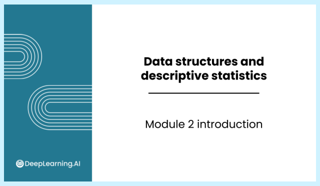
DAG C3M2 scripts

Last updated: Nov 5, 2024, 12:30am

Video title	lecture	Isabel	Sean	slides	alpha
L0v1 – Module 2 introduction	-	✅	✅	✅	✅
L1v1 – Beyond lists	✅	✅	✅	✅	✅
L1v2 – Installing and using modules	✅	✅	✅	✅	✅
L1v3 – DataFrames	✅	✅	✅	✅	✅
L1v4 – Attributes and methods	✅	✅	✅	✅	✅
L1v5 – Series	✅	✅	✅	✅	✅
L1v6 – Dictionaries and NumPy arrays	✅	✅	✅	✅	✅
~~L1v7 – Pair programming with LLMs~~	~~✅~~	~~❌~~	~~❌~~	~~❌~~	~~✅~~
L2v1 – Reading CSV files into a DataFrame	✅	✅	✅	✅	✅
L2v2 – Selecting columns	✅	✅	✅	✅	✅
L2v3 – Selecting rows	✅	✅	✅	✅	✅
L2v4 – Filtering	✅	✅	✅	✅	✅
L2v5 – Sorting	✅	✅	✅	✅	✅
L2v6 – Applying functions	✅	✅	✅	✅	✅
L3V1 – Counts, sums, & histograms	✅	✅	✅	✅	✅
L3V2 – Central tendency, variability, and skew	✅	✅	✅	✅	✅
L3V3 – Categorical data	✅	✅	✅	✅	✅
L3V4 – Correlation	✅	✅	✅	✅	✅
L3V5 – Segmentation by one feature	✅	✅	✅	✅	✅
L3V6 – Segmentation by multiple features	✅	✅	✅	✅	✅

Introduction

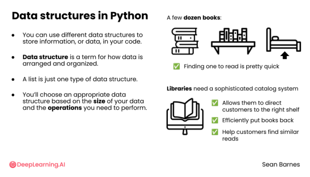
L0V1 – Module 2 introduction

Visual	Script
🎙️ TH  Data structures and descriptive statistics Module 2 introduction DeepLearning.AI	Welcome to Module 2: Data Structures and Descriptive Statistics! In this module, you'll discover how to organize, analyze, and derive meaningful insights from your data using a powerful Python library called Pandas. You'll learn how to use essential data structures, including DataFrames and Series, that allow you to work efficiently with large datasets. In the first lesson, you'll explore data structures beyond lists, learning how DataFrames can store and organize complex data just like a spreadsheet, but with much more analytical power. The second lesson focuses on selecting, sorting, and filtering your data - crucial skills that let you focus on exactly the information you need. You'll learn how to pull out specific columns and rows, sort your data in meaningful ways, and filter to find precisely what you're looking for. Throughout this lesson and the next, you'll use a real world data set of survey responses about learning to code. In the final lesson, you'll learn how to calculate descriptive statistics using pandas – from counts and sums to measures of central tendency to correlations between features and segmentation. You'll also create visualizations to help communicate your findings effectively, including histograms, bar charts, and scatter plots. By the end of this module, you'll be able to transform massive data sets into actionable insights using Python. Follow me to the first lesson to begin exploring these powerful data structures. See you there!

Data structures

L1v1 – Beyond lists

Visual	Script
🎙️ TH Data structures and descriptive statistics Beyond lists DeepLearning.AI	In the previous module, you worked extensively with lists, which store data in a sequence. It turns out that the **way** you store information when you're coding is quite important. It determines how you can interact with that information, how easy it is to access, and how efficient your analysis will be.



Data structures in Python

- You can use different data structures to store information, or data, in your code.
- **Data structure** is a term for how data is arranged and organized.
- A list is just one type of data structure.
- You'll choose an appropriate data structure based on the **size** of your data and the **operations** you need to perform.

A few **dozen books**:

✓ Finding one to read is pretty quick

Libraries need a sophisticated catalog system

✓ Allows them to direct customers to the right shelf
✓ Efficiently put books back
✓ Help customers find similar reads

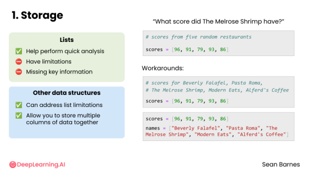
DeepLearning.AI Sean Barnes

In Python, **[CLICK]** you can use different data structures to store information, or data, in your code. **[CLICK]** "Data structure" is a fancy term for how data is arranged and organized. **[CLICK]** A list is just **one** type of data structure.

Think of it this way, if you have just **[CLICK]** a few dozen books lying around, it doesn't matter much how you store them. They could be in a **[CLICK]** stack, **[CLICK]** on a shelf, or in a **[CLICK]** pile under your bed. **[CLICK]** Finding one to read is pretty quick no matter what.

[CLICK] Libraries however need a sophisticated **[CLICK]** cataloging system. This system **[CLICK]** allows them to direct customers to the right shelf, **[CLICK]** efficiently put books back, and so on. Libraries often separate fiction from nonfiction as well, to **[CLICK]** help customers find similar reads.

In summary, the number of books you have, and the kinds of operations you need to do, determine the way you store the books. Similarly, **[CLICK]** you'll choose an appropriate data structure in Python based on the size of your data and the operations you need to perform.



1. Storage

Lists
✓ Help perform quick analysis
⛔ Have limitations
⛔ Missing key information

Other data structures
✓ Can address list limitations
✓ Allow you to store multiple columns of data together

"What score did The Melrose Shrimp have?"

```python
# scores from five random restaurants

scores = [96, 91, 79, 93, 86]
```

Workarounds:

```python
# scores for Beverly Falafel, Pasta Roma,
# The Melrose Shrimp, Modern Eats, Alferd's Coffee

scores = [96, 91, 79, 93, 86]
```

```python
scores = [96, 91, 79, 93, 86]
names = ["Beverly Falafel", "Pasta Roma", "The Melrose Shrimp", "Modern Eats", "Alferd's Coffee"]
```

DeepLearning.AI Sean Barnes

In the previous module, you worked with 5 or 10 years of library data, or around 100 restaurant grades. **[CLICK]** Lists helped you perform quick analysis, and you didn't need to scale just yet to millions of rows.

However, lists **[CLICK]** have a few key limitations – in particular, storage, flexibility, and efficiency. The big picture is that **[CLICK]** other data structures available to you can address these limitations. Let's talk about storage.

If you have a list like **[CLICK]** this:

```python
scores = [96, 91, 79, 93, 86] # scores from five random restaurants
```

Say you wanted to answer a question like **[CLICK]** "what score did The Melrose Shrimp have?" You're **[CLICK]** missing some key information! Can you tell what it is? **[pause for thought]** That would be the restaurant names! While it's useful to have just a list of scores — for example, to calculate the average restaurant score – having the name of the restaurant is even more useful.

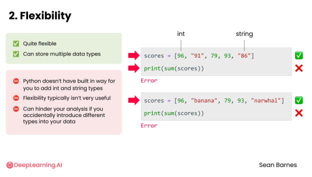
There are **[CLICK]** workarounds, like **[CLICK]** adding comments:

Python
```python
# scores for Beverly Falafel, Pasta Roma, The Melrose Shrimp,
Modern Eats, Alferd's Coffee
scores = [96, 91, 79, 93, 86]
```

Or **[CLICK]** keeping track of two separate lists:

Python
```python
scores = [96, 91, 79, 93, 86]
names = ["Beverly Falafel", "Pasta Roma", "The Melrose Shrimp",
"Modern Eats", "Alferd's Coffee"]
```

But it's clunky. Other types of data structures **[CLICK]** allow you to store multiple columns of data together, in the same way you can in a spreadsheet.



2. Flexibility

✅ Quite flexible
✅ Can store multiple data types

⛔ Python doesn't have built in way for you to add int and string types
⛔ Flexibility typically isn't very useful
⛔ Can hinder your analysis if you accidentally introduce different types into your data

```python
scores = [96, "91", 79, 93, "86"]
print(sum(scores))
```
Error

```python
scores = [96, "banana", 79, 93, "narwhal"]
print(sum(scores))
```
Error

DeepLearning.AI
Sean Barnes

Another aspect of lists is that they're quite flexible – in a sense too flexible. For example, lists in Python **[CLICK]** can actually store multiple data types.

For example, **[CLICK]** this list is **[CLICK]** perfectly valid:

Python
```python
scores = [96, "91", 79, 93, "86"]
```

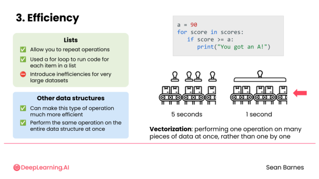
But what do you think happens when you **[CLICK]** run this line of code? **[pause for thought]**

Python
```python
print(sum(scores))
```

This line of code produces an **[CLICK]** error. And that's because **[CLICK]** Python doesn't have built in way for you to add **[CLICK]** int and **[CLICK]** string types. As far as Python is concerned, your list could look like **[CLICK]**this:

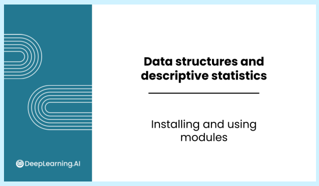
Python
```python
scores = [96, "banana", 79, 93, "narwhal"]
```

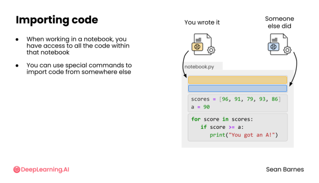
	So it **[CLICK]** doesn’t support adding these up. So on one hand lists are quite flexible and allow you to store different types in one data structure. But that **[CLICK]** flexibility typically isn’t very useful, and **[CLICK]** can actually hinder your analysis if you accidentally introduce different types into your data.
🖥️ Screencast 🔗 [C3M2L1V1_Library data]()	For example, if you have this Plainsville public library data, in each column, which you previously represented using a list, you always expect the same type of data. So in column A, you're always going to have an integer representing the year, B and C are always going to be strings, and D through G are always going to be integers that represent currency amounts. Your data is more useful for analysis because it’s organized in this way.


3. Efficiency
Lists
✅ Allow you to repeat operations
✅ Used a for loop to run code for each item in a list
⛔ Introduce inefficiencies for very large datasets
Other data structures
✅ Can make this type of operation much more efficient
✅ Perform the same operation on the entire data structure at once
`a = 90`
`for score in scores:`
`   if score >= a:`
`      print("You got an A!")`
 	
Vectorization: performing one operation on many pieces of data at once, rather than one by one
DeepLearning.AI
Sean Barnes
	Finally, consider the way lists **[CLICK]** allow you to repeat operations. In the previous module, you used a **[CLICK]** **for** loop to run code for each item in a list:
	
	Python
	`a = 90`
	`for score in scores:`
	`    if score >= a:`
	`        print("You got an A!")`
	
	This is a great solution, and **for** loops are a very common tool in Python programming. However, they **[CLICK]** introduce inefficiencies for very very large datasets – i.e. for data at scale.
	
	When this **for** loop runs, it essentially steps through each operation in order, running the indented code for each score in turn. Imagine a **[CLICK]** conveyor belt of scores going past you one by one, and your job is to **[CLICK]** stamp each score in turn if it’s an A.
	
	However, there are certain specialized data structures used in data analytics and machine learning that **[CLICK]** can make this type of operation much more efficient. You’ll encounter two of them over the next couple of videos. Instead of operating on each item in the data structure individually, you can **[CLICK]** perform the same operation on the entire data structure at once. If you imagine the **[CLICK]** conveyor belt of scores going past, in this example you have a **[CLICK]** very wide stamp that can stamp all 5 scores at once if needed.

	In the first example, if each task takes 1 second, the for loop code would take **[CLICK]** 5 seconds while the more efficient code would take just **[CLICK]** 1 second. At scale – for very large datasets – this efficiency adds up significantly. This process is called **[CLICK]** vectorization. Vectorization can be an intimidating word, but it essentially means **[CLICK]** performing one operation on many pieces of data at once, rather than one by one.
🎙️ TH	So while lists are incredibly useful, they aren't the only data structure! Follow me to the next video to learn how the Pandas library can provide some powerful data structures for data analytics.

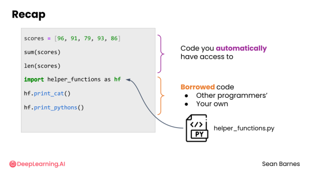
L1v2 – Installing and using modules

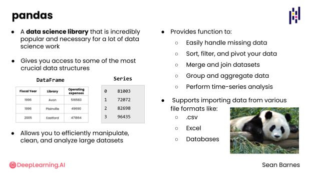
Visual	**Script**
🎙️ TH  **Data structures and descriptive statistics** Installing and using modules DeepLearning.AI	What do you do if you need a power tool, but you don't have one? **[eyebrows/duckface]** Borrow it, of course! It turns out you can do this in code too – borrowing someone else's tools!


Importing code
• When working in a notebook, you have access to all the code within that notebook
• You can use special commands to import code from somewhere else
You wrote it
Someone else did
notebook.py
`scores = [96, 91, 79, 93, 86]`
`a = 90`
`for score in scores:`
`    if score >= a:`
`        print("You got an A!")`
DeepLearning.AI
Sean Barnes
	When you're working in a **[CLICK]** notebook, you of course have access to all the code within that notebook – the **[CLICK]** variables, **[CLICK]** loops, and so on.
	
