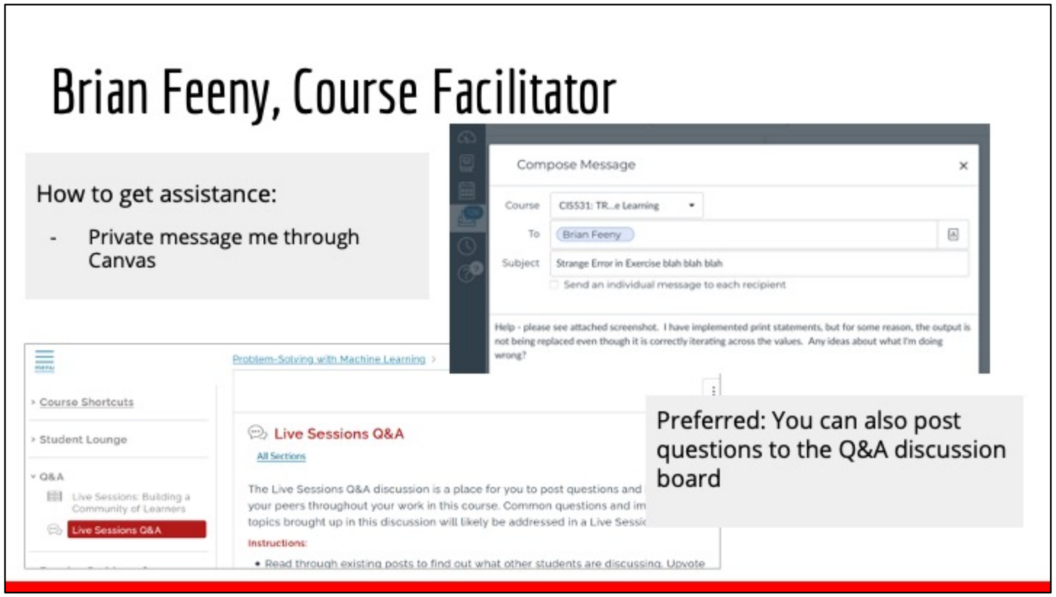
CIS531v2:
Problem-Solving With Machine Learning

Live Session 1

Brian Feeny

Note: This presentation is based substantially on work done by Melynda Eden, eCornell Machine Learning Facilitator, for version 1 of this course.



My name is Brian Feeny, and I am your course facilitator. If you have specific questions for me about the course material, assignments, or questions about project code, send me a message through Canvas. You can post questions or comments about the course material to the Live Sessions Q&A discussion board, but please do not post significant amounts of code, or full or partial solutions.

Today's Live Session

CIS531v2 - Course Info and Overview

Data Science / Machine Learning - features and labels, loss function

k-nearest-neighbors algorithm

Assignments - Class Discussions, Frame a Machine Learning Problem, Applications and limitations of k-NN, Use a Jupyter Notebook

Code - NumPy, Indexing / Slicing, Subsetting, Functions

Lab Project tips & tricks

Assignments for next week - Euclidean Distance and Facial Recognition

CIS531v2: Problem-Solving With Machine Learning

- Rigorous course that includes discussions, written assignments, coding exercises, and a project
- Prior knowledge and familiarity with Python, linear algebra (matrix multiplication), statistics, and basic calculus (derivatives) **is assumed**
- It is recommended that you try to get through modules 1, 2 and 3 the first week, saving the second week for module 4

This course is CIS531 Problem-Solving With Machine Learning. This is the first of seven courses that make up the Machine Learning Certificate. This course will help you reframe real-world problems in terms of supervised machine learning, and the final project of this course is implementing the k-Nearest Neighbors algorithm to build a facial recognition system.

The machine learning courses are rigorous, and require prior experience with Python programming. You should be familiar with linear algebra and understand matrices and matrix operations. There are two self-paced linear algebra courses available to you in Canvas (low-dimension and high-dimension). If you haven't already done so, I encourage you to go through these linear algebra courses, as they will give you a better understanding of matrix operations that are applied in modules 3 and 4 and future courses that make up this certificate program.

Regarding this course, I highly recommend getting through modules 1, 2 and 3 the first week, and having the second week to go through module 4 which includes the Euclidean Distance assignment and the Facial Recognition project. This final project is challenging, and may take some time to complete.

Deeper Dives from Dr. Weinberger

Lecture 1 "Supervised Learning Setup" -Cornell CS4780 Machine Learning for Decision Making SP17
https://www.youtube.com/watch?v=MrLPzBxG95I

Lecture 2 "Supervised Learning Setup Continued" -Cornell CS4780 SP17
https://www.youtube.com/watch?v=zj-5nkNKAow

Lecture 3 "k-nearest neighbors" -Cornell CS4780 SP17
https://www.youtube.com/watch?v=oymtGlGdT-k

Lecture 4 "Curse of Dimensionality / Perceptron" -Cornell CS4780 SP17
https://www.youtube.com/watch?v=BbYV8UfMJSA&t=2478s

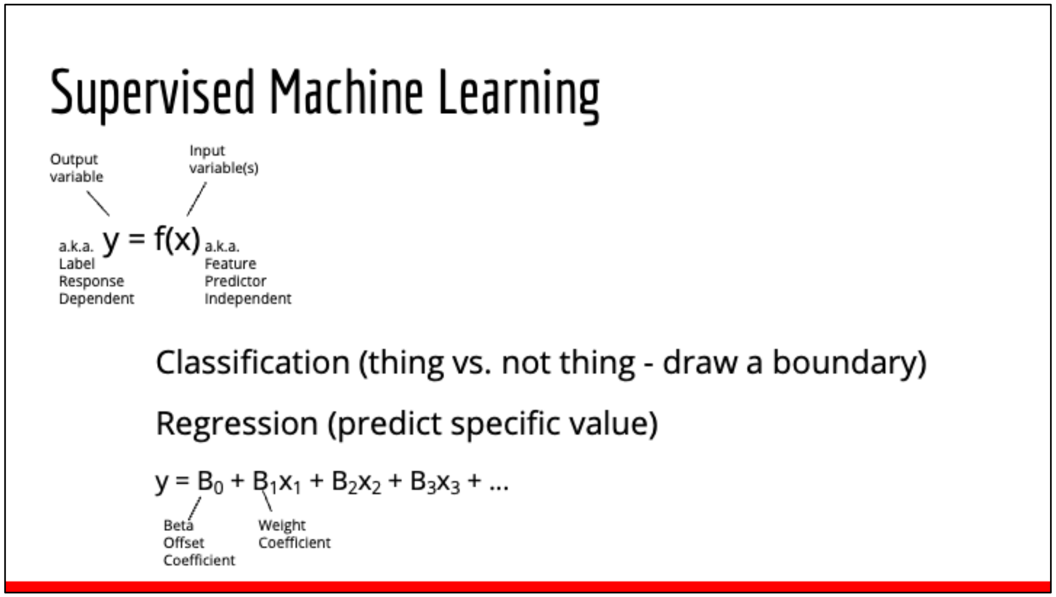
Lecture 4 begins concepts that apply to a future course.

Avoid the Appearance of Plagiarism

https://drive.google.com/file/d/19w90Kmz8zP4ajS2rfpeL9D03ss3ZN0D2/view?usp=sharing

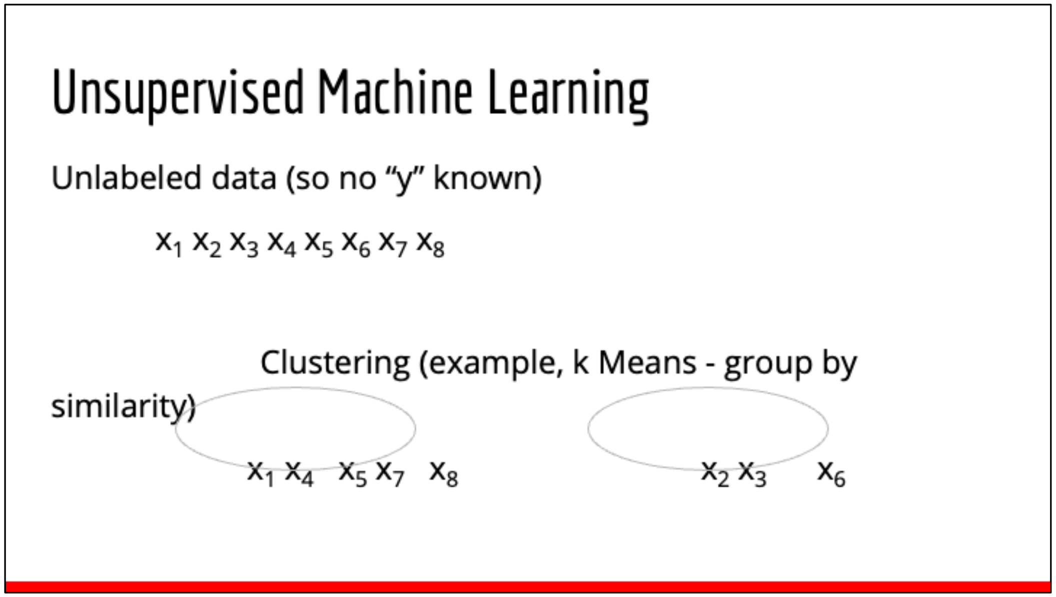
Key points:

- Show your work - comment out, do not delete, mistakes, print statements, test cells, etc. Messy notebook that shows original thought is better than pristine code that happens to look like someone else's work
- Reference helpful websites - put links to discussion boards, Q&A forums (Stack Overflow, etc.) that contained information you found useful in developing your code
- When in doubt, ask your facilitator!



There are different types of machine learning. With supervised learning, your data is labeled - you have input variables (x) and an output variable (Y). Input variables, also called independent variables, inputs, predictors, features - all referring to the same thing. Outputs, also called labels, responses, or dependent variables - all referring to the same thing.

