Working with Multidimensional Data Using NumPy

by Janani Ravi

Course Overview

Course Overview

Hi. My name is Janani Ravi and welcome to this course on Working with Multidimensional Data Using NumPy. A little about myself. I have a master's degree in electrical engineering from Stanford and have worked at companies such as Microsoft, Google, and Flipkart. At Google I was one of the first engineers working on real-time collaborative editing in Google Docs and I hold four patents for its underlying technologies. I currently working on my own startup, Loonycorn, a studio for high-quality video content. In this course we learn the simple and intuitive functions and classes that NumPy offers to work with data of high dimensionality. We start off with basic operations to explore multi-dimensional data such as creating, printing, and performing basic mathematical operations on arrays. We study indexing and slicing of array data and iterating over these lists. We'll see how images are basically three-dimensional arrays and how they can be manipulated with NumPy. We then move on to complex indexing functions. NumPy arrays can be indexed with conditional functions as the less arrays of indices. We'll then see how broadcasting rules works. This allows NumPy to perform operations on arrays with different shapes. We'll study array operations such as the np. argmax, which are very commonly used when working with ML problems. Then we'll move on to studying how NumPy integrates with other libraries in the PyData stack. We cover specific implementations with SciPy as well as with Pandas. At the end of

this course you will be very comfortable using the array manipulation techniques that NumPy has to offer to get your data in the right form for extracting in sites.

Exploring Multidimensional Data Using NumPy

Module Overview

Hi and welcome to this course on Working with Multidimensional Data Using NumPy. If you've used the Python programming language for data analysis or for machine learning algorithms, chances are you've used NumPy. NumPy is a very cool scientific computing package using all kinds of data analytics. NumPy supports a wide variety of mathematical computations such as linear algebra, Fourier transform, random number capabilities and so on. The basic building block used by NumPy is a very powerful n-dimensional array. The NumPy package in addition to the ability to represent these n-dimensional arrays also contains a whole suite of operations that can be performed very efficiently and effectively on these arrays. There is a suite of open source software available in the Python programming language for math, science, and engineering and NumPy forms the core of this suite. Many of the other packages such as Pandas, SciPy, Statsmodels, etc., are built on top of NumPy. In addition to multidimensional array representations, NumPy offers a large collection of high-level mathematical functions which can operate on these arrays.

Prerequisites and Course Overview

Before we get started with the course let's see what the pre-reqs are so that you can make the most of your learning. This course assumes that you're comfortable programming in Python. If this is your first course in the Python programming language, I'd recommend that you take one of these other courses on Pluralsight first to brush up on your Python basics. Here are some of the skills that you need to be familiar with in order to tackle the demos that are in this course. You need to be comfortable programming in Python 3 because that's what we'll be using for all the demos. All our demo code will written using Jupyter notebooks. Jupyter notebooks are basically a browser= based interactive shell. These notebooks can be thought of as a must-have tool for any Python programmer. NumPy is a numeric computing package which means that you need to

be familiar with mathematics at a high-school level. NumPy is all about performing operations on matrices. You should be comfortable with high-school level basic matrix math. Let's take a look at the topics that we'll cover during this course. We'll start off by exploring multidimensional data. You'll see how to create arrays, print these arrays, perform basic operations on them. We'll then move on to intermediate operations such as shape manipulation of arrays, making deep and shallow copies. The next module will focus on indexing operations and arrays. There are many ways in which arrays can be indexed in NumPy. We'll see how we can use array indices to index into other arrays. We'll then see how broadcasting works and see how we can stack vectors together. NumPy can be used for data exploration as well. You can build histograms to see the distribution of your data. There are a variety of other miscellaneous functions such as argmax and argmin, which are very commonly used in machine learning operations and we'll study all of these. In the next module we'll focus on how NumPy integrates with other libraries in Python. We'll see examples of how NumPy works with SciPy as well as Pandas. This will give you an idea of how well NumPy is integrated with other libraries in the Python ecosystem.

Creating Arrays

If you're a data analyst or a data engineer or a data scientist who's working with Python, chances are that you use one or more of these packages. Statsmodels is a Python package used for estimating statistical models, performing statistical tests and for statistical data exploration. Scikit-image is a Python package which is primarily used for image manipulation and image processing. Scikit-learn is a very popular and widely used package for machine learning. If you're starting with machine learning in Python the scikit-learn library is what you'll typically start off with. If you're working with data that is best represented in a tabular format with rows and columns, chances are that you're using the Pandas library. Let's say you explored, transformed, and aggregated a bunch of data and you want to use it for insight, then you might have used matplotlib to visualize your data. Now under all of these libraries which are presented in the open source system lies NumPy. The multidimensional NumPy array is the code building block which is used to represent data in all of these packages. The best way to learn NumPy is to get hands-on coding with NumPy, which is why this course will mainly focus on demos. We start off by seeing how we can create multidimensional arrays in NumPy. Here is my Jupyter notebook. All the data that I'm going to use for examples in this course are under the data directory. These datasets are also available for you to download on Pluralsight. This is our current working directory and here is where we'll write all the code for all the demos in this course. Here is my Jupyter notebook running. I'm going to create a brand new Python 3 notebook and I'm going to give it a meaningful

name. All of the names will start off with the module number and the demo number that we're currently working on. When Jupyter notebook using the Anaconda 3 distribution the NumPy package typically comes along with your Python installation. If you do not have NumPy you can simply get it by using pip install. Notice the exclamation point before your command to indicate that this command should be executed on your shell. NumPy is already installed here; we're ready to get started. Import the NumPy module so that we can call functions and classes from within here. NumPy is typically aliased as np within your Python code. Let's create our very first array here called array_one. This we can do by calling the np. array function and you can pass in a list of any kind of data elements into that function and your NumPy array will be created. Simple NumPy arrays might seem very similar to Python lists, but NumPy arrays have a variety of functions that are available that can be used to manipulate the array. These functions are not available on Python list. Here we have our second array, array_two, which is once again an array of numbers. We specify the numbers in the form of a Python list and use the np. array function to create a NumPy array. A very commonly used function in NumPy is the np. zeros function. This is typically invoked when you want to create an array of the desired shape and here we want an array which is 3, 4 with three rows and four columns. In addition to creating the array the np. zeros function initializes this array to have all elements equal to 0. This is a very quick way to initialize your array in order to use it within your other calculations. This is a very popular function. Very similar to np. zeros is the function np. ones. This creates an array of the shape that you specify and initializes all the elements to one. You can also explicitly specify to NumPy that you want your array elements to be of a specific type. This you do by passing a dtype argument. Here the dtype is np. int16. Sixteen-bit integers. When your array is printed to screen, notice that it says that the data type is int16. If you want to set up an np array, that is you want to create the array, but you do not want to initialize the nps in the array you'll use the np. empty function. NumPy also offers the eye function, which allows you to set up a square matrix with 1s along the diagonals and 0s elsewhere. Here is the 3x3 matrix or the 3x3 array and you can see 1s along the main diagonal. NumPy offers other functions as well, which allow you to set up arrays with specific values very easily. The arange function takes in a start value and end value which is exclusive and a step function to create an array. The resultant array is an array of even numbers which start at 2. It goes up to 18. Notice that 20 is not included and steps are of 2. That is 2, 4, 6, 8, other elements in that array. The step size between the elements in your array can be floating point numbers as well. Here is an array which starts at 0 with the step size of 0. 3 and goes up to 2. NumPy can create two-dimensional arrays if you pass in a list of lists to initialize your NumPy array. Here we have a list of lists with two rows and three columns and that is how our NumPy array is created. Every NumPy array has a shape member variable which you can use to

determine the dimensions of the array. You can see that this is a 2 x 3 array, two rows and three columns. A common operation when working with NumPy arrays is to create the array first, for example using a function such as arange and then to reshape it to the shape that you're interested in. Here we are reshaping this array to be 3 x 2 with three rows and two columns. Here you can see the resultant 3 x 2 array. You have to be careful when using the reshape function on a NumPy array, though. If you try to shape an array into a shape that it won't fit in, you'll get an error like you see on the screen. What we tried to do here was to take an array, which was originally of length 6 and tried to reshape it into a 3 x 4 array. A 3 x 4 array requires 12 elements; 3 multiplied by 4 is equal to 12, which means this reshape was bound to fail. Let's say you have an existing array of a certain shape such as an array_nd which is a 3 x 2 array. Let's say you want to create another array which has exactly these dimensions, you can use the np. ones_like method. Ones_like basically creates an array with exactly the same dimensions as another array which you already have, but pre-fills this array with 1s.