	However, **[CLICK]** you can also use special commands to import code from somewhere else, whether **[CLICK]** you wrote it or **[CLICK]** someone else did.

🖥️ Screencast 🔗 **C3m2l1v2 Importing helper functions**	Here are a few tasks you've been asked to accomplish. Let's import some helper functions like print_cat and print_pythons that have already been written to do some cool things. You'll also test importing the pandas library, which you'll use throughout this module, and test the unique function just to make sure that it's working correctly. First of all, in the same folder as this Import testing notebook that you are working with right now, there's also this file helper_functions.py. This is a Python file that contains a couple of fun functions. And here is what those functions look like. This is the entire helper_functions.py. Don't worry too much about what's going on here or what this file is, you're just going to borrow these two functions to use them in your notebook. You could just copy and paste all the code into your notebook. However, that

	can get clunky if you want to borrow a lot of code, or you don't have easy access to the file. If you want to use those functions, you can use the command **import** helper functions. Now, I can access those functions print cat and print pythons. I can run helper functions dot print cat. Hit shift, enter, and there's the cat. Now, nowhere in this notebook is the function print cat defined, but it's defined in the helper functions file. I can also run helper functions dot print pythons. Hit shift, enter, and there are some pythons. So this is a great way to borrow code from yourself. You have some code in another file, you'd like to use it in this one. It's a little annoying to have to write this long helper functions bit every single time. One thing that you can do in Python is say import helper functions as HF so you're giving helper functions a nickname and that just makes it a lot faster to type. So if I hit shift enter now, instead of saying helper functions, dot print pythons, you can say HF dot print pythons. Shift enter. There you go. And it just makes your coding life a lot faster and easier.

**Recap**
`scores = [96, 91, 79, 93, 86]`
`sum(scores)`
`len(scores)`
`import helper_functions as hf`
`hf.print_cat()`
`hf.print_pythons()`
Code you **automatically** have access to
Borrowed code
• Other programmers'
• Your own
helper_functions.py
DeepLearning.AI Sean Barnes
	So to recap, many functions you **[CLICK]** automatically have access to like **[CLICK]** `sum` and **[CLICK]** `len`. You can also **[CLICK]** **[CLICK]** borrow **[CLICK]** other programmers' code, including **[CLICK]** your own code that you've previously written, using the **[CLICK]** `import` command.
	
	As you just saw, the **[CLICK]** `import helper_functions as hf` command allows you to borrow all of the functions contained in **[CLICK]** this other Python file, helper_functions.py. So now in addition to `sum` and `len` and the many other functions you have access to, you can also use **[CLICK]** hf.print_cat and hf.print_pythons.

**pandas**
• A **data science library** that is incredibly popular and necessary for a lot of data science work
• Gives you access to some of the most crucial data structures
`DataFrame`
Fiscal Year / Library / Operating expenses
1996 / Avon / 516583
1996 / Plainville / 49690
2005 / Eastford / 471864
`Series`
`0 81003`
`1 72072`
`2 82698`
`3 96435`
• Allows you to efficiently manipulate, clean, and analyze large datasets
• Provides function to:
◦ Easily handle missing data
◦ Sort, filter, and pivot your data
◦ Merge and join datasets
◦ Group and aggregate data
◦ Perform time-series analysis
• Supports importing data from various file formats like:
◦ .csv
◦ Excel
◦ Databases
DeepLearning.AI Sean Barnes
	Throughout these modules you'll be using the pandas library, **[CLICK]** a data science library that is incredibly popular because it's very useful for a lot of data science work. It **[CLICK]** gives you access to some of the most crucial data structures, including **[CLICK]** DataFrames which store data in rows and columns, and **[CLICK]** Series which are like a very powerful, jazzed up list.
	
	Pandas **[CLICK]** allows you to efficiently manipulate, clean, and analyze large datasets. It **[CLICK]** provides functions to **[CLICK]** easily handle missing data; **[CLICK]** sort, filter, and pivot your data; **[CLICK]** merge and join datasets; **[CLICK]** group and aggregate data, and **[CLICK]** perform time-series analysis. It also **[CLICK]** supports importing data from various file formats like **[CLICK]** CSV, **[CLICK]** Excel, and even **[CLICK]** databases.

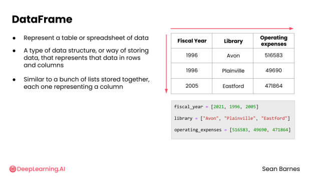
	Unfortunately pandas has nothing to do with the cute, cuddly animal from real life — it has a different and a bit less fun origin story. But **[CLICK]**here's a picture of a panda to make up for that fact.
🖥️ Screencast 🔗 [**C3m2l1v2 Importing pandas**]()	If you want to import and use pandas, as well as test out the unique function, you can "import pandas". Now, in this case, pandas wasn't written by you or probably anyone you know. This is a bunch of code written by others in the Python community that provides you with incredibly powerful data analysis tools. Now, same as before, when importing, you can give pandas a nickname. And what you'll very, very commonly see is import pandas as pd. It's just a lot shorter and easier to work with. So hitting shift enter, now pandas has been imported and you can use all of that cool code. Just a quick example, you'll explore a lot more in the coming videos, but say you have a list of ASCII faces like this. At a glance, it's a little hard to tell how many different ones there are. However, if you need to just get a list of unique faces, you can use the pandas unique function, pd.unique. So you're going to borrow this unique function from the pandas library. A library essentially means a Python file with a bunch of functions in it. If you hit shift enter, you get an array, which is a special Pandas data type similar to a list, with the unique faces. It turns out there were five unique faces, but think about how complicated of an operation that might have been if you had to write the code from scratch yourself. It would've required a loop and a conditional or two, and maybe that wouldn't even be the best or fastest way of finding these unique values. So borrowing this unique function from the pandas library was like borrowing a power tool from a friend – you get a superpowered capability at no cost!
🎙️ TH	Pandas has a lot of cool functions, but it also gives you access to two data structures that can handle incredibly complex analysis beyond what lists are capable of. Join me in the next video to learn more about DataFrames.

L1v3 – DataFrames

Visual	**Script**
🎙️ TH Data structures and descriptive statistics DataFrames DeepLearning.AI	When you're working with data, you're often working with data stored in rows and columns, which you saw is difficult to represent with a single list. DataFrames will be your main tool in Python to represent this type of data.

🖥️ Screencast 🔗 **C3m2l1v3_data frames exploration demo**	Previously, you worked with the Plainville library data, but now you have library data from many more libraries as well. Your first instinct when you see this data should be to explore it: the number of observations; you can already see that there are some missing values here where maybe there was no data. How do you go about doing this type of exploration in code? First, save the data as a csv, which is called `library_data`. Taking it over to a notebook, your first step is to load the data into Python. Then you want to explore that data, and start to identify some possible interesting research questions. First, you'll need to import pandas as pd. Then you're going to create a data frame by reading in a CSV file using a function from the library you just imported. You'll learn more about this line of code in a later lesson. If you hit shift enter and run this code, now you have all that data saved in the variable df. And what is the type of df? Well, it is a panda's DataFrame. What does that look like? If you just put df here in the cell, you're gonna get a really nice display of what this looks like. Hitting shift enter, there is all of your data, and it's essentially like a spreadsheet: it's a table of rows and columns. Is there anything you notice that's different about this than a spreadsheet? **[pause for thought]** There are a few differences. One that may jump out at you is that the numbering starts at 0. If you go back to your spreadsheet, technically index 2 is the first row of data. But as with lists in Python, your data frames start at zero. You also just see a preview of the data, not all of the rows, but you can see that there are about 5,100 rows and 7 columns. You could also print df, if you hit shift enter, you'll see that you still get the same exact data, but it's just displayed less nicely. Just having df, the variable name, and running that cell does the same thing but looks nicer. This is a feature of the Jupyter notebook, it's not a Python feature specifically. So what else might you want to know about this data? Maybe the names of all of the columns. If you type df.columns, shift enter, that command displays all of the names of the columns very succinctly. Fiscal year, library, county, and so on. What types of data are stored in the data frame? Remember types are crucial in Python. In DataFrames, each column has an assigned data type. If you type df.dtypes, for data types, you're essentially asking the computer, what are the data types of each column in this DataFrame? Shift Enter. Here,

	you have int64, so fiscal year is an integer. Then you have library and county, which are objects. Object can be an intimidating word, which you'll tackle in a future module, but in this case library and county are strings — so these are text data types. In pandas, specifically, when reading your data from a CSV file, it will use the object type for text which is a bit more flexible. You don't really need to worry about the particulars. Lastly, you have population, library visits, operating income, and expenditures, which are all floats. You could summarize all that information with df. info. Hit shift enter, you see that this is a data frame. There's 5,105 rows. You see all the columns and their types. What else do you see here that's new? **[pause for thought]** Well, that would be the non null count. If you go back to your library data, you saw earlier that some of these cells are empty. There's nothing in there. Maybe the data was lost, but whatever the reason is, it's just left blank. Blank doesn't mean zero, because, for example, this library definitely serves some people, otherwise why would it exist? It just means the information is missing. The lower this number is, the more null data points you have. For example, for the population of the service area, it looks like this data point was missing almost 500 times. Let's take a closer look at how data frames work in Python.

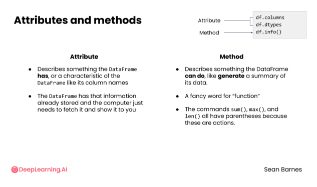

DataFrame
• Represent a table or spreadsheet of data
• A type of data structure, or way of storing data, that represents that data in rows and columns
• Similar to a bunch of lists stored together, each one representing a column
Fiscal Year · Library · Operating expenses
1996 · Avon · 516583
1996 · Plainville · 49690
2005 · Eastford · 471864
`fiscal_year = [2021, 1996, 2005]`
`library = ["Avon", "Plainville", "Eastford"]`
`operating_expenses = [516583, 49690, 471864]`
DeepLearning.AI
Sean Barnes
	You saw how DataFrames **[CLICK]** represent a table or spreadsheet of data. They are **[CLICK]** a type of data structure, or way of storing data, that represents that data in **[CLICK]** rows and **[CLICK]** columns.
	
	You can think of data frames as **[CLICK]** similar to a bunch of lists stored together, each one representing a column. For example, to represent the library data, you could have a list variable called **[CLICK]** fiscal year, with all the years; and a **[CLICK]** list variable called library with the library names; all the way to **[CLICK]** operating_expenses. A data frame combines all those lists into a single data structure.

🎙️ TH	DataFrames have many advanced capabilities, which you'll see throughout this module. Follow me to the next video to see a key difference between attributes and methods.

L1v4 – Attributes and methods

Visual	**Script**

🎙️ TH	DataFrames store data in rows and columns. They also come with some extra information – attributes, or characteristics, and methods, which allow you to perform actions on the stored data.

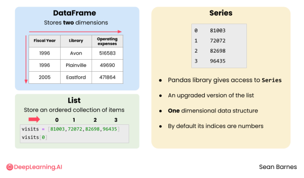
 **Attributes and methods**
Attribute `df.columns` `df.dtypes`
Method `df.info()`
Attribute
• Describes something the `DataFrame` **has**, or a characteristic of the `DataFrame` like its column names
• The `DataFrame` has that information already stored and the computer just needs to fetch it and show it to you
Method
• Describes something the DataFrame **can do**, like **generate** a summary of its data.
• A fancy word for “function”
• The commands `sum()`, `max()`, and `len()` all have parentheses because these are actions.
DeepLearning.AI Sean Barnes
	What do you notice about the different commands you learned in the previous video? **[pause for thought]**
	
	Python
	`df.columns`
	`df.dtypes`
	`df.info()`
	
	Columns and dtypes do not need parentheses, while info does. This difference is because columns and dtypes are **[CLICK]** attributes, while info is a **[CLICK]** method. An attribute **[CLICK]** describes something the DataFrame **has**, or a characteristic of the DataFrame like its **[CLICK]** column names. A method **[CLICK]** describes something the DataFrame can **do**, like **generate** a summary of its data.
	
	“Method” is just **[CLICK]** a fancy word for function. They are technically slightly different, but the difference is out of scope for this course. **[CLICK]** The commands `sum()`, `max()`, and `len()` all have parentheses because these are actions. In the case of sum, the **[CLICK]** computer has to go and add up all the values. In the case of df.dtypes and attributes in general, **[CLICK]** the DataFrame has that information already stored and the computer just needs to fetch it and show it to you. **[CLICK]** No calculation needed.

