```
X_mat.shape
```

Out[84]:
```
(70000, 784)
```

In [85]:
```
X_mat = X_mat.reshape(X_mat.shape[0], 28, 28)
```

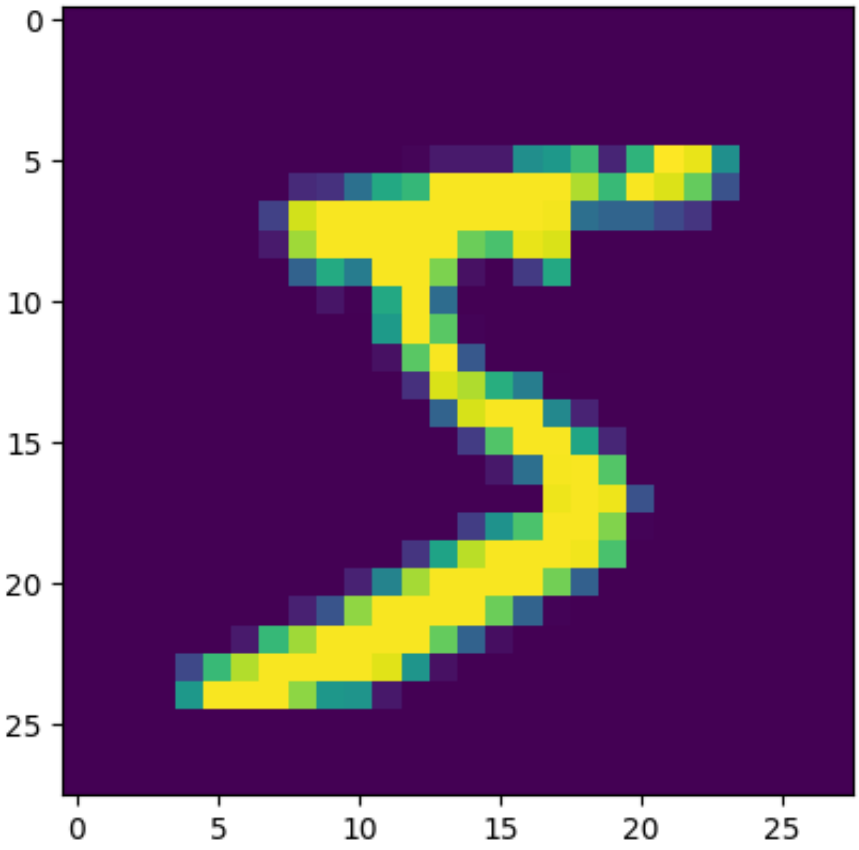
matplotlib is short for mathematical plotting library. It's **a collection of tools for visualising/plotting data**. **pyplot is a library within matplotlib** that has some tools **more specific to just creating plots**. The standard names we use to import these are mpl for matplotlib and plt for matplotlib.pyplot.

In [86]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Here we can use the pyplot plot type **imshow to view one of those 784 pixel images**.