Printing Arrays

If you are debugging data in Python you'll want to print your arrays to screen. Let's see how you can do it. We'll import the NumPy library as usual and let's create our very first one-dimensional array using np. arange and we can print it out and you can see the array is printed neatly within square brackets. We'll now create a two-dimensional array here using the reshape function and then print it out to screen. Notice the double square brackets in order to indicate the two dimensions of the array. As you add more dimensions to your NumPy array, here we have a three-dimensional array, which is 2 x 3 x 4. There are a total of 24 elements here. When you print it to screen you'll have as many square brackets as there are dimensions in the array. This array c can be thought of as containing two arrays of size 3 x 4 within it. Well, there was nothing remarkable so far. What if you had a really huge array, one with 10, 000 elements, for example? When you print it out to screen you'll see that most of the array has been encompassed in the ellipsis, the ... indicating that there are elements there, but they haven't been displayed. Let's now recreate an array with ten-thousand elements, but this time we'll reshape it into a two-dimensional array with 100 rows and 100 columns. Notice that when we print this out, once again we see the ellipsis indicating that there are elements there, but they haven't been displayed. There are some print options that you can configure in order to have your arrays display in a specific way. Np. set_printoptions allows you to format your array display. Once you specify the format, all arrays that you display within this notebook will follow the same format. The threshold indicates the total number of array elements which you need to have, which triggers the summarization or the

display of the ellipsis. Here we are saying we want no summarization. We want all elements of the array to be printed. Let's try printing our 100 x 100 array once again. Notice that we have no summaries here, no ellipsis. All elements of the array are displayed.

Basic Array Operations

In this demo we'll study the basic operations that can be performed on NumPy arrays. We'll start off by creating two one-dimensional arrays, A and B, one which contains three 10s and another which contains three 5s. Both of these arrays have the same shape, which means you can add them together as you see on the screen. They will be added element wise. Notice that every element of the resultant array is the sum of individual elements in the original arrays that were added together. You can also perform element wise subtractions of arrays using the minus sign. Notice that the result is an array of 5s because 10 minus 5 is 5. If two arrays have the same dimensions you can perform element wise multiplication as well. Notice that the result is 50, n multiplied by 50. In all of these operations which are performed element wise the resultant array has the same dimensions as the original two arrays. You can perform element wise division as well, a divided by b will give you an array of 2s. You can perform mathematical operations on an array using a scalar as well. Here the modular operator will be applied to every element of array a. Five modular 3 is equal to 1 and that is our resultant array. The resultant array has the same dimensions as the array a. You can apply conditional operators to NumPy arrays as well and the result of this condition will be also in an array. A less than 35, that is true for all elements of array a. The result is an array whose shape matches array a with an array of Boolean trues. Let's perform another conditional evaluation of this array. We want to check whether all elements of this array a are greater than 25. This is clearly false. Each element is just 10. We get an array of false values. Let's try some mathematical operations on two arrays. Here are two arrays A and B and they are 2 x 2 arrays initialized as you see on screen. Now if you use the multiplication operator to multiply A with B you will get an element wise result. Here every element of the array A is multiplied with the corresponding element of the array B. The first element at 0, 0 is 1. It's multiplied by the element 2 with a result of 2. Similarly the element at the bottom right at 1, 1 is 1 multiplied by 4, gives you the result 4. But if you remember your high school mathematics, when you multiply two matrices together you perform matrix multiplication. If you want to perform matrix multiplication you'll use the dot function in NumPy. Here is the result of a. b, matrix multiplication just to refresh your memory. An element at a particular row and a particular column is the sum of an element-wise multiplication of all elements in that row multiplied by all elements in that column. You can get the same matrix multiplication result by calling np. dot and passing in

the two matrices that you want multiplied. If you remember your high-school math, the number of columns in the first matrix A should be equal to the number of rows in the second matrix B in order for matrix multiplication to be possible. Here are some other ways that you can manipulate NumPy arrays. As you can see, NumPy arrays can be treated like any other primitive data type in Python. This is element-wise multiplication once again as we multiply A by the scalar 3. Every element of the array A is multiplied by 3 and the resultant array is assigned to A again. The mathematical operation along with the assignment operator, the += or the multiplication equal to or the division equal to, can be used with NumPy arrays on both sides of the assignment. This is the same as B is equal to b + a, and here is the result. Let's instantiate a new NumPy array which contains the ages of four different individuals. If you want to find the sum of all their ages you can simply call the sum function on a NumPy array. Other useful functions are the min function to find the minimum age. Here you can see that it's 12. Max function will help you find the maximum age, which is 20. You can apply mathematical operations on all elements along a specific access in NumPy. For example, along all rows or all columns. For this let's set up a two-dimensional array. This is a 3 x 4 array. Now you can call the numbers. sum function, but this time we specify an access argument and we set axis=0. The result is an array which contains the sum of all elements along all columns. Axis=0 sums up the columns. If you want sums along all rows you'll use axis=1. This is the second access and the resultant array has the sum of all elements in every row. Similarly you can specify the axis argument to other mathematical functions as well such as the min. Min along axis=1 finds the minimum value for each row. The minimum element in the first row is 0 and that's what is there in the result.

Universal Functions

Mathematical operations such as sine course, exponent etc., are all available in NumPy. They are called universal functions. Here is an array of angles specified in degrees. The angles range from 0 to 90 degrees. Instead of expressing these angles in degrees we want to convert them to radians, which means we multiply by pi and divided by 180. Notice that the pi mathematical constant is available as a part of NumPy and this mathematical operation applies to every element of the angles array. You can use the np. sine universal function to find the sine of all of the angles that we set up. Notice that sine of 90 is equal to 1, which is what we'd expect. Now instead of performing a mathematical computation to calculate the angle in radians we can simply use the np. radians function. This function is applied to every element in the input array and the result is an array of angles in radians. Just like sine values we can use the np. cos function to find cosine values of our angles in radians. NumPy offers the entire suite of these functions. Np. tan will give

you the tan values. You can also find the sine inverse of our sine values by using np. arcsine. The sine inverse gives us angle in radians. If you want to convert angles and radians to degrees instead of performing the mathematical computation yourself, you can simply use np. degrees. Other functions that you can perform on array are statistical functions. Let's set up an array of test_scores. If you want to find the mean of test_scores you can call np. mean. Here is the average of our test scores. You can find other statistical values as well such as the median by calling np. median. You can use NumPy functionality to read in a file of data. Np. genfromtxt will generate an array from the data that is present in the salary. csv file. The delimiter for the csv file is a comma and that's what we've specified in the input argument. Salaries is a one-dimensional array of salaries as you can see on screen. The shape of the salaries data indicates that there are roughly 1150 salaries in the csv file. When you print out the shape of a vector or a one-dimensional array you get the length of the vector followed by a comma and nothing specified there because it's a 1-D array. When you start exploring salary data you might first want to gather statistical information about this data. A way to get a quick feel for the salary information is to calculate the mean, median, standard deviation, and the variance and there are functions for all of these available in NumPy.

