Julieta Matevosyan, Data Science learning notes

Data Scientist with Python Track

Type conversion

Using the + operator to paste together two strings can be very useful in building custom messages.
In case we have calculated the return of investment and want to summarize the results in a string. Assuming the floats, savings and result are defined, we can try something like this:
print("I started with $" + savings + " and now have $" + result + ". Awesome!")
This will not work, though, as we cannot simply sum strings and floats.
To fix the error, we'll need to explicitly convert the types of our variables. More specifically, we'll need str(), to convert a value into a string.

Similar functions such as int(), float() and bool() will help you convert Python values into any type.

Print out the number of o's on the variable place by calling count() on place and passing the letter 'o' as an input to the method. We're talking about the variable place, not the word "place"!
To print out the number of o's in place use.

print(place.count('o'))

List Methods

Most list methods will change the list they're called on. Examples are:
append() - adds an element to the list it is called on
remove() - removes the first element of a list
reverse() - reverses the order of the elements in the list it is called on.
*You can use the reverse() method to reverse the order of the elements in areas
areas.reverse()

To import a specific part of a package you can use.
e.g. from math import radians

Loop over the dictionary

europe = {'spain':'madrid', 'france':'paris', 'germany' : 'berlin',
      'norway':'oslo', 'italy':'rome', 'poland':'warsaw', 'austria':'vienna' }
Each dictionary has keys and values: key=country, value=capital
# Iterate over europe
for key, value in europe.items():

```
print ('the capital of '+str(key)+' is '+str(value))
the capital of france is paris
the capital of Norway is oslo…
```

Loop over Numpy array

If we're dealing with a 1D Numpy array, looping over all elements can be as simple as:
for x in my_array :

If we're dealing with a 2D Numpy array, it's more complicated. A 2D array is built up of multiple 1D arrays. To explicitly iterate over all separate elements of a multidimensional array, we'll need this syntax:
for x in np.nditer(my_array) :

Loop over Pandas DataFrame

Iterating over a Pandas DataFrame is typically done with the iterrows() method. Used in a for loop, every observation is iterated over and on every iteration the row label and actual row contents are available:

```
for lab, row in brics.iterrows() :
```

If we want to add a column to a DataFrame by calling a function on another column, the iterrows() method in combination with a for loop is not the preferred way to go. Instead, you'll need to use apply().
・A column COUNTRY should be added to cars, containing an uppercase version of the country names.
Using .apply(str.upper)
cars = cars["country"].apply(str.upper)

Chapter 3. Python Data Science Toolbox (Part 1)

Exercises

If the key is in the dictionary langs_count, add 1 to the value corresponding to this key in the dictionary, else add the key to langs_count and set the corresponding value to 1. Use the loop variable entry in your code.

```
# Initialize an empty dictionary: langs_count
langs_count = {}

# Extract the 'lang' column from DataFrame: col
col = df['lang']
```

```
# Iterate over lang column in DataFrame
for entry in df['lang']:

# If the language is in langs_count, add 1
    if entry in langs_count.keys():
        langs_count[entry]+=1
# Else add the language to langs_count, set the value to 1
    else:
        langs_count[entry]=1

# Print the populated dictionary
print(langs_count)
```

***

Define the function count_entries(), which has two parameters. The first parameter is df for the DataFrame and the second is col_name for the column name.
Complete the bodies of the if-else statements in the for loop: if the key is in the dictionary langs_count, add 1 to its current value, else add the key to langs_count and set its value to 1. Use the loop variable entry in your code.
Return the langs_count dictionary from inside the count_entries() function.
Call the count_entries() function by passing to it tweets_df and the name of the column, 'lang'. Assign the result of the call to the variable #result.

```
# Define count_entries()
def count_entries(df, col_name):
    """Return a dictionary with counts of
    occurrences as value for each key."""
# Initialize an empty dictionary: langs_count
    langs_count = {}
# Extract column from DataFrame: col
    col = df[col_name]
# Iterate over lang column in DataFrame
    for entry in col:
# If the language is in langs_count, add 1
        if entry in langs_count.keys():
            langs_count[entry] += 1
# Else add the language to langs_count, set the value to 1
        else:
            langs_count[entry] = 1
# Return the langs_count dictionary
    return langs_count
# Call count_entries(): result
result = count_entries(tweets_df, 'lang')
```

```
# Print the result
print(result)
```

Nested functions

To avoid writing out the same computations within function repeatedly.

