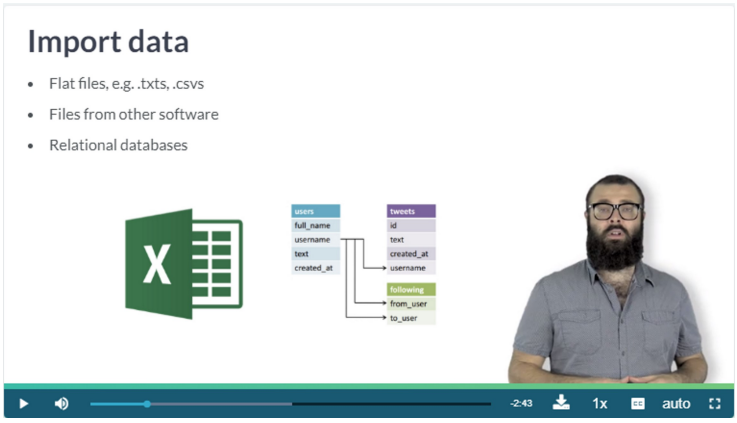
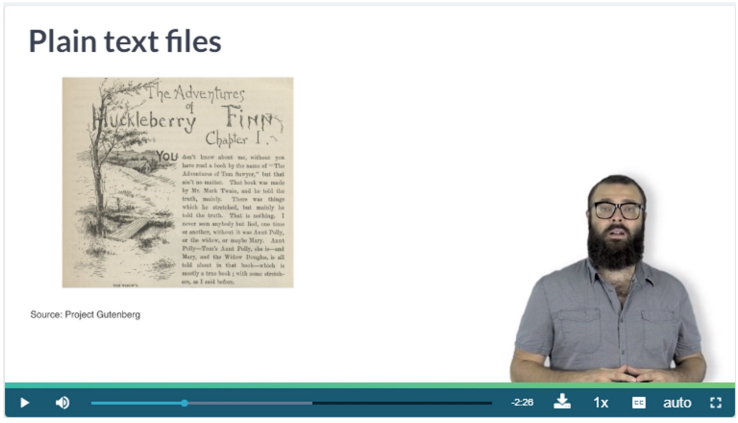
05. Introduction to Importing Data in Python

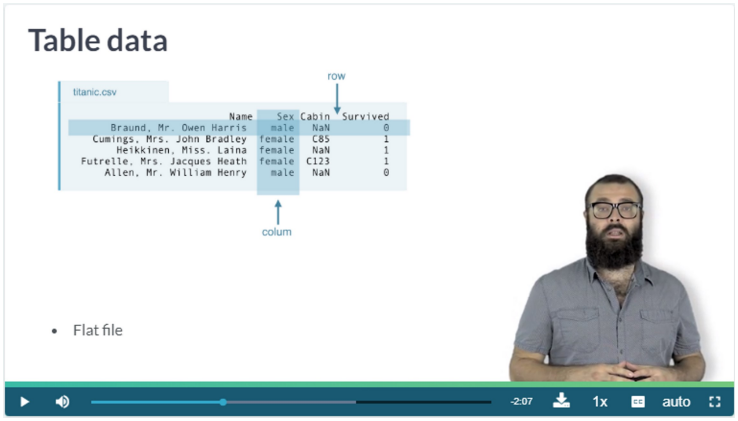
13 January 2020 14:57

1. Introduction and flat files

1.1 Welcome to the course!







Reading a text file

```
filename = 'huck_finn.txt'
file = open(filename, mode='r')  # 'r' is to read
text = file.read()
file.close()
```

Printing a text file

```
print(text)
```

```
YOU don't know about me without you have read a book by
the name of The Adventures of Tom Sawyer; but that
ain't no matter. That book was made by Mr. Mark Twain,
and he told the truth, mainly. There was things which
he stretched, but mainly he told the truth.  That is
nothing. never seen anybody but lied one time or
another, without it was Aunt Polly, or the widow, or
maybe Mary. Aunt Polly--Tom's Aunt Polly, she is--and
Mary, and the Widow Douglas is all told about in that
book, which is mostly a true book, with some
stretchers, as I said before.
```

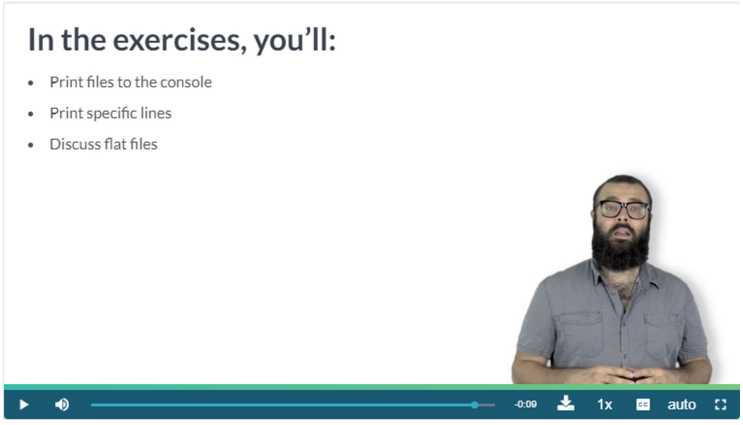
Writing to a file

```
filename = 'huck_finn.txt'
file = open(filename, mode='w')  # 'w' is to write
file.close()
```

Context manager with

```
with open('huck_finn.txt', 'r') as file:
    print(file.read())
```

```
YOU don't know about me without you have read a book by
the name of The Adventures of Tom Sawyer; but that
ain't no matter. That book was made by Mr. Mark Twain,
and he told the truth, mainly. There was things which
he stretched, but mainly he told the truth.  That is
nothing. never seen anybody but lied one time or
another, without it was Aunt Polly, or the widow, or
maybe Mary. Aunt Polly--Tom's Aunt Polly, she is--and
Mary, and the Widow Douglas is all told about in that
book, which is mostly a true book, with some
stretchers, as I said before.
```



1.2 Exploring your working directory

In order to import data into Python, you should first have an idea of what files are in your working directory.

IPython, which is running on DataCamp's servers, has a bunch of cool commands, including its **magic commands**. For example, starting a line with `!` gives you complete system shell access. This means that the IPython magic command `! ls` will display the contents of your current directory. Your task is to use the IPython magic command `! ls` to check out the contents of your current directory and answer the following question: which of the following files is in your working directory?

Possible Answers

- [ ] huck_finn.txt
- [ ] titanic.csv
- [x] moby_dick.txt

```
In [1]: !ls
moby_dick.txt
```

1.3 Importing entire text files

In this exercise, you'll be working with the file `moby_dick.txt`. It is a text file that contains the opening sentences of Moby Dick, one of the great American novels! Here you'll get experience opening a text file, printing its contents to the shell and, finally, closing it.

Instructions:

- Open the file `moby_dick.txt` as *read-only* and store it in the variable `file`. Make sure to pass the filename enclosed in quotation marks `''`.
- Print the contents of the file to the shell using the `print()` function. As Hugo showed in the video, you'll need to apply the method `read()` to the object `file`.
- Check whether the file is closed by executing `print(file.closed)`.
- Close the file using the `close()` method.
- Check again that the file is closed as you did above.

```
# Open a file: file
file = open('moby_dick.txt', mode = 'r')

# Print it
print(file.read())

# Check whether file is closed
print(file.closed)

# Close file
file.close()

# Check whether file is closed
print(file.closed)
```

1.4 Importing text files line by line

For large files, we may not want to print all of their content to the shell: you may wish to print only the first few lines. Enter the `readline()` method, which allows you to do this. When a file called `file` is open, you can print out the first line by executing `file.readline()`. If you execute the same command again, the second line will print, and so on.

In the introductory video, Hugo also introduced the concept of a **context manager**. He showed that you can bind a variable `file` by using a context manager construct:
`with open('huck_finn.txt') as file:`
While still within this construct, the variable `file` will be bound to `open('huck_finn.txt')`; thus, to print the file to the shell, all the code you need to execute is:

```
with open('huck_finn.txt') as file:
    print(file.readline())
```

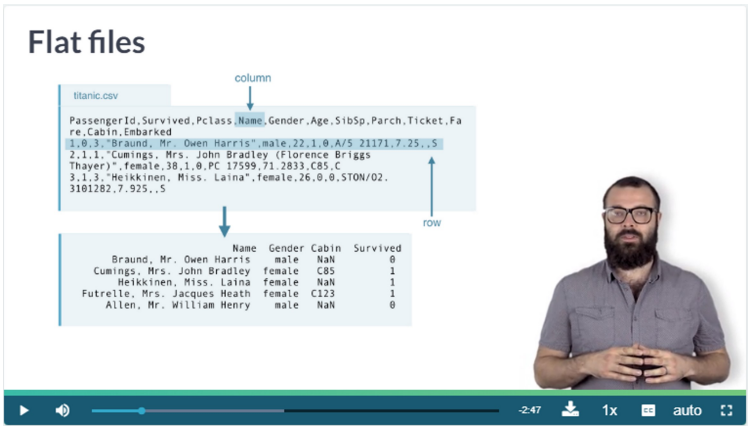
You'll now use these tools to print the first few lines of `moby_dick.txt`!

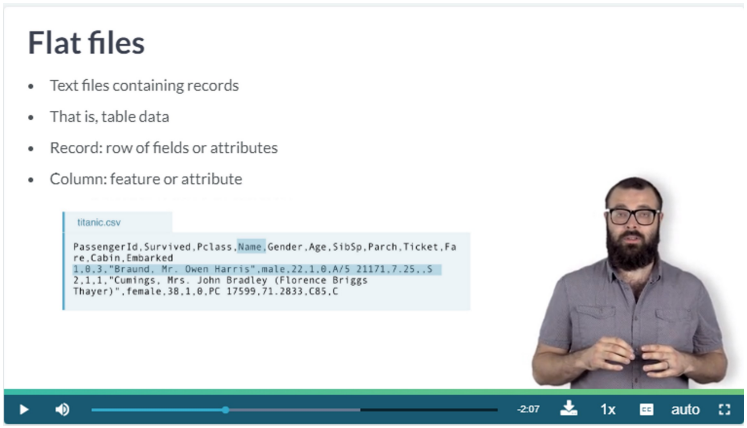
Instructions:

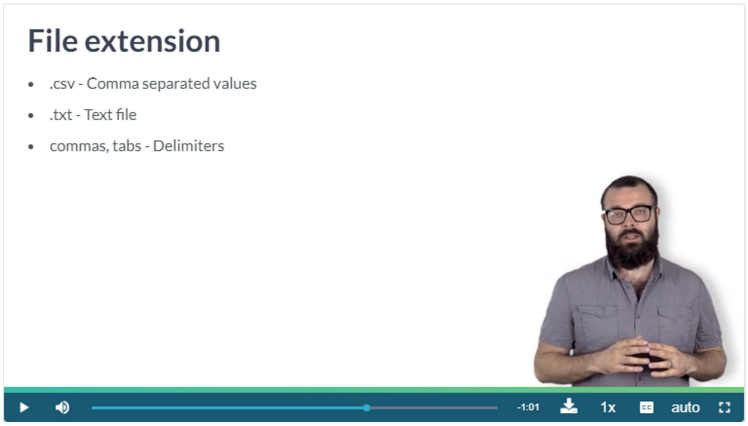
- Open `moby_dick.txt` using the `with` context manager and the variable `file`.
- Print the first three lines of the file to the shell by using `readline()` three times within the context manager.