	As a real world analogy, think about how you answer different questions about yourself. Questions like **[CLICK]** what’s your name, **[CLICK]** what’s your age — you probably don’t need to think to answer those. They’re your **[CLICK]** attributes or characteristics. But what if I asked you a question like **[CLICK]** “how many days has it been since you visited a park?” **[pause for thought]** You probably have to think about when that was, then count up the number of days it has been since then. You have to do some calculation to get the answer, which is more similar to using a **[CLICK]** method or function. I doubt you have the answer memorized just in case someone asks! The **[CLICK]** computer takes the same kinds of shortcuts. For **[CLICK]** commonly accessed, fundamental information about a DataFrame, Python stores that information as an attribute, which is fast to access. For **[CLICK]** more complex questions, like **[CLICK]** generating summary data, or in general **[CLICK]** most operations, those are methods – the computer has to take some action or do some calculation to get you the answer. So, **[CLICK]** big picture, you’ll need to differentiate between a DataFrame’s

	attributes and methods. **[CLICK]** When you're accessing an attribute, like `columns` or `dtypes`, you don't need to use parentheses. **[CLICK]** When you're using a method, you will use parentheses, and often arguments to the method as well. Luckily there aren't too many attributes – you'll probably find yourself using columns, dtypes, and others like the dimensions of the DataFrame – the number of rows and columns.
🎙️ TH	Now that you've explored the DataFrames data structure, let's take a look at the Pandas series – it's a data structure that represents a single column of data in your DataFrame. Follow me to the next video to see how it works.

L1v5 – Series

Visual	**Script**
🎙️ TH Data structures and descriptive statistics Series DeepLearning.AI	Often when working with a table of data, you're interested in just one row or just one column. The **Pandas Series data structure** allows you to analyze one dimensional data like a row or column.


DataFrame
Stores **two** dimensions
<table><tr><th>Fiscal Year</th><th>Library</th><th>Operating expenses</th></tr><tr><td>1996</td><td>Avon</td><td>516583</td></tr><tr><td>1996</td><td>Plainville</td><td>49690</td></tr><tr><td>2005</td><td>Eastford</td><td>471864</td></tr></table>
List
Store an ordered collection of items
0 1 2 3
`visits = [81003,72072,82698,96435]`
`visits[0]`
Series
`0 81003`
`1 72072`
`2 82698`
`3 96435`
• Pandas library gives access to `Series`
• An upgraded version of the list
• **One** dimensional data structure
• By default its indices are numbers
DeepLearning.AI
Sean Barnes
	A DataFrame **[CLICK]** stores **two** dimensions – **[CLICK]** rows and **[CLICK]** columns. However, you'll often need to focus your analysis on just one column of data. For that, you'll need a one dimensional data structure.
	
	Think back to **[CLICK]** lists in Python. The list data structure allows you to **[CLICK]** store an ordered collection of items. Here's an example of a list containing **[CLICK]** 5 years of total library visits for the Plainville library. Do you remember how to access the first element of this list? **[pause for thought]**
	
	You would access the first element using **[CLICK]** `visits[0]` , which selects the first item from the list. So lists have **[CLICK]** numerical indices – you use a number to access each row.
	
	The **[CLICK]** Pandas library gives you access to the Series data structure, which you can think of as an **[CLICK]** upgraded version of the list. Like a list, a Series is a **[CLICK]** one dimensional data structure. When you create a Series, **[CLICK]** by default its indices are also numbers. So there are a lot of similarities. The skills you developed while programming with lists will be incredibly helpful, and you'll also get to do many operations faster and more efficiently.

🖥️ Screencast 🔗 [C3M2L1V5_Series]()	You have a couple tasks before you. You want to store the first five years of Plainville Public Library visits, and you want to find the average visits per year. You've done this operation before with lists. Here is a variable named visits, that stores all of the total visits. And as you just reviewed, this list has indices 0

through 4.

Okay, so if you hit Shift Enter, now you can use the visits variable. You can say, for example, print visits 0, which is the value for 1996. If you want to find the average number of visits per year, sum up all the visits and then divide by the number of visits. Hit Shift Enter, and the average is somewhere around 95,000.

How can you do this faster and more easily using Pandas? First, you can create a Pandas series, call it visits series, using this list. Pd.series allows you to create a series from your visits lists. Hit shift enter, and now take a look at visits series. Hit shift enter.

This has exactly the same information as before. On the left, you have indices 0, 1, 2, 3, 4, and on the right you have the values. Taking a look at the original visits list, these two data structures might look the same. However, because this is a Pandas Series, you now have access to many more functions; in particular, data analysis functions.

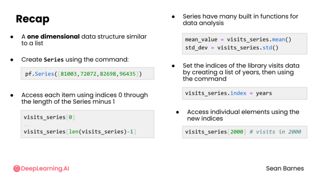
So how do you find the average of this series? In this case, you can type visits series dot mean. What value do you expect to be printed if you run this command? Shift Enter, gives you a floating point number, 95,210.8, the same as the average of the list, but now you calculated it with a single command. And it doesn't stop there. You can say visits series .std for standard deviation. Shift Enter, it's around 8,866.

How would you calculate the standard deviation from the list? It's a much more complicated operation, and you'd have to write your own code to do so. But because the pandas library is built for data analysis, using a series gives you so many more options.

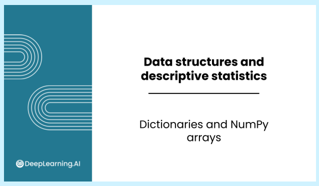
Another really useful feature of series is that you can assign different indices to that series. When you think about what this data means, each value for total visits is not associated with a number 0 through 4, but actually with a year. With a Pandas series, you can associate the two. Instead of having indices that are numbers like this, you can assign the indices to whatever you would like.

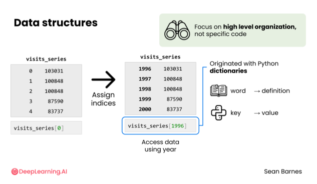
Start by creating a list of the years that you need to label each number of visits. Now, you want to take visits series dot index. Index is the set of all of these numbers here that correspond to each value, and you want to assign it the value years. What you're doing is taking the values in years, and you're assigning them to these indices here.

If you hit shift enter, now you can check out this series again. You have this beautiful data structure with each year corresponding to each number of visits corresponding with that year. So now, instead of having to remember which

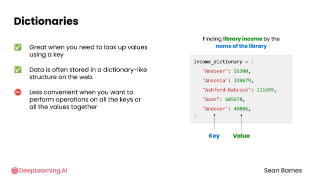
	year goes with which index as you would with a list, you can access each year's data directly. For example, you can say visits series, open brackets, but instead of saying 0, or 1, or 2, how do you think you would access the value for the year 2000? **[pause for thought]** In this case, you can just write 2000. And there you get the value for the year 2000. Same thing for 1996. You get 103,000, which is the number of visits from 1996. Indices can be numbers, as you saw in this example, and they can also be strings.
 **Recap** • A **one dimensional** data structure similar to a list • Create `Series` using the command: `pf.Series([81003,72072,82698,96435])` • Access each item using indices 0 through the length of the Series minus 1 `visits_series[0]` ... `visits_series[len(visits_series)-1]` • Series have many built in functions for data analysis `mean_value = visits_series.mean()` `std_dev = visits_series.std()` • Set the indices of the library visits data by creating a list of years, then using the command `visits_series.index = years` • Access individual elements using the new indices `visits_series[2000] # visits in 2000` DeepLearning.AI Sean Barnes	To recap the key aspects of Series. • They're **[CLICK]** a one dimensional data structure similar to a list. • You can **[CLICK]** create them using the command **[CLICK]** pd.Series() and providing that function a **[CLICK]** list • You can then **[CLICK]** access each item as you would with a list, using indices 0 through the length of the Series minus 1 • You saw how **[CLICK]** Series have many built in functions for data analysis, like **[CLICK]** .mean() and .std(), making working with them faster and easier compared with lists. • You also saw how to set the indices of the Series, allowing you to define how you want to access the values. You **[CLICK]** set the indices of the library visits data by creating a list of years, then using the command **[CLICK]** `visits_series.index = years` • You can then **[CLICK]** access individual elements using the new indices, like **[CLICK]** `visits_series[2000]` to get the number of visits in the year 2000
🎙️ TH	You've seen how the Pandas Series data structure allows you to quickly perform analysis on data. There's a lot more to explore and you'll learn a ton of techniques in the coming videos. For now, follow me to the next video to learn more about other common data structures you'll see in Python.

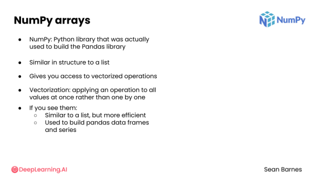
L1v6 – Dictionaries and NumPy arrays

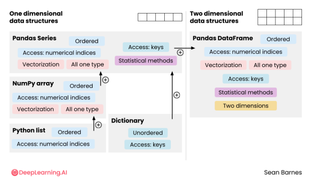
Visual	**Script**
🎙️ TH  **Data structures and descriptive statistics** Dictionaries and NumPy arrays DeepLearning.AI	While you'll primarily be working with Pandas DataFrames and Series in this course, you should also be familiar with two other common data structures: dictionaries and numpy arrays.

 **Data structures**
Focus on **high level organization,** not specific code
`visits_series`
`0 103031`
`1 100848`
`2 100848`
`3 87590`
`4 83737`
`visits_series[0]`
Assign indices
`visits_series`
`1996 103031`
`1997 100848`
`1998 100848`
`1999 87590`
`2000 83737`
`visits_series[1996]`
Access data using year
Originated with Python **dictionaries**
word → definition
key → value
DeepLearning.AI Sean Barnes
	Throughout this video, try to **[CLICK]** focus on the high level organization of each data structure rather than the specific code. You won't be coding with these explicitly in this course.
	
	So, when **[CLICK]** working with Series, you saw how **[CLICK]** assigning your own indices allowed you to **[CLICK]** access data about library visits using the year rather than memorizing an index value.
	
	This feature **[CLICK]** originated with Python dictionaries. In a **[CLICK]** real world dictionary, you use a **[CLICK]** word to look up its **[CLICK]** definition, rather than searching for a number. In a **[CLICK]** Python dictionary, you use a **[CLICK]** *key* to access a **[CLICK]** *value*. These key value pairs allow you to model meaningful relationships in your data.

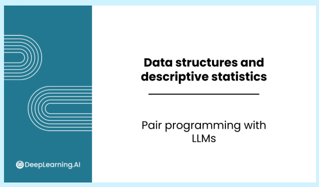
🖥️ Screencast 🔗 [C3m2l1v6 Dictionaries]()	Here's how you could store this same data about library visits using a dictionary. Dictionaries in Python are defined using curly braces, whereas lists use brackets. Each line in this dictionary has two values. The key, that's what you're going to use to look up the data, in this case the year, and the value, in this case the number of library visits. Hitting shift enter, now you can access each value the same way you did with the series. So visits dictionary 1996 is the value for 1996. For 2000, it's the value for 2000. Here's another example of a dictionary with strings as keys. Each key is the name of a library, and each value is the income made by that library in the year 1996. If you hit Shift Enter, now you can use the income dictionary for Avon, for example. And the value this command accesses is 685,678.

 **Dictionaries**
✅ Great when you need to look up values using a key
✅ Data is often stored in a dictionary-like structure on the web.
⛔ Less convenient when you want to perform operations on all the keys or all the values together
Finding **library income** by the **name of the library**
`income_dictionary = {`
`"Andover": 56300,`
`"Ansonia": 320679,`
`"Ashford-Babcock": 111699,`
`"Avon": 685678,`
`"Andover": 40806,`
`}`
Key Value
DeepLearning.AI Sean Barnes
	Dictionaries are **[CLICK]** great when you need to look up values using a key, like **[CLICK]** finding library income by the name of the library. **[CLICK]** Data is often stored in a dictionary-like structure on the web. They're **[CLICK]** less convenient when you want to perform operations on all the keys or all the values together.

 **NumPy arrays** NumPy • NumPy: Python library that was actually used to build the Pandas library • Similar in structure to a list • Gives you access to vectorized operations • Vectorization: applying an operation to all values at once rather than one by one • If you see them: ◦ Similar to a list, but more efficient ◦ Used to build pandas data frames and series DeepLearning.AI Sean Barnes	You'll likely also encounter data stored in NumPy arrays. **[CLICK]** NumPy is another Python library that was actually used to build the Pandas library. I know, I know, there are so many layers here! But it shows the progress you can make when you stand on the shoulders of those before you 🙂 A NumPy array is very **[CLICK]** similar in structure to a list, but it **[CLICK]** gives you access to vectorized operations. Remember that **[CLICK]** vectorization means applying an operation to all the values in a data structure at once rather than one by one. You won't really need to create NumPy arrays directly – just **[CLICK]** remember

	if you see them that they're **[CLICK]** similar to a list, but more efficient for numerical computation, and that they're **[CLICK]** used to build Pandas Series and DataFrames.
 One dimensional data structures Two dimensional data structures Pandas Series Ordered Access: numerical indices Vectorization All one type Access: keys Statistical methods NumPy array Ordered Access: numerical indices Vectorization All one type Python list Ordered Access: numerical indices Dictionary Unordered Access: keys Pandas DataFrame Ordered Access: numerical indices Vectorization All one type Access: keys Statistical methods Two dimensions DeepLearning.AI Sean Barnes	Are you noticing all the similarities between some of these data structures? They're all related to each other. A **[CLICK]** Python list is an **[CLICK]** ordered one dimensional data structure where elements of different types can be stored together, and these elements are **[CLICK]** accessed using numerical indices starting from zero. A **[CLICK]** numpy array is a more powerful version of a list that allows **[CLICK]** vectorization when performing mathematical operations. It also uses numerical indices starting from zero, and they **[CLICK]** typically allow only one type of data. A **[CLICK]** dictionary is an **[CLICK]** unordered data structure that allows **[CLICK]** access to values using keys rather than numerical indices. A **[CLICK]** pandas Series is like a **[CLICK]** combination of an array and a dictionary. It is an **[CLICK]** ordered one dimensional data structure that **[CLICK]** can have numerical indices starting from zero. ● However you can also assign it different indices, allowing you to **[CLICK]** access values using keys. ● Pandas Series also have built in **[CLICK]** vectorization and **[CLICK]** allow only one type because they're built using NumPy arrays ● And they have **[CLICK]** built in statistical methods to facilitate analysis. A **[CLICK]** data frame is a two dimensional data structure with rows and columns. Each column is a **[CLICK]** Series. So you get **[CLICK]** all the power and flexibility of a Series – vectorization, built-in computational methods – and in **[CLICK]** two dimensions!
🎙️ TH	It's interesting to see how complex data structures can be built out of simpler ones. In the next video, you'll take a look at how these data structures are used to represent different types of data in the real world. I'll see you there!