Indexing and Slicing Arrays

Indexing and slicing are very efficient ways to access a subset of elements from arrays, but these also tend to be confusing. Let's create an array that we can use for our slicing operations. The np. arange function will create an array where the elements are from 0 to 10 and raise to 2 will square all these numbers. The resultant array is the square of all numbers from 0 to 10. We can use the square brackets to index individual elements in the array. Array elements start at index 0, which means a of 2 will access the third element in this array. The square bracket notation to access individual array elements is exactly the same as in Python lists. When you work with NumPy arrays you might often see a negative index within square brackets. The - sign indicates that we should start counting the elements from the end of the array, not the beginning. The very last element in this array will be represented by the index -1; Minus 2 refers to the second-to-last element in this array. You can access a subset of elements within a NumPy array by specifying an index range. Here we want all the elements starting at index 2 up to but not including index 7. Specifying a subset of the array using indexes in the manner that you have seen is called slicing an array. This is important to note. Any time we specify an index range within square brackets the end index is excluded. Negative indices can be used with index ranges as well. Here we want to start at index 2 and we want to go up to but not include index -2. That is the second-to-last

element. Notice that the resulting array does not include the second-to-last element. When you omit the end index in an index range that simply means that you want to go to the end of the array. Here we start at index 2 and go all the way through until the end. If you omit the start index, that means you want to start at the beginning of the array. Here we start at the beginning and go up to the index 6. We do not include the index 7. You can access specific elements within an array by specifying a step as well. Notice that here we want to start at the very beginning. Our end index is 11 and our step is 2. The resulting array will start off at the very beginning of array A and access every alternate element with a step of 2. We can also step through an array in a reverse direction. In fact, this is a very tricky way to reverse an array. Let's see how. By omitting the start index and the end index we have indicated that we want to start at the beginning of the array and go up until the very end, but we want to step -1. That is we want to step from the end of the array backwards. This is what reverses a NumPy array. All elements within a NumPy array may not be of the same data type. Here is a mixed array with strings as well as integers. The B type here is kind of strange. It has a less-than sign, a U, and a 7. The less-than sign indicates that this array is laid out in little endian style. The greater-than sign indicates big endian, and U indicates unicode. What you see here is a two-dimensional array. If you specify just one index within square brackets you'll access this array rule wise. Index 0 refers to the first row of this array. Index 1 refers to the second row and index 2 refers to the third and final row in this array. In order to access a single element in a two-dimensional array you need to specify two indices within square brackets. Student of 0, 1 is the student, Beth in row 0 and column 1. You can specify index range operations within square brackets for multidimensional arrays as well. The first index range refers to row data in rows 0 and 1 and remember, 2 is excluded. And we're interested in columns 2 and 3. The cells that are included in the result are the ones that satisfy both of these conditions. They are in rows 0 and 1 and they are in columns 2 and 3. You can choose to completely omit the indexes for specific dimensions. Yes, this is a little hard to read, but it's actually simple when you see what exactly is going on. The range for the first dimension that is for the rows is completely omitted. That means we want to select all of the rows and the second dimension has a 1:2, which means we are interested in the column 1. The cells which belong to all rows of column 1 are the ones selected. Here is another example for you to get familiar with the syntax. Student to student, all rows because we only have the colon specified there, we're interested in columns 1 and 2. So all rows in column 1 and 2. Let's parse using negative indexing with multidimensional arrays as well. The negative index applies only to the first dimension, that is to the rows. In the second dimension after the comma we only have the colon indicating that we're interested in all columns. No columns ranges are specified, we're interested in all columns. The result will be elements in all columns for the row -1. Minus 1 indicates that we have to count from the end, that is the last row.

Some more negative index practice. Here at the first dimension we're interested in the rows -3 to -1. Minus 1 is not included. The end index is not included. We're interested in columns starting with -2 going to the very end. Because are just three rows in our students table, minus 3 refers to the first row. If you count backwards it refers to the first row. That means we are interested in the first two rows, minus 3 and minus 2. On the column front, -2 up to the very end indicates that we are interested in the second-to-last column going up to the last column. If you're interested in all the elements along a particular dimension you can also use the ellipsis to specify this. Here we want the elements from the row with index 0 and all columns as specified by the ellipsis and that's what we get in the result. Here we are interested in all rows, but only in that column with index 1, which is the second column.

Iterating Over Arrays

Once we've created a NumPy array and performed some mathematical operations on them it's often the case that we might want to iterate over the elements in the array in order to do something with them. Let's see how to do that in this clip. Let's create an array that we've seen before, an array of the squares of all numbers from 0 to 10. This is the way we've set it up. We can iterate over a NumPy array using a for in loop. For I in a we print out the square root of every element. Iterating over single dimensional arrays or vectors is straightforward. Let's set up our multidimensional 2-D array and let's see how we can iterate over it. A simple for in loop like this, for i in students, we'll iterate over the first dimension of the array so it will iterate over the elements in the first dimension. The elements in the first dimensions are arrays themselves. A simple iteration of this kind will give us the elements in every row of this 2-D array. There is a function that you can use in NumPy to access every element individually within the array and that is the flatten function. Flatten allows you to iterate over every element of an array. This is row major flattening by default. We'll first iterate over all the elements which are present in a single row, then move on to the next row and then to the third and so on. Notice that the names of all our students, which are in the first row of index 0 are printed first. You can pass in an input argument to the flatten function in order to flatten your array in a different order. When you specify order is equal to F, this is column major flattening of your array. Elements which belong to the first column are printed out first, then we move on to the next column, to the next column, and so on. If you observe the result here on screen, Alice 65 and 71 are all elements which are in the first column of our array. Let's create a multi-dimensional array which has three rows and four columns. This is an array that we have seen before. Now there is another way to iterate over elements in an array using the nditer function. By default, nditer iterates over the array in row

major form. You go through one row first, then to the elements in the next row and so on. Just like in the case of flatten you can use the order = F input argument in order to iterate over the elements in your array in column major form. The nditer function might seem very similar to flatten. In this context, however, there is a significant difference. Flatten returns a result array with the elements in flattened order. Nditer simply allows you to iterate over the elements that are already in the order that you're interested in. The operation of iterating over every element in your array is not very efficient. It might be the case you want the innermost dimension of your loop to be expressed as a vector or as a single dimensional array. You want to iterate over all the outer dimensions. You can do this by using the external_loop flags parameter in nditer. Order = F forces a column major iteration. Every column is expressed as a 1-D array so the innermost dimension is expressed as a vector or a 1-D array. When you use the nditer function for iteration this is a read-only iteration you can't assign to elements in your array. This will be an error. You can see that assignment destination is read-only. Notice the use of the ellipsis to assign to specific elements that you're iterating over. When you use the nditer function you have the option of making the iteration a read-write iteration. You'll simply specify op_flags as equal to readwrite. Here you can assign to the values of the array and when you take a look at the result, once again the ellipsis is used to signify the current element that we're iterating over. We've iterated over every element here and if we look at the result, X or original array contains the element-wise multiplication of X by itself.

Reshaping Arrays

One of the most common operations that you might want to perform on a NumPy array is reshaping the array. This is extremely common when you're working with machine learning problems which involve newer networks. We'll initialize a two-dimensional array of countries, Germany, France, Hungary, and Austria and their corresponding capitols. This array has two rows and four columns; it's a 2 x 4 array. Another function that you can use in NumPy to flatten an array is the ravel function. Now this flattens the array to a 1-D vector. It functions in a very similar way to flatten, except that a copy of the array is made using ravel only if needed; only if the array shape has changed. NumPy also allows you a very easy way to transpose your matrices. You can call the T member variable of any array and that contains the transposed matrix. Then all your rows have become columns and columns have become rows. The transposed matrix T is a NumPy array as well and you can call ravel on it to flatten it and you'll now get the elements in a different order. By default, the ravel function flattens the array row-wise. Our array A has two rows and four columns. We can now reshape this array to have four columns and two rows by calling the

reshape function. Specify the dimensions within reshape as input arguments and here you can see visually is the result of the reshape. Our original array contained two rows and four columns, the first row comprised of country names, the second row of capitols. On the right you can see our reshaped 4 x 2 array. The data contained within this reshaped array is exactly the same as the original data. By performing a reshape, NumPy considers the number of columns that are present in the reshaped array. It then takes that number of elements from the original array. There are two columns in the reshaped array. NumPy picks the first two elements in the first row from the original array. Next, after having filled in the first row with two elements, it then moves downwards, the next two elements pick before the second row are Berlin and Paris and then Hungary and Austria. Understanding how NumPy reshapes the array will spare you from unpleasant surprises when you work on real-world code. Let's move on to another example. We want to create a 2-D array with three rows and five columns. We can do this with a simple arange and reshape. You can reshape the same 15-element array that we created using arange to a completely different shape. This is with five rows and three columns. As long as the new shape of the array accommodates all the elements in the original array, the reshape is possible. Five multiplied by three or three multiplied by five are both equal to 15; both these reshapes are possible. Once again notice how NumPy picks the elements from the original array, which was a 1-D vector created by arange to fit into the reshaped array. And here is an example of a reshape function that will fail. We have 15 elements which we create using arange. We try to reshape it into a 3 x 6 array. A 3 x 6 array requires the number of elements to be a multiple of 18. This reshape will clearly fail. Let's take another example. Once again, we've created an array of strings that is an array of countries. This is a 1-D array. The reshape function on a NumPy array can be called with negative arguments for dimensions as well. Negative 1 has a special meaning, which indicates that we don't know what the value of that dimension is. Here we are saying that we want three columns in the resultant arrays and we don't know the number of rows that depends on the number of elements that are present in the original array. NumPy is then free to create as many rows in the result as it needs to in order to fit the original data. The resultant array should definitely have three columns, though. Here is another example of using -1. Here we have said that we want three rows in the result. We don't know the number of columns that might exist when you divide it into three rows. NumPy will go ahead and create as many columns as needed in order to fit the shape on the original data. Here is an example of a reshape that will not work. We have six elements in the original array. We are trying to get a reshape array with four rows. The original elements do not fit. Six divided by four is not an integral value and this reshape will fail.