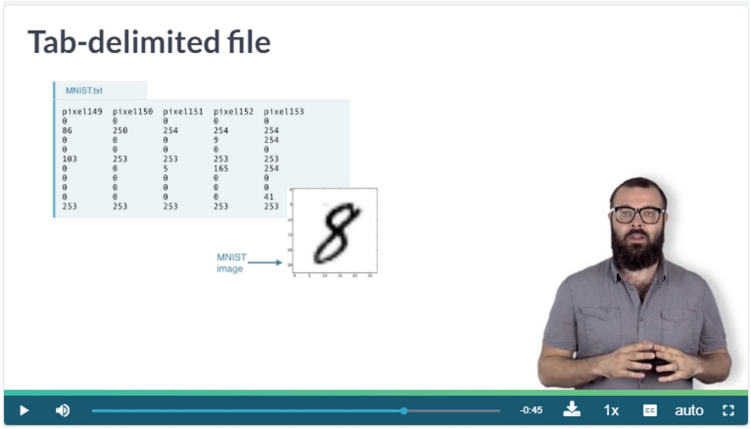
```
# Read & print the first 3 lines
with open('moby_dick.txt') as file:
    print(file.readline())
    print(file.readline())
    print(file.readline())
```

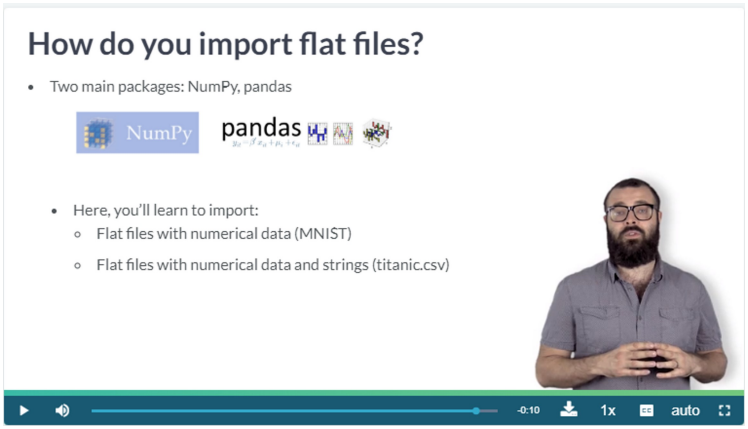
1.5 The importance of flat files in data science











1.6 Pop quiz: examples of flat files

You're now well-versed in importing text files and you're about to become a wiz at importing flat files. But can you remember exactly what a flat file

is? Test your knowledge by answering the following question: which of these file types below is NOT an example of a flat file?

Possible Answers
- [ ] A .csv file.
- [ ] A tab-delimited .txt.
- [x] A relational database (e.g. PostgreSQL).

1.7 Pop quiz: what exactly are flat files?

Which of the following statements about flat files is incorrect?

Possible Answers
- [ ] Flat files consist of rows and each row is called a record.
- [x] Flat files consist of multiple tables with structured relationships between the tables.
- [ ] A record in a flat file is composed of *fields* or *attributes*, each of which contains at most one item of information.
- [ ] Flat files are pervasive in data science.

1.8 Why we like flat files and the Zen of Python

In PythonLand, there are currently hundreds of *Python Enhancement Proposals*, commonly referred to as *PEP*s. **PEP8**, for example, is a standard style guide for Python, written by our sensei Guido van Rossum himself. It is the basis for how we here at DataCamp ask our instructors to **style their code**. Another one of my favorites is **PEP20**, commonly called the *Zen of Python*. Its abstract is as follows:

> *Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.*

If you don't know what the acronym BDFL stands for, I suggest that you look **here**. You can print the Zen of Python in your shell by typing import this into it! You're going to do this now and the 5th aphorism (line) will say something of particular interest.

The question you need to answer is: **what is the 5th aphorism of the *Zen of Python*?**

Possible Answers
- [ ] Flat is better than nested.
- [ ] Flat files are essential for data science.
- [ ] The world is representable as a flat file.
- [ ] Flatness is in the eye of the beholder.

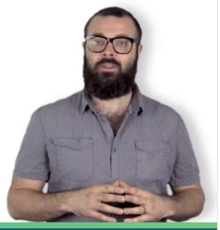
```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.9 Importing flat files using NumPy

Why NumPy?

- NumPy arrays: standard for storing numerical data
- Essential for other packages: e.g. scikit-learn

- loadtxt()
- genfromtxt()

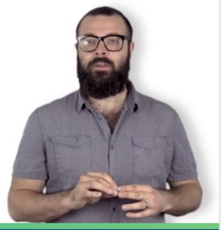


-1:32 1x auto

Importing flat files using NumPy

```
import numpy as np
filename = 'MNIST.txt'
data = np.loadtxt(filename, delimiter=',')
data
```

```
[[  0.   0.   0.   0.   0.]
 [ 86. 250. 254. 254. 254.]
 [  0.   0.   0.   9. 254.]
 ...,
 [  0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.]]
```

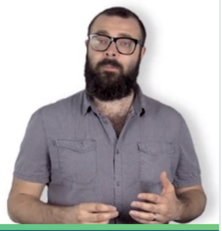


-1:20 1x auto

Customizing your NumPy import

```
import numpy as np
filename = 'MNIST_header.txt'
data = np.loadtxt(filename, delimiter=',', skiprows=1)
print(data)
```

```
[[  0.   0.   0.   0.   0.]
 [ 86. 250. 254. 254. 254.]
 [  0.   0.   0.   9. 254.]
 ...,
 [  0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.]]
```



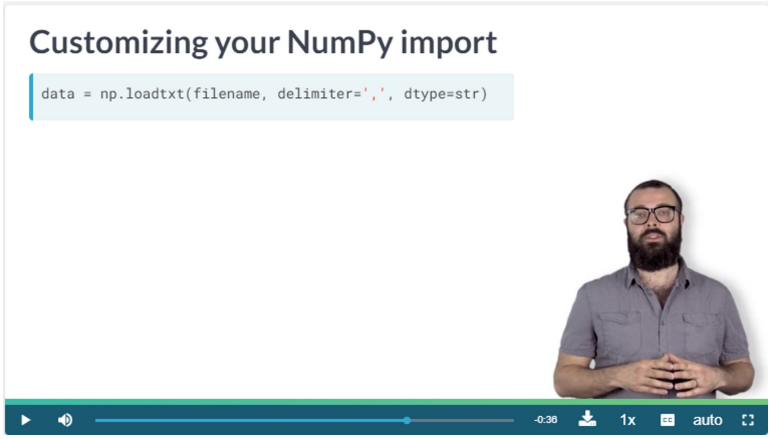
-0:56 1x auto

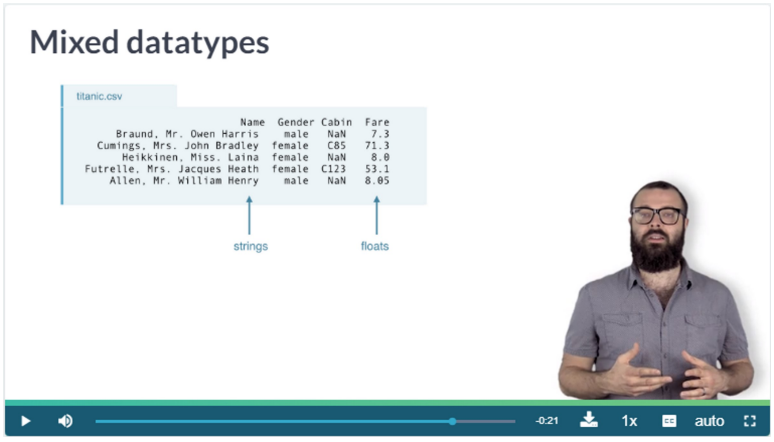
Customizing your NumPy import

```
import numpy as np
filename = 'MNIST_header.txt'
data = np.loadtxt(filename, delimiter=',', skiprows=1, usecols=[0, 2])
print(data)
```

```
[[  0.   0.]
 [ 86. 254.]
 [  0.   0.]
 ...,
 [  0.   0.]
 [  0.   0.]
 [  0.   0.]]
```

-0:47 1x auto





1.10 Using NumPy to import flat files

In this exercise, you're now going to load the MNIST digit recognition dataset using the numpy function loadtxt() and see just how easy it can be:

- The first argument will be the filename.
- The second will be the delimiter which, in this case, is a comma.

You can find more information about the MNIST dataset **here** on the webpage of Yann LeCun, who is currently Director of AI Research at Facebook and Founding Director of the NYU Center for Data Science, among many other things.

Instructions:

- Fill in the arguments of np.loadtxt() by passing file and a comma ',' for the delimiter.
- Fill in the argument of print() to print the type of the object digits. Use the function type().
- Execute the rest of the code to visualize one of the rows of the data.

```
# Import package
import numpy as np

# Assign filename to variable: file
file = 'digits.csv'

# Load file as array: digits
digits = np.loadtxt(file, delimiter=',')

# Print datatype of digits
print(type(digits))

# Select and reshape a row
im = digits[21, 1:]
im_sq = np.reshape(im, (28, 28))
```

```
# Plot reshaped data (matplotlib.pyplot already loaded as plt)
plt.imshow(im_sq, cmap='Greys', interpolation='nearest')
plt.show()
```

1.11 Customizing your NumPy import

What if there are rows, such as a header, that you don't want to import? What if your file has a delimiter other than a comma? What if you only wish to import particular columns?

There are a number of arguments that `np.loadtxt()` takes that you'll find useful:

- `delimiter` changes the delimiter that `loadtxt()` is expecting.
  - You can use `','` for comma-delimited.
  - You can use `'\t'` for tab-delimited.
- `skiprows` allows you to specify *how many rows* (not indices) you wish to skip
- `usecols` takes a *list* of the indices of the columns you wish to keep.

The file that you'll be importing, `digits_header.txt`, has a header and is tab-delimited.

Instructions:

- Complete the arguments of `np.loadtxt()`: the file you're importing is tab-delimited, you want to skip the first row and you only want to import the first and third columns.
- Complete the argument of the `print()` call in order to print the entire array that you just imported.

```
# Import numpy
import numpy as np

# Assign the filename: file
file = 'digits_header.txt'

# Load the data: data
data = np.loadtxt(file, delimiter='\t', skiprows=1, usecols=[0,2])

# Print data
print(data)
```

1.12 Importing different datatypes

The file `seaslug.txt`

- has a text header, consisting of strings
- is tab-delimited.

These data consists of percentage of sea slug larvae that had metamorphosed in a given time period. Read more **here**.

Due to the header, if you tried to import it as-is using `np.loadtxt()`, Python would throw you a `ValueError` and tell you that it `could not convert string to float`. There are two ways to deal with this: firstly, you can set the data type argument `dtype` equal to `str` (for string).

Alternatively, you can skip the first row as we have seen before, using the `skiprows` argument.

Instructions:

- Complete the first call to `np.loadtxt()` by passing `file` as the first argument.
- Execute `print(data[0])` to print the first element of `data`.
- Complete the second call to `np.loadtxt()`. The file you're importing is tab-delimited, the datatype is `float`, and you want to skip the first row.
- Print the 10th element of `data_float` by completing the `print()` command. Be guided by the previous `print()` call.
- Execute the rest of the code to visualize the data.