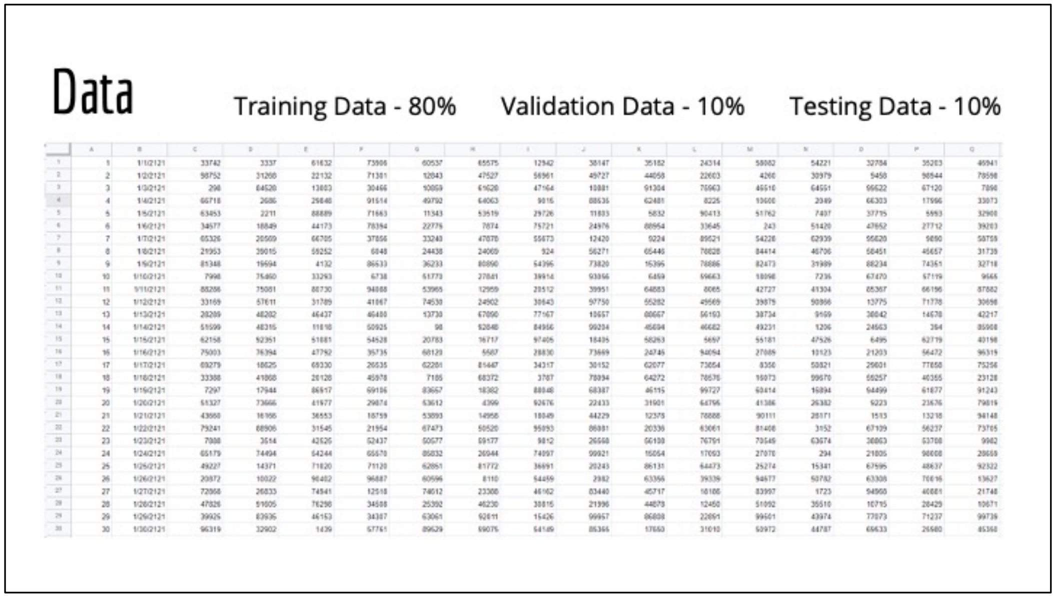
Regression and classification problems are two types of supervised learning problems. With classification, we are looking to classify something into one or more classes. We could classify emails as “spam” or “not spam”. If we have two classes, it is binary classification. We can have more than two classes; for example, we can classify images of fruit: “apples”, “oranges”, “bananas”, and “strawberries” for example. With a regression problem, you are trying to predict something that is a real or continuous value. For example, you might predict the price of a house based on square footage, zip code, number of bedrooms, and number of bathrooms.



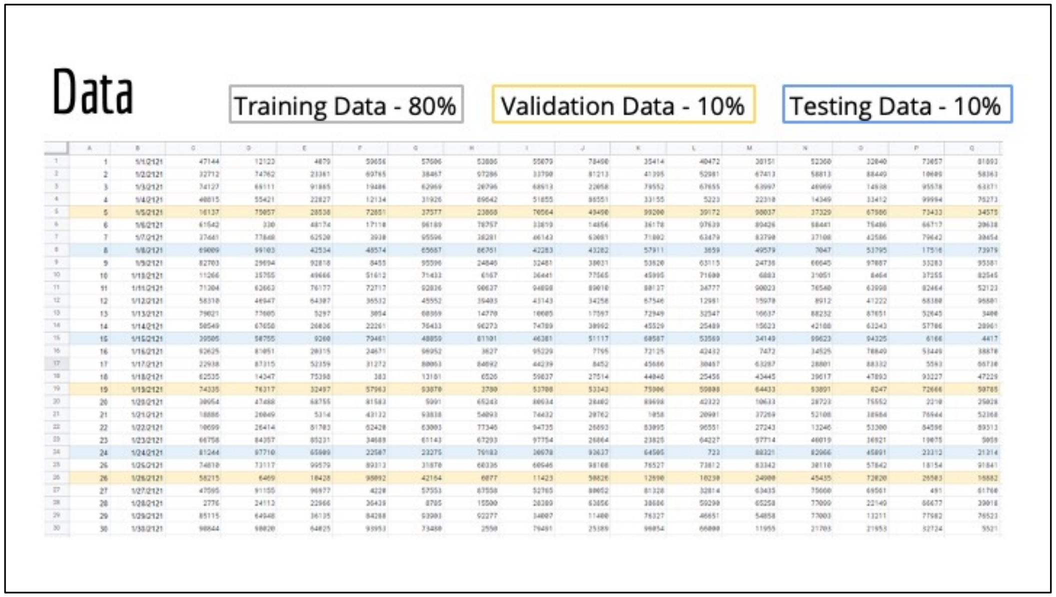
With unsupervised learning, there are input variables (x), but the data is not labeled, so no corresponding output variables, so there are no correct answers. Unsupervised learning can be used as an exploratory data analysis technique used to discover the underlying structure of the data or distribution of the data. An example of an unsupervised learning problem can be customer segmentation, or understanding different groups of customers with similar buying habits. You might have lots of customer data, but the data is not divided into named categories, so your data is unlabeled. Unsupervised learning usually involves clustering or association, where you are trying to find similarities within the data set and breaking the data into logical groups. K-means clustering is an example of an unsupervised machine learning algorithm. With k-means clustering, there are no labels, so there are no predefined classes. Given k groups to look for, k-means iterates over the data to find similarities, and segments the data into k clusters, where each data point belongs to only one group. Data points in each group should be similar, with differences between data points belonging to different groups.

Sometimes people confuse k Means and k-Nearest Neighbors, but they are two different algorithms. k-Nearest Neighbors, which will be learned in depth in this course, is a supervised learning algorithm, and as I just mentioned, k Means is unsupervised. The focus of this course and certificate program is supervised machine learning, and we’ll talk more about k-NN in a minute.

.

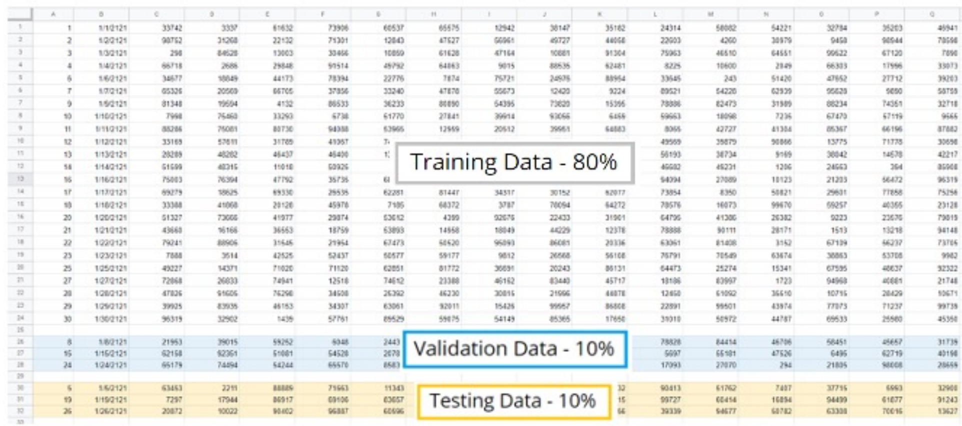


In supervised learning, we take our data set, and we divide our data into training, validation, and testing data.

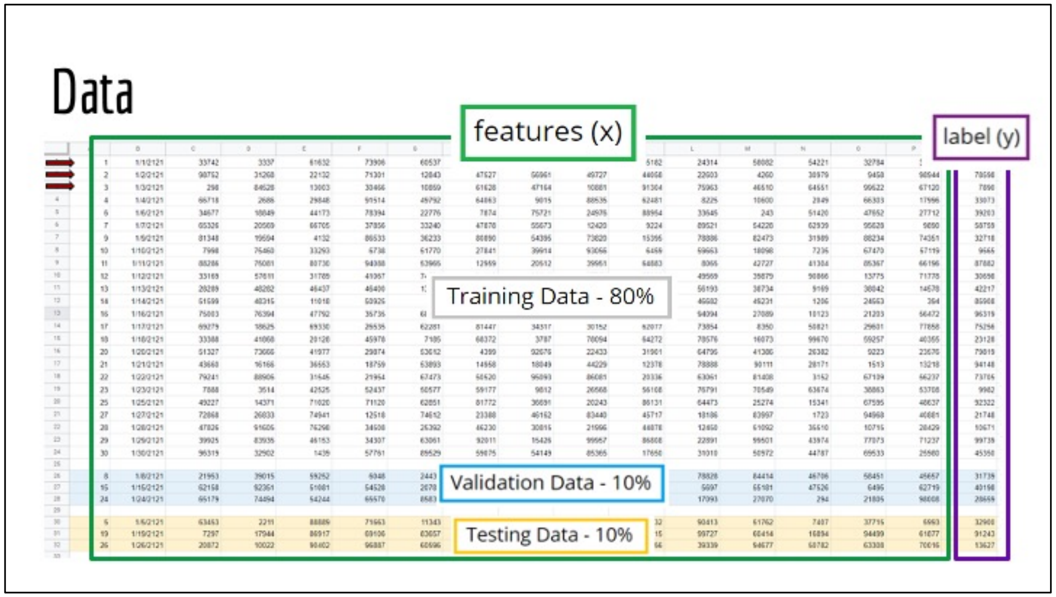


Some data scientists just divide data into training and testing, but it is better to also carve out a portion of the data to serve as validation data.

Data

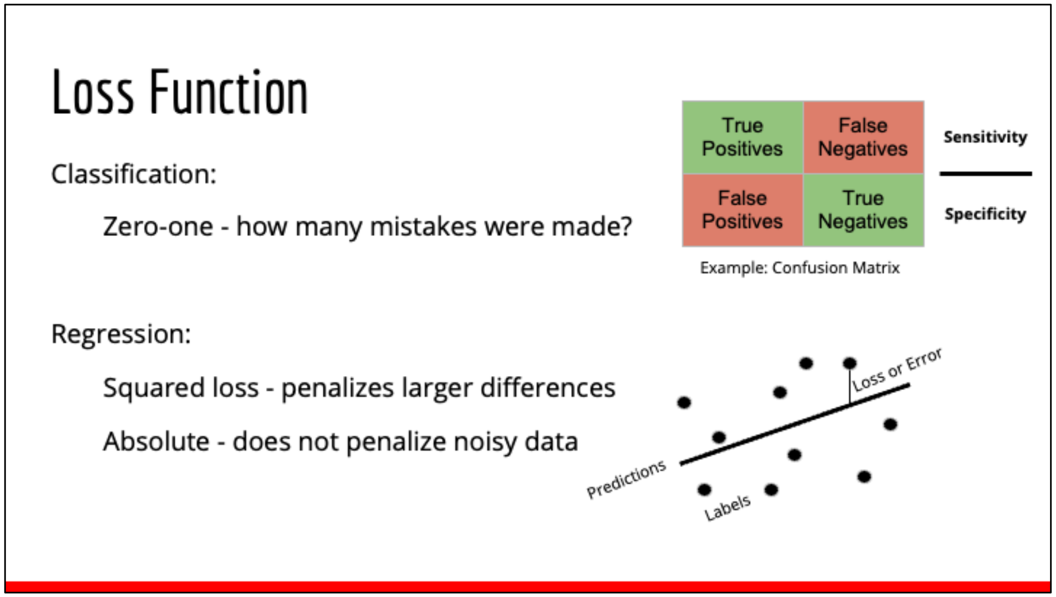


As explained in the course, it is ideal to train the data on the training data set, and evaluate the model on the validation set. We make improvements, and then retrain, and run the model against the validation data again. Continue doing this until you reach the desired level of accuracy as measured by the loss function. Then, make one final evaluation using the test data set. Since we only use the test data once, we avoid the issue of our model adapting to the test data, so we can trust the accuracy of our model analysis.