Splitting Arrays Horizontally and Vertically

Let's say you have many elements stored in a NumPy array. You might want to split the array for easier computation or for some other purpose. That's very easy to do with NumPy as well. Let's set up a simple array comprising of nine elements from 0 through 8. You can use the np. split function in order to split this array. You'll see that you want three pieces of this array and you'll get three equal-sized sub arrays in the result. If you try to split the original nine-element array into four equal-sized sub arrays, that is clearly an error. The split will not work because this does not result in an equal division. You can be more specific about your split as well. Here you can see that I want to split the array X at index 4 and then at index 7 and that's what you'll see in the result. Here once again is the 2 x 4 array of countries and capitols that you've seen before. You can split this array horizontally by using the using the hsplit function. The second argument 2 is the number of sections that we want this array to be split into. If you print out the result you can see that we have two resultant arrays, two equal sections that have been horizontally split from the original array. The variable p1 held the first section of the split array. The variable p2 holds the second section. We can also choose to horizontally split this array into four sections. Every column of the original array will be one section of the split result, and as you can imagine, a four-column array cannot be horizontally split into three equal-sized sections. This is clearly an error. Just like the X split there is also the V split, which will allow you to split into vertical equal-sized sections. Our original array had two rows. We split it into two equal-sized sections, each section comprises of a single row.

Image Manipulation

At this point we are pretty comfortable with NumPy. Let's see how we can use NumPy for image manipulation. Any image, whether it's a jpeg image or a png image, when programmatically represented is just basically an n-dimensional array. Every cell that you see depicted in this image matrix holds some value to represent a single pixel in that image. Now what exactly is this number that can be used to represent a pixel depends on the type of image. Let's say you have a colored image. Then every pixel will require that RGB value to represent color. RGB stands for red, green, and blue for color images. Each of these R, G, and B have an integer value assigned to it from 0 to 255. This is how you'd represent a pixel that is red in color. R value will be equal to 255; G and B will be equal to 0. A green image on the other hand will have the G value set to 255; R and B will be equal to 0. A blue pixel will be represented by this tuple once again. R and G will be 0; B will be 255. So we have three values to represent color. These are the three channels of a colored image. Colored images are three-channel images. Images can also be represented using

grayscale. Here you need just one value to represent a particular pixel in this image, and that value represents the intensity of that pixel. Every pixel's intensity information can be represented by a number between 0 and 1. For example, a grayscale image will be a matrix where every pixel can be represented by one value, a single-channel image, the 1 value is to represent the intensity. So the two basic kinds of images that can represented by arrays are single-channel and multi-channel images. Given this information you should be able to figure out that images can now be represented by a three-dimensional matrix. The first two dimensions are obvious. They represent the height and width of the image. Along the third dimensions are the numbers used to represent the number of channels in an image, for a colored image the third dimension will have shape 3 and for a black and white or a grayscale image the third dimension will be 1. So if you have a 6 x 6 grayscale image the shape of your image matrix will be 6, 6, 1 where 1 represents the pixel intensity. This is a single-channel image. In the case of a 6 x 6 colored image, the shape of the matrix to represent that colored image will be 6, 6, and 3. Three refers to the number of channels in a colored image, RGB values to represent a single pixel. Now that we've understood how images are represented we can move on to our demo of image manipulation. For this demo along with NumPy we'll also use the SciPy library. Remember, the SciPy library is the Python-based open source ecosystem for all kinds of mathematical and scientific software. Let's get a random image using the misc module in SciPy. This gets a colored image of a raccoon face. Let's check out the shape of this image. Notice that this is a 768 x 1024 image. These dimensions refer to the height and width of the image in pixels and the third dimension is the number of channels in this colored image equal to 3. This is an image of a raccoon. We'll see that in a little bit. This is represented using a NumPy array. If you simply print out the contents of this array, you will see the RGB values which make up this color image. If we want to view this image and convince ourselves that it's indeed an image of a raccoon, we'll need to install visualization software. We'll use matplotlib. The pyplot. imshow function will display this image and there you can see our raccoon. Now this image is a NumPy array, which means we can perform slicing operations on it to select only a portion of the image. Let's print out this portion of the image and there you can see our slice of the image array. You can also split this image array into two equal-sized sections. Let's display the section. Each section is an image of its own. This is the upper section and this is the lower section. The np. split function performed a vertical split by default. We can also perform a horizontal split by specifying the axis input argument. Axis=1 will split the image horizontally. We have split this image into two sections, b1 and b2. Here is b1 and this contains the left portion of the image and b2 contains the right portion of the image. If you want put together two NumPy arrays you can use the concatenate function. The variables a1 and a2 here contain our vertically split images. When we concatenate them we get the original image of the raccoon back. Let's try

concatenate with our horizontally split images, which are stored in variables b1 and b2. Notice the result. Notice that concatenate stacks the arrays vertically one on top of the other. Concatenate puts together two arrays, column-wise or along the columns by default. We can change this by specifying an axis to the concatenate function. Axis=1 will concatenate b1 and b2 row-wise giving back our original raccoon image.

Shallow Copies Using View

Once you have a NumPy array you can create a view over it. You can think of the view as a shallow copy of the underlying array. In the case of shallow copies, any edits that you make to the copy will also reflect in the original array. Let's set up an array of fruits. This is a 1b array and let's set up two baskets, which are views into the fruits array. The view function on a NumPy array creates shallow copies. Basket 1 and basket 2 are shallow copies of the original fruits array. The contents of the shallow copies may be the same as the original array; however, shallow copies are unique objects though and this can be very clearly seen when you apply the id function to the shallow copies as well as to the original array. All the IDs are different, implying that these are all different objects. Here is another test. Basket_1 is fruits will be false because basket 1 is a view of fruits; it's not the fruit object itself, but basket_1. base is fruits will return true because the base object from which basket 1 has been derived is the fruits array. Let's modify one of our baskets, that is basket_2 to set the first element to strawberry. It used to be apple. Now when we print out the values of basket_2 you can see that the first element has been updated. A modification to basket_2 will also affect the fruits array and if you print them both out you can see that the underlying contents have been updated. The first elements of the fruits array as well as the shallow copy basket_2 are now the same and because basket_1 is also a shallow copy of fruits that has also been updated. Strawberry is now the first element for all of these arrays. Let's say that you now create an entirely new NumPy array and assign it to the basket_1 variable. This array contains the fruits peach, pineapple, banana, and orange. You'll find that this is an entirely new object. It's not a view of an existing array, which means that the contents of basket_1 are completely different from the contents of the original fruits array. The variable basket_2 still contains a shallow copy of the original fruits array. We'll now reshape basket_2 to be a two-dimensional array, a 2 x 2 array. This we do by setting the value for the shape parameter. When we print basket_2 you'll find that the shape has changed. The interesting thing here is that the shape of the original fruits array hasn't been affected. You can reshape a view or a shallow copy without affecting the shape of the original array.

Deep Copies Using Copy

Sometimes it happens that you have to make deep copies of existing arrays. This is not always efficient, especially if the original array is huge with many thousands of elements, but this is something that you have to do so let's learn how to do it. We'll work with the same fruits array as before, but this time we create a basket, which is a copy of the fruits array by calling fruits. copy. The basket variable now contains a deep copy of the original fruits array. This is not a view or a shallow copy. Basket is a completely different object from fruits. Basket is fruits will give you false. Basket. base is fruits will also give you false. That means basket hasn't been derived from fruits. In fact, it's a completely different copy, a deep copy. Let's change the first element of our basket. We'll set it to strawberry and if you print out the basket array you'll see that the first element has indeed been updated. If you go back and check out the original array of which we made the copy, the fruits array, you'll find that it remains unchanged. Basket is now a completely different array. If you change the shape of the basket array the shape of the original fruits array will remain unchanged as well. And with this we come to the very end of this module where we introduced multidimensional arrays using NumPy and explored how we can work with them. NumPy is a very important scientific computing package in the Python ecosystem and the n-dimensional array that it offers forms the basic building blocks of many other packages and libraries. NumPy allows you to perform a wide variety of operations on multidimensional arrays using very simple and intuitive operators. Some of what we covered in this module included basic operations on arrays, universal functions, reshaping, splitting of arrays, image manipulation, and making shallow and deep copies. In the next module we'll focus on indexing operations and you'll see the wide variety of complex indexing that is possible with NumPy.