```
# Assign filename: file
file = 'seaslug.txt'

# Import file: data
```

```
data = np.loadtxt(file, delimiter='\t', dtype=str)

# Print the first element of data
print(data[0])

# Import data as floats and skip the first row: data_float
data_float = np.loadtxt(file, delimiter='\t', dtype=float, skiprows=1)

# Print the 10th element of data_float
print(data_float[9])

# Plot a scatterplot of the data
plt.scatter(data_float[:, 0], data_float[:, 1])
plt.xlabel('time (min.)')
plt.ylabel('percentage of larvae')
plt.show()
```

1.13 Working with mixed datatypes (1)

Much of the time you will need to import datasets which have different datatypes in different columns; one column may contain strings and another floats, for example. The function np.loadtxt() will freak at this. There is another function, np.genfromtxt(), which can handle such structures. If we pass dtype=None to it, it will figure out what types each column should be.

Import 'titanic.csv' using the function np.genfromtxt() as follows:

```
data = np.genfromtxt('titanic.csv', delimiter=',', names=True, dtype=None)
```

Here, the first argument is the filename, the second specifies the delimiter , and the third argument names tells us there is a header. Because the data are of different types, data is an object called a **structured array**. Because numpy arrays have to contain elements that are all the same type, the structured array solves this by being a 1D array, where each element of the array is a row of the flat file imported. You can test this by checking out the array's shape in the shell by executing np.shape(data).

Accessing rows and columns of structured arrays is super-intuitive: to get the ith row, merely execute data[i] and to get the column with name 'Fare', execute data['Fare'].

After importing the Titanic data as a structured array (as per the instructions above), print the entire column with the name Survived to the shell. What are the last 4 values of this column?

Possible Answers

- [ ] 1,0,0,1.
- [ ] 1,2,0,0.
- [x] 1,0,1,0.
- [ ] 0,1,1,1.

```
In [1]: data = np.genfromtxt('titanic.csv', delimiter=',', names=True, dtype=None)

In [2]: np.shape(data)
Out[2]: (891,)
```

1.14 Working with mixed datatypes (2)

You have just used np.genfromtxt() to import data containing mixed datatypes. There is also another function np.recfromcsv() that behaves similarly to np.genfromtxt(), except that its default dtype is None. In this exercise, you'll practice using this to achieve the same result.

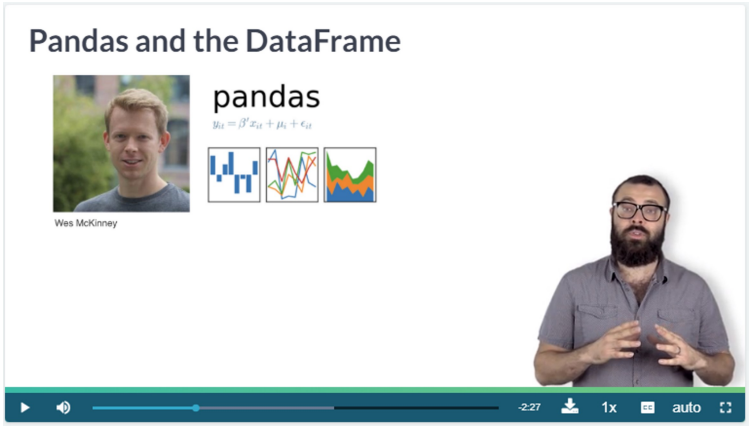
Instructions:

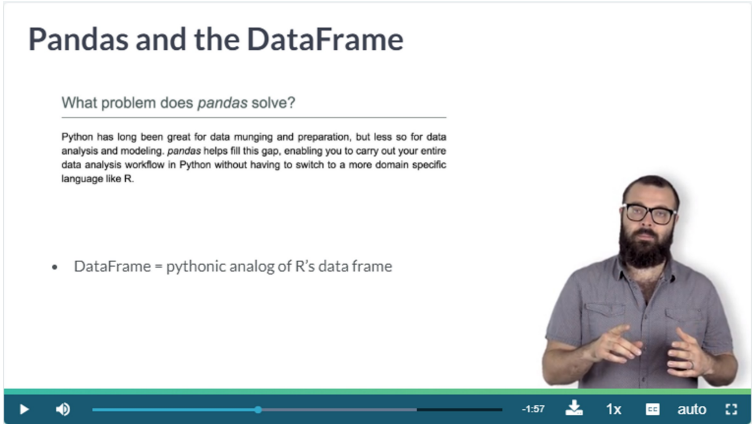
- Import titanic.csv using the function np.recfromcsv() and assign it to the variable, d. You'll only need to pass file to it because it has the defaults delimiter=',' and names=True in addition to dtype=None!
- Run the remaining code to print the first three entries of the resulting array d.

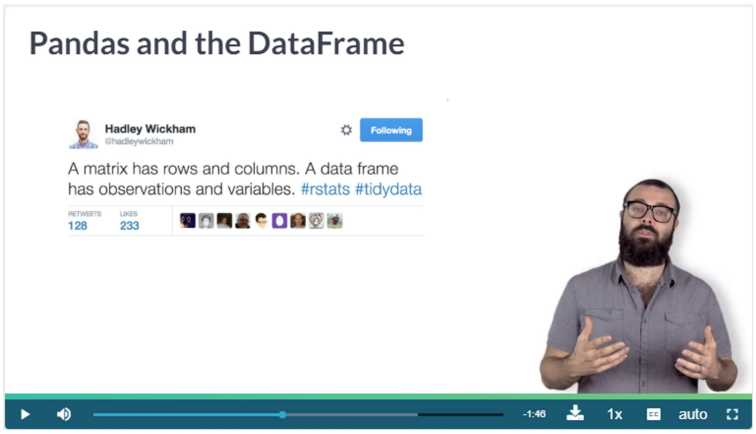
```
# Assign the filename: file
file = 'titanic.csv'
```

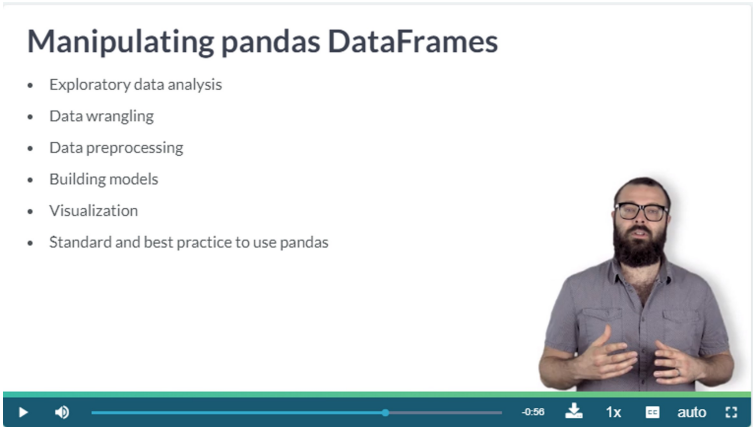
```
# Import file using np.recfromcsv: d
d = np.recfromcsv(file)

# Print out first three entries of d
print(d[:3])
```

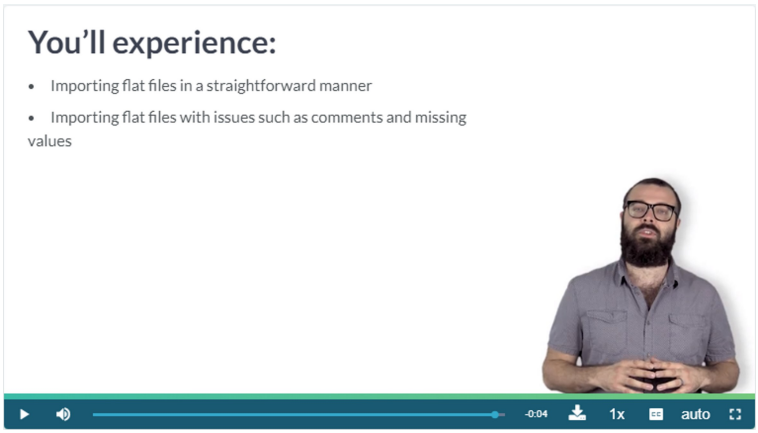
1.15 Importing flat files using pandas











1.16 Using pandas to import flat files as DataFrames (1)

In the last exercise, you were able to import flat files containing columns with different datatypes as numpy arrays. However, the DataFrame object in pandas is a more appropriate structure in which to store such data and, thankfully, we can easily import files of mixed data types as DataFrames using the pandas functions read_csv() and read_table().

Instructions:

- Import the pandas package using the alias pd.
- Read titanic.csv into a DataFrame called df. The file name is already stored in the file object.
- In a print() call, view the head of the DataFrame.

```
# Import pandas as pd
import pandas as pd

# Assign the filename: file
file = 'titanic.csv'

# Read the file into a DataFrame: df
df = pd.read_csv(file)

# View the head of the DataFrame
print(df.head())
```

1.17 Using pandas to import flat files as DataFrames (2)

In the last exercise, you were able to import flat files into a pandas DataFrame. As a bonus, it is then straightforward to retrieve the corresponding numpy array using the attribute values. You'll now have a chance to do this using the MNIST dataset, which is available as digits.csv.

Instructions:

- Import the first 5 rows of the file into a DataFrame using the function pd.read_csv() and assign the result to data. You'll need to use the arguments nrows and header (there is no header in this file).
- Build a numpy array from the resulting DataFrame in data and assign to data_array.
- Execute print(type(data_array)) to print the datatype of data_array.