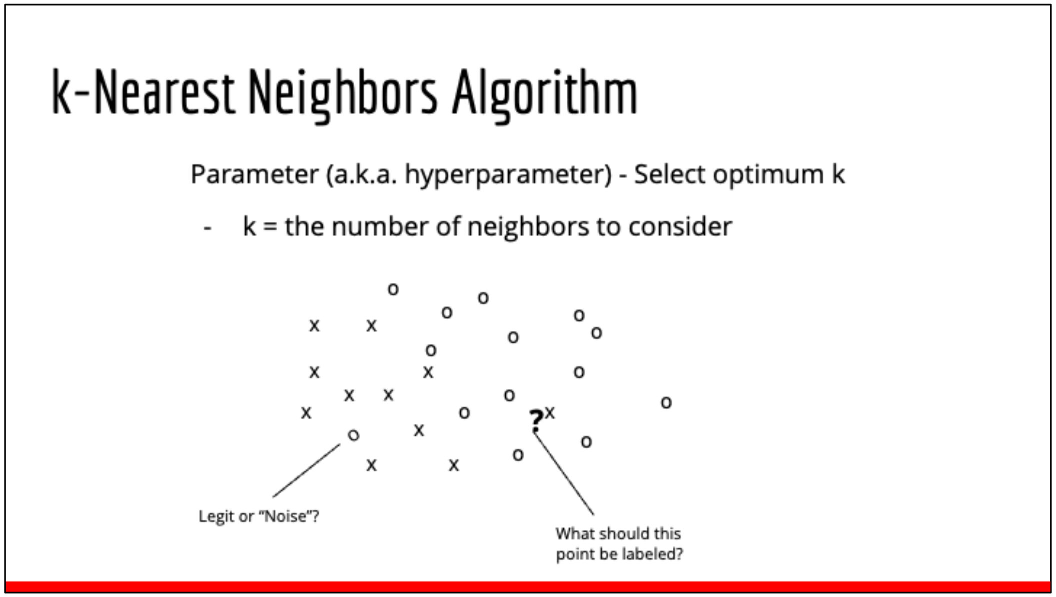
When thinking about our data in terms of input variables (x) and output label (y), we have n number of observations or data points. So in this data set, which to make things simple, looks like a small spreadsheet, each row is an observation, or a data point. All of the columns, except the last column represent features, so basically, one feature per column. If this data is for facial recognition, one column might be the width of the right eye, and one column might be the distance between the eyes. The last column is the label column - a person's name.

We give the model the training data, which includes the features and the label for this subset of the observations. When evaluating the model, first with the validation data, we **don't give the model the label column**. We give the model the features for each observation, and ask the model to predict the label. Then, we evaluate the model by comparing the label that the model gives to the true label. We can make changes to the model, retrain it, and evaluate it again with the validation data, hopefully improving with each iteration. Finally, we test the model with the test data. Again, we just give the model the features for each observation, and ask the model to predict the label. We determine the accuracy of the model by comparing the model's predicted label with the true label.



As talked about in the course, we measure the accuracy of our model with a loss function. As we make improvements to the model, the goal is to increase accuracy - minimize mistakes, which will minimize the loss function. This course talks about the different loss functions - zero-one, squared, and absolute losses.

Zero-one is the simplest loss function. It counts how many mistakes were made. This can be used for classification problems, when you want to know how many incorrect labels were given versus the total number of labels.
Squared loss function is often used in regression problems. Squaring removes negative signs, and also gives more weight to larger differences. So, if we have noisy data, the squared loss function may not work well because it will give weight to data that is way off. In this case we may want to use absolute loss function, which is the absolute differences between our predicted and actual values. So, noisy data is not penalized like it is with the squared loss function.



k-Nearest Neighbors takes labeled data, with the labels indicating a particular class, and the algorithm will predict the label for a new data point, test data, based on the labels of nearby training points. We can choose how many neighbors are considered. If we set k to 3, the three closest neighbors are considered. The three labels are taken into account, and majority rules. If there is a tie, the label chosen will be that of the nearest neighbor.

By increasing the number of nearest neighbors, we reduce the complexity of the decision boundary, thereby eliminating noise (mislabeled data points). However, if we increase the number of nearest neighbors too high, groups of data points will then be mislabeled in order to produce a smoother decision boundary. So, it is a balance to find the best k value to maximize accuracy.

Since we're relying on similarity between the new data point and the nearby training data points, the way we are determining which data are the closest, or most similar matters. The distance function should return a distance that measures the similarity of the data in a meaningful way, such that data points that are measured as closer together are in fact more similar than data points measured as being farther apart.

k-Nearest Neighbors works best if you have data that fit into a few classes, rather than having data with so many features that very few data points are similar. As dimensionality increases (specifically, having a high number of uncorrelated dimensions), similarity among data points decreases.

Defining “Noise” in Data

https://medium.com/mlearning-ai/dont-make-me-come-over-there-440f7eece4f3

NumPy

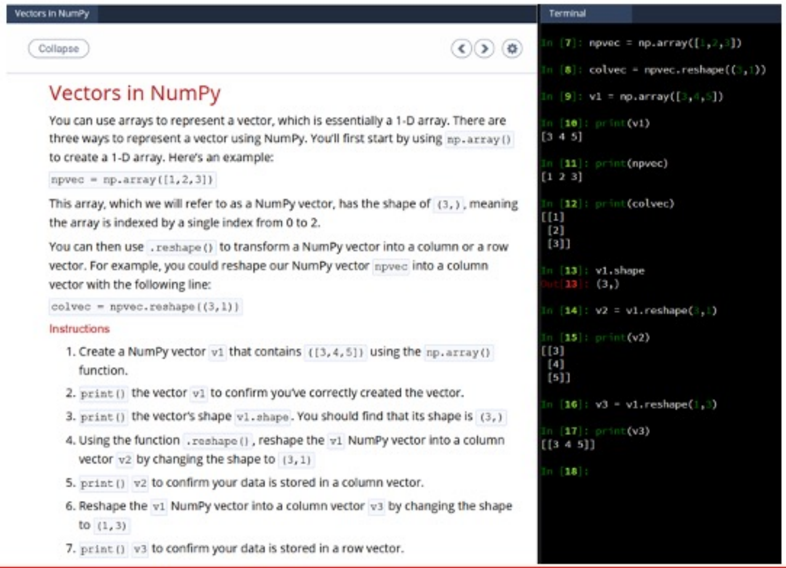
Module 3 - NumPy and Jupyter Notebooks

Codio environment

- Introduction to Numpy
- Practice Matrix Multiplication
- Additional NumPy Exercises

The Codio environment is built into the course, and you'll see it in modules 3 and 4. In module 3, there are a series of exercises to help you learn NumPy. NumPy, is a Python library for scientific computing. As the course explains, using NumPy will allow for much more efficient code than writing loops in Python.

In the codio environment, you can walk through several exercises that will teach you some important NumPy functions. The Introduction to NumPy section in the course steps through arrays, attributes of arrays, vectors, and matrixes. Today we will look at reshape, some matrix operations, as well as indexing and slicing, and I'll give you a few extra examples that you can work through that might help you understand the NumPy functions better. I encourage you to take these examples and work through them on your own in codio. It's important to understand the NumPy functions and understand how they work. You'll likely need to use some of these functions in the final project, so it's important to gain an understanding of them in the codio practice environment.



In codio you will have these instructions for reshaping a vector, which is good, but this is a pretty simple example.

Linear Algebra Concepts

Dot product - the sum of the products of the components of two vectors

Dot product is sometimes called the scalar product or the inner product

Dot product formula for vectors x and y:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2$$

Next, let’s briefly go over some linear algebra concepts that will be needed for module 4.

Dot product is the sum of the products of the components of two vectors. The dot product is the same thing as inner product. Be sure to go all the way through the NumPy codio exercises, and understand how to do dot product. You will need to be able to do this in the final project.

Dot Product - examples from Linear Algebra course

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

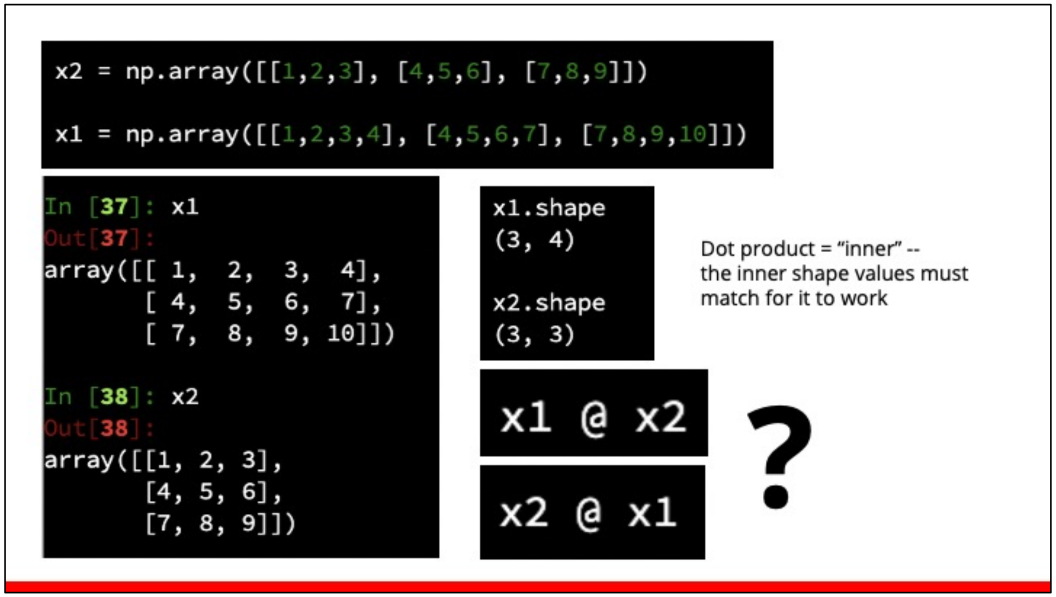
$$\mathbf{x} \cdot \mathbf{y} = 1 \cdot 3 + 2 \cdot 4 = 11$$

$$\mathbf{x} = \begin{pmatrix} -2 \\ 3 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = (-2) \cdot 4 + 3 \cdot 1 = -8 + 3 = -5$$

Here is are two examples of dot product. Both examples are taken from the Linear Algebra course that you all have access to.