Complex Indexing Using NumPy

Module Overview

Hi, and welcome to this module on Complex Indexing Using NumPy. Indexing involves accessing specific elements within an array and NumPy gives you a wide variety of ways in which these elements can be accessed. NumPy has this really cool feature where you can access specific elements by specifying Boolean conditions. You specify the condition, it will generate a Boolean array, and this Boolean array can be used to index into another array. We tend to use NumPy

when we are working with numeric data; however, NumPy arrays can also be used to store structured data. This can be thought of as a precursor to DataFrames as in Pandas. A very powerful feature that NumPy offers is the idea of broadcasting. Broadcasting allows you to work with scalars and arrays and with mismatched arrays provided they match certain broadcasting rules.

Indexing Arrays Using Arrays

In NumPy you can use an array of integer elements to index into another NumPy array. In this demo we'll see how. Import the NumPy module as we've done before; also import the csv module. We'll be reading from a csv file in this demo as well. Set up an array using arange which holds the squares of all numbers from 0 to 11. This is the array that we'll work with. We've already seen how we can use square brackets and one index to access specific elements within this array. Here are the elements at index 2, 6, and 8. NumPy also allows you to use an array of indices to index into another array. Here we have an array which contains the index 2, 6, 8 and we can use indx_1 to index into array A. Notice that within square brackets we've specified an array of indices rather than individual index values. The result is an array with those elements which are present at indices 2, 6, and 8 from the original array A. Let's create a two-dimensional array of indices. This is indx_2. This contains a 2 x 2 and each of these array element values are indices that we're going to use to index into our original array A. Note here that the array A is a 1-D array and we're using an index array within square brackets. That index array is a 2-D array. Interestingly enough, this is completely valid in NumPy. The shape of the resulting array is the shape of the index array that we specified within square brackets. The elements of this result array are the elements which are present at index 2, 4, 8 and 10 of the original array A. There are different interesting and useful ways of indexing NumPy data. Set up an array of food items. This is a 2-D array 3 x 4. The food array contains three rows and four columns. We'll now initialize two separate arrays here, row and column. Both of these are 2 x 2 arrays, two-dimensional arrays with two rows and two columns each. NumPy allows you to use both of these arrays in conjunction to index into the original food array that we set up. Let's understand the result here. The image that you see on the right-hand side is the original 3 x 4 food array and the result array is available on the left, highlighted in red. The elements in the result were chosen by taking the index value from the row array and the column array element-wise, one element at a time. The first element in the result is Blueberry at index 0, 0, 0 row and 0 column, and the last element in our result array is from row 2 and column 3. This is the element Clove. A couple of important things to note here. The row and the column arrays are both of the same shape, 2 x 2. When we use row and column arrays to index into our

food array, the resultant array is of the same shape as row and column, that is 2 x 2. If you want to access a specific element of a multidimensional array you'll specify two indices within square brackets. Here we access the element at row 2, column 0, which is Mustard. You can use array indexing with the assignment operator as well. Here we've used the row and column array to index into the food array and assign those elements to all zeros. If you now display the food array you'll see that the elements at the four corners of this food array have now been set to 0 with a single assignment statement. We've updated multiple elements by using array indexing. Let's perform this same operation using the original array A that we'd set up. We had indx_1 which accessed the elements at index 2, 6, and 8. Now let's assign all of the elements at these indices to 9, 9, 9. This is something that is possible to do with a single assignment statement using array indexing. The elements at indices 2, 6, and 8 are now equal to 9, 9, 9.

Fancy Indexing with GDP Data

You'll see that these fancy indexing operations that we learned about in the last clip are often used in NumPy. Let's work on another example this time with GDP data. Along with NumPy we'll also use the Pandas library. The Pandas library represents data in the form of DataFrames, a tabular structure with rows and columns. We'll also use matplotlib for visualization. We'll use Pandas to read in the csv files in the data directory gdp_pc. csv. We're interested only in values for 2016 so that's the column that we access from our GDP data. The. values function returns the GDP data for 2016 in the form of a NumPy array. The shape of this array will give us the number of countries for which we have the GDP information. This is GDP per capita. We have it for a total of 264 countries. When the second dimension of the shape parameter is missing, this is a 1-D array. Let's visualize the per-capita GDP information using matplotlib. Notice how the plot function simply takes in a NumPy array and displays a graph. Let's use the np. median command to find the median GDP across these 264 countries and the result you see is NaN or not a number. This means that NumPy was unable to calculate the median and the reason for this is because our GDP data contains missing values. When you print out the entire array you'll see that several entries are NaNs. Let's see we want our gdp_16 NumPy array to contain data only for those countries where the GDP value is not missing. You can do this using fancy indexing. Within the square brackets of our GDP array we've specified a condition. We only want those elements where the element is not NaN, where the element is a valid number. This will automatically filter out the invalid or missing elements from our NumPy array and you can see the result here. All the NaNs have been filtered out and the shape of the resultant array will give us the number of entries in this array which is now 229. It has fallen from 264 to 229 when we removed the NaNs.

And now if you try to calculate the median GDP value we'll get a meaningful result. This is the median GDP per capita across 229 countries. Once the NaNs have been removed from our NumPy array we can calculate mean as well as perform other arithmetic operations. NumPy has a special method called count_nonzero, which allows us to count the number of elements that match a particular condition. Here we want to count those elements, only those countries where the GDP per capita is greater than 40, 000. The syntax here is interesting. We index into the gdp_16 array. We only include those elements which satisfy our condition of gdp_16 greater-than 40, 000. There are 32 such countries in our list. Another useful operation on arrays is to sort them in order. The np. sort function creates a sorted array and in the sorted result we only want to display the first n elements which we specify using a slicing operation within square brackets. Np. sort by default sorts out array elements in ascending order. Here are the countries with the lowest GDP, the lowest 10. We can slice our sorted result a little differently. Here are the countries with the highest GDP. The sort has been in ascending order and we want to see the countries towards the end of the sorted result, those with the highest per capita GDP values. Conditional indexing is really useful because you can now slice the data based on the values within a particular column of an array. The result array here will give you all countries which have GDP values greater than 657. 06. Once again you can use the np. count_nonzero to count the number of records that match this condition, which is 152 in our case. We can flip the condition here to count the remaining records. You can see that the result here is 77; 77 + 152 is equal to 229, which is the total number of records that we have available. The count_nonzero function can be used with more complex logical operations as well. Here we've specified two conditions using the ampersand or the and operator. A GDP value will be counted only if it matches both of these conditions. If it's greater than 6570 and less than about 16609.