```
# Assign the filename: file
file = 'digits.csv'

# Read the first 5 rows of the file into a DataFrame: data
data = pd.read_csv(file, nrows = 5, header=None)

# Build a numpy array from the DataFrame: data_array
data_array = data.values

# Print the datatype of data_array to the shell
print(type(data_array))
```

1.18 Customizing your pandas import

The pandas package is also great at dealing with many of the issues you will encounter when importing data as a data scientist, such as comments occurring in flat files, empty lines and missing values. Note that missing values are also commonly referred to as NA or NaN. To wrap up this chapter, you're now going to import a slightly corrupted copy of the Titanic dataset titanic_corrupt.txt, which

- contains comments after the character '#'
- is tab-delimited.

Instructions:

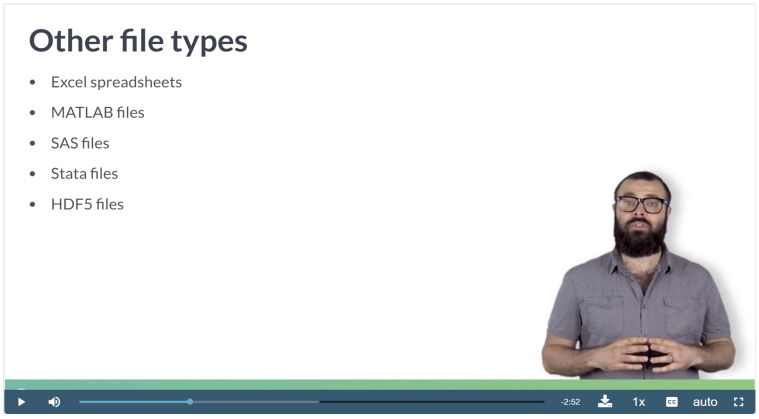
- Complete the sep (the pandas version of delim), comment and na_values arguments of pd.read_csv(). comment takes characters that comments occur after in the file, which in this case is '#'. na_values takes a list of strings to recognize as NA/NaN, in this case the string 'Nothing'.
- Execute the rest of the code to print the head of the resulting DataFrame and plot the histogram of the 'Age' of passengers aboard the Titanic.

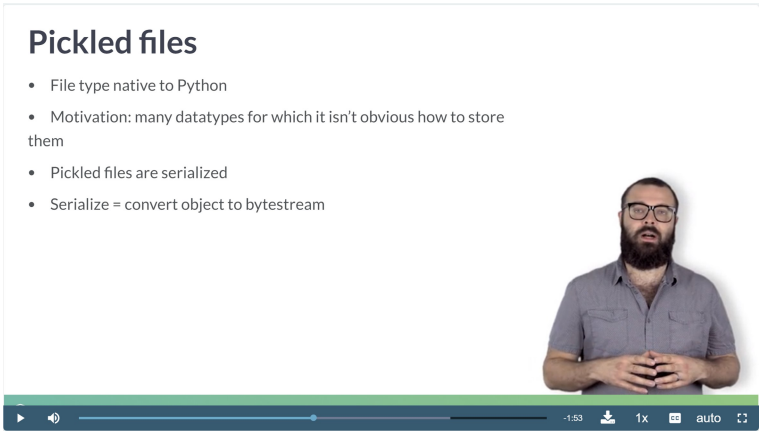
```
# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt

# Assign filename: file
```

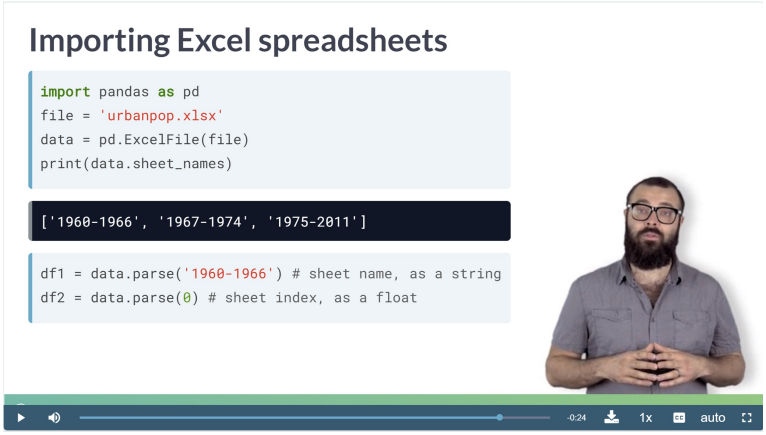
```
file = 'titanic_corrupt.txt'

# Import file: data
data = pd.read_csv(file, sep='\t', comment='#', na_values='Nothing')

# Print the head of the DataFrame
print(data.head())

# Plot 'Age' variable in a histogram
pd.DataFrame.hist(data[['Age']])
plt.xlabel('Age (years)')
plt.ylabel('count')
plt.show()
```

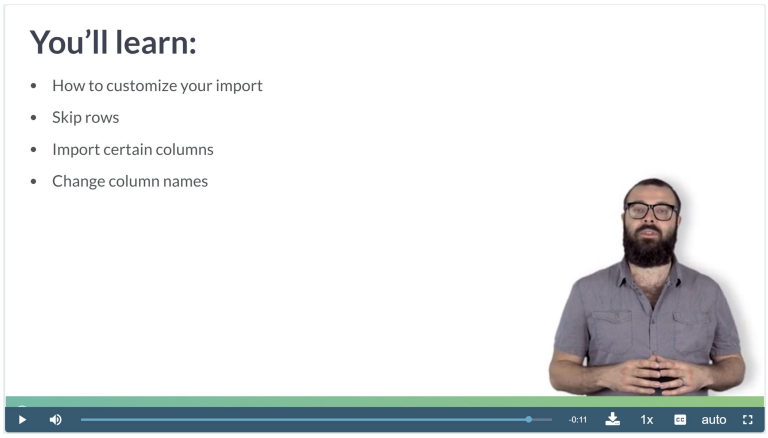
2. Importing data from other file types

2.1 Introduction to other file types









2.2 Not so flat any more

In Chapter 1, you learned how to use the IPython magic command `! ls` to explore your current working directory. You can also do this natively in Python using the **library os**, which consists of miscellaneous operating system interfaces.

The first line of the following code imports the library `os`, the second line stores the name of the current directory in a string called `wd` and the third outputs the contents of the directory in a list to the shell.

```
import os
wd = os.getcwd()
os.listdir(wd)
```

Run this code in the IPython shell and answer the following questions. Ignore the files that begin with `.`.

Check out the contents of your current directory and answer the following questions: (1) which file is in your directory and NOT an example of a flat file; (2) why is it not a flat file?

Possible Answers

- [ ] `database.db` is not a flat file because relational databases contain structured relationships and flat files do not.
- [x] `battledeath.xlsx` is not a flat because it is a spreadsheet consisting of many sheets, not a single table.
- [ ] `titanic.txt` is not a flat file because it is a `.txt`, not a `.csv`.

```
In [1]: !ls
battledeath.xlsx  titanic.txt

In [2]: import os

In [3]: wd = os.getcwd()

In [4]: os.listdir(wd)
Out[4]: ['titanic.txt', 'battledeath.xlsx']
```

2.3 Loading a pickled file

There are a number of datatypes that cannot be saved easily to flat files, such as lists and dictionaries. If you want your files to be human readable, you may want to save them as text files in a clever manner. JSONs, which you will see in a later chapter, are appropriate for Python dictionaries.

However, if you merely want to be able to import them into Python, you can **serialize** them. All this means is converting the object into a sequence of bytes, or a bytestream.

In this exercise, you'll import the pickle package, open a previously pickled data structure from a file and load it.

Instructions:

- Import the pickle package.
- Complete the second argument of open() so that it is read only for a binary file. This argument will be a string of two letters, one signifying 'read only', the other 'binary'.
- Pass the correct argument to pickle.load(); it should use the variable that is bound to open.
- Print the data, d.
- Print the datatype of d; take your mind back to your previous use of the function type().

```
# Import pickle package
import pickle

# Open pickle file and load data: d
with open('data.pkl', 'rb') as file:
    d = pickle.load(file)

# Print d
print(d)

# Print datatype of d
print(type(d))
```

2.4 Listing sheets in Excel files

Whether you like it or not, any working data scientist will need to deal with Excel spreadsheets at some point in time. You won't always want to do so in Excel, however!

Here, you'll learn how to use pandas to import Excel spreadsheets and how to list the names of the sheets in any loaded .xlsx file.

Recall from the video that, given an Excel file imported into a variable spreadsheet, you can retrieve a list of the sheet names using the attribute spreadsheet.sheet_names.

Specifically, you'll be loading and checking out the spreadsheet 'battledeath.xlsx', modified from the Peace Research Institute Oslo's (PRIO) **dataset**. This data contains age-adjusted mortality rates due to war in various countries over several years.

Instructions:

- Assign the spreadsheet filename (provided above) to the variable file.
- Pass the correct argument to pd.ExcelFile() to load the file using pandas, assigning the result to the variable xls.
- Print the sheetnames of the Excel spreadsheet by passing the necessary argument to the print() function.

```
# Import pandas
import pandas as pd

# Assign spreadsheet filename: file
file = 'battledeath.xlsx'

# Load spreadsheet: xls
xls = pd.ExcelFile(file)

# Print sheet names
print(xls.sheet_names)
```

2.5 Importing sheets from Excel files

In the previous exercises, you saw that the Excel file contains two sheets, '2002' and '2004'. The next step is to import these.

In this exercise, you'll learn how to import any given sheet of your loaded .xlsx file as a DataFrame. You'll be able to do so by specifying either the sheet's name or its index.

The spreadsheet 'battledeath.xlsx' is already loaded as xls.

Instructions:

- Load the sheet '2004' into the DataFrame df1 using its name as a string.
- Print the head of df1 to the shell.
- Load the sheet 2002 into the DataFrame df2 using its index (0).
- Print the head of df2 to the shell.

```
# Load a sheet into a DataFrame by name: df1
df1 = xls.parse('2004')

# Print the head of the DataFrame df1
print(df1.head())

# Load a sheet into a DataFrame by index: df2
df2 = xls.parse(0)

# Print the head of the DataFrame df2
print(df2.head())
```

2.6 Customizing your spreadsheet import

Here, you'll parse your spreadsheets and use additional arguments to skip rows, rename columns and select only particular columns.

The spreadsheet 'battledeath.xlsx' is already loaded as xls.

As before, you'll use the method parse(). This time, however, you'll add the additional arguments skiprows, names and usecols. These skip rows, name the columns and designate which columns to parse, respectively. All these arguments can be assigned to lists containing the specific row numbers, strings and column numbers, as appropriate.

Instructions:

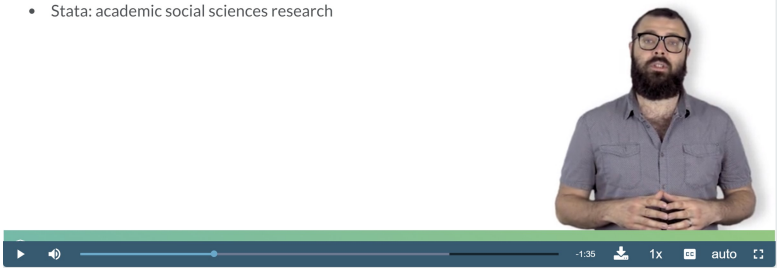
- Parse the first sheet by index. In doing so, skip the first row of data and name the columns 'Country' and 'AAM due to War (2002)' using the argument names. The values passed to skiprows and names all need to be of type list.
- Parse the second sheet by index. In doing so, parse only the first column with the usecols parameter, skip the first row and rename the column 'Country'. The argument passed to usecols also needs to be of type list.