Note that the dot product of x with y is the same as the dot product of y with x. So, order does not matter.



Let's look at a more complex example and make sure we understand what's going on here.

```
In [22]: x1.shape
Out[22]: (3, 4)

In [23]: x2.shape
Out[23]: (3, 3)

In [24]: x1 @ x2
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-f5addec7e971> in <module>()
----> 1 x1 @ x2

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc
 signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 4)

In [25]: x2 @ x1
Out[25]:
array([[ 30,  36,  42,  48],
       [ 66,  81,  96, 111],
       [102, 126, 150, 174]])
```

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30

1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30

1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36

1x3 + 2x6 + 3x9 = 3 + 12 + 27 = 42

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30

1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36

1x3 + 2x6 + 3x9 = 3 + 12 + 27 = 42

1x4 + 2x7 + 3x10 = 4 + 14 + 30 = 48

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30

1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36

1x3 + 2x6 + 3x9 = 3 + 12 + 27 = 42

1x4 + 2x7 + 3x10 = 4 + 14 + 30 = 48

```
In [42]: x2 @ x1
Out[42]:
array([[ 30,  36,  42,  48],
       [ 66,  81,  96, 111],
       [102, 126, 150, 174]])
```

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

4x1 + 5x4 + 6x7 = 4 + 20 + 42 = 66

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

4x1 + 5x4 + 6x7 = 4 + 20 + 42 = 66

```
In [42]: x2 @ x1
Out[42]:
array([[ 30,  36,  42,  48],
       [ 66,  81,  96, 111],
       [102, 126, 150, 174]])
```

```
In [37]: x1
Out[37]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [42]: x2 @ x1
Out[42]:
array([[ 30,  36,  42,  48],
       [ 66,  81,  96, 111],
       [102, 126, 150, 174]])
```

https://jasondeden.medium.com/matrix-multiplication-e2cf007d0755

Inner Product vs. Product, etc.

Most matrix math operations require shapes to match, not just inner values, and produce results of that same shape as well

* + -

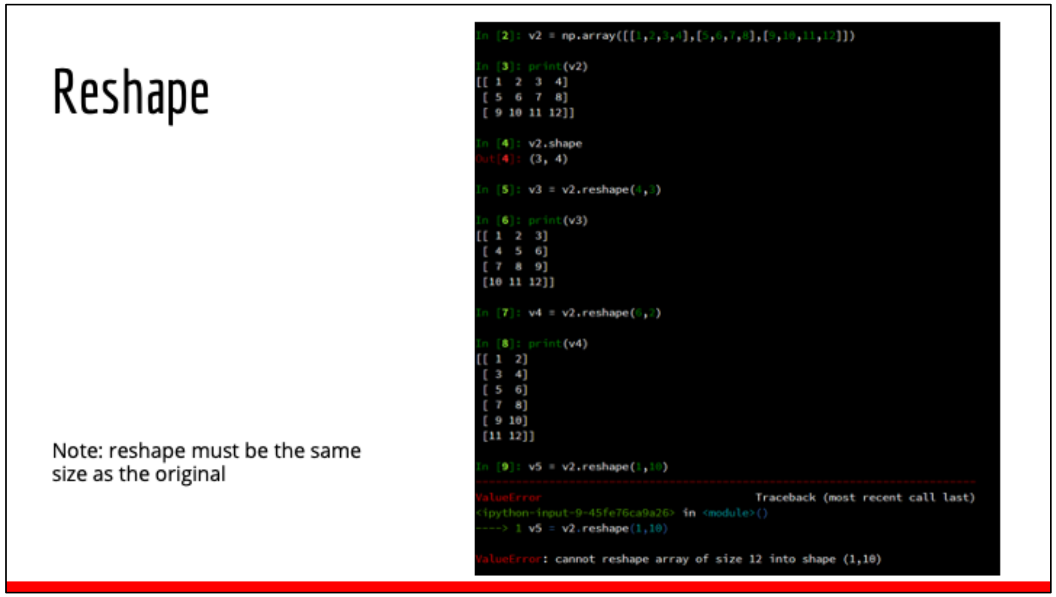
Transpose

```
In [45]: x1.T @ x2
Out[45]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])
```

Transpose

```
In [45]: x1.T @ x2
Out[45]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])
```

```
In [48]: x1
Out[48]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [49]: x1.T
Out[49]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6,  9],
       [ 4,  7, 10]])

In [50]: x1.T.shape
Out[50]: (4, 3)
```



Here is another example that's a little more complex that might help illustrate what reshape can and can't do. So if we have a matrix with 3 rows and 4 columns, we can tell NumPy that we want to reshape this into 4 rows and 3 columns. We could also say that we want six rows and 2 columns. But we can't ask for a shape that is different from the number of elements that we have. For example, we can't say we want to reshape this into 1 row and 10 columns because our array has 12 elements not 10. We would get a similar error if we said reshape into 2 rows and 5 columns, or 3 rows and 2 columns. We have to choose a shape that will exactly fit the number of elements that we have.

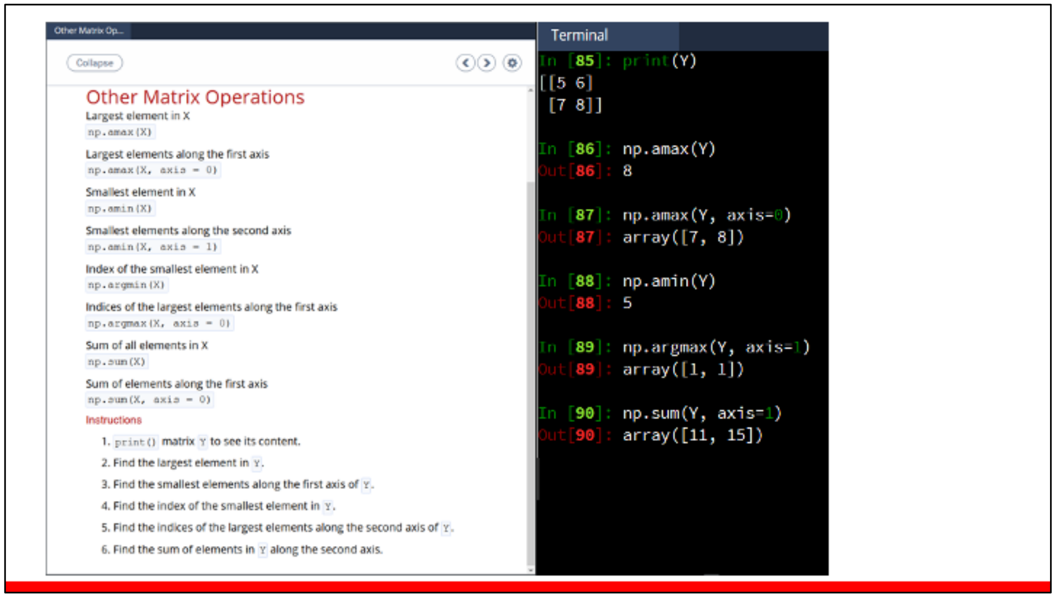
Reshape

```
In [56]: x1
Out[56]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [57]: x1.T
Out[57]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6,  9],
       [ 4,  7, 10]])

In [58]: x1.reshape(4,3)
Out[58]:
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

Reshape

```
In [56]: x1
Out[56]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [57]: x1.T
Out[57]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6,  9],
       [ 4,  7, 10]])

In [58]: x1.reshape(4,3)
Out[58]:
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

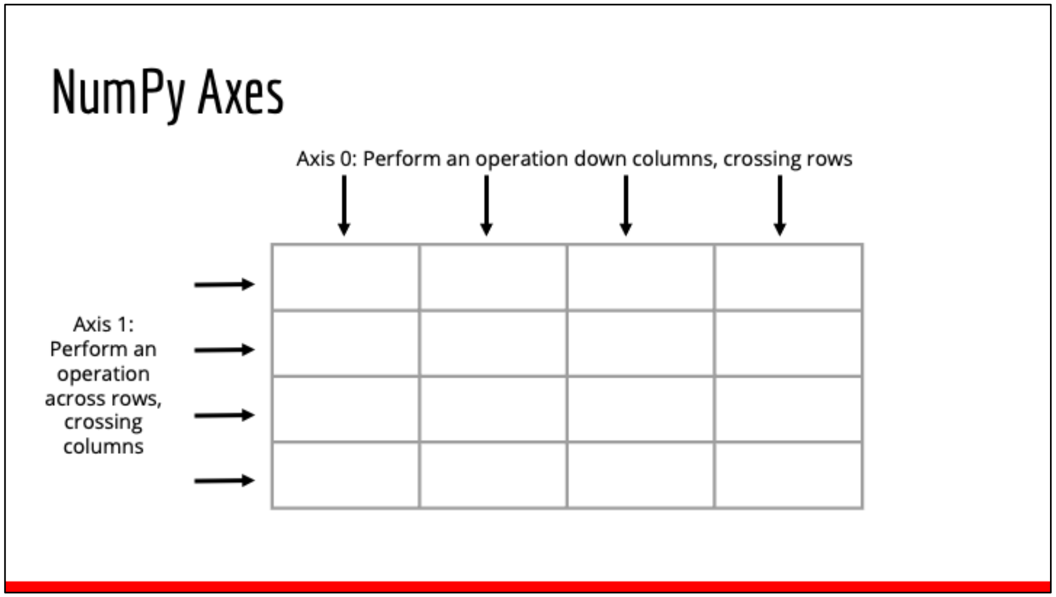
```
In [59]: x1.T @ x2
Out[59]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])

In [60]: x1.reshape(4,3) @ x2
Out[60]:
array([[ 30,  36,  42],
       [ 55,  68,  81],
       [ 83, 103, 123],
       [114, 141, 168]])
```



Let's look at the Other Matrix Operations section of the Introduction to NumPy in codio.