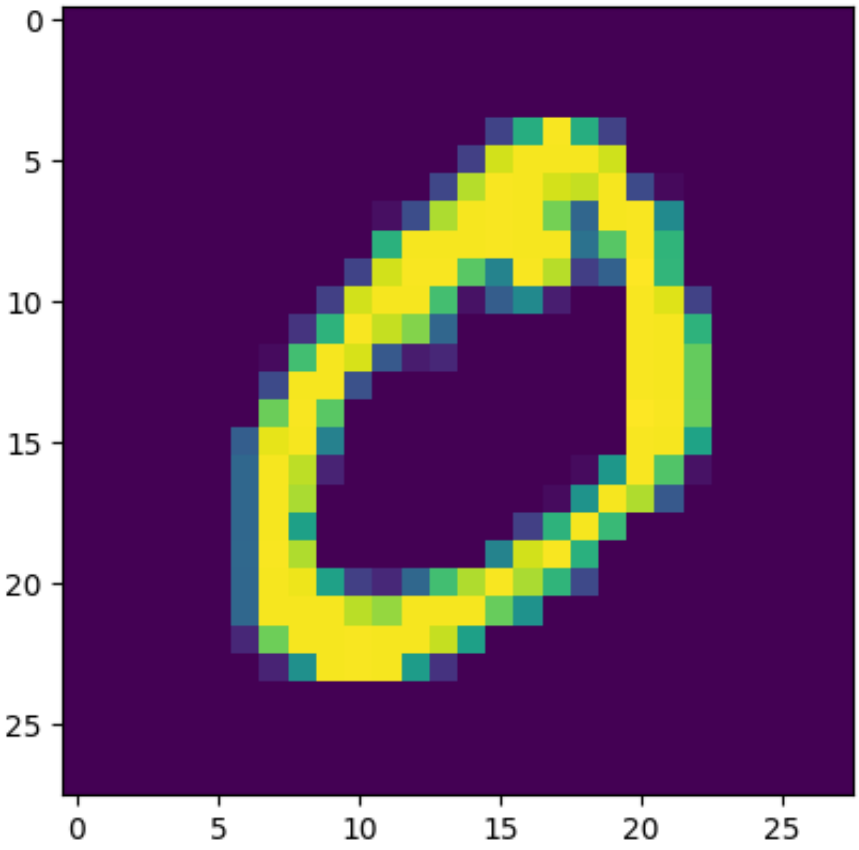
In [87]:
```
plt.imshow(X_mat[0])
```

Out[87]:
```
<matplotlib.image.AxesImage at 0x166266160>
```



In [88]:
```
plt.imshow(X_mat[1])
```

Out[88]:
```
<matplotlib.image.AxesImage at 0x16622fdf0>
```



The first two images were a 5 and 0. And sure enough, our first 2 items in our output data are 5 and 0. So the output data is labels of the number image.

In [89]:
```
y_arr[0:2]
```

Out[89]:
```
array([5, 0], dtype=uint8)
```

When we perform **logical operations on numpy arrays**, we can **create masks**. For
example, writing **y_arr == 3 says: create a new array which is only True at the location of each 3 and False everywhere else**.

In [90]:
```
# this is a mask
y_arr == 3
```

Out[90]:
```
array([False, False, False, ..., False, False, False])
```

We can also **take that mask array** of True and False values **and pass it to another array like it's a collection of indices**. The result is that we'll
get values from that other array only at the indices that were labeled as True.

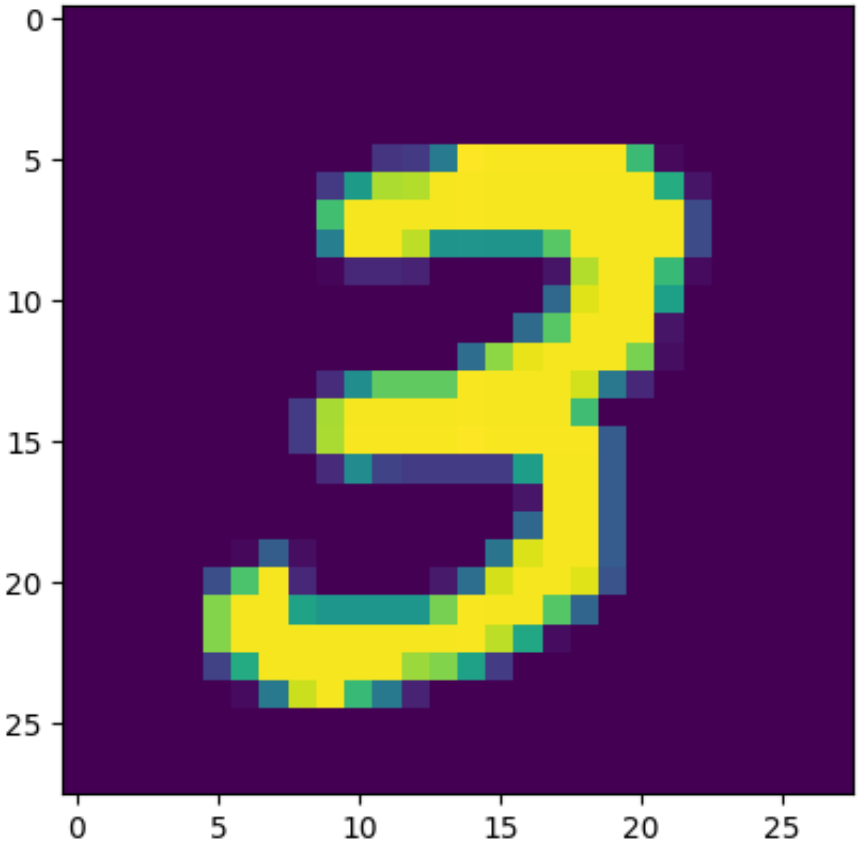
In [91]:
```
only_3 = X_mat[y_arr == 3]
```

Because the new array only_3 is the collection of images only at the location

where our label was 3, we should only have images of 3's left over.