L1v7 – Pair programming with LLMs

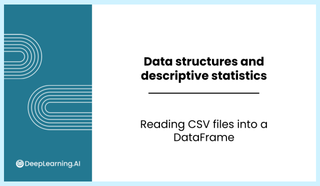
Visual	**Script**
🎙️ TH Data structures and descriptive statistics Pair programming with LLMs DeepLearning.AI	At this point, you may be starting to feel overwhelmed by all the options available to you with Python and the Pandas library. If you learn to pair program effectively with an LLM, it can help reduce that sense of overwhelm and help you write high quality code quickly.

	Traditional pair programming involves two developers working together. Since the advent of LLMs however, now you can have a coding buddy at all hours of the day and night! Remember that LLMs have read a vast amount of text, a lot of it from the internet. And a lot of the text on the internet is about programming — blogs, tutorials, online courses. So LLMs are an incredibly effective pair programmer. However, while they have a lot of background knowledge about Python, they will inevitably be completely unfamiliar with the problem you're trying to solve. When working with an LLM, you'll need to spend some mental effort explaining the problem, which can make your prompt quite long. Pair programming with an LLM often involves a back and forth conversation. You have your window open with the chat and your notebook open for programming.
🔗**Source 1** 🔗**Source 2**	Some best practices to keep in mind when pair programming with an LLM: ● Use a strategy of iterative refinement. For example, you could first describe the problem and ask the LLM to repeat it back to you so you know it understands. Then you could ask for a basic solution, then expand on different aspects of that solution. ● Remember that generated code is often wrong. To help combat hallucinations, ○ Have an expectation of what the answer should look like. You should know roughly what kind of code you're looking for, or the different ways it could go wrong. Once you see solution generated by the LLM, you can then do a gut check and see if it matches your expectations. ○ Remember that testing and experimenting is free; running code is free. There's no harm in just trying the solution! ○ The more specific the task, or the longer the output the LLM is trying to generate, the more likely the errors are. Another way of saying that is the smaller the ask, the better the output. Try to get the LLM to write small chunks one at a time rather than an entire program in one go. That strategy also makes it easier for you to tell if the code is right, and if it's wrong where the problem is. ● Finally, remember that LLMs excel at creative, open-ended tasks. Asking for multiple options can help you decide on the best solution rather than relying on just one answer.
	Doing it ● Coursera interface ● ChatGPT canvas feature

🎙️ TH	Pair programming with an LLM can make you a much more effective programmer, and it can speed up your programming as well. In the upcoming practice lab, you'll practice prompting for pair programming with an LLM. Once you're done, follow me to the next lesson to start performing more rigorous analysis on data frames from selection to sorting and filtering to applying functions to all the values in a column. I'll see you there!

Selection, sorting, and filtering

L2V1 – Reading CSV files into a DataFrame

Visual	Script
🎙️ TH  Data structures and descriptive statistics Reading CSV files into a DataFrame DeepLearning.AI	Let's put Pandas to the test by examining some data about new programmers. You'll start loading the data into your Python notebook so you can analyze it.
🖥️ Screencast 🔗 C3m2l2v1 Importing CSV data 📂 Useful vis	Say you've been asked to investigate a survey of new programmers. You want to load the data into Python, explore it, and then determine any possible interesting research questions. You don't know anything about this data, so your first step is to check out the source. This is publicly available data. It's stored in GitHub, which is a website for sharing code. Don't worry too much about the specifics of GitHub, you're just investigating the data. The data was collected in 2016 by FreeCodeCamp and CodeNewbie. The data itself is located in the clean data directory. You can just go to clean data and the only CSV file here is 2016 FCC new coder survey data. So if you click on that, then you can download that file directly using this little download button. You can also see in the raw data folder, the survey questions. ● Are you already working as a software developer? ● How many hours do you spend learning each week? ● How many months have you been programming for? ● And so on. Once you've downloaded the file, you'll see that it has the same file name as before. This will be relevant in a moment. You can rename the file to something shorter, like survey data. Now, how do you get this file into your code? You won't need to do this step yourself, but just so that you're aware of how it works, you'll need to make sure that your survey data, or your CSV file, is moved into the same folder as

your notebook. The new coders analysis notebook lives in this notebooks folder, and now the CSV file is in the same folder. This will already be done for you in the labs for this course.

Because these two are in the same folder, you can access the CSV file with just the file name. Ultimately, you'll want to create a pandas data frame. So the very first thing you need to do: import pandas as pd. Then create a new data frame, df equals, this will store all your data, and use pd dot read csv. Read csv takes a csv file, like the one you just downloaded, and stores it inside your Python program as a dataframe. The CSV function takes at least one argument, which is the name of the file as a string. Survey Data fdot CSV. You saw a very similar line of code to this one earlier, when working with the library data. Shift enter. Now you have all your data read in.

Your first step once you load in data should be to check it out. For example, you can just type df and hit shift enter to take a look at your data frame.

Each row represents a set of survey answers from one person, and each column is one of the questions that the survey respondents had to answer. There are over 15,000 survey responses. You may also notice that some of these responses have NAN, not a number. Similar to the library data, those cells are empty. There's no information.

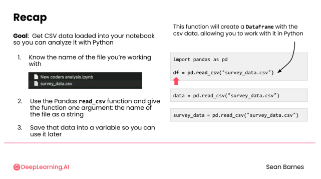
Next, df.columns. Remember, columns is an attribute, not a method. It’s a quick question like “what’s your name?” So no parentheses. You have age, gender, are they a software developer, marital status, months spent programming, and so on. You can also check out the Data types, another attribute. It looks like age is a float, which is a little interesting, maybe age should be an integer. Gender, country, and language are objects, which are strings for your purposes here. And there’s a bool for boolean for isSoftwareDev, so that should be either true or false.

One quick way to see the top n responses is to use the head method. So df. head and you can say how many rows you'd like to see. For example, maybe you want to see the first five rows. Shift enter. This is just another way to take a look at your data.

One thing you might notice is that the early rows have NAN for the number of children, while the later rows do have values, so it seems like maybe this data is sorted in some way.

What if you just wanted to see a few random rows in the data? You can use df.sample, which takes a random sample of n rows in the data. So head takes the first n rows, whereas sample takes a random sample of n rows. Let's say you want five random rows, shift enter, now you get row 13,269 and so on. So, just like in real life, a sample gives you a little more diversity in the responses.

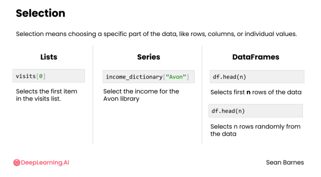
	You might already be thinking of some research questions. What does the distribution of age look like in this survey? How much money have people spent learning to code, or how many months have they spent programming? This column may help you see whether this is truly a survey of beginners. You may also be curious about some relationships between features in the data. What is the relationship between hours spent learning to code each week and the income that a person makes, or between the income a person makes and how many months they spent programming? Maybe the more months they spent programming, the more income they have. So there's a lot to explore here!


Recap
Goal: Get CSV data loaded into your notebook so you can analyze it with Python
1. Know the name of the file you're working with
New coders analysis.ipynb
survey_data.csv
2. Use the Pandas **`read_csv`** function and give the function one argument: the name of the file as a string
3. Save that data into a variable so you can use it later
This function will create a **`DataFrame`** with the csv data, allowing you to work with it in Python
`import pandas as pd`
`df = pd.read_csv("survey_data.csv")`
`data = pd.read_csv("survey_data.csv")`
`survey_data = pd.read_csv("survey_data.csv")`
DeepLearning.AI
Sean Barnes
	Let's quickly recap reading in CSV data using Pandas. Your **[CLICK]** goal is to get some CSV data loaded into your notebook so you can analyze it with Python. You'll need to **[CLICK]** know the name of the file you're working with, and it's easiest if that file is **[CLICK]** located in the same folder as your notebook. In the Coursera labs, this step will already be done for you.
	
	Then you'll **[CLICK]** use the Pandas read_csv function with the command `pd.read_csv()` and give the function one argument: the name of the file as a string, like **[CLICK]** "survey_data.csv". This function will create a data frame with the csv data, allowing you to work with it in Python. Your final step is to **[CLICK]** assign that data into a variable so you can use it later. In the demo, you saw the variable **[CLICK]** `df`, which is a common shorthand for data frame. You could also use **[CLICK]** `data` or **[CLICK]** `survey_data` or whatever other name makes sense to you.

🎙️ TH	Now that you have your data loaded in, you can start to investigate some of the questions you're interested in! Follow me to the next video to see how to select rows and columns as well as slices of your data!

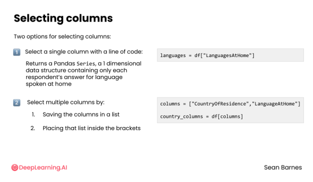
L2v2 – Selecting columns

Visual	**Script**
🎙️ TH	In your data analysis journey so far, you've often cut down a large dataset to work with only a portion of that data. You can do this operation in Python as well, and it's called selection.


Selection
Selection means choosing a specific part of the data, like rows, columns, or individual values.
Lists
`visits[0]`
Selects the first item in the visits list.
Series
`income_dictionary["Avon"]`
Select the income for the Avon library
DataFrames
`df.head(n)`
Selects first **n** rows of the data
`df.head(n)`
Selects n rows randomly from the data
DeepLearning.AI
Sean Barnes
	Selection means choosing a specific part of the data, like rows, columns, or individual values. You've already seen selection in action a few times.
	
	You've also seen selection using **[CLICK]** lists, with code like **[CLICK]** visits[0], which **[CLICK]** selects the first item in the visits list.

	With Python **[CLICK]** dictionaries, you used code like **[CLICK]** `income_dictionary["Avon"]` to **[CLICK]** select the income for the Avon library. And with **[CLICK]** dataframes, you saw: ● **[CLICK]** df.head(n), which **[CLICK]** selects the first n rows of the data ● **[CLICK]** df.sample(n), which **[CLICK]** selects n rows randomly from the data When working with data frames, you'll often need to select only specific columns or rows. Let's take a look, starting with columns.
🖥️ Screencast 🔗 [c3m2l2v2 selecting columns]()	One thing you may be interested in doing is selecting one column of the data. For example, if you want to see how many unique languages are spoken at home, you may only be interested in the "LanguageAtHome" column. In order to select the language at home column, you start with the name of the data frame, df. You will select the column using square brackets, just like you would from a list, a series, or a dictionary. Inside the brackets and in quotes, put the name of the column you want to select, in this case, LanguageAtHome. If you want to use the result later, you'll want to assign it into a variable: languages. Shift enter. Let's take a look at languages. What is the type of languages? Languages is a series. This is why series are so important, because each column in a data frame is a series. If you think about it, each column is just one dimension. Having more than one column is what gives you two dimensions in a data frame. Let's examine languages a little more. What do you expect to be the length of the languages variable? **[pause for thought]** Hitting shift enter, 15,620, the same as the number of responses in the data. Remember it's 15,619 plus the zeroth row gives you 15,620. What about the number of unique languages? You can say languages dot unique. Shift enter, and wow, there's really a lot of unique languages here, and some languages that are spoken by relatively small populations in the world, which is really, really fascinating. It looks like a pretty diverse, global survey. You can also select multiple columns. For example, maybe you want to analyze both CountryOfResidence and LanguageAtHome together to see how they're related. First, make a new variable, columns, and make a list of the columns you're interested in. Store them each as a string. Hitting shift enter, type of columns is just a list, and the length of columns is two, there's two items in the list. To select those two columns from the data frame, so you can say df, use brackets to select, and what are you going to select? Columns!

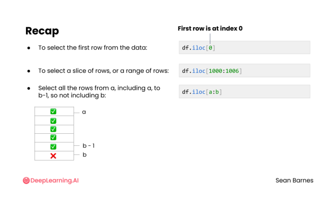
	Assign the result into a variable, like country columns. Shift enter. What is the type of country columns? This is a data frame. Why is it not a series? Because now you have two dimensions. You have two columns, not just one. If you execute countryColumns dot head, each row has both a country and a language spoken at home. That makes two dimensions.