The code on the right follows the steps on the left. I step through some of this, but I won't talk through every example. I will make these slides available so you can go through them later if you want.

Here we can see that np.amax gives us the largest value of the matrix, which is 8, and np.min gives us the smallest value, which is 5. Np.amax along the axis 0 gives us 7 and 8, which means we're wanting the max value for each column. Np.argmax gives us the indices of the largest value, 8, which is [1, 1], meaning the 8 is in the row 1, column 1, which remember is the second row and second column. Finally, np.sum gives us the sum of the first row, 11, and the sum of the second row, 15.

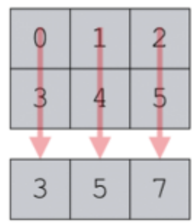


In a NumPy array, axis 0 is the “first”axis. It is the axis that runs down the rows. Axis 1 is the “second” axis, and it runs across the columns.

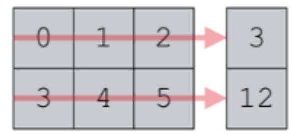
NumPy Axes, NumPy Sum

https://www.sharpsightlabs.com/blog/numpy-axes-explained/

axis = 0, np.sum()



axis = 1, np.sum()



If we set the axis to 0, when we sum, we're collapsing/crossing the rows, and summing each column. If we set the axis to 1, when we sum we're collapsing/crossing the columns, and summing across each row.

So, check out that link and read through the explanation, and I think numpy axes will make more sense.

```
In [103]: Y2 = np.array([[9,5,4,3],[7,8,6,2]])

In [104]: print(Y2)
[[9 5 4 3]
 [7 8 6 2]]

In [105]: np.amax(Y2)
Out[105]: 9

In [106]: np.amax(Y2, axis=0)
Out[106]: array([9, 8, 6, 3])

In [107]: np.amax(Y2, axis=1)
Out[107]: array([9, 8])

In [108]: np.amin(Y2)
Out[108]: 2

In [109]: np.amin(Y2, axis=0)
Out[109]: array([7, 5, 4, 2])

In [110]: np.amin(Y2, axis=1)
Out[110]: array([3, 2])
```

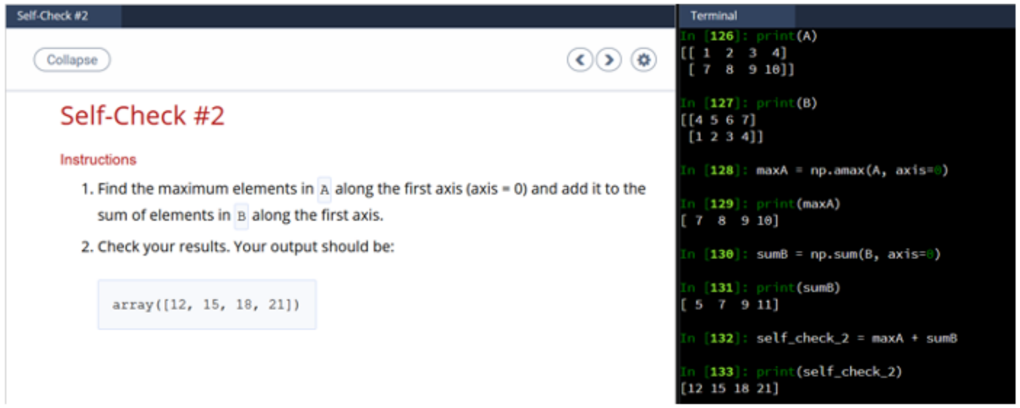
Additional Examples

```
In [111]: np.argmax(Y2, axis=0)
Out[111]: array([0, 1, 1, 0])

In [112]: np.argmax(Y2, axis=1)
Out[112]: array([0, 1])

In [113]: np.argmin(Y2, axis=0)
Out[113]: array([1, 0, 0, 1])

In [114]: np.argmin(Y2, axis=1)
Out[114]: array([3, 3])

In [115]: np.sum(Y2)
Out[115]: 44

In [116]: np.sum(Y2, axis=0)
Out[116]: array([16, 13, 10,  5])

In [117]: np.sum(Y2, axis=1)
```

Here is another example that is not in the instructions. I encourage you to try this on your own in codio.

Create a matrix Y2 like I did here. So, we have Y2 with elements 9, 5, 4, 3 in the first row, which is row zero, and elements 7, 8, 6, and 2 in the second row, which is row one. So, as we would expect, 9 is returned as the maximum value for the matrix, and 2 is the minimum. I think with this larger matrix, it's easier to understand the other functions. Here we see that np.amax for axis=0 returns the maximum value for each column. Here we have 4 columns, which are columns 0 through 3. The max value for column zero is 9 from the top row, for column 1, it is 8 from the bottom row, for column 2, it is 6 from the bottom row, and for column 3, it is 3 from the top row. Next, np.amax for axis=1 returns the maximum value for each row, so 9 from row 0, and 8 from row 1. Since we have two rows, np.amin for axis 0 basically returns the opposite values, the minimum for each column, 7, 5, 4, 2. Np.amin for axis 1 returns the minimum for each row, 3 for row 0 and 2 for row 1.

So, amax and amin return values, and argmax and argmin returns indices. So, let's look at np.argmax along the 0 axis, and we get 0,1,1,0, which tells us for each column 0 through 3, which row contains the maximum value. So, for column 0, 9 is row 0, so that's the first zero returned. For column 1, 8 is in row 1, so that's the first 1 that's returned, for column 2, the 6 is in row 1, so that's the next 1 that was returned, and finally, for column 3, the 3 is in row 0. For np.argmax along the axis 1, we get back the number for the column that contains the max value for each row. So, for row 0, column 0 contains the max value, which is 9, and for row 1, column 1 contains the max value, which is 8. Same concept for argmin. For np.argmin, axis = 0, we're getting back the number for each column that contains the minimum value for each row. So, for column 0, row 1 contains the minimum value, 7. For column 1, row 0 contains the minimum value 5, for column 2, row 0 contains the minimum value 4, and finally for column 3, row 1 contains the minimum value 2. And for np.argmin, we get back 3, 3 because for row 0, the minimum value, which is 3, is in column 3, and for row 1, the minimum value, which is 2, is also in column 3. And, np.sum gives us 44 which is the sum of the entire matrix. Np. sum for axis=0 gives us the sum of each row, and np.sum for axis=1 gives us the sum of each column, which this screenshot is missing, but should be 21 and 23.

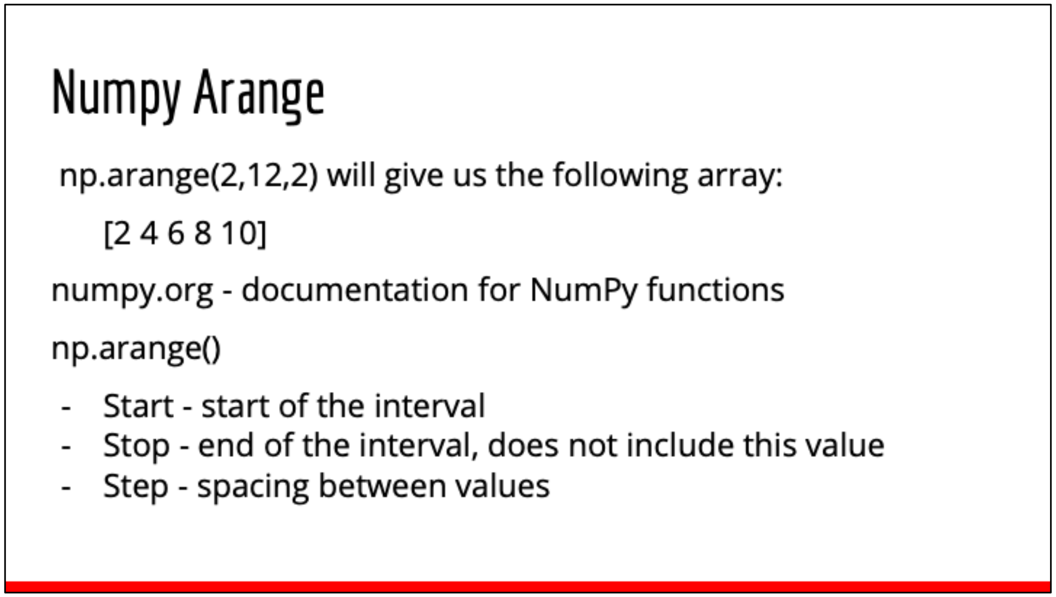


As part of the NumPy exercises, there are self checks. These are not for a grade, but all of these exercises are very important to do. You'll need to understand these NumPy functions in order to do the final project. This self check has to do with finding the maximum elements in a matrix along the first axis (axis=0), and adding it to the sum of the elements in a second matrix along the first axis.