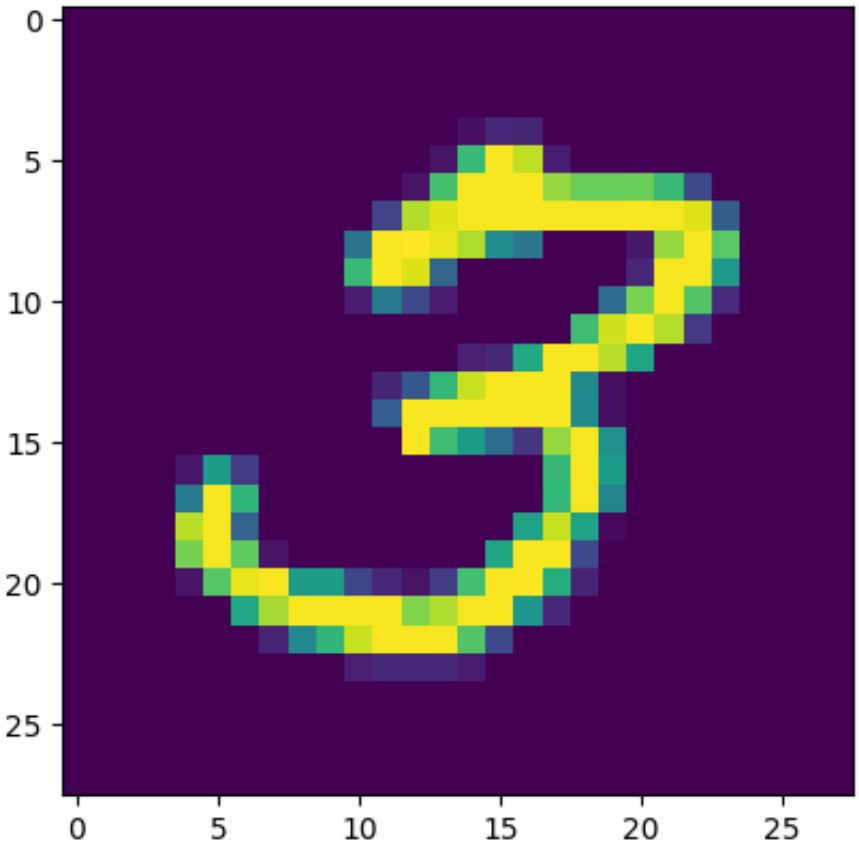
In [92]:
```
plt.imshow(only_3[0])
```

Out[92]:
```
<matplotlib.image.AxesImage at 0x1661e9310>
```



In [93]:
```
plt.imshow(only_3[42])
```

Out[93]:
```
<matplotlib.image.AxesImage at 0x1663fb0d0>
```



Pandas

What is pandas?

Pandas is a library for data analysis, with a data storing system built
off of Numpy. It can handle data from a wide variety of formats and provides
powerful
tools for viewing and manipulating that data.

Some additional resources:
Introduction to pandas - Interactive tutorial created by Google

Pandas documentation - Lots of good info here for both beginner and advanced users.

We import pandas with the shortened name `pd` by running `import pandas as pd`

In [94]:
```
import pandas as pd
```

Let's first create a NumPy array to work with.

We'll use the NumPy function `randint` to create an array of size (4,3)

filled with random integers ranging from 0 to 100

In [95]:
```
np.random.seed(42) # Set random seed for reproducible results
mydata = np.random.randint(low=0, high=101, size=(4,3))
print(mydata)
```

```
[[51 92 14]
 [71 60 20]
 [82 86 74]
 [74 87 99]]
```

We can now directly convert this array into a **pandas DataFrame**

DataFrames are the primary data storing objects used in pandas, and they can be built from arrays using `pd.DataFrame()`. Let's try passing our array in and store it in the variable `df`

In [96]:
```
df = pd.DataFrame(mydata)
```

Now let's see what `df` looks like

In [97]:
```
df
```

Out[97]:

	0	1	2
0	51	92	14
1	71	60	20
2	82	86	74
3	74	87	99

Ok, nothing too new, but we see an important difference from NumPy arrays: **DataFrames include labeled indexes and columns**

In this case, we just have the default labels of 0, 1, 2, etc, but the power of pandas starts to show when we specify the labels on our data.