```
# Parse the first sheet and rename the columns: df1
df1 = xls.parse(0, skiprows=[0], names=['Country', 'AAM due to War (2002)'])

# Print the head of the DataFrame df1
print(df1.head())

# Parse the first column of the second sheet and rename the column: df2
df2 = xls.parse(1, usecols=[0], skiprows=[0], names=['Country'])

# Print the head of the DataFrame df2
print(df2.head())
```

2.7 Importing SAS/Stata files using pandas

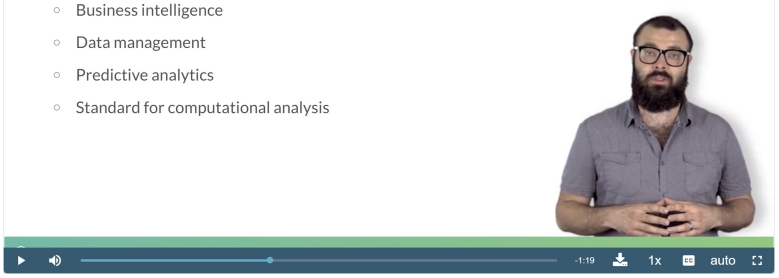
SAS and Stata files

- SAS: Statistical Analysis System
- Stata: “Statistics” + “data”
- SAS: business analytics and biostatistics
- Stata: academic social sciences research



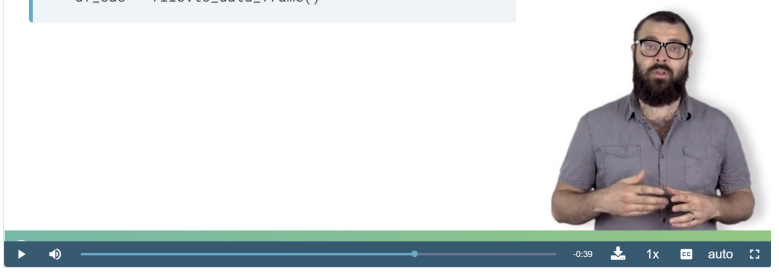
SAS files

- Used for:
  - Advanced analytics
  - Multivariate analysis
  - Business intelligence
  - Data management
  - Predictive analytics
  - Standard for computational analysis



Importing SAS files

```
import pandas as pd
from sas7bdat import SAS7BDAT
with SAS7BDAT('urbanpop.sas7bdat') as file:
    df_sas = file.to_data_frame()
```



Importing Stata files

```
import pandas as pd
data = pd.read_stata('urbanpop.dta')
```

2.8 How to import SAS7BDAT

How do you correctly import the function SAS7BDAT() from the package sas7bdat?

Possible Answers

- [ ] import SAS7BDAT from sas7bdat
- [ ] from SAS7BDAT import sas7bdat
- [ ] import sas7bdat from SAS7BDAT
- [x] from sas7bdat import SAS7BDAT

2.9 Importing SAS files

In this exercise, you'll figure out how to import a SAS file as a DataFrame using SAS7BDAT and pandas. The file 'sales.sas7bdat' is already in your working directory and both pandas and matplotlib.pyplot have already been imported as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
```

The data are adapted from the website of the undergraduate text book **Principles of Econometrics** by Hill, Griffiths and Lim.

Instructions:

- Import the module SAS7BDAT from the library sas7bdat.
- In the context of the file 'sales.sas7bdat', load its contents to a DataFrame df_sas, using the method to_data_frame() on the object file.
- Print the head of the DataFrame df_sas.
- Execute your entire script to produce a histogram plot!

```
# Import sas7bdat package
from sas7bdat import SAS7BDAT

# Save file to a DataFrame: df_sas
with SAS7BDAT('sales.sas7bdat') as file:
    df_sas = file.to_data_frame()

# Print head of DataFrame
print(df_sas.head())

# Plot histogram of DataFrame features (pandas and pyplot already imported)
pd.DataFrame.hist(df_sas[['P']])
plt.ylabel('count')
plt.show()
```

2.10 Using read_stata to import Stata files

The pandas package has been imported in the environment as pd and the file disarea.dta is in your working directory. The data consist of disease extents for several diseases in various countries (more information can be found **here**).

What is the correct way of using the read_stata() function to import disarea.dta into the object df?

Possible Answers

- [ ] df = 'disarea.dta'
- [ ] df = read_stata.pd('disarea.dta')
- [x] df = pd.read_stata('disarea.dta')
- [ ] df = pd.read_stata(disarea.dta)

2.11 Importing Stata files

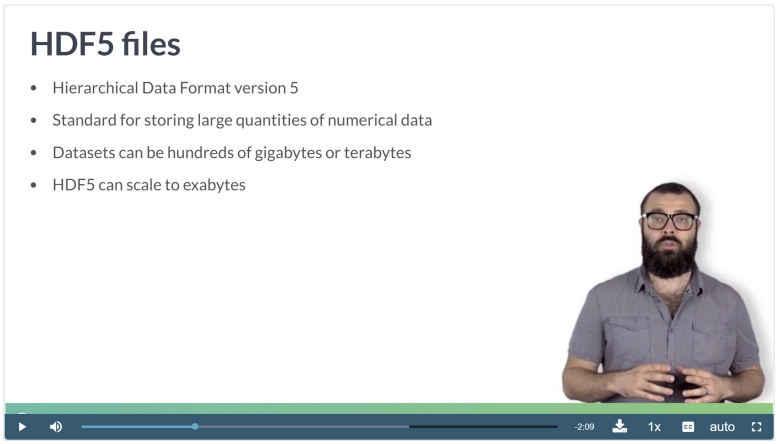
Here, you'll gain expertise in importing Stata files as DataFrames using the pd.read_stata() function from pandas. The last exercise's file, 'disarea.dta', is still in your working directory.

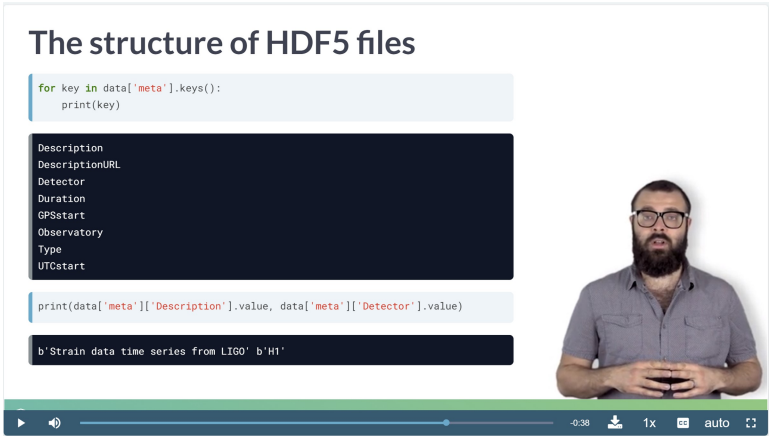
Instructions:

- Use pd.read_stata() to load the file 'disarea.dta' into the DataFrame df.
- Print the head of the DataFrame df.
- Visualize your results by plotting a histogram of the column disa10. We've already provided this code for you, so just run it!

```
# Import pandas
import pandas as pd

# Load Stata file into a pandas DataFrame: df
df = pd.read_stata('disarea.dta')

# Print the head of the DataFrame df
print(df.head())

# Plot histogram of one column of the DataFrame
pd.DataFrame.hist(df[['disa10']])
plt.xlabel('Extent of disease')
plt.ylabel('Number of countries')
plt.show()
```

2.12 Importing HDF5 files





2.13 Using File to import HDF5 files

The h5py package has been imported in the environment and the file LIGO_data.hdf5 is loaded in the object h5py_file.

What is the correct way of using the h5py function, File(), to import the file in h5py_file into an object, h5py_data, for *reading* only?

Possible Answers

- [ ] h5py_data = File(h5py_file, 'r')
- [x] h5py_data = h5py.File(h5py_file, 'r')
- [ ] h5py_data = h5py.File(h5py_file, read)
- [ ] h5py_data = h5py.File(h5py_file, 'read')

2.14 Using h5py to import HDF5 files

The file 'LIGO_data.hdf5' is already in your working directory. In this exercise, you'll import it using the h5py library. You'll also print out its datatype to confirm you have imported it correctly. You'll then study the structure of the file in order to see precisely what HDF groups it contains.

You can find the LIGO data plus loads of documentation and tutorials **here**. There is also a great tutorial on Signal Processing with the data **here**.

Instructions:

- Import the package h5py.
- Assign the name of the file to the variable file.
- Load the file as read only into the variable data.
- Print the datatype of data.
- Print the names of the groups in the HDF5 file 'LIGO_data.hdf5'.

```
# Import packages
import numpy as np
import h5py

# Assign filename: file
file = 'LIGO_data.hdf5'

# Load file: data
data = h5py.File(file, 'r')

# Print the datatype of the loaded file
print(type(data))

# Print the keys of the file
for key in data.keys():
    print(key)
```

2.15 Extracting data from your HDF5 file

In this exercise, you'll extract some of the LIGO experiment's actual data from the HDF5 file and you'll visualize it.

To do so, you'll need to first explore the HDF5 group 'strain'.

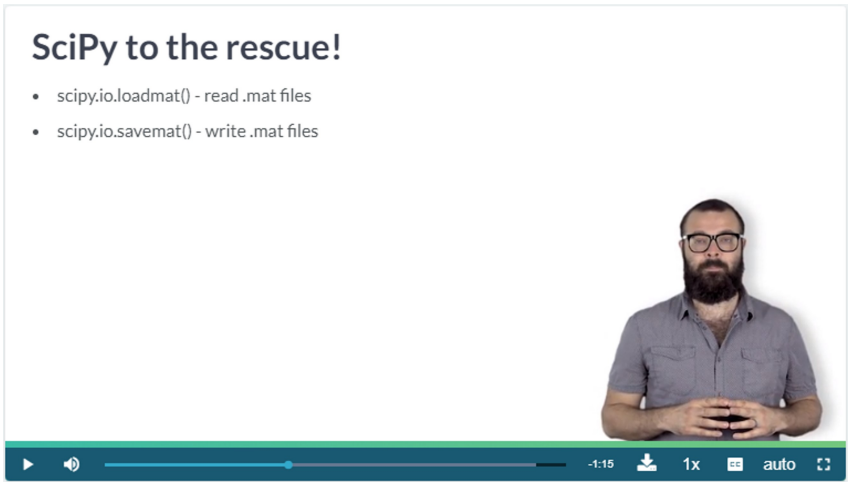
Instructions:

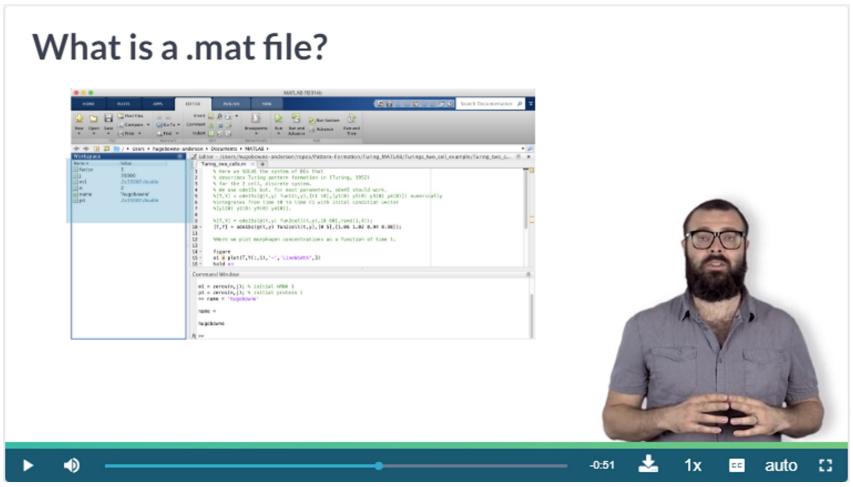
- Assign the HDF5 group data['strain'] to group.
- In the for loop, print out the keys of the HDF5 group in group.
- Assign to the variable strain the values of the time series data data['strain']['Strain'] using the attribute .value.
- Set num_samples equal to 10000, the number of time points we wish to sample.
- Execute the rest of the code to produce a plot of the time series data in LIGO_data.hdf5.