 **Selecting columns**
Two options for selecting columns:
1️⃣ Select a single column with a line of code:
Returns a Pandas `Series`, a 1 dimensional data structure containing only each respondent's answer for language spoken at home
`languages = df["LanguagesAtHome"]`
2️⃣ Select multiple columns by:
1. Saving the columns in a list
2. Placing that list inside the brackets
`columns = ["CountryOfResidence","LanguageAtHome"]`
`country_columns = df[columns]`
DeepLearning.AI Sean Barnes
	To play that back, you have **[CLICK]** two options for selecting columns.
	● You can **[CLICK]** select a single column with a line of code like **[CLICK]** `df["LanguageAtHome"]`, which **[CLICK]** returns a Pandas Series, a 1 dimensional data structure containing each respondent's answer for language spoken at home. You can then save what's returned in a variable, like **[CLICK]** `languages`
	● You can also **[CLICK]** select multiple columns by **[CLICK]** saving the columns in a list, then **[CLICK]** placing that list inside the brackets

🎙️ TH	So that's selecting columns! Follow me to the next video to see how to select and examine individual rows and slices of rows in a data frame.

L2v3 — Selecting rows

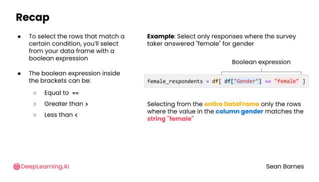
Visual	**Script**
🎙️ TH	While it's common to work with a subset of columns – one or two columns – you may also want to analyze a subset of rows in your data frame.
🖥️ Screencast 🔗 c3m2l2v3 selecting rows	What if someone asks you to verify the 1001st response and just double check if that response had an error or not? Maybe they're concerned that some values might be missing. Start with the name of the data frame, df.iloc. iloc stands for integer location. If you go back up and look at your data frame, on the left, you have these numbers that represent the index, or location, of the row, which is an integer. This is the default way that the index is assigned when you read in the data. Use square brackets to select the row that you want using ~~the~~ iloc ~~method~~, and you can enter 1000 to get the 1001st row. Remember, the index counts from 0, so 1000 gets you the 1001st row. You can say the response 1000 equals df dot iloc 1000. Shift Enter. Now you have that 1001st response, saved into this variable response 1000. What do you expect to be the type of response 1000? Hitting shift enter, this is also a series. It's relatively short because there's only one value for each column and there's around 20 columns or something like that. Hitting shift enter you can actually see this value. So this response was submitted by a 25 year old male, who lives in the USA, speaks English, and they spent about 20 hours learning to code a week. And so on.

	After investigating this row, you could respond to your colleague and let them know that there wasn't a major issue with this submission. Now, because response 1000 is a series, you can also select individual values from that series as well using the index on the left, like age, gender, and so on. So you can say, for example, response1000, age, shift enter, and that will give you 25. Another common task is for you to select a range of rows in your data frame. For example, say you wanted to select rows 1000 to 1005. Maybe your colleague was concerned that these rows might only have missing values. Maybe there was a server issue when those rows were being submitted. To select the rows, start with the name of the data frame, df and dot iloc for integer location, and add brackets to enable your selection. Start with the first row you want to select, in this case 1000, followed by a colon. Your selection is going to include this row. Then go all the way to the value one after the row that you want. This code may look a little strange. What this command does is it selects the rows starting at 1000, going up through 1001, 1002, 1003, 1004, and finally 1005. It will not select 1006. You can save this selection in a variable like range of rows. What is the type of range of rows? Shift enter, this is a data frame. Why? Because now you have two dimensions. If you just take a look at range of rows on its own, well, you essentially have a slice of the data frame. You have all of these rows from 1,000 to 1,005. And it doesn't look like any of these rows is full of missing values. This operation is called slicing the data frame, because you're cutting out a section of it to examine further. It's almost like cutting a slice of cake for yourself. You're gonna make two cuts to define the start and end of the slice, and you're gonna get that slice of cake.
 **Recap** • To select the first row from the data: `df.iloc[0]` • To select a slice of rows, or a range of rows: `df.iloc[1000:1006]` • Select all the rows from a, including a, to b-1, so not including b: `df.iloc[a:b]` First row is at index 0 a b - 1 b DeepLearning.AI Sean Barnes	Selecting rows can feel complicated as you're getting used to it, so to recap: ● You'll use a command like **[CLICK]** `df.iloc[0]` **[CLICK]** to select the first row from the data. Similar to lists, data frames are zero-indexed, so the counting starts at 0. The **[CLICK]** first row is at index zero. **[CLICK]** To select a slice of rows, or a range of rows, you'll use a very similar command, this time with a colon to select the range. So if you have a line of code like **[CLICK]** `df.iloc[1000:1006]`, that will select **[CLICK]** rows 1000 through 1005, not including 1006. More generally, if you have a command like **[CLICK]** `df.iloc[a:b]`, that will **[CLICK]** select all the rows from **[CLICK]** a, **[CLICK]** including a, to **[CLICK]** b-1, so **[CLICK]** not including b.

A note on slicing `df.iloc[1000:1006]` • Might seem counterintuitive that you type in the value 1006 when slicing, but you don't get that value • **One of the main benefits**: The length of the slice is always end minus start 1000 Start 1001 1002 1003 1004 1005 End 1006 1006 - 1000 = 6 ✅ Length of this slice is 6. ❌ If slice included last value, the length would be 7 DeepLearning.AI Sean Barnes	It **[CLICK]** might seem counterintuitive that you type in the value 1006 when slicing, but you don't get that value. Why is the last value not included in the slice? There are a few benefits, but **[CLICK]** one of the main ones is that the length of the slice is always end minus start. So for example **[CLICK]** 1006 - 1000 equals 6, so the **[CLICK]** length of this slice is 6. **[CLICK]** If the slice included the last value, the length would be 7, which is more awkward.
🎙️ TH	Great work slicing and dicing your data frame! Selecting rows and columns is the foundation of filtering your data. Follow me to the next video to see how.

L2v4 – Filtering

Visual	**Script**
🎙️ TH	You've now seen how to filter your data in spreadsheets – selecting rows that only meet a certain condition. How can you accomplish that task in Python?
🖥️ Screencast 🔗 c3m2l2v4 filtering	Check out these two tasks. First, you want to analyze the gender breakdown of the respondents. And second, you want to examine the respondents who spent more than 30 hours per week learning to code. Starting with the "Gender" column, the first thing you'll want to do is analyze the unique responses for gender. So first you can select the gender column dot unique. Shift enter. The options included male, female, genderqueer, trans, and agender. And NAN, which is not a number. These are people who left that field blank. You could filter the dataset to only include female respondents, just as an example. Start with the DataFrame, df. You're going to select from the DataFrame by adding brackets. What do you want to select? You want to select those respondents whose gender equals female. Let's fill in each of those parts. Start with female, which is the easier part. This is just a string. Make sure it's lowercase, your case here has to match the case of the data. Now, how do you produce the gender of each respondent? That would be df gender. So, what is this line of code going to do? It's gonna go through each response in the data frame. For that response, it's going to look at the value in the gender column. If the gender is equal to female, then it will include that row in the filtered data frame. If not, then it won't include that row. In this little subset here that you calculated earlier, the first row would not be included; and the second, third, and fourth rows would also not be included. Only this fifth response, 1004, would be included. Go ahead and save that result to a variable, for example, female respondents. Shift enter. What is the type of female respondents? It is a data frame. Let's take a look at the first 5 rows of that data frame. You can see that you've successfully filtered out any other genders. Notice that the original integer index is maintained. So you

	know that rows 0, 1, and 2 had a different value for gender, and so on. If you want to just do an analysis on the female respondents, now you can do that. You can do a couple of other calculations, too. So for example, the length of female respondents, 2,840. This value is the number of rows in this data frame. So not a whole lot: out of 15,000, there were about 3,000 female respondents. This operation is exactly the same as in a spreadsheet selecting a filter, clearing it, and only choosing the value female. Your next task is to examine respondents who spent more than 30 hours per week learning to code. The hours spent learning to code column name is a little long, so you can just copy that with Command or Control C. How can you select those people? Start with the data frame, df, and select rows in that data frame where the column, Hours Spent Learning to Code, is greater than 30. Then store that result in Above 30 Respondents. Shift Enter. What is the type of Above 30 Respondents? Well, it's a data frame. That makes sense. You can take a look at the first five rows. Already you can tell this is a little more rare than a female respondent because the first 43 rows are not included.. You can see hours spent learning of 56, 35, 40, 40, 40. These are the people who are spending a lot of time on coding! This first respondent is 19, so maybe they're studying in school in addition to some time on the side, and for these last three respondents, it's basically their full time job learning to code, and they are in their 30s. You can also check the length of this data frame, that'll give you the number of rows, about 1500. So it's a fair amount, right? If you have the length of this data frame over the length of the original data frame, about 10 percent of respondents are spending more than 30 hours a week learning to code, which is a pretty interesting finding. In a spreadsheet, this is a little more complicated of an operation. You'll select the column hours spent learning to code, filter by condition, greater than 30. Now you have those same first five respondents.
 **Recap** • To select the rows that match a certain condition, you'll select from your data frame with a boolean expression • The boolean expression inside the brackets can be: ○ Equal to `==` ○ Greater than `>` ○ Less than `<` **Example**: Select only responses where the survey taker answered "female" for gender Boolean expression `female_respondents = df[ df["Gender"] == "female" ]` Selecting from the **entire DataFrame** only the rows where the value in the **column gender** matches the **string "female"** DeepLearning.AI Sean Barnes	To summarize, in order to filter your data – that is, **[CLICK]** to select the rows that match a certain condition, you'll select from your data frame with a boolean expression. For example, you saw how to **[CLICK]** select only responses where the survey taker answered "female" for gender using the command **[CLICK]** `df[ df["Gender"] == "female"]`. You're **[CLICK]** selecting from the entire data frame only the rows where the value in the **[CLICK]** column gender matches the string **[CLICK]** "female".

	You also saw that **[CLICK]** the boolean expression inside the brackets can be **[CLICK]** equal to, **[CLICK]** greater than, it can also be **[CLICK]** less than. So you have a lot of options for filtering.
🖥️ Screencast 🔗 **C3m2l2v4 pt 2 filtering**	Take a look at this piece of code right here. You're selecting rows that match this boolean expression, so what does that actually do? First, save this result to the variable gender is female. If you look at the type of gender is female, it's a series. Interesting. Since this is a series, you can also slice that series by, for example, taking the first five values. Shift enter. The values in the series are false, false, false, true, true. So this expression creates a series of Boolean values. Each value is true if the gender in that row is female, and false in any other case. If you look at df.head, the original data frame, the genders were male, male, male, female, female. These genders match up with the booleans in the series. When you select from your data frame using this expression, the computer is going to look at each value in the Series one by one. If the value is true, it is going to include that row in the selection. If the value in the Series is false, it's not going to include that row; it's going to throw it out. And that's why in female respondents, the first two indices were 3 and 4. Those were the indices of the first two rows where the series had the value true.
🎙️ TH	Filtering gives you a lot of flexibility for analyzing subsets of your data. In the next video, you'll combine filtering with sorting, another powerful tool for analyzing your data. I'll see you there!

L2v5 – Sorting

Visual	**Script**
🎙️ TH	Sorting is an incredibly powerful operation that you can complete quickly and flexibly with Python. Let's take a look.
🖥️ Screencast 🔗 **c3m2l2v5 sorting**	You have a few more tasks to complete. First, sort the respondents by their age from youngest to oldest. Then, identify respondents who spent the most time per week learning to code. Then combine those two sorting conditions together to sort by age, then time spent learning to code. For the first task, start with your data frame. You'll use the method sort values. Now, by default, sort values is not going to change your existing data frame, it's going to create a new one. You want to store the result of this sorting operation in a variable. You can call this one sorted by age. Sort values takes at least one argument because you need to tell the computer how you want to

sort the data, by what column. You can also give it more arguments as well. Second, you could tell it in what order you want to sort: ascending–lowest to highest–or descending–highest to lowest.

Start by specifying the column by equals age. This looks a little different than some of the other arguments you've used before because you have this by equals. By, here, is called a “named argument”. Named arguments allow you to provide a function or method with the arguments in any order that you want. As you program in pandas more and more, you'll see this more and more commonly, since you won’t always remember which order the arguments are defined by default.

Here, you're sorting by age, and, ascending equals true. So you do want this to be in ascending order. Shift enter. What do you expect this result to be? Well, this should be a data frame, which it is, that's great. And it should be the same length as the original. You didn't remove any rows, you just changed the order.

Take a look at the head of this resulting data frame. How about the top 10 values? These are the youngest respondents: a couple who were 10 and a few who were 11. There are some very young people in this survey! It looks like this first 10 year old had spent over two years programming already. That’s pretty impressive.

In the sort values method, ascending equals true is actually the default value. Remember earlier you learned that sort values needs “at least one” argument. So the minimum is 1; you need “by equals”. So **this** line of code is actually equivalent to **this** line of code. If you don't include ascending equals, it will automatically default to true.

I'll just show you that in action. Shift enter, you’re recalculating sorted by age, this time with no ascending argument. Take the first 10 values, and these are sorted in ascending order, the same as the original result

Now for the second task. Identify respondents who spent the most time per week learning to code. Df dot sort values. Which column are you gonna sort by? You're gonna sort by hours spent learning to code. In this case, you want the highest values. There are a couple of ways to do that, but one convenient way is to say ascending equals false. You actually want the highest values first.

