Module 4
Video Transcripts

Video 1: Packages (3:08)

In this video, we'll learn how to install packages in that specific tasks. A package is a directory of Python scripts, that includes functions, methods and types. Think of these as collections of functions, methods and types you can install together. These are useful because they have a lot of pre-written code, so you don't need to write all your own code from scratch. Two packages that we'll learn about in this module are NumPy and Pandas. So, how do you install a package?

First, you need to download a package maintenance system called 'pip' from the sewer route. Then, open your terminal and type 'python3 get-pip.py'. Now, you're ready to install a package. To do that, type 'pip3 install' followed by the package name. For example, you can type 'pip3 install numpy' or 'pip3 install pandas'. Now that your package is installed, you need to import it before you can use it. To do so, simply write 'import' followed by the package name. Now we can use functions for these packages.

For example, the NumPy package has the function sign which returns one if an argument is positive, negative one if the argument is negative, and zero if the argument is zero. Let's try using the function 'sign()' on '5'. We get an error. That's because any time you want to use a function from a package, you need to refer to the package like this. Writing out the package name each time can get tedious, so a common thing to do is to alias a package when installing it; We're writing by writing 'import' followed by the 'package_name', followed by 'as', followed by the 'alias', you want to give your package.

The commonplace nickname for 'numpy' is 'np'. Now, you can call the package by the alias, so to use the 'sign()' function, you can write 'np.sign()'. Instead of installing the entire package, if you want to just install a specific function, you can do that as well. You can type '#from package_name import function_name'. So, 'from numpy import sign'. Now, the function, but not the rest of the functions from that package are imported and ready to use. For this example, we imported 'sign()' from numpy, but you can do this with any function from any package.

****

Video 2: Getting Started with NumPy Arrays (3:39)

Let's get started with an important package for data science: NumPy. NumPy is a Python package that stands for Numeric Python. Probably the most useful tool in this package is a new data type called an array, which we'll explore in this video. It also includes some sophisticated functions, and linear algebra, and random number capabilities. For this video, let's focus on the extremely useful data type included in NumPy, the NumPy array.

To understand the benefits of a NumPy array, you need to first understand the limitations of a Python list. Let's say we're working on an experiment where we need to find the densities of multiple materials. We calculate the masses, and the volumes of these materials, and we record this data in two separate lists. To find the density, we need to divide the mass by the volume. In Python, we can divide with the forward slash.

When we divide the 'mass' list by the 'volume' list, we get an error. Unfortunately, Python can't do calculations on the built-in data type list. To find the densities of these objects, you would have to divide each mass by each volume in a separate calculation. Here we only have three materials, but imagine if you had a thousand. Even just 10 materials would take way too long. This is where the NumPy array comes in handy.

It works almost exactly like a list, with a few exceptions, one of which is that you can perform calculations across arrays. Let's look at an example. To create a NumPy array, we first need to import NumPy. We'll use the common alias np. NumPy comes with a function called array that will turn its input into an array data type. So, let's call the function array on mass and volume. Don't forget to refer to NumPy to use the function.

Now that the masses and volumes are in arrays, not lists, we can perform calculations on them. To find the density, we can divide the mass by the volume like this. Let's look at the contents of density. See how easy that was? A few final notes on NumPy arrays before we conclude: NumPy arrays can only contain one data type.

If you're converting a list with multiple data types into an array, the NumPy array function will automatically turn all the elements into a single data type. Also note that you can index, subset, and slice NumPy arrays the same way you can with a list. You can also use comparison operators the same way you've already seen. Once you're comfortable with simple arrays like the ones we saw in this video, you're ready to learn about more complex NumPy arrays with two or more dimensions.

****

Video 3: Getting Started with 2d NumPy Arrays (3:34)

Let's look at how NumPy arrays can be extended. Rather than creating a NumPy array with one dimension, you can create NumPy arrays that have multiple dimensions. Let's look at the syntax for creating a two dimensional NumPy array, and then retrieving the data you want from it. Here, we have mass and volume lists. You can create a 2D NumPy array from a list of lists, so let's combine these two lists into one like this.