Indexing with Boolean Arrays

We saw a real-world example of how fancy indexing works. Let's break this down. Fancy indexing is actually indexing with Boolean arrays. Let's create a simple 2-D array. This contains four rows and four columns and its elements are the numbers from 0 through 15. Let's check for a specific condition within this array. We want all elements of a which are greater than 9 and we store the result in index Boolean. Now when we print this out you'll see that the result is an array of Booleans. Within this Boolean array result we have the value true for all elements of a which have a value greater than 9. The other elements of this Boolean array are set to false. The result of a conditional evaluation on a NumPy array is an array of Booleans with the same shape as the original array on which we applied the condition. In NumPy we can now use this Boolean array to

index into the original array. When you use a Boolean array within square brackets the result will give you all the elements of the original array for which the corresponding Boolean condition was true. This is what we shortened into one statement when we used fancy indexing. From the array A we want to select all those elements for the condition a greater than 9 is satisfied. The result is the same as we've seen before. Let's generate another Boolean array, this time with the condition that the element within the array should be less than 6 and here is the result. Now typically in programming languages a Boolean value of true is considered nonzero. We can use np. count_nonzero to see how many true values exist in our Boolean array. True values are those which match the condition that the element is less than 6. You can use np. sum to count the number of elements that match a specific condition as well. Here we'll sum up the number of elements that match the condition that the element value is less than six. The answer here is clearly six just like before. We can get a count of elements in a row-wise manner which satisfy the condition. This we do by specifying an additional axis parameter to np. sum. The condition is that the element should be less than six and the axis is equal to one, indicating that the sum should be row-wise. Let's parse the result here. The first element is four, indicating that the count of elements which satisfy the condition along the first row were all four elements, there were two elements which satisfied the condition a less than six, along the second row. In the third and the fourth row there are no elements which satisfy the condition that the element is less than six. The np. any function can be used to see whether there is any element within the array that satisfies the condition. This function will return true if there are any elements in the array A whose value is greater than eight. Yes, there are, which is why it's true. Np. all will return true if all elements match the condition that has been specified, are all elements in the array A less than 10? Clearly not in our array, which is why the result is false. Are all elements in array A less than 100? Yes, indeed. Our elements range from 0 to 15; they're all less than 100, which is why the result is true here. You can consider elements in a row-wise or column-wise manner as well using np. any or np. all if you specify the axis parameter. Axis=1 refers to row-wise computation. We want to know whether all elements in each of the rows are less than nine. You can see from the result that in the case of the first two rows the answer is true. In the case of the third and the fourth rows the answer is false. In the third row only the element eight is less than nine; the remaining elements are greater than nine. All elements in the fourth row are greater than nine.

Arrays with Structured Data

Generally you'll use NumPy to work with very large numeric arrays because that is what is NumPy's strength, but you can also use NumPy with structured data. Let's say we have three

separate arrays; name, studentid, and score, which contain information about four students; Alice, Beth, Cathy, and Dorothy. We can store all of this in one structure array called student_data. We'll initially initialize this array to zeros. The first parameter here is the shape of the array. The four here indicates a 1-D array with four records. The interesting thing here is in how we specify the data type of this structured array. The structure is names and within names we have the name, studentid, and the score columns. You can also specify formats individually for each column of this data. The name column is a string. The studentid column is of type integer and the score if of type of floating point. And here is our student_data array initialized to zeros. There are a couple of interesting things to note here. The data type is specified on a per-column basis because that's how we had set it up. NumPy knows that the name column is a string so when we use np zeros to initialize a string column it will simply set it to an np string. In a structured NumPy array we can assign values to the matrices by specifying the name of the column. Notice how we access the individual columns by name in student_data array. Once you've assigned values for all the individual columns you can print out student_data and you can see that the student_data tabular structure is as you would expect. Individual columns can be accessed by specifying the column names. This will give you all the names of the students. You can access the student id by specifying studentId within square brackets, and you can access the score by specifying score within square brackets. If you're familiar with the Pandas library in Python you'll see that the NumPy array here functions very much like a Pandas DataFrame. This NumPy array containing structured data can be accessed using indices as well. Here we access index 1, which is the second row in our array. You can also combine row indices along with column names. Here we are interested in the row -1. That is the last row, but only in the name column. This is Dorothy. Structured data can be indexed using Boolean arrays as well, specified using conditions. We want to find all students who have a score value of greater than 85 and we want to access their names. This returns the names of Alice, Beth, and Cathy who satisfy this condition.

Broadcasting

One of the most powerful features that NumPy offers is this idea of broadcasting. Broadcasting is what allows NumPy to work with arrays of mismatched shapes. In order to understand what broadcasting really means and to see the rules that it has to operate under, let's first start off with the definition of broadcasting. The definition that you see on screen here is quite a mouthful, but it's pretty clear that broadcasting refers to how NumPy treats arrays with different shapes when you perform operations on them. Now you may not realize it, but we've already used broadcasting in the NumPy operations that we've seen so far. What we've seen so far are

broadcasting scalars. You can always broadcast scalars to work with NumPy arrays. Broadcasting is best understood visually; let's see what it means to broadcast scalars to work with arrays. Let's consider a very simple array here, an array which contains six elements which are all 1s and you want to multiply this array with the scalar with the value 10. Now this is an operation that we've already performed early on when we used NumPy. We saw that every element of the array will be multiplied by the scalar and this is what is broadcasting. Now what exactly happens here is that the scalar 10 is replicated to match the shape of the array, to match the six elements that are present in this array. Once the scalar has been replicated so that the dimensions match the original array we can now perform element-wise multiplication to get the result, which is an array of 10s, six elements all equal to 10. NumPy has broadcast the smaller array, which in this case is the scalar to match the dimensions of the larger array to perform this arithmetic operation. Broadcasting works with multidimensional arrays and scalars as well. Here we have a 3 x 6 array represented by the variable arr and a scalar 10. This multiplication operation will first replicate the scalar s along the columns and then the rows so that the array shapes match and element-wise multiplication can be performed. Now that we've understood broadcasting, specifically when we use arrays and scalars, we can go back to the broadcasting definition. This describes how NumPy treats arrays with different shapes during arithmetic operations and these arithmetic operations are performed element-wise. To allow these element-by-element operations, the smaller array has to be broadcast to match the shape of the larger array. When we use scalars, broadcasting seems fairly straightforward because it's very easy to replicate the scalar to match the dimension of the array that it operates with; however, broadcasting also works with two arrays subject to certain constraints. We've already seen how broadcasting with scalars works. Let's now look at broadcasting with arrays. You can only broadcast the smaller array over the larger one if the shapes of the two arrays match exactly or are compatible in some ways. Let's see what this compatibility means. Here are two arrays and we want to perform some arithmetic operation between these two arrays. This arithmetic operation we want to perform in an element-wise manner. The first array has a shape of 3 x 6, the second array has a shape of 3 x 1. Let's start off by looking at these dimensions in the reverse order. The last dimension of the smaller array is 1 and the remaining dimensions of the smaller array perfectly match the larger array, which means the smaller array can be broadcast over the large one for arithmetic operations. This we do by replicating the last dimension of the array to match the larger array. Once the smaller array has been replicated, the shapes of the two arrays perfectly match each other, you can now perform element-wise arithmetic operations on these two arrays. Let's consider another example here. Here we have a 3 x 6 array and a 3 x 2 array. Now notice the last dimension of this array. This is not equal to one. In fact, it's equal to two. The smaller array shape is different from the larger

array and its last dimension is not equal to one, which means the smaller array cannot be broadcast over the larger array. Element-wise arithmetic operations cannot be performed with these two arrays. We'll get a broadcast error. Let's consider another example here. We have the original 3 x 6 array, a second array is 2 x 1. The last dimension of the second array is equal to one; however, notice that the remaining dimensions do not match. Three by six versus two by one, which means the smaller array cannot be broadcast over the larger array. Element-wise arithmetic operations between these two arrays will not work. There will be a broadcast error. It's pretty clear from our broadcasting definition here that the shapes of the two arrays have to be compared element-wise and they are subject to certain constraints before broadcasting is possible and the dimensions of the two arrays in bold are compared in reverse starting with the trailing dimensions and working forward. If broadcasting rules are met, that is all the remaining dimensions are exactly the same and the trailing dimension of the smaller array is equal to 1, then the smaller array can be stretched to be equal to the shape and dimensions of the larger array and arithmetic operations can be performed element-wise on them. One thing to note when you use NumPy broadcasting is the fact that no actual copies of the smaller array are made, which is why broadcasting is very computationally efficient as well as memory efficient. Broadcasting rules state that the two arrays need to be of compatible shapes and compatible means that all dimensions of the two arrays need to match or one of the two dimensions need to be equal to one, so that array can be stretched to match the shape of the larger array. In short, broadcasting is a very powerful feature of NumPy which allows arrays of different shapes to be combined. It is memory efficient because actual copies of the smaller array are not made. It's also computationally efficient because the looping operations under the hood are in C rather than Python. We've seen that when we perform operations that are between an array and a scalar, scalars are very easy to broadcast. They're just replicated along all axes of the larger array. You'll never get an error when you broadcast scalars and scalar broadcasting is independent of the size and shape of the other array. Where you might run into trouble and have to watch out is when you broadcast one array over another. The shapes of the two arrays need to be compared and have to be compatible. Start from the trailing dimensions and check to see whether the shapes match. Every dimension needs to match or one of the dimensions need to be one.