Store that result in sorted by hours. Shift enter. The type of this variable is a data frame. It should be the same length as the original, you didn't remove or add anything. Now, take a look at the first 10 values. It looks like the highest value was 100. 100 hours per week is a lot of time spent learning to code! That’s pretty interesting.

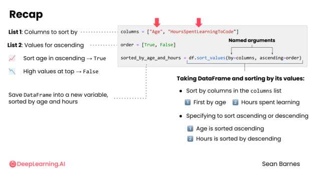
By the way, just for fun, how can you tell how many respondents reported 100 hours of coding per week? Select hours spent learning to code equals 100, and you can just find the length of that data frame, 34.

In order to sort by two values, You're going to use a similar approach to what you did for selecting by multiple columns. Instead of just having one value here, you're going to have a list of values. So, sorted by age and hours, df dot sort values. You use the same method to sort by one or more columns. The only thing that will change is your input to that method.

You're going to need an argument for by, and an argument for ascending. For the columns, first you want to sort by age, then you want to sort by hours spent learning to code. This is a list of length two. For the order, again, you're going to have a list because you have two columns. You want the age to be ascending order, and then you want the hour spent learning to code to be in descending order. This way, for each age, starting with the youngest, you can see who spent the most hours learning to code first, and then work your way down the list.

So, sortedByAgeAndHours is going to be df dot sort values by columns, ascending equals order. This is a really funky looking piece of code, you'll go through that step by step in a moment, but let's just hit shift enter for now, and what type do you expect this value to be? It should be a data frame, and again the length should be exactly the same as the original. You didn't add or remove anything, you're just moving things around.

What do you expect to see when you look at the top ten values? You should expect to see these first two rows in exactly the same order, because this individual had a higher hours spent learning to code than the next one. And then you expect to see a bunch of 11 year olds with a descending number of hours. Shift enter. The first two rows are in fact the same as before, but now you have a bunch of 11 year olds displayed in descending order of their hours spent coding, starting with 10 hours, 6, 4, down to not a number.



Recap

List 1: Columns to sort by

List 2: Values for ascending

📈 Sort age in ascending → `True`

📉 High values at top → `False`

Save `DataFrame` into a new variable, sorted by age and hours

```
columns = ["Age", "HoursSpentLearningToCode"]

order = [True, False]

sorted_by_age_and_hours = df.sort_values(by=columns, ascending=order)
```

Named arguments

Taking DataFrame and sorting by its values:

- Sort by columns in the `columns` list
  1️⃣ First by age 2️⃣ Hours spent learning
- Specifying to sort ascending or descending
  1️⃣ Age is sorted ascending
  2️⃣ Hours is sorted by descending

DeepLearning.AI Sean Barnes

Let's break down those lines of code together, because there's a lot happening:

```Python
columns = ["Age", "HoursSpentLearningToCode"]
order = [True, False]

sorted_by_age_and_hours = df.sort_values(by=columns,
ascending=order)
```

	First, you're creating two lists. The **[CLICK]** first list columns contains the columns you want to sort by. In this case **[CLICK]** age, then **[CLICK]** hours. The **[CLICK]** second list contains the values for ascending — in the first case, you want to **[CLICK]** sort age in ascending order so the value is true, but in the second case you want the **[CLICK]** higher values at the top so you sort with ascending as false. The third line, starting on the right. You're **[CLICK]** taking the data frame and sorting its values. You're **[CLICK]** sorting by the columns in the columns list — so **[CLICK]** first by age, then by **[CLICK]** hours spent learning. Then you're **[CLICK]** specifying whether you want to sort ascending or descending using the list order. So **[CLICK]** first, Age is sorted in ascending order, and then **[CLICK]** hours is sorted in descending order. The sort values method creates a new data frame, so you **[CLICK]** save that data frame in a new variable, sorted by age and hours. **[CLICK]** "By" and **[CLICK]** "ascending" are called **[CLICK]** named arguments. Named arguments allow you to give your arguments in any order. You'll often see them when working with Pandas.
🎙️ TH	Excellent work with sorting! There's only one more video left to go in this lesson. Follow me to the next video to really take advantage of vectorization and see how to apply many operations at once to columns in your data. I'll see you in the next video!

L2v6 – Applying functions

Visual	**Script**
🎙️ TH Data structures and descriptive statistics Applying functions DeepLearning.AI	Say you want to perform an operation on every value in a column at once. You have a powerful option for that task: the apply method.
🖥️ Screencast 🔗 [c3m2l2v6 apply]()	Say you've been given a new task to create a new years spent learning to code column based on the months spent learning to code, using the months to years helper function. Months to years takes an integer as input and converts it to a decimal representing the number of years spent. So, first you want to import your helper functions as hf, since you know this function is stored in the helper functions file. Shift Enter, and to check if months to years is working properly, you might want to first print hf dot months to years 12. That should give you 1. There you

go, 1. So this function is working as expected, and now you can go ahead and use that function on your data.

So when you think about the process of converting this entire column to years, heading over to your spreadsheet, how would you do that? You would probably insert a column, call it Years Spent Programming, and use a formula to divide by 12. Then double click the fill handle to fill that formula into all the other cells. In Python, you can perform a very similar process of applying this operation (dividing by 12) quickly and easily to every value in a column, in a vectorized way so that it happens super fast.

Heading back to your Python notebook, you can carry out this process using the apply function. First, remind yourself what the data looks like.

You're working with the Months Spent Programming function. So first you want to select that column, Months Spent Programming, and you can then use apply to perform some operation on every value in Months Spent Programming. What operation do you want to perform? Well, the argument to the apply method is the name of a function. In this case, hf dot monthsToYears.

What this line of code does is, you're selecting the column you want to do some operation on. You're using the apply method to perform some operation on every value in this series, so first on 1, then 2, then 24, and so on. What operation do you want to do on each of those? hf dot months to years.

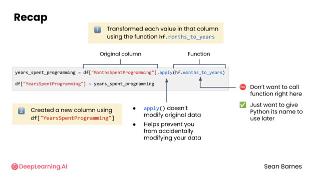
Now, similar to sort_values, which you learned about in an earlier video, apply does not, by default, modify your original data, but rather creates a new series. So you'll want to save that series in a variable, yearsSpentProgramming. Hit Shift Enter, and take a look at yearsSpentProgramming.

First the type, what type do you expect? **[pause for thought]** This is a series. That makes sense because you started with a series (df months spent programming). While you’ve changed the values in the series, you haven't changed the way that information is stored.

The length of years spent programming is the same as the length of the data frame. There's one value for each row. And take a look at the first ten values. These have changed, and you might want to compare with the original values. So first, for one month, that's 0.08 of a year. For two months, that's a little bit more than twice that, 0.17. And 24 months is two years. Great.

So what this line of code did up here was create a new series, this time with the values representing the number of years that that person had spent

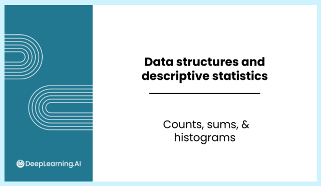
	programming. Your last step is to add this new information to your data frame. So, add a new column. Years spent programming equals the new series that you created, your spent programming. What should your data frame look like now? Well, the same information as before, except at the end, you now have a column, Years Spent Programming, that represents the number of years that that person spent programming.


Recap
1️⃣ Transformed each value in that column using the function `hf.months_to_years`
`years_spent_programming = df["MonthsSpentProgramming"].apply(hf.months_to_years)`
`df["YearsSpentProgramming"] = years_spent_programming`
2️⃣ Created a new column using `df["YearsSpentProgramming"]`
• `apply()` doesn't modify original data
• Helps prevent you from accidentally modifying your data
⛔ Don't want to call function right here
✅ Just want to give Python its name to use later
 Original column
Function
DeepLearning.AI
Sean Barnes
	To recap what just happened, you used two lines of code to add a new column to your data frame that was calculated from another column:**[CLICK]**
	
	Unset
	`years_spent_programming =`
	`df["MonthsSpentProgramming"].apply(hf.months_to_years)`
	`df["YearsSpentProgramming"] = years_spent_programming`
	
	First, you took the **[CLICK]** original column months spent programming and applied a **[CLICK]** function to it, meaning you **[CLICK]** transformed each value in that column using the function hf.months_to_years.
	
	Notice that **[CLICK]** hf.months_to_years, even though it's a function, doesn't have parentheses. This is pretty nuanced, but the reason is you **[CLICK]** don't want to call that function right here, you **[CLICK]** just want to give Python its name to use later. You're essentially saying, hey, here's a function, run it once for every value in this column.
	
	By default, **[CLICK]** apply **[CLICK]** doesn't modify your original data frame, which is a safety feature that **[CLICK]** helps prevent you from accidentally modifying your data. You looked at the result, checked if it's what you needed, and then **[CLICK]** created a new column using `df["YearsSpentProgramming"]`.

🎙️ TH	Great work using apply! It's one of the more abstract functions you'll use in this course, but incredibly useful. That brings you to the end of this lesson! You know your way around DataFrames and Series now, including selecting, sorting, filtering, and applying. You'll test your skills in the upcoming practice lab and practice assignment. Once you've finished, follow me to the final lesson in this module: descriptive statistics. I'll see you there!

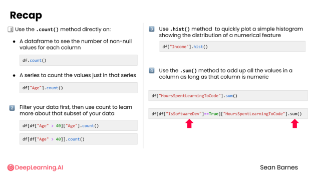
Descriptive statistics

L3v1 – Counts, sums, & histograms

Visual	Script
🎙️ TH  Data structures and descriptive statistics Counts, sums, & histograms DeepLearning.AI	Now that you've learned to sort, filter, and apply functions, let's get into some of your core descriptive statistics. You can get far with foundational descriptive analyses like counts and sums!
🖥️ Screencast 🔗 **c3m2l3v1 counts sums and histograms**	Let's continue on with your analysis of the new coder survey, which is the same data source you explored in the previous lesson. You might want to create a new notebook to store all of your descriptive statistics. You have a few tasks before you. You want to count and visualize the distributions of some different features, and sum up a couple of features as well. Start, as always, by importing pandas as pd and reading in the data into the variable df, for data frame. Remind yourself of what this data looks like using df dot head. For the first task, to count and visualize the distribution of age, you can start by selecting the age column directly. To visualize this distribution, you can use the method dot hist for histogram. Shift enter, and you get a nice visualization of the distribution of this feature. You will learn much more in depth how to make really beautiful histogram visualizations in the next module. So most respondents are in their 20s and 30s, and it is more of a skewed distribution. Now, it was possible on the survey for a respondent to leave these questions blank. For that reason, there are different numbers of null values in each column. So, NAN is null, or empty, which means there was no response for that answer. You can see, for example, that of these first four respondents, four of them did not share their marital status. So, first remind yourself of the number of rows: 15,620. To count the number of people who included their age, you can say df age, select the column, dot count. And that gives you 13,613. Now remember, you can also select using a condition. So for example, say you're interested in the number of people who were 40 or above, it's kind of this tail of the distribution. It's hard to gauge exactly how many people that may be. So you can select Df, where the value for the column age was greater than 40,

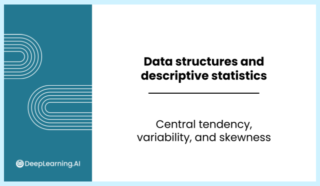
	Now, this entire thing gives you a data frame. So from this data frame, you want to select the age column. You're first filtering to any of the rows where the value for age is greater than 40. Then you're just going to grab the age column from there. You can use the method count on that age column, which gives you 1,500. The result is that there are 1,500 rows where the user's age was greater than 40. Another option was to find the length of that filtered data frame. That would have worked as well. Although you'll often want to examine a feature in detail, you can actually use count on the entire data frame. This shorthand will tell you the number of non null values for every column. So for age, you already calculated this value directly with age dot count. At a glance, it looks like learning via books had the least responses, presumably because not everyone learns via books. You can also see that marital status, student debt owed, and income had relatively few responses, as well as number of children, likely because not as many respondents had children or wanted to share that info. If you want to visualize the distribution of income, you can again use hist() since income is a continuous feature. You can see that overall it's a skewed distribution, which you might expect. For hours spent learning to code, you can also use hist. You can see that more than half of people spend between 0 and 20 hours learning to code. For a feature like marital status, you'll see in an upcoming video how to visualize categorical data quickly. Now, to sum the total hours spent learning and money spent learning to code. First select a column, dot sum, and you get that these respondents spend over 200,000 hours per week collectively learning to code, which is pretty cool. And for money spent learning to code, collectively respondents have spent over 16 million dollars. So, a lot of money overall. You can also sum conditionally. For example, if you wanted to sum the hours spent learning to code only for those who are a software developer, you could first select from df, where the is software dev column is true, select the hours spent learning to code column, and then sum. That gives you 57,000 hours each week for those who are already software developers. You may remember the sum function from working with lists, and you may think to use it directly on this column of the data frame, if you want to sum the money spent learning to code, but shift enter gives you nan, not a number. That's because the type of this column is a series, not a list. So, as soon as the computer sees "Nan", not a number, it's going to stop there and just say, well

	there's a missing value in your data structure and I don't know how to handle it. So .sum() will gracefully handle missing values — and sum all the other values — while the built in Python function sum() won't know how to handle these values.