To turn this into an array, we use NumPy array function just like before. Let's call this array 'massvolume_array'. We see that the type of 'massvolume_array' is 'numpy.ndarray'. This means that it's an array from a NumPy package, and 'nd' means it could have any number of dimensions. Here's what it looks like. We can also see the shape of a NumPy array. To see the shape of a NumPy array, we type the name of the array, followed by '.shape'.

This will give us the number of rows and the number of columns. In this case, we have two rows and three columns. Each column is one material. The first row has the mass, and the second row has the volume. Note that the '.shape' here is an attribute. This is something we haven't seen before. An attribute gives you more information about the structure of an object. It's not the same as a method even though the syntax looks similar. You can tell it's an attribute and not a method because it's not followed by parentheses.

A 2D NumPy array is similar to a list of lists, where you can perform calculations on that elements simultaneously. You can also subset more flexibly. For example, let's say you want to get the volume of the third material; you can subset with '[1, 2]'. This tells you that you want to index from the row in index 1, and within that row, you want the volume of index

2. We now know that the volume of that material is 10. You can also subset entire rows. If you just want to see the first row, you index 0.

Sub setting in NumPy arrays is very flexible. You can select any number of rows and any number of columns as long as the sub setting for row is separated from the sub setting for the columns with a comma. For example, if you want to select the masses and volumes of the second and third element, you'd want both rows in the second and third column, so you index rows 0 to 1, using '[0:2,' and then after the comma, you index 1 and 2, using '1:3]'. Remember, the last index isn't included.

****

Video 4: Looping Over a 2d NumPy Array (3:45)

Let's discuss looping over a NumPy array. As you may have suspected, looping over a 1D numpy array will be the same as looping over a list. But looping over a 2D numpy array will be slightly different. Let's start with a 1D numpy array. Here, I pasted the code to create an array that you should already be familiar with. We 'import numpy as np' and create a list of masses. We then turn that list into an array called 'mass_array' using numpy's array function.

Now, if we want to loop over it, we do the same thing we do to a list. To find weight in Newtons of an object on Earth, multiply the mass by 9.8. Here we can do this with a 'for loop'. This loop creates a new variable weight, which is the product of the mass and it's 9.8. Then it prints out each weight. This should seem straight forward so far, but let's move over to a 2D numpy array. Here, we have a list of lists called 'massvolume'.

The first sub-list is the list of masses, and the second sub-list is a list of volumes. To create the 'massvolume_array', we use the array function in numpy. Now, if you try to loop over this array the same way, we see that it simply prints out all the values at once, meaning it doesn't loop to each individual value.

We need to work around here. NumPy's nditer function is useful here. To loop through each individual value with this 2D numpy array, you can use 'np.nditer' function ends in parentheses, the name of the array. Here, you can see that each value printed out individually. We can use this in combination with subsetting to loop through only specific values in the array.

For example, if I want to find the weight by multiplying the masses by 9.8, I can use only the values that represent mass, and not the values that represent volume. In this array we know that the masses are in the first row, so index '0'. Here, we loop through the values using the nditer function on 'massvolume_array', index '0'. We multiply each value by 9.8, and assign that value to weight; then we print out each weight. This syntax is great to remember when you want to perform calculations on only a certain row or column.

****

Video 5: Getting Started with Pandas Creating DataFrames (6:47)

By now, you should have a general understanding of Python syntax. In this video, we're going to shift gears towards working with real data. A common way that data could be stored is in tables of rows and columns, similar to what you'd find in a spreadsheet. Each row is an observation, and each column is a variable measuring some value of that observation. This is where the pandas library will come in handy.

The pandas library includes data structures and functions that allow you to go through the entire data analysis and workflow within Python, so you don't have to rely on other specialized programming languages. Arguably, the most important thing that pandas contributes is the pandas DataFrame. A DataFrame is a data structure that allows you to neatly store data in rows and columns.

This is similar to a data structure we already saw, the NumPy array. However, a 2D NumPy array allows you to only store data of one type. So you can store all integers or all strings, but not a mix. More often than not, your data will contain multiple data types. So you'll want to use a more flexible data structure, like the pandas DataFrame. Let's look at three methods of creating a DataFrame. Before we do anything, we need to import pandas. We'll alias it with a common alias, 'pd'.