Additional Examples

```
In [134]: C = np.array([[4,55,6,7],[11,2,33,4]])

In [135]: print(C)
[[ 4 55  6  7]
 [11  2 33  4]]

In [136]: print(Y2)
[[9 5 4 3]
 [7 8 6 2]]

In [137]:  maxC = np.amax(C, axis=0)

In [138]: print(maxC)
[11 55 33  7]

In [139]: sumY2 = np.sum(Y2, axis=0)

In [140]: print(sumY2)
[16 13 10  5]

In [141]: self_check = maxC + sumY2

In [142]: print(self_check)
[27 68 43 12]

In [143]: 
```

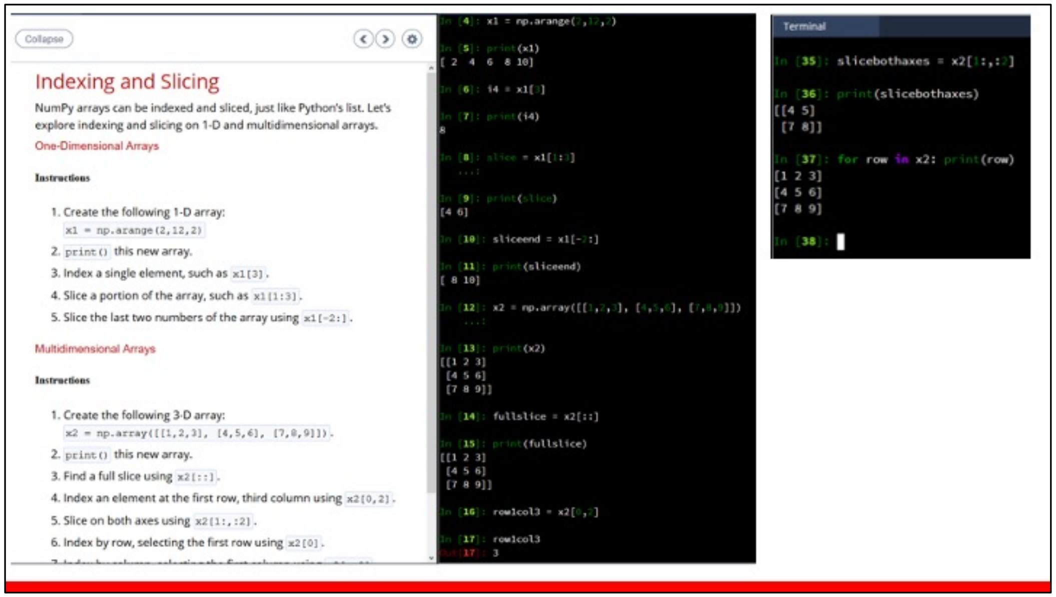
Let's do an additional example of np.sum. Here we have two matrices, matrix C and also matrix Y2. So, just like in the previous example, we're going to get the max elements from matrix C along the first axis, and then we're going to add them to the sum of the elements from the first axis in matrix Y2. So, the max elements from matrix C along axis zero are 11, 55, 33, and 7. The sum of the elements in matrix Y2 along the first axis (axis=0) are 16, 13, 10, and 5. So now, let's add these together and we get 27, 68, 43, and 12.



So, why do we get 2, 4, 6, 8, and 10 with np.arange(2,12,2)? And, how can we better understand NumPy functions?
So for any NumPy function that we don't understand, we can look at NumPy.org at the documentation.

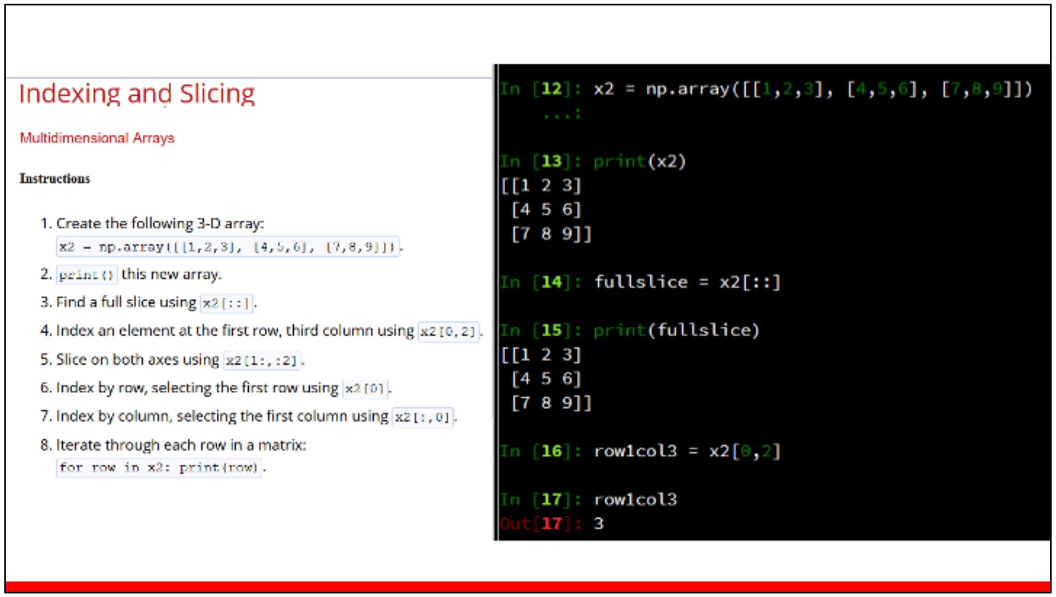
The documentation for arange explains that the first number indicates the starting point, the second number is the stopping point, and the third number indicates the step, which is the spacing between values. It's important to note that the stop indicates the end of the interval, but does not include this value. So we're creating an array that starts with 2's, increases by 2's, and goes up to but not including 12. If we don't give NumPy the first number to indicate the starting point, it will default to zero. If we don't give NumPy the second number to indicate the end, it will default to the length of the array, and if we don't give NumPy the third number for the step, it will default to 1.

(ndarray object gets created - an n-dimensional array - collection of items of all the same type, zero-based index)

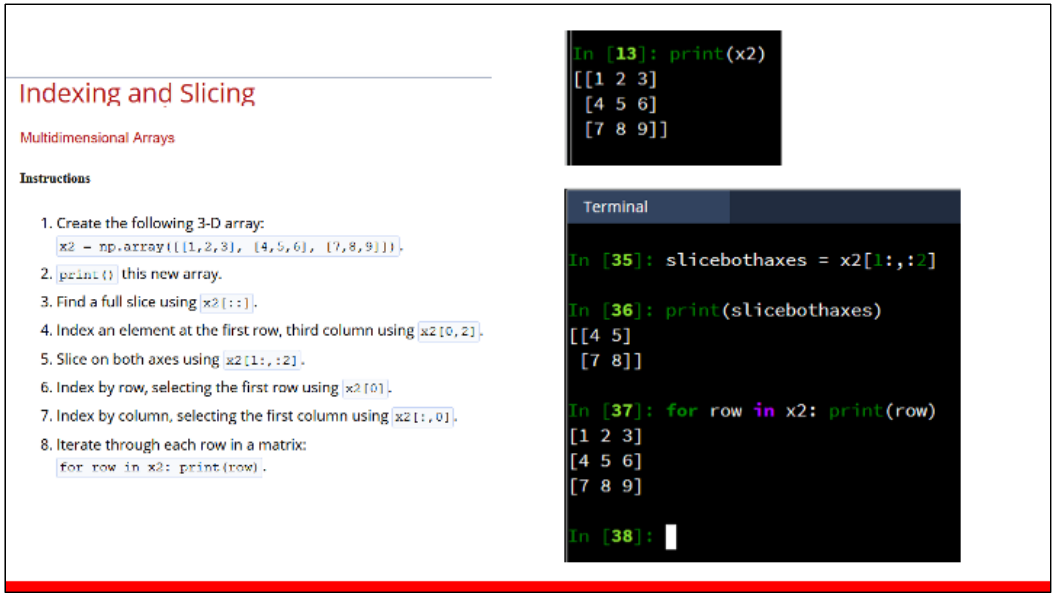


The next part of the Introduction to NumPy tutorial goes through Indexing and Slicing, so let's take a look at that more closely.

So, first it's asking us to create a 1-dimensional array using np.arange(2,12,2). And this produces the array 2, 4, 6, 8, 10.



Let's look at indexing and slicing examples. This slide steps through the exercises in the Indexing and Slicing section for Multidimensional Arrays. First, we create the array x2, and then we see that we can create a full slice, which is basically the original array, using open bracket, colon, colon, close bracket. We can return the element at the first row, third column with open bracket, zero comma two, close bracket because the zero refers to the row zero, and the 2 refers to the column 2, which is the third column.



In this example, we can slice on both axes by using open bracket, one colon comma colon two. This gives us the slice that is row 1 on, and all columns up to but not including column 2, so we skip row 0, and get rows 1 and 2, and we get column 0 and 1. This example can be a little confusing, so make sure you understand what's going on here.

Although it's not shown in the screenshot here, if you try the example on number 6, open bracket colon comma zero close bracket, it will return the first row, 1, 2, and 3. Example 7 shows us how to select the first column with open bracket colon comma zero close bracket. Finally, example 8 will print each row, so it will basically return the whole array. So, if we don't include a comma, we're providing row operations only. If we do include a comma, everything before the comma is row instructions, and everything after the comma are column instructions. Let's look at some more examples.

Array Slicing Under the Covers

We slice in 1-dimension by providing the following information:

x1[start:end]

X1[start:end:step]

x1[::]

Note: values can be negative, in which case it starts from the end and counts backwards

If we don't specify start, will assume 0

If we don't specify end, will assume the length of the array in that direction

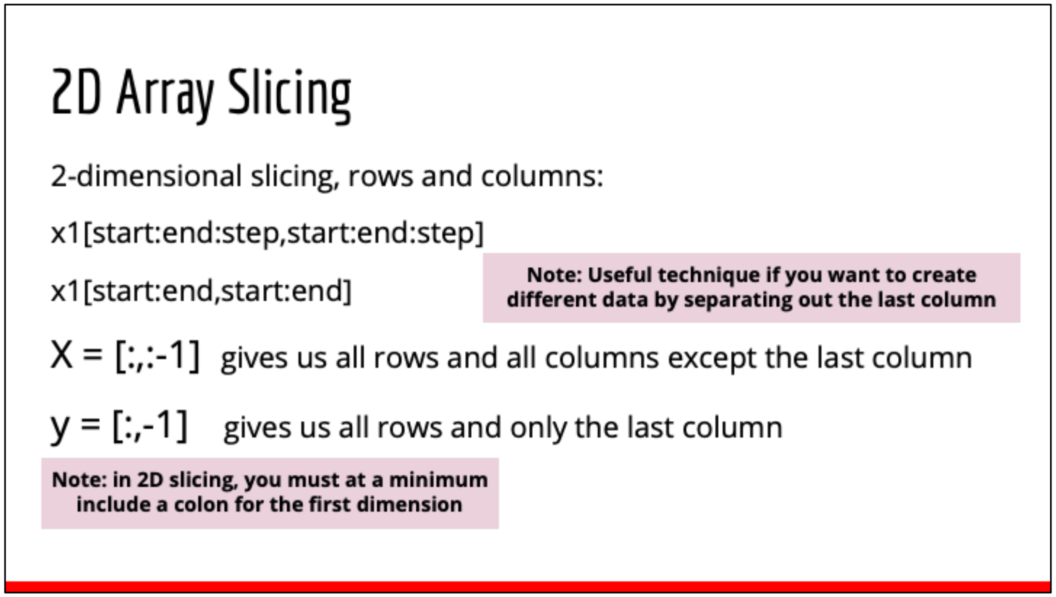
If we don't specify the step (interval), will assume 1

x1[::] is the same as x1[0:len(x1):1] -- or simply x1

We can give NumPy slicing instructions as shown. We can provide some of these parameters, or none. The basic slice syntax is start end step. We can specify the start, or the end, or the step, or none. If we just put the name of the array, then open bracket, colon, colon, close bracket, we get the whole array back. Why? Because, if we don't tell NumPy where to start, it will assume 0. If we don't tell NumPy where to end, it will assume we want the whole thing, and if we don't tell NumPy an step value, which is the interval, NumPy will put in 1 because it assumes we don't want to skip values.