Let's imagine that our data represents the scores of four students on three different midterm exams. We'll call the students Bob, Ann, Steve, and Laura and store them in a list. We'll also create the midterms:

In [98]:
```
students = ["Bob", "Ann", "Steve", "Laura"]
midterms = ["Midterm 1", "Midterm 2", "Midterm 3"]
```

We can now use these as arguments to `pd.DataFrame()` as follows:

In [99]:
```
df = pd.DataFrame(data=mydata, index=students, columns=midterms)
```

What does `df` look like now?

In [100…
```
df
```

Out[100…

	Midterm 1	Midterm 2	Midterm 3
Bob	51	92	14
Ann	71	60	20
Steve	82	86	74
Laura	74	87	99

We have an informative table showing our data!

Another important property that differentiates DataFrames from NumPy arrays is their ability to store data of different types. For example, a single NumPy array cannot store both integers and strings, but a DataFrame can.

Here's an example to show this: let's create a DataFrame which stores the name, height (in meters), age (in years), and birth month of five people

In [101…
```
name = ["Bob", "Ann", "Steve", "Laura", "Jack"]
height = [1.78, 1.70, 1.60, 1.83, 1.72]
age = [32, 24, 20, 74, 66]
birth_month = ["September", "April", "June", "September", "March"]
```

There are multiple ways to put multiple lists into a DataFrame. For now, we'll use the format: `pd.DataFrame({"label": data})`, separating our different categories with commas.

In [102…
```
people_df = pd.DataFrame({"Name": name,
                          "Height (m)": height,
                          "Age (years)": age,
                          "Birth Month": birth_month})
```

In [103…
```
people_df
```

Out[103…

	Name	Height (m)	Age (years)	Birth Month
0	Bob	1.78	32	September
1	Ann	1.70	24	April
2	Steve	1.60	20	June
3	Laura	1.83	74	September
4	Jack	1.72	66	March

Large Data Example

Let's now look at how a large machine learning dataset is stored in a pandas DataFrame and what methods we can use to view and understand the data.

We'll load in the MNIST dataset as before, storing the data samples in `X` and the labels in `y`

In [104…
```
#X, y = mnist["data"], mnist["target"]
y = y.astype(int)
```

Now, what type of object is `X`?

In [105…
```
type(X)
```

Out[105… `pandas.core.frame.DataFrame`

A pandas DataFrame!

We can check the shape of `X` using the DataFrame property `.shape`

In [106…
```
X.shape
```

Out[106… `(70000, 784)`

We interpret this as: `X` has 70,000 samples, each with 784 data points. This is the MNIST dataset, so we have 70,000 different grayscale pictures, each with 784 pixels of intensities ranging from 0 to 255.

With so many samples, we can't view them all at once! A common way to take a quick look at the data is to use the `.head(n)` method, which shows the first *n* samples of the DataFrame.

In [107…
```
X.head() # shows first 5 rows by default
```

Out[107…

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

5 rows × 784 columns

In [108…

```
X.head(8) # show first 8 rows
```

Out[108…

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0

8 rows × 784 columns

We can look at some of the statistical information of each feature, such as mean and standard deviation, using `.describe()`

In [109…

```
X.describe()
```

Out[109…

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8
count	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

8 rows × 784 columns

We can index through DataFrames similarly to NumPy arrays using `.iloc`.

For example, `X.iloc[0]` returns the data points of the first sample.

In [110…
```
X.iloc[0]
```

Out[110…
```
pixel1      0
pixel2      0
pixel3      0
pixel4      0
pixel5      0
           ..
pixel780    0
pixel781    0
pixel782    0
pixel783    0
pixel784    0
Name: 0, Length: 784, dtype: int64
```

To get the actual numerical values in a NumPy array, we use `.values`

For example, `X.values[0]` returns a NumPy array containing the pixel intensity values of the first sample, which we'll store in `tmp`

In [111…
```
tmp = X.values[0]
print(type(tmp))
print(tmp[210:230]) # Look at pixels 210 through 229
```

```
<class 'numpy.ndarray'>
[253 253 253 251  93  82  82  56  39   0   0   0   0   0   0   0   0
0
   0   0]
```

We can view the columns of a DataFrame using `.columns`

In [112…
```
X.columns
```

Out[112…
```
Index(['pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6', 'pixel7',
       'pixel8', 'pixel9', 'pixel10',
       ...
       'pixel775', 'pixel776', 'pixel777', 'pixel778', 'pixel779', 'pixel780',
       'pixel781', 'pixel782', 'pixel783', 'pixel784'],
      dtype='object', length=784)
```

You can access the contents of a column simply using the column's name.