The first method we'll look at is converting a dictionary to a pandas DataFrame. Here we have a predefined dictionary of colleges your cousin applied to, and whether or not he was accepted. To convert this dictionary to a DataFrame, let's first name our DataFrame. I'm going to name it 'my_df'. Then I'm going to call the alias 'pd.' and the function '.DataFrame( )'. Dataframe takes the argument of where your data lives. In this case, it's the 'dictionary' so that's all you need to enter. We can print out this DataFrame to see what it looks like.

You can see it's very neat. You'll have 'college' and 'accepted' as the columns and the rows are labeled with their index numbers. Let's compare that to the dictionary so you can see the difference. I'm going to print out the dictionary and we see that the pandas DataFrame is much neater and easier to read. For example, if you wanted to know if your cousin was accepted to 'Hunter', here, you can quickly glance in it, and see that you have 'True' under 'accepted'.

Whereas here, you'd need to see where 'Hunter' is, and then map it to the acceptance status here. Let's try out another method, creating a DataFrame from a list of lists. I'm going to first create this list, and call it 'list_of_lists' Recall that a list is created with '[ ]'. In these square brackets, I'm going to have sub-lists of the college your cousin applied to, and his acceptance status. The first sub-list is going to say '['Columbia' , True]'. The second is going to say '['Stanford' , False]', and so on.

Now that I have my list of lists, I can convert it to a DataFrame. Again, the first thing I'm going to do is name my DataFrame. I'll use 'my_df' again and I'll use the DataFrame function from the pandas library. The function takes the argument of where your data lives; in this case, it's in 'list_of_lists'. The other thing that you need to do here is enter the column names. When we converted a dictionary to a DataFrame, we didn't need to do that because the keys are automatically applied to be the names of the columns.

Whereas here, Python has no way of knowing what you want to call your columns unless you specify the columns parameter. The columns parameter takes a list of names in this case, '['college' , 'accepted']', and that's all you need to do. Again we could print out the DataFrame to see what it looks like. And we see a nice, neat way to store your data. Okay, I'm going to show one more method, even though there are many other methods that you could use. This is a very common method that you can use when your data's sent to you in a '.csv' format.

Before doing anything, make sure your data is saved in the same dictionary as your Jupyter Notebook. For this example, we'll use a dataset from the Pew Research Center that contains responses from Americans to questions about social media used, and attitudes towards the internet. I'm going to call this DataFrame 'df' and create it by importing data from the csv. To do that, I'm going to use the 'pd.read_csv'. Then I'll enter a string that has the name of the '.csv' file.

A quick way to do this, and to make sure you have no typos, is to type the quotation marks for the string, and then hit Tab on your keyboard. You'll see a list of suggestions, and you can scroll down and select the '.csv' that you want. When you run this code, the entire dataset will be saved as the DataFrame 'df'. So, it's ready to analyze. So, these are three methods to creating a DataFrame. To review, we created a DataFrame from a 'dictionary', we also created one from a 'list of lists', and finally, we created a DataFrame by importing data from a '.csv'.

****

Video 6: Getting Started with Pandas Slicing and Filtering DataFrames (5:20)

Let's work with the data set from the Pew Research Center. This data contains responses from Americans to questions about social media usage and attitudes towards the Internet. Here, we import the data using the pandas function 'read_csv', and assigning it to the variable social. Before we can analyze any data, we need to learn how to select data from specific rows or columns from the DataFrame. Let's try selecting the column that contains responses to the question, what is the main reason you think the Internet has been a good or bad thing for society?

This column is called 'pial11ao@'. 'Pial' stands for Pew Internet and American Life, which is the survey where this was taken from. There are three ways we can select a column. The first way is using square brackets. Simply write the name of the DataFrame, in our case, 'social [ ]'. In the brackets, enter a string with the name of the column. Printing this out, we see some interesting responses. Notice that the data type of this column is a pandas Series. This is a data type you probably haven't seen before.

A Series is simply a column of a DataFrame. It's like a list or an array, but it's labeled. If you use double square brackets when selecting the column, you'll see that the data type is a DataFrame. You're creating a new DataFrame here that contains only one column. You can also use double square brackets to select multiple columns. Here I'll create a DataFrame with two columns, 'pial11', which is a response to whether the participant thinks that overall the internet has been mostly a good or mostly a bad thing for society and 'pial11ao@', which is their explanation for that answer.