Broadcasting Scalars and Arrays

In this demo we'll see some of the broadcasting rules that we studied in the last clip in action. We'll initialize two arrays that we'll work with. Each of these arrays have five elements. These are a and b. Both of these arrays have identical shapes, which means element-wise operations such as

multiplication can be performed on these arrays. Let's initialize a scalar 10 which is assigned to the variable c. Broadcasting always works when you perform arithmetic operations that involve an array and a scalar. Scalar dimensions can be thought of as one, which means that scalar can be replicated to match the shape of the array so that this multiplication can be carried out. Arithmetic operations which involve a scalar and multidimensional arrays work as well. Here is a 4 x 3 array of all ones. This can be multiplied by c to get an array of all 10s. This is a 4 x 3 array as well. Let's see how broadcasting works with arrays. Here we have two lists of heights and weights. Heights are expressed in centimeters and weights are expressed in kilograms. We convert the heights and weights of these six individuals into a single NumPy array. This is a 2 x 6 array. Think of these as student details that are stored in the student_bio NumPy array. The first row is the heights of these students in centimeters and the second row is the weights of these students in kilograms. Let's say we now want to express these students' heights in feet and we want to express their weight in pounds. In that case we need to multiply every element of this array by the factor that it convert centimeters to feet and kilograms to pounds. Factor_1 is the 1_D array or the vector that has the elements to perform this conversion. The shape of a 1-D array as we've seen before, is 2, which is the number of elements in the 1-D array, nothing. And the shape of the student_bio array is 2, 6. Now these arrays are not compatible for broadcasting so when we try to multiply the student_bio by factor_1 you'll get an error. The error very clearly says that these operands could not be broadcast together. Look at the last dimension of both of these arrays. For the first array it's six and for the second array it's nothing. The last dimension for one of these arrays has to be equal to one for these arrays to be broadcast together with no error. Let's change the shape of the array that is used to represent the multiplication factor that we convert centimeters to feet and kilograms to pounds. Factor_2 is now a two-dimensional array, not a 1-D array, which means its shape is 2, 1. Now when we try to multiply factor_2 with student_bio broadcasting works. You'll see that this gives us a valid result. This result contains all the students' heights expressed in feet and their weight expressed in pounds.

Automatic Reshaping

We've done a bunch of array reshaping already using the reshape function. Let's take a look at automatic reshaping in NumPy arrays. We'll use arange to create an array with 30 elements ranging from 0 to 29. We can reshape NumPy arrays by assigning values to the shape property of an array. Notice that the shape we want for this array is 2-1 3. Minus 1 indicates that we don't know the dimension of that particular axis when we reshape it, fitted based on the underlying data. Here you can see the resultant array shape. This is a 2 x 5 x 3 array. The value 5 has been

automatically assigned by NumPy based on how many elements existed in the array that we reshaped. Let's reshape the same array once again, but this time we specify that the third dimensions is unknown. The first two dimensions are two and three, and here is the resultant array. The underlying array has 30 elements, two multiplied by three, multiplied by five gives us 30. The unknown dimension is five and this has been automatically assigned by NumPy during the reshape.

Stacking Arrays

NumPy also allows us to stack arrays either vertically or horizontally. Let's see some of the functions that we can use for array stacking. We'll first initialize two arrays here, x and y. Both of these are 2 x 2 arrays. You can print out the shapes and the array contents. You can see Germany and Berlin, France and Paris are in the same array. Hungary, Budapest, Austria, and Vienna are in the second array. One of the functions that NumPy offers to join a sequence of arrays along an existing axis is the concatenate. We've used concatenate earlier and by default concatenate stacks the arrays column-wise. One array is stacked on top of another. By default the axis parameter to concatenate function is equal to 0, which is column-wise stacking of arrays. If you want the concatenation to occur row-wise or horizontally, you'll specify axis is equal to one. This will result in the two arrays being stacked one beside the other. The second array is stacked right next to the first one. NumPy provides a bunch of other convenient functions that call np. concatenate under the hood to allow you to join arrays such as the np. stack. This function can join a sequence of arrays along a new axis and by default it will be a column-wise stack, one array on top of another. You can pass in more than two arrays to np. stack. Here are some arrays that we've encountered before for studentId, name, and scores and you can stack all of these together to get information for one student in one column. Column-wise stacking gives us a 3 x 4 array. If you want to stack along a different axis you simply specify the axis parameter to np. stack. This is row-wise stacking of the arrays. The row-wise stacking of the same three arrays results in a 4 x 3 array. If you find using the axis parameter confusing you can use the convenient functions vstack and hstack. Vstack will vertically stack the arrays along axes to 0, and hstack will horizontally stack the array along the rows. The arrays will be stacked next to one another. Vstack and hstack might be more intuitively used and more readable when you're writing NumPy code.

Histograms

NumPy can also be used to plot histograms on your data. Histograms allow you to view distributions of your numeric data and are a great tool for analysis. In this demo we'll use the NumPy library to bin our data, to place our data in the right categories and we'll use matplotlib to visualize the distribution. The np. histogram function is what we'll use to compute a histogram over an array of data. The first argument to np. histogram is the data over which we want the histogram to be computed. That is our actual data points. The bins parameter can be used to explicitly specify our histogram bins. Here the bins are 0, 1, 2, and 3, and the result will show you the distribution of our data points in the individual bins. There are 0 instances in bin 0, two instances in bin 1, and one instance in bin 2. If you observe our original data there are 0 data points with value 0, two data points with value 1, and one data point with value 2. Histograms are more interesting when they are viewed over larger data sets. Let's create a random normal distribution in NumPy using the np. random. normal function. This is a normal distribution with a mean of two and a standard deviation of 0. 5 and the resulting NumPy array assigned to data will have 10, 000 points. Let's create a histogram over this data using np. histogram. We want 50 bins in our histogram. The histogram function as we've seen before returns two values. The first is n, which gives the occurrences of data that fall within each bin. That is the frequency of data within each bin. The bin_edges here refers to the values on the x-axis of our histogram, which marks the boundaries of each bin. There is a bin edge corresponding to each of the 50 bins in our histogram. We can now use matplotlib to view our histogram. We plot the bin_edges on the x-axis and the actual histogram n on the y-axis. And the result you can see is our normal distribution of a mean of two and a standard deviation of 0. 5.

Miscellaneous Operations

NumPy offers a bunch of array-related functions which are very useful when you're working with machine learning models. These are very specific to how machine learning requires data to be formatted how it outputs predictions and so on. Let's take a look at some of these functions here. We'll work with salary data that is present in the salary. csv file. The delimiter is comma and we can read in this file using np. genfromtxt. Salaries is a 1-D NumPy array and it contains a total of 1147 salary records. NumPy offers a function called the argmax. This will return the indices of the maximum values along a particular axis of a NumPy array and this is the function that we'll use in our machine learning model that we'll see in the next module. Here the return value is a scalar, a single number, because the highest salary occurred at index 246. If you check out the salary at index 246 you'll see that it is $850, 000. This is the highest salary in our 1-D array. Exactly like the argmax, NumPy also has the argmin function, which returns the indices of minimum values along

a particular axis. Because the input to argmin is a 1-D array there is just one axis that it has to work with. The minimum salary in our array is at index 282 and we can view this and it is around $11, 400. The argsort function returns the indices of sorted values in this array, not the actual sorted values themselves, but an array of indices. If you take a look at the resultant array of sorted indices you can see that 282 has the lowest salary value and 246 has the highest salary value. You can also specify the sorting algorithm that you want used when you specify the argsort. Here we've specified kind is equal to mergesort and that is our sorting algorithm. We can use array indexing to display our salaries in sorted order. Np. argsort will return the array indices and we can use this result to get the salaries in sorted order. Another useful NumPy function is the np. where. Now if you want to find the indices of all salaries which satisfy a certain condition and here the condition is that the salary should be greater than 100, 000, you can use the np. where. The result for np. where is an array of indices. All of the indices represent salaries which are over $100, 000. In our salary data of 1147 records we have 72 salaries which are over 100K. Once again, you can use array indexing to find the actual salaries which are over 100K and here is the result. You can specify more complicated conditions in order to extract specific elements from your NumPy array. Here we want to find all salaries that are above the average. The average will be calculated by np. mean. Once we have the condition we can use np. extract to extract those elements from the salaries array which have salaries greater than the average value. And on this note we come to the very end of this module. In this module we covered complex indexing operations to access specific elements within a NumPy array. We saw that using conditions generated a Boolean array and this Boolean array could be used to index into the original array. This is what allows us to find those elements in the original array that satisfy our conditions. We saw that NumPy arrays, even though they're typically used with numeric values can also work with structured data. These arrays are the precursor to DataFrames which are used in Pandas. We also studied broadcasting rules in NumPy, which is a very powerful feature which allows NumPy to work with mismatched arrays or with an array and a scalar. In the next module we'll see how NumPy works with other Python libraries such as SciPy, Pandas, and TensorFlow. In fact, we'll build a real machine-learning model using TensorFlow that utilizes NumPy computations.