Recap
1 Use the `.count()` method directly on:
• A dataframe to see the number of non-null values for each column
`df.count()`
• A series to count the values just in that series
`df["Age"].count()`
2 Filter your data first, then use count to learn more about that subset of your data
`df[df["Age" > 40]["Age"].count()`
`df[df["Age" > 40]].count()`
3 Use `.hist()` method to quickly plot a simple histogram showing the distribution of a numerical feature
`df["Income"].hist()`
4 Use the `.sum()` method to add up all the values in a column as long as that column is numeric
`df["HoursSpentLearningToCode"].sum()`
`df[df["IsSoftwareDev"]==True]["HoursSpentLearningToCode"].sum()`
DeepLearning.AI Sean Barnes
	To play that back, you can **[CLICK]** use the .count() method directly on **[CLICK]** a dataframe to see the number of non-null values for each column. Or you can use it on **[CLICK]** a series to count the values just in that series, for example by selecting a column first.
	
	You can also **[CLICK]** filter your data first, then use count to learn more about that subset of your data, either on **[CLICK]** an individual column or **[CLICK]** the whole subset.
	
	You also saw that you can **[CLICK]** use the hist method to quickly plot a simple histogram showing the distribution of a numerical feature. For categorical features, you'll need to use a different strategy which you'll see in a coming video.
	
	And finally, you can **[CLICK]** use the sum method to add up all the values in a column as long as that column is numeric. And you can **[CLICK]** select conditionally – so first, **[CLICK]** filtering your data – and then **[CLICK]** sum the column as well.

🎙️ TH	Counts and sums are the foundation of any analysis, and the hist method from Pandas will help you quickly visualize the distribution of your numerical features. Follow me to the next video to analyze distributions more thoroughly, including measures of central tendency, variability, and skewness.

L3v2 – Central tendency, variability, and skewness

Visual	**Script**
🎙️ TH  Data structures and descriptive statistics Central tendency, variability, and skewness DeepLearning.AI	Now that you've investigated counts and sums in your data, you'll want to get more information about the distribution of each numerical feature.
🖥️ Screencast 🔗 [c3m2l3v2 central tendency, variability, and skew]()	The next step in your analysis might be to describe the distributions of the hours spent learning to code and age features. Let's start with hours spent learning to code. Remember that dot hist allows you to generate a histogram. Here's a quick reminder of what that distribution looks like. You can now calculate your descriptive statistics.

First off, this is a really long column name, so you may want to save this into a new variable. You're essentially giving it a nickname. So instead of typing this long column name every time, now you can just use the hours variable. So let's find hours dot mean. The mean value is around 15 hours per week, which makes sense; that seems to be the center of mass here.

You can also check the median, which is 10. That makes sense, because in a skewed distribution, the mean is pulled more towards the tail of the distribution, in this case towards the higher values. You also have hours dot std to get the standard deviation, which is around 14. Not quite as meaningful with a skewed distribution as it would be for data that was a little more normally distributed.

Variance is calculated using the var function. Variance is the standard deviation squared. It’s useful for calculating all kinds of other statistics. All of these common descriptive statistics are very, very straightforward to calculate. You don't need any arguments in these functions, you can just call them directly on your column. As a quick reminder, all of these calculations will generate the same results as using the name of the column directly.

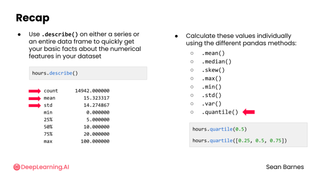
You may also want to get your quantiles, which essentially your percentiles. Remember, quantiles define specific milestones in the distribution of your data. For example, you can have hours dot quantile 0.5. What value do you expect this to be? **[pause for thought]** That should be the median, 10. 50 percent of the data is above, 50 percent of the data is below that value.

Now, say you're constructing a boxplot and you want to get the first 25 percent of the data, the first 50 percent of the data, and the first 75 percent of the data. You can create a list of those values here, and instead of giving the quantile method this one value, 0.5, you can give it a list of values as well. So you can say hours dot quantile, quantiles. You’re using this list right here as the argument to this method, which finds the 25th, 50th, and 75th percentiles. Nice, clean numbers there.

What's cool about all of these descriptive statistics is you can get them pretty easily with hours dot describe(). Shift enter, and that gives you a lot of statistics right away. How many values did you have, what was the mean, standard deviation, the min, max, the different quantiles including the median. That's a super nice shorthand.

For age, you can apply the describe method after selecting the column. You can see there are fewer responses here than for the hours spent learning. The average person was 30 years old. The median person was 27, so again a little bit of a skewed distribution, the standard deviation is smaller, and it's in years, and so on. The oldest person here was 86. The youngest was 10, as you've

	already identified. You may notice that skewness is not included in dot describe; however, you can use the dot skew method. This gives you the same result, same interpretation as the skew function in a spreadsheet. For hours dot skew, that gives you a value of 2. So the distribution of hours spent learning to code is more skewed than the distribution of the values in the age column. You've now seen the describe method applied to a series. Remember that the type of any given column is a series. It's one dimensional. You can also use describe on a data frame. You're about to get a lot of information! Df dot describe gives you a really nice looking table of all of the features and their distributional statistics, as long as they were numerical features. You can't calculate the standard deviation of a categorical feature. So there's a lot to investigate here, and you can get a lot of information quickly.

 **Recap**
• Use `.describe()` on either a series or an entire data frame to quickly get your basic facts about the numerical features in your dataset
`hours.describe()`
`count 14942.000000`
`mean 15.323317`
`std 14.274867`
`min 0.000000`
`25% 5.000000`
`50% 10.000000`
`75% 20.000000`
`max 100.000000`
• Calculate these values individually using the different pandas methods:
○ `.mean()`
○ `.median()`
○ `.skew()`
○ `.max()`
○ `.min()`
○ `.std()`
○ `.var()`
○ `.quantile()`
`hours.quartile(0.5)`
`hours.quartile([0.25, 0.5, 0.75])`
DeepLearning.AI Sean Barnes
	Let's recap the tools you have available for central tendency, variability, and skewness for numerical features:
	● You can **[CLICK]** use .describe() on either a series or an entire data frame to quickly calculate descriptive statistics about the numerical features in your dataset, including **[CLICK]** count, **[CLICK]** mean, **[CLICK]** median, and so on.
	● You can also **[CLICK]** calculate these values individually using the different pandas methods available to you: **[CLICK]** mean, **[CLICK]** median, **[CLICK]** skew, **[CLICK]** max, **[CLICK]** min, **[CLICK]** std, **[CLICK]** var, and **[CLICK]** quantile
	● Most of these methods takes **[CLICK]** no arguments, except for the **[CLICK]** quantile method, for which you'll specify either **[CLICK]** one, or a **[CLICK]** list to get multiple quantiles at once.

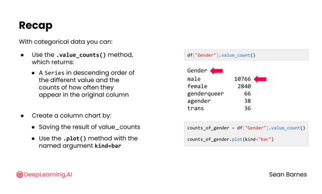
🎙️ TH	Think of how powerful the .describe() function is. You can get so much information in just a snap of your fingers! Once you've dealt with your numerical features, you'll want to move on to your categorical ones. Follow me to the next video to see how.

L3v3 – Categorical data

Visual	**Script**
🎙️ TH **Data structures and descriptive statistics** Categorical data DeepLearning.AI	You'll often be working with categorical or non-numeric data, which is analyzed differently from numeric data. Python gives you a couple of key tools to perform those analyses, including the value counts method and plotting.

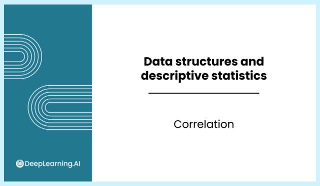
🖥️ Screencast 🔗 **c3m2l3v3 categorical data**	In working with your categorical features, say you've been asked to describe the distribution of the gender feature and find the top five languages that are spoken at home. Now, if you don't have all the column names memorized, you can use df dot d-types to remind yourself which columns are actually categorical, which includes any column that's anything that's not numeric. So gender, country of residence, language at home, is software dev, and marital status. You'll want to first select the column you want to do some analysis on, which is the gender column. Your main angle of attack for categorical data is going to be the value counts method. Shift enter. Value counts gives you every unique value in the Series and how many times it occurred. Males made up about 78% of non-null responses; females made up about 21%; and the three gender diverse answers made up around 1%. You can see that male was about three times more likely than the next most common response, female, and the three non-binary genders were not strongly represented in the data. You can also graph the distribution of gender. Because you're going to be using these value counts to generate plots and maybe do some other analyses, you'll want to save those to a variable so you can use them later. Counts of gender, equals, and then that exact same command that you used before, shift enter. Look at the type of counts of gender. It's a series. One thing you can do with a series is plot it in a bar plot. So, when you had a numeric column, you did something like df income dot hist. For a non numeric column, these are just comments, so that you can compare what happens with numeric and non numeric Series, you're gonna take counts of gender dot plot. Now, you need to tell pandas a little more information than that; specifically, with a named argument, what kind of plot you wanna make. In this case, a bar plot. So, Shift Enter, and you get this straightforward graph showing exactly the information you just saw, but in a very digestible way. You might say, well, this is a column chart, not a bar chart, that's just what it's called. If you try to say kind equals column, you get this scary looking red text. Column is not a valid plot type for this method. You can use line, bar, and all these other options that you'll take a look at in the next module. Don't be afraid to make a mistake! If you put column and you get an error, you can ask an LLM, you can try to read through that error. Use your strategies! Next up, you want to find the top five languages spoken at home. Let's say counts of languages equals df. Select the column you want to work with,

	language at home, dot value underscore counts. Remember the type of counts of languages. Oops, I almost typed kangaroos. Counts of languages is a series. You can do anything with counts of languages that you would do with a series. For example, you can take a look at counts of languages dot head, the first five of these. You can see that English is by far the most commonly spoken language at home, almost ten times more common than the second most common answer, Spanish, with Russian, Hindi, and Portuguese rounding out the top five. What else can you do with a series? Well you can check the length of it. What would the length of this series tell you? **[pause for thought]** Well it would tell you how many different languages appear in this column. 148. So there are 148 different languages that were entered into this survey as a language spoken at home. Now, if you want to visualize counts of languages, you might think to run dot plot, kind equals bar. Check it out. Shift enter. Okay. It gave you exactly what you asked for, which is this huge bar chart of 148 different languages. It's really challenging to come to any kind of conclusions from this graph, although it is cute that Pandas tried. Since you're just interested in the top five languages here, you can slice your series just like you would with any other series. So if you want the top five languages, you can say counts of languages, and then select the first five values from that series, 0 inclusive to 5 exclusive. And there you go! You have the top five languages spoken at home. The type of Top 5 Languages is also a series. After all, a slice of cake is still cake. So now you can go ahead and plot it. Top 5 Languages dot plot, kind equals bar, and now you get a nice, simple bar chart showing the conclusions about the languages spoken at home.


Recap
With categorical data you can:
• Use the **`.value_counts()`** method, which returns:
  • A `Series` in descending order of the different value and the counts of how often they appear in the original column
• Create a column chart by:
  • Saving the result of value_counts
  • Use the **`.plot()`** method with the named argument **`kind=bar`**
`df["Gender"].value_count()`
`Gender`
`male        10766`
`female       2840`
`genderqueer    66`
`agender        38`
`trans          36`
`counts_of_gender = df["Gender"].value_count()`
`counts_of_gender.plot(kind="bar")`
DeepLearning.AI
Sean Barnes
	So, with categorical data, you can use the value_counts method, which returns a Series in descending order of the different values and the counts of how often they appear in the original column.
	
	You can also create a column chart by first saving the result of value_counts, then using the .plot method with the named argument kind equals bar.

🎙️ TH	Once you've wrapped your mind around the distributions of your categorical features, you can dig deeper into the relationships in your data. Follow me to the next video to see how to do that.

L3v4 – Correlation

Visual	Script
🎙️ TH  Data structures and descriptive statistics Correlation DeepLearning.AI	Correlation is your go-to for quantifying the strength and direction of the relationship between two numerical features. Pandas allows you to quickly calculate correlations between multiple features, as well as create scatter plots.
🖥️ Screencast 🔗 **c3m2l3v4 correlation**	Say you want to find the correlation between income and money spent learning to code. Maybe you think that the more money someone has, the more money they are able to spend on programming. You may also be interested in the correlation between age and months spent programming. Maybe the older someone is, the more time they've spent programming. Typically when you're looking at correlations, the first thing you want to do is look at a scatterplot. You can remind yourself of the columns that you're working with here. Previously, when you were plotting a histogram and a bar chart, you started with the series, like the column, because you only needed to plot one column. In a scatterplot, you need to plot two columns, so you want to start with the data frame. That way you have a reference to multiple columns. DF dot plot, same function as you use for a bar chart. Kind is going to be scatter. And here comes the new part. You have two new named arguments. x, which in this case you can go ahead and start with income. X is income and Y is money spent learning to code. So X is whatever column is going to be on your x axis and you just need to specify the column name. You don't need to select the column because you already have a reference to the data frame and you're just kind of pointing the computer to which column you want it to graph. Shift Enter. Looking at the resulting plot, maybe this is not the pattern you were expecting? It doesn't seem like income on the X axis has too much of an effect on income on the Y axis. Most of the values are relatively low. For the second scatterplot you were interested in, just copy and paste that code, df dot plot, kind equals scatter, and now you want to visualize the relationship between age and months spent programming. Shift Enter. This plot also has quite an interesting relationship. You can see that a lot of people have zero months spent programming, they're just starting out. But there does seem to be some kind of positive relationship here. In order to calculate the Pearson correlation coefficient, you're going to use the pandas dot core method. The core method is used with a data frame, not

an individual column, because you need two columns to find the correlation!