The step is not necessary. If we tell NumPy x1 open bracket colon close bracket, so that we only have one colon, NumPy is going to look for the start and end values, and assume the step is 1

So this syntax will give us 1-dimensional slicing, basically giving us row-level operations.



So in 2-dimensions, to define a slice with rows and columns, we use a comma.
Remember, it’s not necessary to define any of the parameters, and often we leave off the step because we don’t need an interval.
If we have a dataset and we have several columns of features, and the last column is our labels, we might want to split that data into X and y, features and labels.
So if we tell NumPy open bracket, colon, comma, colon, minus 1, close bracket, we’re not giving a start or stop value for the rows, so we want all rows, and we’re not giving a start value for columns, but our end value is minus 1, meaning give us all columns except the last one.
If we tell NumPy open bracket, colon, comma, minus 1, we’re asking for all rows, and this time we’re only providing the starting value for columns, which we’re saying minus 1, so give us only the last column.
In the final project, the x and y data are already done for you, but I thought this example would help illustrate 2-dimensional slicing.

It is possible to index and slice in 3-dimensions, but that’s beyond the scope of this class.
You can visit NumPy.org and look at the documentation for more information and examples on Indexing and Slicing.

Additional Examples

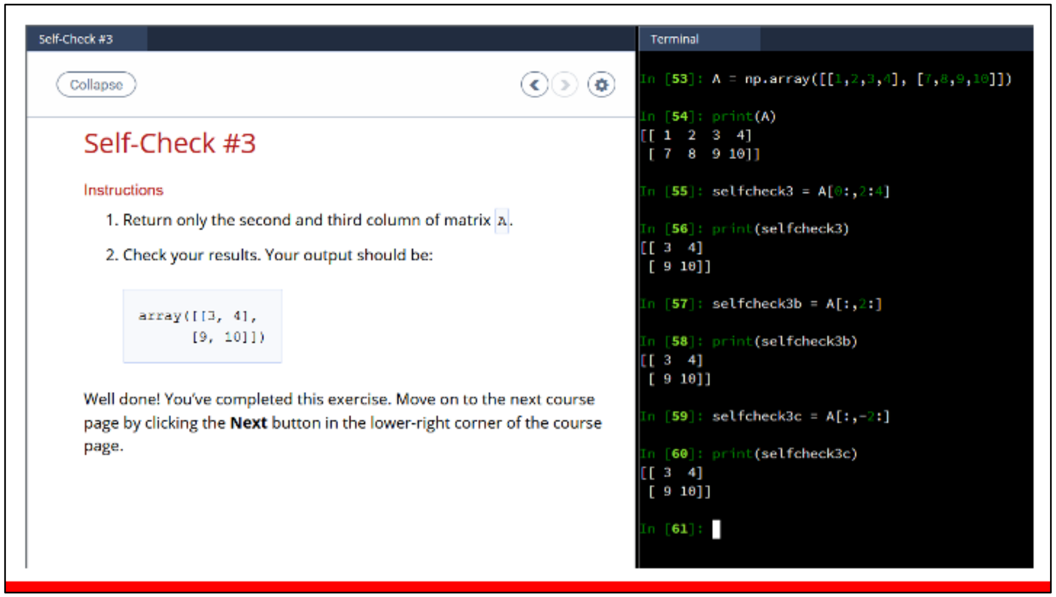
```
In [39]: print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [40]: slice1 = x2[:,:2]

In [41]: print(slice1)
[[1 2]
 [4 5]
 [7 8]]

In [42]: slice2 = x2[::2]

In [43]: print(slice2)
[[1 2 3]
 [7 8 9]]
```

```
In [44]: slice3 = x2[0:,:2]

In [45]: print(slice3)
[[1 2]
 [4 5]
 [7 8]]

In [46]: slice4 = x2[1::2]

In [47]: print(slice4)
[[4 5 6]]

In [48]: slice5 = x2[::2,::2]

In [49]: print(slice5)
[[1 3]
 [7 9]]
```

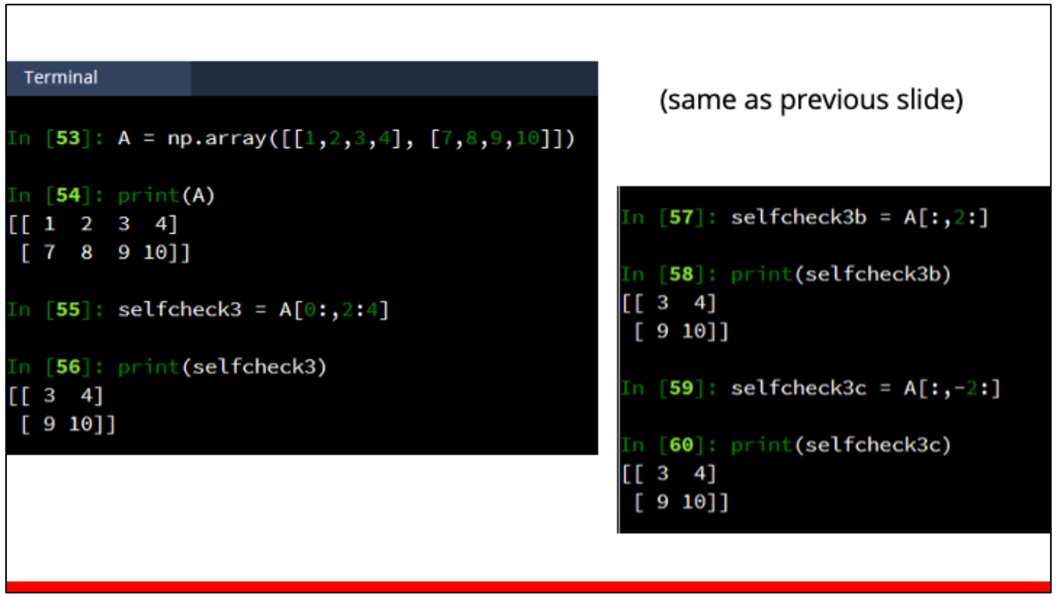
Let's look at these five examples to better understand the explanation from the previous slides.

These examples use the same array we saw earlier, x2.
So, slice 1, which is open bracket colon comma colon 2 will return the first two columns because its asking for all rows, and columns up through column 2, which is column zero and one. Slice 2 is an interesting example.
Slice 2, which is open bracket colon colon 2, will return row zero and row 2. Why does it return rows zero and row 2? Since we did not use a comma, this only operated at the row level. Since we didn't put in a value before the first colon, we are not specifying a starting point, so the default is zero. Since we didn't put in a value between the colons, we didn't specify a stopping point, so we will get the full length of the array, but we did put a 2 after the second colon, which tells the slicing operation the interval step is 2, which means, return every other row.
Now, let's look at slice 3, which includes a comma, so we're giving

instructions for rows and columns. Slice 3 is open bracket zero colon comma colon two. This is basically the same as slice 1, but we've specified all rows, and column up through column 2, so we get all rows of columns zero and one.
Next, slice 4 is open bracket one colon colon 2 close bracket, and it returns row 1 only, which is the second row. Here's another example that does not include a comma, so it's a row operation. We're saying start with row 1 and return every other row. Since there are three rows, we only get row 1, the second row.
Finally, slice 5 is a fun example that focuses on defining the step, which is the interval. You can see that it returns the four corners of the matrix x2. It does this because we're asking for every other row and every other column. So slice 5 is open bracket colon colon two comma colon colon two. So, this eliminates row 1 and column 1 (the second row and the second column), and returns 1, 3, 7, 9.



Next is self-check 3. Make sure you work through this and the other exercises on your own, and do additional examples until you understand the functions. This slide shows three different ways to solve this exercise.



This is the same as the previous slide - enlarged so we can see it better.

Additional Examples

```
Terminal

In [70]: print(A)
[[ 1  2  3  4]
 [ 7  8  9 10]]

In [71]: slice1 = A[::3]

In [72]: print(slice1)
[[1 2 3 4]]

In [73]: slice2 = A[0:,:3]

In [74]: print(slice2)
[[1 2 3]
 [7 8 9]]
```

```
In [75]: slice3 = A[0:,2:3]

In [76]: print(slice3)
[[3]
 [9]]

In [77]: slice4 = A[1:,2:4]

In [78]: print(slice4)
[[ 9 10]]

In [79]: slice5 = A[0:,2:4]

In [80]: print(slice5)
[[ 3  4]
 [ 9 10]]
```

Here are some additional examples using matrix A that we created for self-check 3. I encourage you to try these examples and maybe experiment on your own as well to better understanding slicing and indexing.

Slice 1 is a row operation. The start and end is not specified, but the interval is three. Since we only have rows 0 and 1, we only get back row zero.
Slice 2 contains a comma, so we're giving instructions for rows and columns. We're asking for all rows, and columns up through column 3.
Slice 3 asks for all rows, and columns starting with column two, up through and not including column 3, so this returns all rows of column 2 only.
Slice 4 is interesting. Here we're saying starting with row 1, return column 2 up through column 4, so we get two values back, 9 and 10, which is row 1, columns 2 and 3.
Finally slice 5 is similar to slice 3 but returns one more column

because the end is set at column 4 rather than 3.