Leveraging Other Python Libraries with NumPy

Module Overview

Hi and welcome to this module on Leveraging Other Python Libraries with NumPy. NumPy is closely integrated with many other Python libraries used for scientific computing and data analysis such as SciPy and Pandas. We'll see an example of how we can use NumPy arrays to work with the interpolation function in the SciPy library. We'll then use NumPy functions along with the Pandas library to analyze the Titanic dataset.

Working with Pandas

In this demo we'll see how we can use a NumPy array computation along with Pandas DataFrames. Pandas is another open source Python library which provides very efficient data analysis tools. Pandas represents all data in a tabular format in rows and columns, which is a very useful and intuitive way to look at data. If you don't already have Pandas installed on your machine you can get the latest version using a simple pip install. Import the NumPy and Pandas libraries and we'll use Pandas in order to read in the Titanic dataset. We'll use the pd. read_csv function. The Titanic dataset is an extremely popular one, which you're sure to encounter when you get started with machine learning. The Titanic dataset contains all of this information about the passengers on the Titanic; a passenger ID, whether they survived or not, their name, age, sex, their ticket number, the fare they paid, the cabin class, and so on. Notice that the data in the Pandas DataFrame is in tabular format in rows and columns. When we're exploring and analyzing this data let's say we're not interested in the precise cabin that the particular passenger occupied. We can simply drop that column from our DataFrame. The drop function drops the column; it's not present in the resultant DataFrame. Pandas has useful functions that allow you to check for missing values. Here we want to see how many of our records do not have values in the age field which we can do using the is_any function. You can see that we do not have age information for 177 records. The Titanic had passengers who had embarked at three different ports; the ports of Cherbourg, Southampton, and Queenstown. Let's say that we want to find aggregate values of all columns for each port. We first group by the Embarked column and then we call the aggregate function and pass in a NumPy function, the np. mean that allows you to find average values. The result is a DataFrame which has been grouped on the port at which the passenger embarked. C for Cherbourg, Q for Queenstown, S for Southampton. The average values are interesting when you look at the Survived column. Zero implies that the passenger did not survive and one implies that the passenger survived the sinking of the Titanic. You can see from the data here that 55% of the passengers who embarked at Cherbourg survived. You can also see the average fare paid by passengers for each port of embarkation. The average fare for Cherbourg is $59. It's pretty clear

here when you look at the average data that a higher survival rate is associated with higher fares. You can also select specific columns from the Pandas DataFrame to see the information that you're interested in. Here you can see a juxtaposition of the age and the fare paid, both averages. You can calculate the number of passengers for whom we have records using NumPy. We'll use the np. count_nonzero function and we'll count on the basis of the Passenger ID column. This gives us a total number of passengers in our dataset. In order to find the passengers who survived, we can use the count_nonzero function once again, but this time on the Survived column. Remember if Survived is equal to one, that means that the passenger survived the sinking. Getting the percentage of survivors from our dataset is now a simple mathematical computation. Printing out the result will show you that of a total of 891 passengers for whom we have information, three-hundred-and-forty-two survived. The percentage survived was 38%. Let's find the records for all the female passengers in our Pandas DataFrame. We simply find all those rows where sex is equal to female. Let's take a quick glance at the resulting DataFrame. Yes, all of these passengers are female. We can now use np. count_nonzero to find the number of female passengers who survived. Let's print it out to screen and you can see that a total of 233 out of 342 survived were female. You can perform the same series of operations to find the number of males who survived and you can see that these were just 109. In the stories of the Titanic that we've heard it's pretty well known that first-class passengers tended to have survived because they got on to the life boats first. Let's see if our data bears this out. We'll group based on the passenger class and count the number of survivors in each class. And here is the result. Let's see if our information was correct and you can see that it is indeed true. A large number of first-class passengers survived.

Working with SciPy

In the first example of this module we'll see how we can use NumPy with the SciPy scientific computing package. SciPy is open source and it contains scientific and numeric tools for Python. In this example we'll use the SciPy libraries, specifically the interpolate 1b function within the SciPy library. We'll use NumPy as matplotlib. For this demo we'll work with the Ages data that is present in the Ages. csv file. The delimiter is a comma. This file has a header which is why we skip the header when we read in the data using NumPy. Let's take a look at this ages NumPy array that we just read in. This has three columns and a number of different rows. The first column represents specific years. These are the years in which a census was conducted. It starts at 1890 and goes up to the year 2000. The next column represents the average age of males in the census data. Here are the averages across different years. And the third column contains the

average ages of females when the census was conducted. Let's slice the ages array to get two different arrays. The array x contains all the years for which the census was conducted and array y contains the average age of males for each of these years. Both x and y here are NumPy arrays and you can confirm this by checking the type of x and y. We'll now use this x and y data which is present in these NumPy arrays to perform some interpolation. We interpolate a 1-D function. We want to find the y values of x values that are not actually present. Let's say we want to find the average age of males for the year 1946. This is not explicitly present in our data, but it can be found using interpolation. We'll perform interpolation to find the average age of males for all of these years; 1896, 1907, 1912, and so on. The data for these years are not explicitly present in our dataset, but can be found using interpolation. We've got the interpolation function f; we can now pass in xnew to find the corresponding ynew. The NumPy array ynew contains our interpolation result. Here are the average ages of males for those specific years. Let's now use matplotlib to visualize our interpolation results. Here we are going to plot the values from our original dataset, which will be represented by the dashes and the values from our interpolated result, which will be represented using circles and here is our matplotlib visualization, the original data and our interpolated results. Let's total the average age of females for all of these years in the variable zed. Zed is a NumPy array as well and we set up a new interpolation. This is x with zed. Now we can find the average age of females for all of those years which are not present in the original dataset. And here is the result in the form of a NumPy array. Once again, we can visualize this using matplotlib. The dashes are the original dataset; the circles are the interpolated results. This is just one example of a library function from the SciPy package, but as you can see, SciPy and NumPy integrate seamlessly.

Vectorization

If you have a function that works on scalar data NumPy allows you to vectorize these functions very easily. This is useful, which is why it's important that you understand how it works. We'll import the NumPy library and set up a simple function which either adds or subtracts two scalars. If a is greater than b, we'll return a-b, otherwise we return a + b from this function. If you pass in the input arguments 5 and 7 this will return the sum as a is less than b and you can see the result is 12. Vectorizing this function will allow us to use this function on arrays, not just with scalars, and this we can do very easily by simply calling np. vectorize and passing in the function name. We can now call this vectorize function by passing in two lists instead of two scalars. This vectorize function will apply the original addsubtract function to individual elements of both of the arrays that we've passed it and we'll get an array as the result. The elements of this array have been

gotten by adding or subtracting the corresponding elements in the input arrays. If the shapes of the input arrays to this vectorized functions are not compatible, then we'll get an error which says that the operands could not be broadcast together.

Summary and Further Study

And with this we come to the very end of this module and to the end of this course. Here we saw how NumPy is closely tied with other Python packages such as SciPy and Pandas and we worked on two specific examples. We saw how we can use the interpolation function in SciPy along with NumPy arrays and we also worked on the Titanic dataset where we used NumPy functions along with Pandas DataFrames. If you're interested in data analysis with Python libraries, here are some other courses on Pluralsight that might help you. Pandas Fundamentals is an extremely popular class on the Pandas library. If it's machine learning that you're interested in, here is how you can get started; How to Think About Machine Learning Algorithms, is a beginners' course in machine learning using Python, or if you're interested in deep learning frameworks such as TensorFlow, Understanding the Foundations of TensorFlow on Pluralsight will help you.

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

Level	Beginner
Rating	★★★★½ (36)
My rating	★★★★★
Duration	1h 43m
Released	22 Jun 2018

Share course