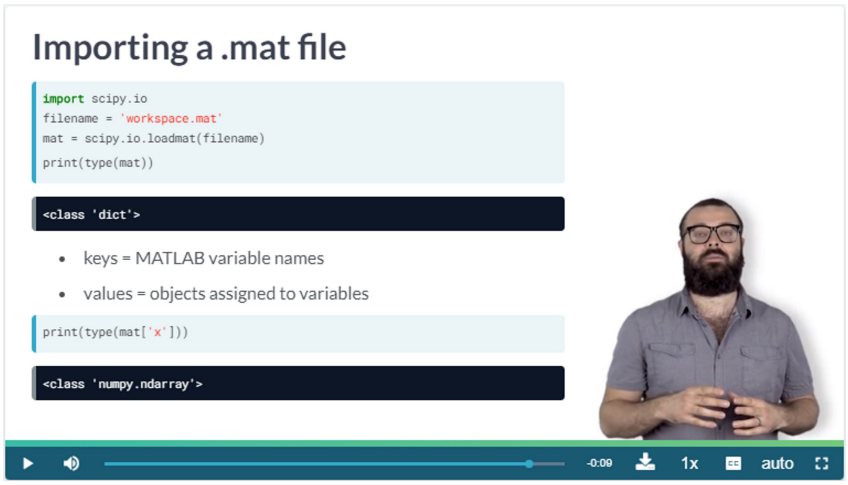
```
# Get the HDF5 group: group
group = data['strain']

# Check out keys of group
for key in group.keys():
    print(key)

# Set variable equal to time series data: strain
strain = data['strain']['Strain'].value

# Set number of time points to sample: num_samples
num_samples = 10000

# Set time vector
time = np.arange(0, 1, 1/num_samples)

# Plot data
plt.plot(time, strain[:num_samples])
plt.xlabel('GPS Time (s)')
plt.ylabel('strain')
```

plt.show()

2.16 Importing MATLAB files







2.17 Loading .mat files

In this exercise, you'll figure out how to load a MATLAB file using scipy.io.loadmat() and you'll discover what Python datatype it yields. The file 'albeck_gene_expression.mat' is in your working directory. This file contains **gene expression data** from the Albeck Lab at UC Davis. You can find the data and some great documentation **here**.

Instructions:

- Import the package scipy.io.
- Load the file 'albeck_gene_expression.mat' into the variable mat; do so using the function scipy.io.loadmat().
- Use the function type() to print the datatype of mat to the IPython shell.

```
# Import package
import scipy.io

# Load MATLAB file: mat
mat = scipy.io.loadmat('albeck_gene_expression.mat')

# Print the datatype type of mat
print(type(mat))
```

2.18 The structure of .mat in Python

Here, you'll discover what is in the MATLAB dictionary that you loaded in the previous exercise.

The file 'albeck_gene_expression.mat' is already loaded into the variable mat. The following libraries have already been imported as follows:

```
import scipy.io
import matplotlib.pyplot as plt
import numpy as np
```

Once again, this file contains **gene expression data** from the Albeck Lab at UCDavis. You can find the data and some great documentation **here**.

Instructions:

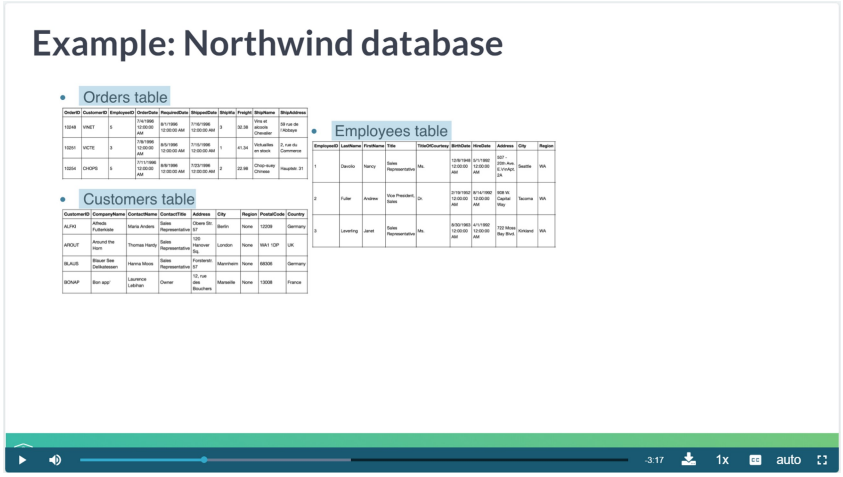
- Use the method .keys() on the dictionary mat to print the keys. Most of these keys (in fact the ones that do NOT begin and end with '__') are variables from the corresponding MATLAB environment.
- Print the type of the value corresponding to the key 'CYratioCyt' in mat. Recall that mat['CYratioCyt'] accesses the value.
- Print the shape of the value corresponding to the key 'CYratioCyt' using the numpy function shape().
- Execute the entire script to see some oscillatory gene expression data!

```
# Print the keys of the MATLAB dictionary
print(mat.keys())

# Print the type of the value corresponding to the key 'CYratioCyt'
print(type(mat['CYratioCyt']))
```

```
# Print the shape of the value corresponding to the key 'CYratioCyt'
print(np.shape(mat['CYratioCyt']))

# Subset the array and plot it
data = mat['CYratioCyt'][25, 5:]
fig = plt.figure()
plt.plot(data)
plt.xlabel('time (min.)')
plt.ylabel('normalized fluorescence (measure of expression)')
plt.show()
```

3. Working with relational databases in Python

3.1 Introduction to relational databases



The Orders table

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress
10248	VINET	5	7/4/1996 12:00:00 AM	8/1/1996 12:00:00 AM	7/16/1996 12:00:00 AM	3	32.38	Vins et alcools Chevalier	59 rue de l'Abbaye
10251	VICTE	3	7/8/1996 12:00:00 AM	8/5/1996 12:00:00 AM	7/15/1996 12:00:00 AM	1	41.34	Victuailles en stock	2, rue du Commerce
10254	CHOPS	5	7/11/1996 12:00:00 AM	8/8/1996 12:00:00 AM	7/23/1996 12:00:00 AM	2	22.98	Chop-suey Chinese	Hauptstr. 31

-2:24 1x auto

Tables are linked

- Orders table

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress
10248	VINET	5	7/4/1996 12:00:00 AM	8/1/1996 12:00:00 AM	7/16/1996 12:00:00 AM	3	32.38	Vins et alcools Chevalier	59 rue de l'Abbaye
10251	VICTE	3	7/8/1996 12:00:00 AM	8/5/1996 12:00:00 AM	7/15/1996 12:00:00 AM	1	41.34	Victuailles en stock	2, rue du Commerce
10254	CHOPS	5	7/11/1996 12:00:00 AM	8/8/1996 12:00:00 AM	7/23/1996 12:00:00 AM	2	22.98	Chop-suey Chinese	Hauptstr. 31

- Customers table

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	None	12209	Germany
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London	None	WA1 1DP	UK
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim	None	68306	Germany
MORAF	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille	None	13008	France

- Employees table

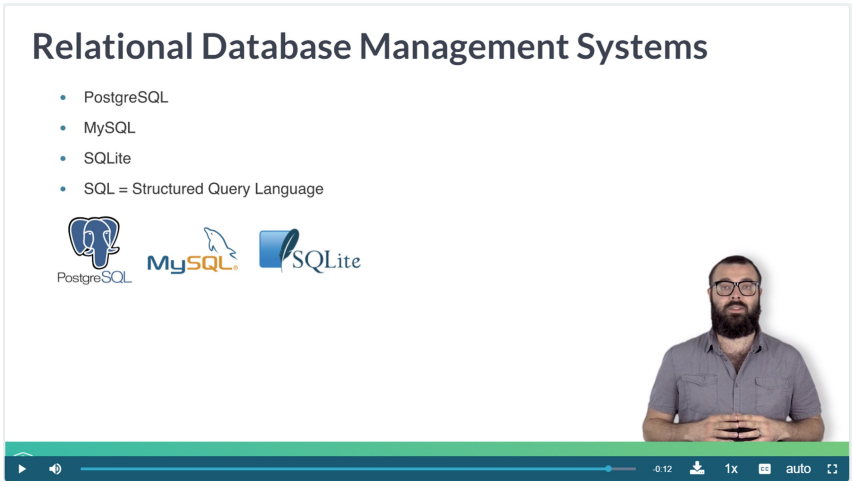
EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region
1	Davolio	Nancy	Sales Representative	Ms.	12/8/1948 12:00:00 AM	5/1/1992 12:00:00 AM	507 - 20th Ave. E.\r\nApt. 2A	Seattle	WA
2	Fuller	Andrew	Vice President, Sales	Dr.	2/19/1952 12:00:00 AM	8/14/1992 12:00:00 AM	908 W. Capital Way	Tacoma	WA
3	Leverling	Janet	Sales Representative	Ms.	8/30/1963 12:00:00 AM	4/1/1992 12:00:00 AM	722 Moss Bay Blvd.	Kirkland	WA

-1:39 1x auto

Relational model

- Widely adopted
- Codd's 12 Rules/Commandments
  - Consists of 13 rules (zero-indexed!)
  - Describes what a Relational Database Management System should adhere to to be considered relational

-0:45 1x auto



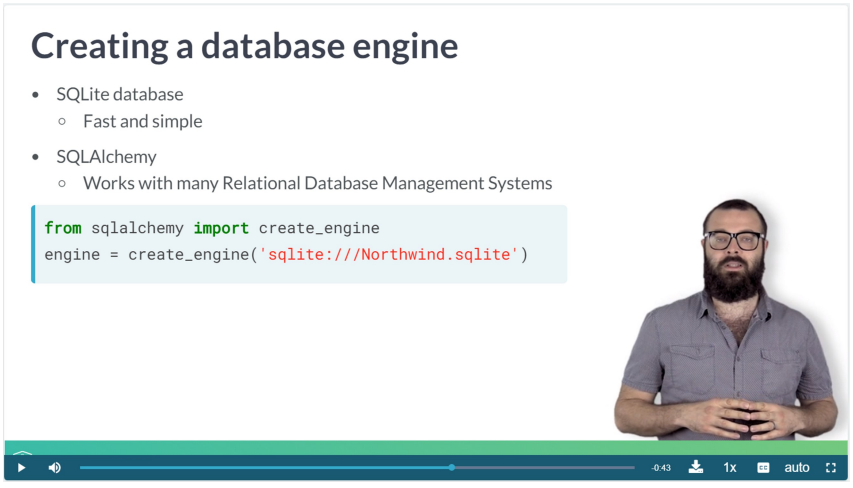
3.2 Pop quiz: The relational model

Which of the following is not part of the relational model?

Possible Answers

- [ ] Each row or record in a table represents an instance of an entity type.
  Each column in a table represents an attribute or feature of an instance.
- [ ] Every table contains a primary key column, which has a unique entry for each row.
- [x] A database consists of at least 3 tables.
- [ ] There are relations between tables.