Additional Examples

```
In [82]: x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])

In [83]: print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [84]: slice1 = x2[1:3,1:3]

In [85]: print(slice1)
[[5 6]
 [8 9]]

In [86]: slice2 = x2[0:2,1:3]

In [87]: print(slice2)
[[2 3]
 [5 6]]
```

Here are more examples with matrix x2. Here slice 1 says start with row 1 up through but not including row 3, and column 1 up through but not including column 3. Slice 2 says start with row zero up to but not including row two, and column 1 up to but not including column 3.

Additional Examples

```
In [89]: x3 = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])

In [90]: print(x3)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

In [91]: slice1 = x3[0:4:2,1:3]

In [92]: print(slice1)
[[ 2  3]
 [10 11]]

In [93]: slice2 = x3[1:4:2,0:4:2]

In [94]: print(slice2)
[[ 5  7]
 [13 15]]
```

Here's one last slide of examples, this time with matrix x3 which has four rows and four columns, so zero through three and zero through three. Slice 1 asks for row zero up to but not including row 4, with a step of 2, so every other row, and asks for column 1 up to but not including column 3, so we get back column 1 and column 2 of row 0 and row 2. Finally, slice 2 says start at row 1, go up to but not including row 4, and get every other row AND start at column zero, go up to but not including column 4, and get every other column. So we get back four elements. The 5, which is row 1 column zero, the 7 which is row 1 column 2, the 13 which is row 3 column 0 and 15 which is row 3 column 2.

```
In [64]: x1
Out[64]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [65]: x1[-1]
Out[65]: array([ 7,  8,  9, 10])

In [66]: x1[::-1]
Out[66]:
array([[ 7,  8,  9, 10],
       [ 4,  5,  6,  7],
       [ 1,  2,  3,  4]])
```

What would we get if we did this?:

`x1[2:0:-1,3:1:-2]`

```
In [64]: x1
Out[64]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [65]: x1[-1]
Out[65]: array([ 7,  8,  9, 10])

In [66]: x1[::-1]
Out[66]:
array([[ 7,  8,  9, 10],
       [ 4,  5,  6,  7],
       [ 1,  2,  3,  4]])
```

What would we get if we did this?:

`x1[2:0:-1,3:1:-2]`

```
In [73]: x1[2:0:-1,3:1:-2]
Out[73]:
array([[10],
       [ 7]])
```

Can you explain why?

More NumPy Goodness

`np.mean()`

`np.median()`

`np.square()`

`np.sqrt()`

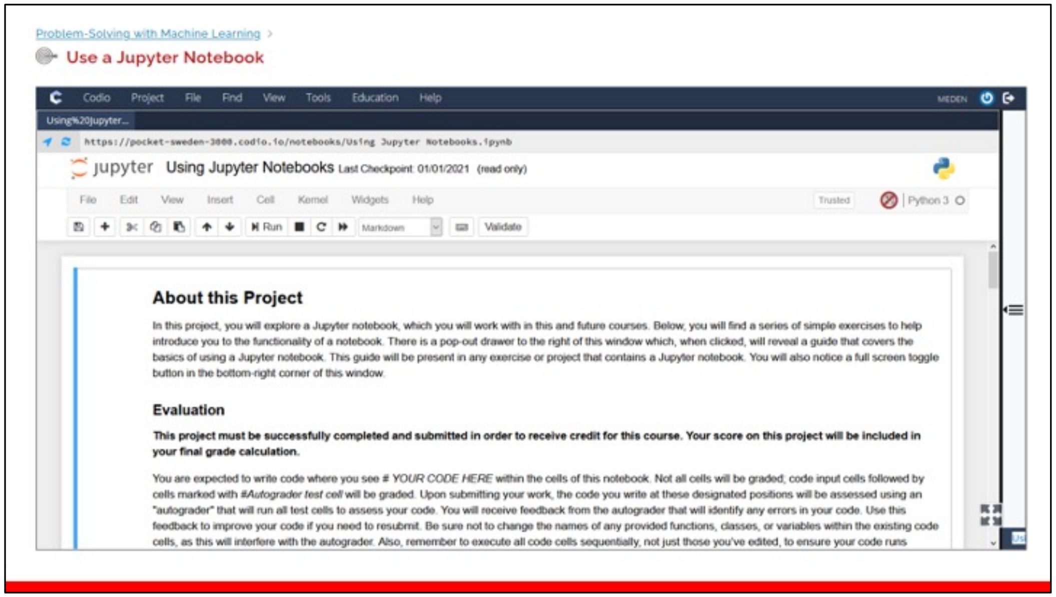
`np.log()`

`np.exp()`

`np.repeat()`

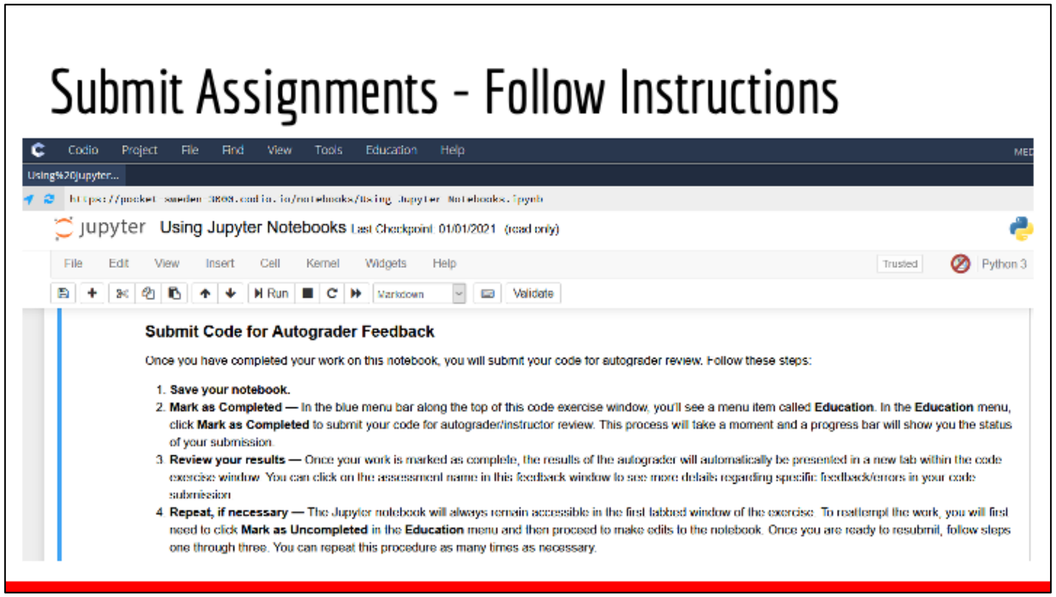
Nesting functions can be useful:

```
np.square(np.sqrt(x)) = x

np.log(np.exp(x)) = x
```



Also built into the course in the codio environment are Jupyter notebooks. So you won't need to download anything to do the coding assignments in this course. Jupyter Notebook is an open-source application that can be to test, document, run, and share code.

Let's look at this assignment, Use a Jupyter Notebook. There are detailed instructions that will step you through using a Jupyter notebook, so just read through and do what is asked of you for each of the steps. This is a very straightforward assignment, and should not take very long.



Once you've completed the assignment, follow the instructions for submitting the assignment. Notice that it tells you to save the notebook and then mark it as complete. As the instructions state, you'll click on the Education tab on the menu above, and one of the options is Mark as Complete. Then the Autograder will run and you will soon see your results. You can make changes if needed, and resubmit the assignment.

Challenging Lab Survival Tips and Tricks

Print statements: at each step, print the current output

```
print("The value at step 3 = {}".format(variablename))
```

"b" - Insert cells and use them to perform experiments

Helpful hint: label a cell "Begin testing" and another one "End testing" at each section and keep your work.

Test Cells and Grader Functions

Replicate the tests, and stages of tests. What are they trying to do?

Look at the output of the grader function. How might that be produced?

https://jasondeden.medium.com/debugging-code-issues-using-print-statements-and-simple-test-data-84d9487d1254

New to Notebooks?

Lots of great tutorials on YouTube that cover the basics. If you’re brand new, check them out.

Specifics for our environment:

https://drive.google.com/file/d/1BM1EAFX3OSdlbu-DVtAVuqInkZHu6p-J/view?usp=sharing

(Low production quality, but helpful content…)

Module 4 Assignments

Euclidean Distance Function Without Loops

Build a Facial Recognition System

There are two assignments for module 4 of this course, and they are due by the last day of this course. You can submit the assignments early, and you can re-submit them if needed, as long as you're submitting before the deadline. The autograder portions of the assignment will give you feedback as soon as you run those portions.

For these two assignments, you will need to know and understand linear algebra concepts like dot products and matrix operations, and how to do matrix operations with NumPy as well as transformations such as reshape. So it is important to have a good understanding of the material in the Linear Algebra courses, as well as a good understanding of the material in modules 1, 2 and 3 of this course.

Questions?

Thank You

End of Live Session 1

Appendix

Optional content that can be covered if

Python Functions

```
def samplefunction(inputarray, scalarvalue):
      import numpy as np #just in case hasn't been
done
      timesscalar = inputarray * scalarvalue
      finalarray = np.square(timesscalar)
      return finalarray
```

```
In [113]: def samplefunction(inputarray, scalarvalue):
     ...:     import numpy as np #just in case hasn't been done
     ...:     timesscalar = inputarray * scalarvalue
     ...:     finalarray = np.square(timesscalar)
     ...:     return finalarray
```

Python Functions

Function Name

Function Input Values
(must be provided when function is called)

```
def samplefunction(inputarray, scalarvalue):
    import numpy as np #just in case hasn't been done
    timesscalar = inputarray * scalarvalue
    finalarray = np.square(timesscalar)
    return finalarray
```

Code to be run based on the input

End of function, values to return follow

One or more values
(multiple can be separated by a comma)
to return after running the function

Note: Indentation matters!

Call a function

```
functionname(input1, input2,...)
```

```
In [116]: x1
Out[116]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [117]: samplefunction(x1,3)
Out[117]:
array([[  9,  36,  81, 144],
       [144, 225, 324, 441],
       [441, 576, 729, 900]])
```