You can also use index numbers in square brackets. If you want to select the first three rows, for example, we'd slice the rows from 0 to 3. A second way you can select data is by using 'loc'. With 'loc' you can select data from one column to another using column labels. Write out the name of your DataFrame and follow it with a '.loc'. After, in square brackets slice the rows you want. Because I want to keep all the rows, I'll just enter a colon. Then, after the comma, slice the columns you want using their labels.

This will come in handy if you have a DataFrame where rows and columns are labeled. If you simply want to use index numbers to select data, use the '.iloc' function instead. In this example, we'll select all the rows and the second column of the DataFrame. Now that you have an understanding of slicing DataFrames let's shift gears towards filtering. This requires you to have a strong understanding of booleans.

Let's look at an example. Say you want to compare the attitudes towards the internet of your younger participants to your older participants. There's a column in our dataset called Age that contains the age of our participants. Let's create a filter variable called 'under35' that will be true if participants are under 35 and false if participants are over 35. To do this, subset the 'age' column and type less than '35'.

We can now use this variable to subset the DataFrame. If we call 'social[under35]', we'll get data only for those participants who are younger than 35. This will be useful when you want to compare different groups in your data or focus on a specific subset of your data. To review, we can select data from a DataFrame with square brackets, and the '.loc' and '.iloc' methods. You can also create filter variables that allow us to look at sections of our data that fit a certain criteria.

****

Video 7: NumPy and Pandas Statistical Tools (4:59)

Now that you've got the basics of NumPy arrays and pandas DataFrames down, we can learn about how powerful the NumPy package really is by looking at how it can be used for statistics. Let's first import pandas and NumPy. Now, let's use pandas to import the data to a DataFrame called 'df'. This code cell contains code for data cleaning, which we haven't covered yet, so don't worry about it for now. The data imported here was collected by the Pew Research Center in 2018 on social media usage.

They asked questions about the frequency of usage, and attitudes towards social media, and made it publicly available. If you'd like to take a closer look at the data, you can download it at this link. Note that this data has survey responses to 70 questions from over 2,000 people. So, unlike other data we have looked at, we won't be able to eyeball the numbers easily. To see the column names, we can use the columns attribute by typing 'df.columns'.

If you're curious, you can take a look at the questions asked in the survey to get an understanding of any of the seven columns in the dataset. In this lesson, you will use the 'intfreq' column, which is a response to the question, how often do you use the internet on a scale of one to five? The 'books1' column, which is in response to how many books have you read in the past year? And the 'age' column which contains the participant's age. Using NumPy functions, we can do a few quick analyses on the data.

Let's say we want to see the mean response to the question about how often do you use the internet. Survey participants responded to this question from one to five. NumPy has the function mean, which takes the argument of the data you're trying to find the mean of. We'll call it 'np.mean', and in '()', enter 'df', which is the name of our DataFrame, and 'intfreq', which is the name of the column with the responses to the question in '[ ]'. We see that the response is about 2.14. Let's try using 'np.median' to find the median response.

Our output says 'nan'; 'nan' stands for not a number, and this output means that one of our, or more, of our responses in this column, is not a number. So, the function can't perform the calculation. There are a few reasons why we might have nans in our data. And we'll discuss that in more detail when we learn about data cleaning. But when you have nans in your data, and you want to find the median, NumPy has another function called 'nanmedian'.

Calling that function, we see that the median of this column, ignoring nans, is '2.0'. Let's take a look at some other columns. One of the questions asked in this survey is how many books have you read in the past year? You may wonder if a response to this question is associated with age. Perhaps the older people are, the more books they read. We can check this out with correlation.

The column 'books1' holds the response to the question by the number of books read, and the column 'age' contains the participant's age. You can use 'np.corrcoef' with these columns to find the correlation. Without getting into statistics too much, it looks like the correlation is around '0.036', which is positive so that higher age is associated with more books read, but the magnitude is rather small, so this correlation may be insignificant.

Take some time to play around with NumPy and analyze some data to really get used to it. If you want to see what functions are available in NumPy, you can type 'np', then hit a Tab key. You'll see the available functions listed here. You can do the same thing with 'pd' with the NumPy package by using the 'mean', 'median', 'nanmedian', and 'corrcoef' functions.

****

---END---