3.3 Creating a database engine in Python



3.4 Creating a database engine

Here, you're going to fire up your very first SQL engine. You'll create an engine to connect to the SQLite database 'Chinook.sqlite', which is in your working directory. Remember that to create an engine to connect to 'Northwind.sqlite', Hugo executed the command engine = create_engine('sqlite:///Northwind.sqlite')
Here, 'sqlite:///Northwind.sqlite' is called the *connection string* to the SQLite database Northwind.sqlite. A little bit of background on the **Chinook database**: the Chinook database contains information about a semi-fictional digital media store in which media data is real and customer, employee and sales data has been manually created.
Why the name Chinook, you ask? According to their **website**,

> *The name of this sample database was based on the Northwind database. Chinooks are winds in the interior West of North America, where the Canadian Prairies and Great Plains meet various mountain ranges. Chinooks are most prevalent over southern Alberta in Canada. Chinook is a good name choice for a database that intends to be an alternative to Northwind.*

Instructions:

- Import the function create_engine from the module sqlalchemy.
- Create an engine to connect to the SQLite database 'Chinook.sqlite' and assign it to engine.

```
from sqlalchemy import create_engine

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')
```

3.5 What are the tables in the database?

In this exercise, you'll once again create an engine to connect to 'Chinook.sqlite'. Before you can get any data out of the database, however, you'll need to know what tables it contains!
To this end, you'll save the table names to a list using the method table_names() on the engine and then you will print the list.

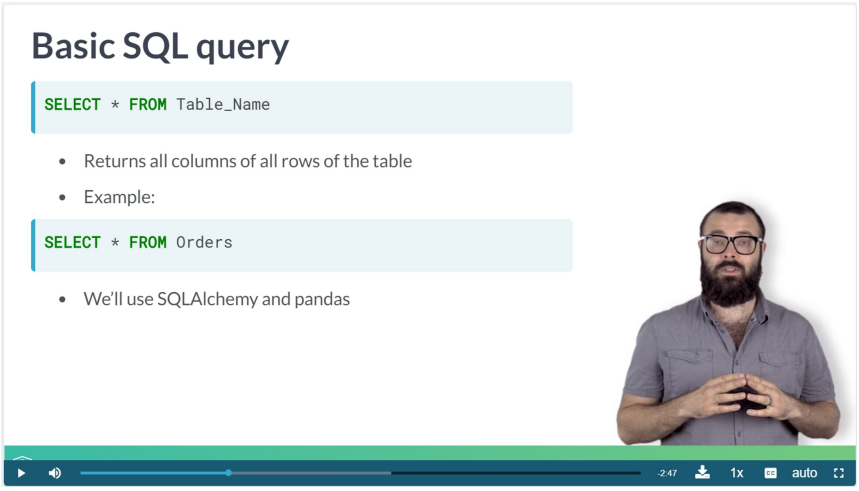
Instructions:

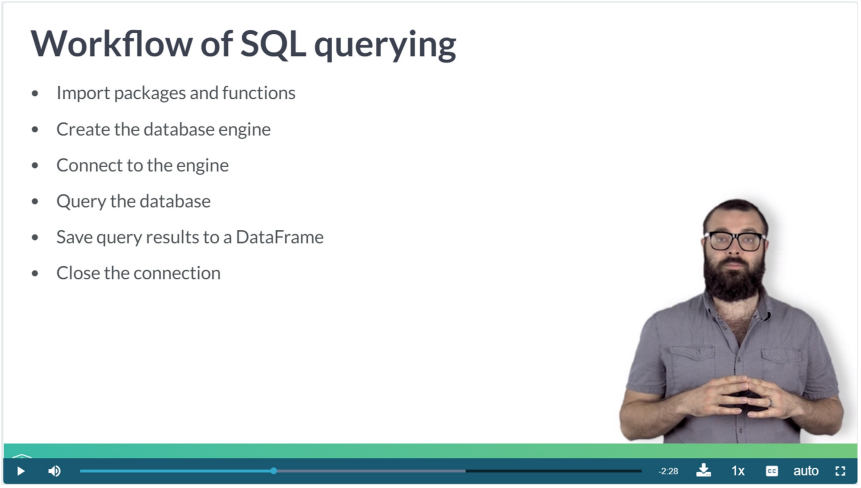
- Import the function create_engine from the module sqlalchemy.
- Create an engine to connect to the SQLite database 'Chinook.sqlite' and assign it to engine.
- Using the method table_names() on the engine engine, assign the table names of 'Chinook.sqlite' to the variable table_names.
- Print the object table_names to the shell.

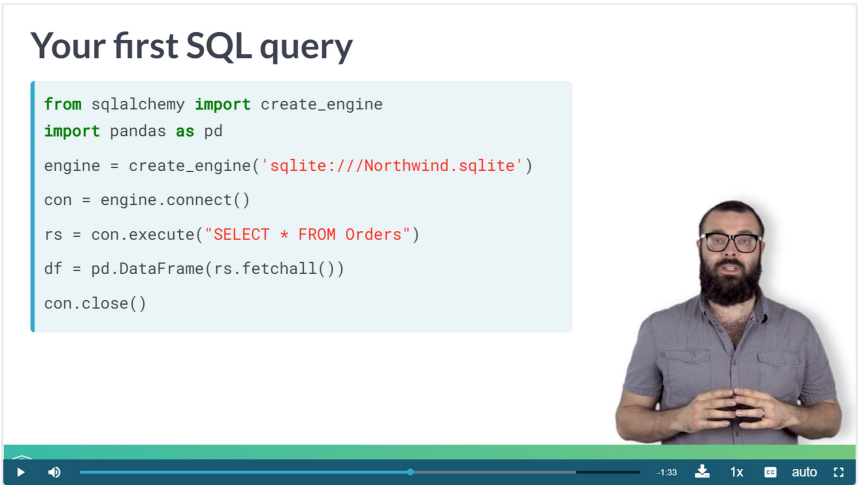
```
# Import necessary module
from sqlalchemy import create_engine

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Save the table names to a list: table_names
table_names = engine.table_names()
```

```
# Print the table names to the shell
print(table_names)
```

3.6 Querying relational databases in Python



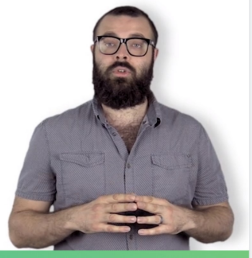




Printing your query results

```
print(df.head())
```

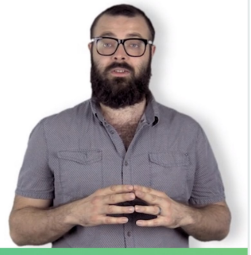
```
      0      1  2                    3                    4
0  10248  VINET  5   7/4/1996 12:00:00 AM   8/1/1996 12:00:00 AM
1  10251  VICTE  3   7/8/1996 12:00:00 AM   8/5/1996 12:00:00 AM
2  10254  CHOPS  5  7/11/1996 12:00:00 AM   8/8/1996 12:00:00 AM
3  10256  WELLI  3  7/15/1996 12:00:00 AM  8/12/1996 12:00:00 AM
4  10258  ERNSH  1  7/17/1996 12:00:00 AM  8/14/1996 12:00:00 AM
```



-1:25 1x auto

Set the DataFrame column names

```
from sqlalchemy import create_engine
import pandas as pd
engine = create_engine('sqlite:///Northwind.sqlite')
con = engine.connect()
rs = con.execute("SELECT * FROM Orders")
df = pd.DataFrame(rs.fetchall())
df.columns = rs.keys()
con.close()
```

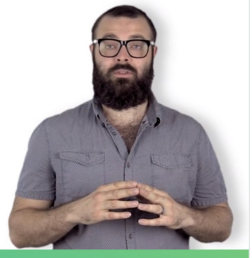


-1:12 1x auto

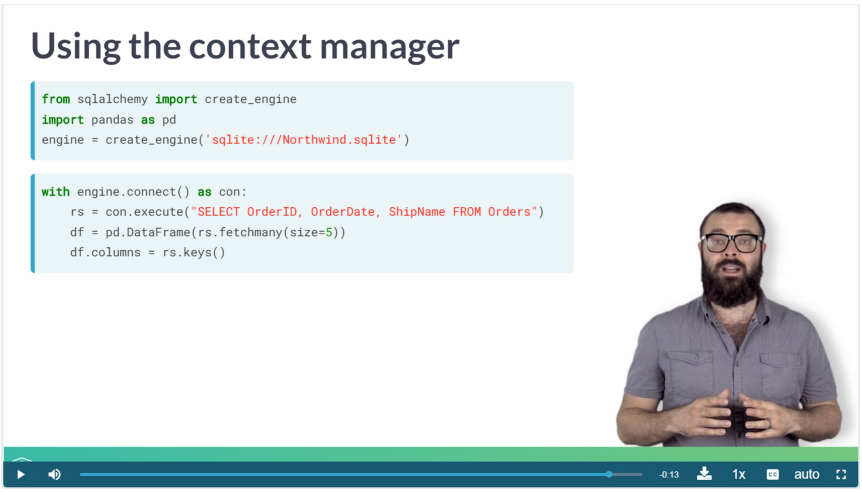
Set the data frame column names

```
print(df.head())
```

```
   OrderID CustomerID  EmployeeID              OrderDate
0    10248      VINET           5   7/4/1996 12:00:00 AM
1    10251      VICTE           3   7/8/1996 12:00:00 AM
2    10254      CHOPS           5  7/11/1996 12:00:00 AM
3    10256      WELLI           3  7/15/1996 12:00:00 AM
4    10258      ERNSH           1  7/17/1996 12:00:00 AM
```



-1:09 1x auto



3.7 The Hello World of SQL Queries!

Now, it's time for liftoff! In this exercise, you'll perform the Hello World of SQL queries, `SELECT`, in order to retrieve all columns of the table `Album` in the Chinook database. Recall that the query `SELECT *` selects all columns.

Instructions:

- Open the engine connection as `con` using the method `connect()` on the engine.
- Execute the query that **selects** *ALL columns* **from** the `Album` table. Store the results in `rs`.
- Store all of your query results in the DataFrame `df` by applying the `fetchall()` method to the results `rs`.
- Close the connection!

```
# Import packages
from sqlalchemy import create_engine
import pandas as pd

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Open engine connection: con
con = engine.connect()

# Perform query: rs
rs = con.execute("SELECT * FROM Album")

# Save results of the query to DataFrame: df
df = pd.DataFrame(rs.fetchall())

# Close connection
con.close()

# Print head of DataFrame df
print(df.head())
```

3.8 Customizing the Hello World of SQL Queries

Congratulations on executing your first SQL query! Now you're going to figure out how to customize your query in order to:

- Select specified columns from a table;
- Select a specified number of rows;
- Import column names from the database table.

Recall that Hugo performed a very similar query customization in the video:

```
engine = create_engine('sqlite:///Northwind.sqlite')
with engine.connect() as con:
    rs = con.execute("SELECT OrderID, OrderDate, ShipName FROM Orders")
    df = pd.DataFrame(rs.fetchmany(size=5))
```

```
    df.columns = rs.keys()
```

Packages have already been imported as follows:

```
from sqlalchemy import create_engine
import pandas as pd
```

The engine has also already been created:

```
engine = create_engine('sqlite:///Chinook.sqlite')
```

The engine connection is already open with the statement

```
with engine.connect() as con:
```

All the code you need to complete is within this context.