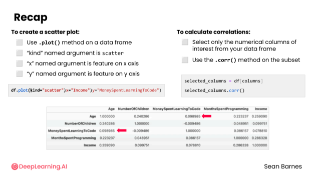
Similar to the describe method, when you call core on a data frame, it will show you the correlations between all pairs of features. So, one thing you might want to do is just select all the numerical features you want, and then you can see how they all correlate with each other.

So, first, create a list of the numerical columns you're interested in. Maybe age, number of children, just for fun so you can see what's there. Money spent learning to code, months spent programming, and income. So the type of the variable columns is a list. You're just creating a list of columns.

Now you want to select those columns from your data frame. So just like you would select any individual column with this command by saying df age, you also saw that you can select a list of columns. Quickly, the type of selected columns is what? **[pause for thought]** Well, it's a data frame. And that's because, if you look at the first five rows, you have both rows and columns. So, you have two dimensions, it's not just one column. And because SelectedColumns is a data frame, you can use the core method. Selected columns works because it only has numerical data; if there were categorical columns here that would cause an error.

Shift enter and you get this big table of results. Diagonally down the middle, all the features have a correlation of 1 with themselves. So age and age: 100 percent correlated. But there's some more interesting results here. You were first interested in income and money spent learning to code. There's a pretty weak correlation of about 0.08. So the more your income goes up, the more money you spend learning to code, but it only explains about 8 percent of that relationship.

You are also interested in age and month spent programming. Here's the result showing the correlation between age and months spent programming. That has a stronger, but still weak, correlation. About 22 percent of the variability in months spent programming can be explained by age. So, as your age goes up, you spend more time programming. Makes complete sense.



Recap

To create a scatter plot:
- Use `.plot()` method on a data frame
- "kind" named argument is `scatter`
- "x" named argument is feature on x axis
- "y" named argument is feature on y axis

```
df.plot(kind="scatter")x="Income")y="MoneySpentLearningToCode")
```

To calculate correlations:
- Select only the numerical columns of interest from your data frame
- Use the `.corr()` method on the subset

```
selected_columns = df[columns]
selected_columns.corr()
```

	Age	NumberOfChildren	MoneySpentLearningToCode	MonthsSpentProgramming	Income
Age	1.000000	0.240286	0.098985	0.223237	0.259090
NumberOfChildren	0.240286	1.000000	-0.009486	0.048951	0.099751
MoneySpentLearningToCode	0.098985	-0.009486	1.000000	0.086157	0.078810
MonthsSpentProgramming	0.223237	0.048951	0.086157	1.000000	0.286328
Income	0.259090	0.099751	0.078810	0.286328	1.000000

DeepLearning.AI

Sean Barnes

When investigating relationships, **[CLICK]** to create a scatter plot you'll want to **[CLICK]** use the dot plot method on a data frame. The **[CLICK]** "kind" named argument is scatter, the **[CLICK]** "x" named argument is whatever feature you want on the x axis, and the **[CLICK]** "y" named argument is whatever feature you want on the y axis.

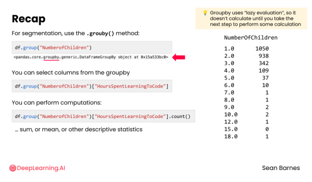
[CLICK] To calculate correlations, you call .corr() on a data frame. If you try to use a dataframe with categorical columns, you'll get an error. So you'll first want to **[CLICK]** select only the numerical columns of interest from your data frame. Then you'll use the **[CLICK]** dot corr method on that new data frame,

	the subset, and that will give you **[CLICK]** this table of correlations. Notice that some of the correlations are **[CLICK]** redundant — half of them to be precise. Correlation is a symmetric operation, so it doesn't matter the order in which you select the columns, the correlation calculated is the same. And think about how much effort this correlation table saves you from manually entering this formula in so many different cells!
🎙️ TH	You're almost to the end of this module! You've learned so many useful methods in Pandas, from reading in data to selecting, sorting, filtering, and calculating descriptive statistics, as well as plotting histograms, column charts, and scatter plots. Follow me to the two videos that cover the final topic in this module: segmentation.

L3v5 – Segmentation by one feature

Visual	**Script**
🎙️ TH	Segmenting your data allows you to see patterns in one feature based on another. It's a powerful tool for analyzing how features interact. Let's take a look at segmentation in Python with Pandas.
🖥️ Screencast 🔗 **c3m2l3v5 segmentation by one feature**	New task. Say you're interested in determining whether hours spent learning to code varies by the number of children that a person said that they had, and, by the number of children **and** whether that person is a software developer. In the first case, you want to segment by one feature, and in the second case, you want to segment by two features. In this video, you'll tackle the first task. Take a look at your data frame. You want to segment by one column, the number of children, and then analyze the hours Spent Learning to Code. Since you need access to both of those columns, you're going to start with the entire data frame. You don't want to start with a series because then you'll be missing one column. So you're going to say df dot group by, by the number of children. This line of code partitions your data into subsets based on the values for number of children. So what happens when you hit Shift Enter? You'd think you might be able to see a table like you did with your data frame, but you just get this pandas dot core dot group by dot generic blah blah blah. What you're looking at here is the **type** of the result of this command. There's nothing to display yet because GroupBy is what's called a lazy evaluation. It doesn't actually compute

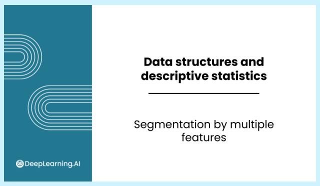
	anything until you take the next step to run some calculation. Select the Hours Spent Learning to Code column, your outcome of interest. And then maybe you want to do a count. So you need to execute a couple of intermediate steps here to generate any results, including selecting the column you want to explore and applying a function to calculate some type of descriptive statistic. Here, you have the hours spent learning to code segmented based on the number of children. There were about a thousand survey respondents who had one child, 938 people who had two children, and so on. After four children, the responses are pretty sparse, and after six, you may as well have had no responses, since there aren't any meaningful insights to conclude from 1 or 2 responses. What may be more interesting to you is the mean hours spent learning to code. For someone with one child, it was 14.1. For two children, it was a little lower, 12.4. Then, about 15, it went back up. Then, 13.8. And, after four children, maybe there's a little bit less valuable information to evaluate due to the small sample sizes within these categories. Overall, there's definitely a drop between one and two children, and after that, maybe it picks back up.


Recap
For segmentation, use the **`.grouby()`** method:
`df.group("NumberofChildren")`
`<pandas.core.groupby.generic.DataFrameGroupBy object at 0x15a533bc0>`
You can select columns from the groupby
`df.group("NumberofChildren")["HoursSpentLearningToCode"]`
You can perform computations:
`df.group("NumberofChildren")["HoursSpentLearningToCode"].count()`
... sum, or mean, or other descriptive statistics
💡 Groupby uses "lazy evaluation", so it doesn't calculate until you take the next step to perform some calculation
`NumberOfChildren`
`1.0     1050`
`2.0      938`
`3.0      342`
`4.0      109`
`5.0       37`
`6.0       10`
`7.0        1`
`8.0        1`
`9.0        2`
`10.0       2`
`12.0       1`
`15.0       0`
`18.0       1`
DeepLearning.AI
Sean Barnes
	For segmentation, you can use the groupby method, starting with your data frame. The type of the **[CLICK]** result of this call is a **[CLICK]** groupby object, so it's not a data frame, but it does have some similarities. For example, **[CLICK]** you can select columns from the group by object, like the **[CLICK]** Hours spent learning to code column you saw in the demo a moment ago.
	
	Once you've selected a column, **[CLICK]** you can perform computations like **[CLICK]** calculating the count, **[CLICK]** sum, mean, or other descriptive statistics, which will then be **[CLICK]** displayed summarized by each group.
	
	[CLICK] Groupby uses what's called "lazy evaluation", so it doesn't calculate any means or counts or even the different groups until you actually take the next step to perform some calculation, like selecting a column and calculating the mean or sum yourself.

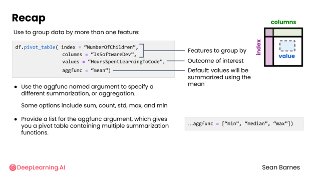
🎙️ TH	So that's segmentation by one feature, which is useful in its own right. If you want to segment by multiple features, however, you'll need to use a different method – a pivot table in Python! Follow me to the next video to see how.

L3v6 – Segmentation by multiple features

Visual	**Script**

🎙️ TH  Data structures and descriptive statistics Segmentation by multiple features DeepLearning.AI	As a final task for describing your data, you may be interested in segmenting by multiple features at once. Let’s take a look at how to do that in Python with Pandas.
🖥️ Screencast 🔗 **c3m2l3v6 segmentation by multiple features**	If you want to segment your data by two categorical features, you're going to want to use a PivotTable. Coming back to the spreadsheet just to refamiliarize yourself, you have two categorical features: number of children and is software dev. Select your data, insert, pivot table. For rows, you want the number of children, and that gives you the same values that you just saw in the previous video. For columns, you have isSoftwareDev, false or true. And for values, hours spent learning to code. The default summarization function is SUM, but you’re more interested in the AVERAGE. Focusing on the first four rows where you have the most data, you have the mean value for people who are not software developers and have one child, people who are software developers and have one child, as well as the overall mean, which is a little more heavily weighted towards the non software developers. There's probably more of them. You can carry out a very similar process very quickly in Python using pandas. I'll just show you that method: PivotTable. PivotTable works very similarly to how it does in a spreadsheet. You assign rows, columns, and values. Remember, you're interested in segmenting by the number of children and isSoftwareDev. The first argument of the pivot table method is called index, but it corresponds to the row. Index will be the number of children. Then for columns, you have isSoftwareDev. And for values, you're interested in hoursSpentLearningToCode. Shift enter. The default aggregation function is the mean, so you're seeing the mean value for each combination of the number of children and whether someone is a software developer. So if someone was not a software developer with one child, they were spending about 14 and a half hours a week learning to code. If they were a software developer, they were spending about 13.4 hours. So it looks like that is likely a significant difference given that n is so high for this number of children. You see the same pattern pretty much across all the data. Going back to your pivot table in a spreadsheet, you had all these options for the summarization function. The pivot table method also has different options for summarization, and these are in the named argument, agg Funk. Funny, I know! It means aggregation function, and the default is the mean. So, if you say mean, you get the exact same result.

	You can also do sum, and instead you get the total number of hours grouped by these two features. Or you could do STD for standard deviation. There doesn't seem to be quite a clear pattern here, but it does weed out the groups with only one respondent. Finally, if you're interested in multiple values, for example, maybe you're interested in the min, median, and max for each of these groups. You could create a pivot table with all of those agg functions because agg funk can be a list. Aggregations is the variable containing the list of functions you're interested in. So shift enter. Now you get the different minimum values. So for each group, the minimum time spent learning to code for the most common groups was zero. Then you have the median in the center two columns. It's roughly similar for these distributions, suggesting that some extreme or outlier values were shifting those means in different directions. And then you have the maximum.

 **Recap**
Use to group data by more than one feature:
`df.pivot_table( index = "NumberOfChildren", columns = "IsSoftwareDev", values = "HoursSpentLearningToCode", aggfunc = "mean")`
Features to group by
Outcome of interest
Default: values will be summarized using the mean
• Use the aggfunc named argument to specify a different summarization, or aggregation.
Some options include sum, count, std, max, and min
• Provide a list for the aggfunc argument, which gives you a pivot table containing multiple summarization functions.
`...aggfunc = ["min", "median", "max"])`
 	
DeepLearning.AI
Sean Barnes
	Recapping the pivot table method, you'll put it to **[CLICK]** use when you want to group your data by more than one feature. Just like a **[CLICK]** pivot table in a spreadsheet has **[CLICK]** rows, **[CLICK]** columns, and **[CLICK]** values, the pivot table method has **[CLICK]** index, **[CLICK]** columns, and **[CLICK]** values. Values should be your **[CLICK]** outcome of interest, and index and columns should be the **[CLICK]** features you want to group by.
	
	By **[CLICK]** default, values will be summarized using the mean. However, you can use the `aggfunc` named argument to specify a different summarization function, or aggregation. **[CLICK]** Some options include sum, count, std, max, and min.
	
	You also saw that you can provide a list for the `aggfunc` argument, which gives you a pivot table containing multiple summarization functions.

🎙️ TH	That takes you almost to the end of this module on data structures and descriptive statistics. You've seen how to read in, store, and manipulate your data to produce powerful summaries. You've also learned many Python subtleties like the difference between attributes and methods, and how to use named arguments. Once you complete the practice assignment and practice lab for this lesson, where you'll continue to work with the Buenos Aires subway data, you'll complete the graded assignment and graded lab for this module. In the graded lab, you'll expand your analysis of the retail sales data from the previous module. I can't wait for you to put into practice all the cool Python methods you've learned in this module. Once you're done, I'll see you in the next module, which covers time series analysis and data visualization. See you there!