For example, we can look at the intensities of pixel1 across different samples by running `X['pixel1']` or `X.pixel1`

In [113…
```
X['pixel1']
```

Out[113…
```
0        0
1        0
2        0
3        0
4        0
        ..
69995    0
69996    0
69997    0
69998    0
69999    0
Name: pixel1, Length: 70000, dtype: int64
```

In [114…
```
X.pixel1
```

Out[114…
```
0        0
1        0
2        0
3        0
4        0
        ..
69995    0
69996    0
69997    0
69998    0
69999    0
Name: pixel1, Length: 70000, dtype: int64
```

Activity

For this activity we will continue to use the mnist datasets we've been working with. This actvitivy will involve 6 steps:

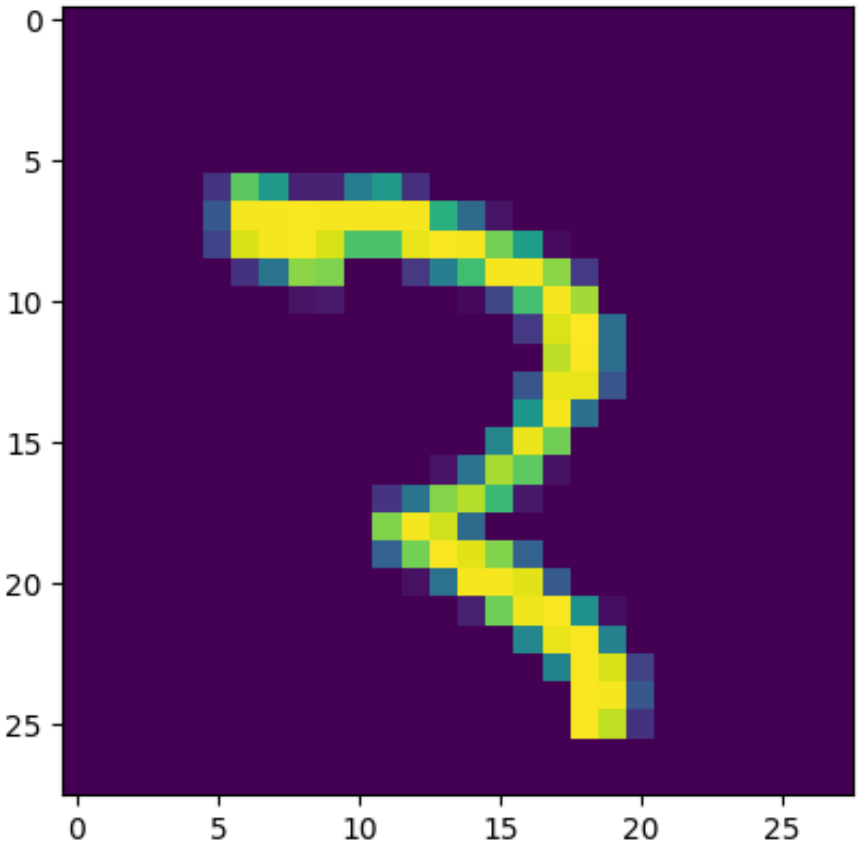
1. Access the data and target values from the mnist data set and save it to new variables called `data` and `target`. **Do not reuse the names X, and y for those variables**.
2. Use the `iloc` function to get the item at index 500 from from `data` and `target` and save them to new variables called `data_500` and `target_500`
3. Use `.values` to convert `data_500` to a numpy array called `d_500_vals`.
4. Use `int(target_500)` to convert `target_500` from a string to an integer called `t_500_vals`.
5. Use `.reshape()` to reshape `d_500_vals` to have shape (28, 28).
6. Finally, use `plt.imshow()` and `print()` to display `d_500_vals` and `t_500_vals` in one cell.

In [115…
```
#Generate the data here:
########################
data, target = mnist.drop(columns='class'), mnist['class']
data_500, target_500 = data.iloc[500], target.iloc[500]
d_500_vals = data_500.values
t_500_vals = int(target_500)
d_500_vals = d_500_vals.reshape(28,28)
########################
```

In [116…
```
#Display d_500_vals and t_500_vals here:
########################
print(t_500_vals)
plt.imshow(d_500_vals)
########################
```

```
3
```

Out[116… `<matplotlib.image.AxesImage at 0x16600ec40>`



In [117…
```
#EXAMPLE OUTPUT
```