Instructions:

- Execute the SQL query that **selects** the columns LastName and Title **from** the Employee table. Store the results in the variable rs.
- Apply the method fetchmany() to rs in order to retrieve 3 of the records. Store them in the DataFrame df.
- Using the rs object, set the DataFrame's column names to the corresponding names of the table columns.

```
# Open engine in context manager
# Perform query and save results to DataFrame: df
with engine.connect() as con:
    rs = con.execute("SELECT LastName, Title from Employee")
    df = pd.DataFrame(rs.fetchmany(size=3))
    df.columns = rs.keys()

# Print the length of the DataFrame df
print(len(df))

# Print the head of the DataFrame df
print(df.head())
```

3.9 Filtering your database records using SQL's WHERE

You can now execute a basic SQL query to select records from any table in your database and you can also perform simple query customizations to select particular columns and numbers of rows.
There are a couple more standard SQL query chops that will aid you in your journey to becoming an SQL ninja.

Let's say, for example that you wanted to get all records from the Customer table of the Chinook database for which the Country is 'Canada'. You can do this very easily in SQL using a SELECT statement followed by a WHERE clause as follows:

```
SELECT * FROM Customer WHERE Country = 'Canada'
```

In fact, you can filter any SELECT statement by any condition using a WHERE clause. This is called *filtering* your records.

In this interactive exercise, you'll select all records of the Employee table for which 'EmployeeId' is greater than or equal to 6.
Packages are already imported as follows:

```
import pandas as pd
from sqlalchemy import create_engine
```

Query away!

Instructions:

- Complete the argument of create_engine() so that the engine for the SQLite database 'Chinook.sqlite' is created.
- Execute the query that **selects** *all* records **from** the Employee table **where** 'EmployeeId' is greater than or equal to 6. Use the >= operator and assign the results to rs.
- Apply the method fetchall() to rs in order to fetch all records in rs. Store them in the DataFrame df.
- Using the rs object, set the DataFrame's column names to the corresponding names of the table columns.

```
# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Open engine in context manager
# Perform query and save results to DataFrame: df
with engine.connect() as con:
    rs = con.execute("SELECT * FROM Employee WHERE EmployeeId >= 6")
    df = pd.DataFrame(rs.fetchall())
    df.columns = rs.keys()

# Print the head of the DataFrame df
print(df.head())
```

3.10 Ordering your SQL records with ORDER BY

You can also *order* your SQL query results. For example, if you wanted to get all records from the Customer table of the Chinook database and order them in increasing order by the column SupportRepId, you could do so with the following query:

```
"SELECT * FROM Customer ORDER BY SupportRepId"
```

In fact, you can order any SELECT statement by any column.

In this interactive exercise, you'll select all records of the Employee table and order them in increasing order by the column BirthDate.

Packages are already imported as follows:

```
import pandas as pd
from sqlalchemy import create_engine
```

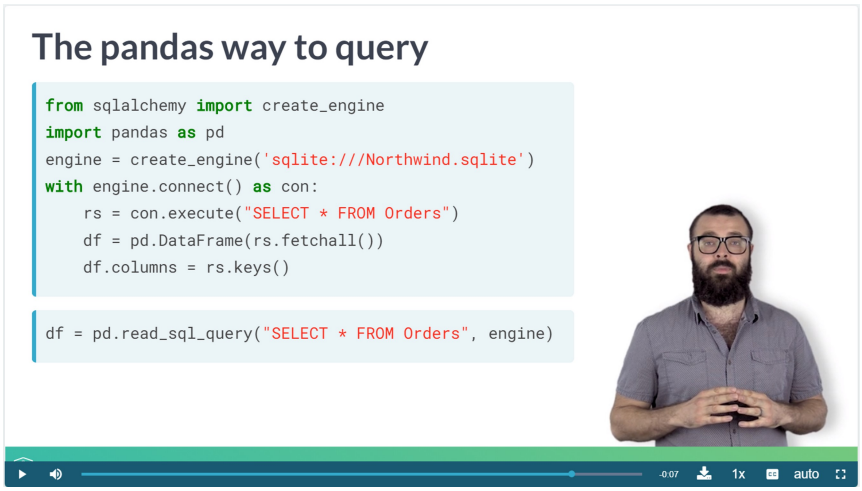
Get querying!

Instructions:

- Using the function create_engine(), create an engine for the SQLite database Chinook.sqlite and assign it to the variable engine.
- In the context manager, execute the query that **selects** *all* records **from** the Employee table and **orders** them in increasing order **by** the column BirthDate. Assign the result to rs.
- In a call to pd.DataFrame(), apply the method fetchall() to rs in order to fetch all records in rs. Store them in the DataFrame df.
- Set the DataFrame's column names to the corresponding names of the table columns.

```
# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Open engine in context manager
with engine.connect() as con:
    rs = con.execute("SELECT * FROM Employee ORDER BY BirthDate ASC")
    df = pd.DataFrame(rs.fetchall())

    # Set the DataFrame's column names
    df.columns = rs.keys()

# Print head of DataFrame
print(df.head())
```

3.11 Querying relational databases directly with pandas



3.12 Pandas and The Hello World of SQL Queries!

Here, you'll take advantage of the power of `pandas` to write the results of your SQL query to a DataFrame in one swift line of Python code!

You'll first import `pandas` and create the SQLite `'Chinook.sqlite'` engine. Then you'll query the database to select all records from the `Album` table.

Recall that to select all records from the `Orders` table in the Northwind database, Hugo executed the following command:

`df = pd.read_sql_query("SELECT * FROM Orders", engine)`

Instructions:

- Import the `pandas` package using the alias `pd`.
- Using the function `create_engine()`, create an engine for the SQLite database `Chinook.sqlite` and assign it to the variable `engine`.
- Use the `pandas` function `read_sql_query()` to assign to the variable `df` the DataFrame of results from the following query: **select** *all* records **from** the table `Album`.
- The remainder of the code is included to confirm that the DataFrame created by this method is equal to that created by the previous method that you learned.

```
# Import packages
from sqlalchemy import create_engine
import pandas as pd

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Execute query and store records in DataFrame: df
df = pd.read_sql_query("SELECT * FROM Album", engine)

# Print head of DataFrame
print(df.head())

# Open engine in context manager and store query result in df1
with engine.connect() as con:
    rs = con.execute("SELECT * FROM Album")
    df1 = pd.DataFrame(rs.fetchall())
    df1.columns = rs.keys()

# Confirm that both methods yield the same result
print(df.equals(df1))
```

3.13 Pandas for more complex querying

Here, you'll become more familiar with the pandas function `read_sql_query()` by using it to execute a more complex query:

a SELECT statement followed by both a WHERE clause AND an ORDER BY clause.

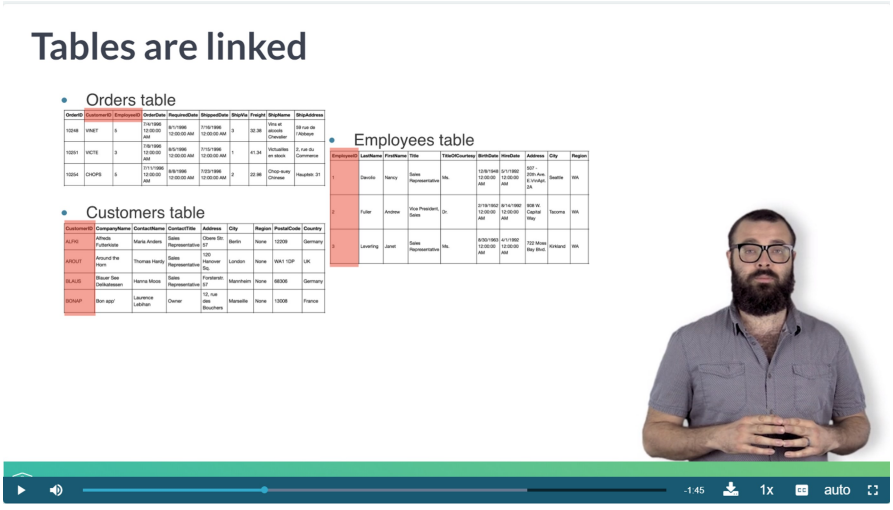
You'll build a DataFrame that contains the rows of the Employee table for which the EmployeeId is greater than or equal to 6 and you'll order these entries by BirthDate.

Instructions:

- Using the function create_engine(), create an engine for the SQLite database Chinook.sqlite and assign it to the variable engine.
- Use the pandas function read_sql_query() to assign to the variable df the DataFrame of results from the following query: **select** *all* records **from** the Employee table **where** the EmployeeId is greater than or equal to 6 and **ordered by** BirthDate (make sure to use WHERE and ORDER BY in this precise order).

```
# Import packages
from sqlalchemy import create_engine
import pandas as pd

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Execute query and store records in DataFrame: df
df = pd.read_sql_query("SELECT * FROM Employee WHERE EmployeeId >= 6 ORDER by BirthDate", engine)

# Print head of DataFrame
print(df.head())
```

3.14 Advanced querying: exploiting table relationships



3.15 The power of SQL lies in relationships between tables: INNER JOIN

Here, you'll perform your first INNER JOIN! You'll be working with your favourite SQLite database, Chinook.sqlite. For each record in the Album table, you'll extract the Title along with the Name of the Artist. The latter will come from the Artist table and so you will need to INNER JOIN these two tables on the ArtistID column of both.

Recall that to INNER JOIN the Orders and Customers tables from the Northwind database, Hugo executed the following SQL query:

"SELECT OrderID, CompanyName FROM Orders INNER JOIN Customers on Orders.CustomerID = Customers.CustomerID"

The following code has already been executed to import the necessary packages and to create the engine:

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine('sqlite:///Chinook.sqlite')
```

Instructions:

- Assign to rs the results from the following query: **select** *all* the records, extracting the Title of the record and Name of the artist of each record **from** the Album table and the Artist table, respectively. To do so, INNER JOIN these two tables on the ArtistID column of both.
- In a call to pd.DataFrame(), apply the method fetchall() to rs in order to fetch all records in rs. Store them in the DataFrame df.
- Set the DataFrame's column names to the corresponding names of the table columns.

```
# Open engine in context manager
# Perform query and save results to DataFrame: df
with engine.connect() as con:
    rs = con.execute("SELECT Title, Name FROM Album INNER JOIN Artist ON Album.ArtistID = Artist.ArtistID")
    df = pd.DataFrame(rs.fetchall())
    df.columns = rs.keys()

# Print head of DataFrame df
print(df.head())
```

3.16 Filtering your INNER JOIN

Congrats on performing your first INNER JOIN! You're now going to finish this chapter with one final exercise in which you perform an INNER JOIN and filter the result using a WHERE clause.
Recall that to INNER JOIN the Orders and Customers tables from the Northwind database, Hugo executed the following SQL query:

"SELECT OrderID, CompanyName FROM Orders INNER JOIN Customers on Orders.CustomerID = Customers.CustomerID"
The following code has already been executed to import the necessary packages and to create the engine:

```
import pandas as pd
from sqlalchemy import create_engine
```

engine = create_engine('sqlite:///Chinook.sqlite')

Instructions:

- Use the pandas function read_sql_query() to assign to the variable df the DataFrame of results from the following query: **select** *all* records **from** PlaylistTrack INNER JOIN Track on PlaylistTrack.TrackId = Track.TrackId that satisfy the condition Milliseconds < 250000.

```
# Execute query and store records in DataFrame: df
df = pd.read_sql_query("SELECT * FROM PlaylistTrack INNER JOIN Track on PlaylistTrack.TrackId = Track.TrackId WHERE Milliseconds < 250000", engine)

# Print head of DataFrame
print(df.head())
```