・Scopes searched:

Local scopes-Enclosing functions-Global-Built-in // LEGB rule

```
Define three_shouts
def three_shouts(word1, word2, word3):
   Returns a tuple of strings
   concatenated with '!!!'.
# Define inner
   def inner(word):
      Returns a string concatenated with '!!!'.
      return word + '!!!'
# Return a tuple of strings
   return (inner(word1), inner(word2), inner(word3))
# Call three_shouts() and print
print(three_shouts('a', 'b', 'c'))
<script.py> output:
   ('a!!!', 'b!!!', 'c!!!')
```

***

Complete the function header of the inner function with the function name inner_echo() and a single parameter word1.
Complete the function echo() so that it returns inner_echo.
We have called echo(), passing 2 as an argument, and assigned the resulting function to twice.
Your job is to call echo(), passing 3 as an argument. Assign the resulting function to thrice.
Hit Submit to call twice() and thrice() and print the results.

```
Define echo
def echo(n):
   """Return the inner_echo function."""
# Define inner_echo
   def inner_echo(word1):
"""Concatenate n copies of word1."""
      echo_word = word1 * n
      return echo_word
```

```
# Return inner_echo
    return inner_echo
# Call echo: twice
twice = echo(2)
# Call echo: thrice
thrice = echo(3)
# Call twice() and thrice() then print
print(twice('hello'), thrice('hello'))
```

***

Complete the function header with the function name shout_echo. It accepts an argument word1 and a default argument echo with default value 1, in that order.
Use the * operator to concatenate echo copies of word1. Assign the result to echo_word.
Call shout_echo() with just the string, "Hey". Assign the result to no_echo.
Call shout_echo() with the string "Hey" and the value 5 for the default argument, echo.
Assign the result to with_echo.

```
define shout_echo
def shout_echo(word1, echo=1):
    "Concatenate echo copies of word1 and three
     exclamation marks at the end of the string."
# Concatenate '!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!'
# Return shout_word
    return shout_word
# Call shout_echo() with "Hey": no_echo
no_echo = shout_echo("Hey")
# Call shout_echo() with "Hey" and echo=5: with_echo
with_echo = shout_echo("Hey", echo=5)
# Print no_echo and with_echo
print(no_echo)
print(with_echo)
# Concatenate echo copies of word1 using *: echo_word
    echo_word = echo*word1
```

```
<script.py> output:
    Hey!!!
    HeyHeyHeyHeyHey!!!
```

***

Lambda function

Lambda x, y: x**y

Map() and lambda functions

So far, you've used lambda functions to write short, simple functions as well as to redefine functions with simple functionality. The best use case for lambda functions, however, are for when you want these simple functionalities to be anonymously embedded within larger expressions. What that means is that the functionality is not stored in the environment, unlike a function defined with def.
To understand this idea better, you will use a lambda function in the context of the map() function.
Recall from the video that map() applies a function over an object, such as a list. Here, you can use lambda functions to define the function that map() will use to process the object. For example:

```
nums = [2, 4, 6, 8, 10]

result = map(lambda a: a ** 2, nums)
```

You can see here that a lambda function, which raises a value a to the power of 2, is passed to map() alongside a list of numbers, nums. The map object that results from the call to map() is stored in result. You will now practice the use of lambda functions with map().
For this exercise, you will map the functionality of the add_bangs() function you defined in previous exercises over a list of strings.
Create a list of strings: spells

```
spells = ["protego", "accio", "expecto patronum", "legilimens"]
# Use map() to apply a lambda function over spells: shout_spells
shout_spells = map(lambda item: item+'!!!', spells)
# Convert shout_spells to a list: shout_spells_list
shout_spells_list=list(shout_spells)

# Print the result
print(shout_spells_list)
```

```
<script.py> output:
   ['protego!!!', 'accio!!!', 'expecto patronum!!!', 'legilimens!!!']
```

The function filter() offers a way to filter out elements from a list that don't satisfy certain criteria.
Your goal in this exercise is to use filter() to create, from an input list of strings, a new list that contains only strings that have more than 6 characters.

```
# Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas', 'gimli',
'gandalf']
# Use filter() to apply a lambda function over fellowship: result
result = filter(lambda member: len(member)>6, fellowship)
# Convert result to a list: result_list
result_list=list(result)
# Print result_list
print(result_list)
# Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""
    hodgepodge = ''
    for word in args:
        hodgepodge += word
    return hodgepodge
```

gibberish() simply takes a list of strings as an argument and returns, as a single-value result, the concatenation of all of these strings. In this exercise, you will replicate this functionality by using reduce() and a lambda function that concatenates strings together.

***

Write a lambda function and use filter() to select retweets, that is, tweets that begin with the string 'RT'.
In the filter() call, pass a lambda function and the sequence of tweets as strings, tweets_df['text']. The lambda function should check if the first 2 characters in a tweet x are 'RT'. Assign the resulting filter object to result. To get the first 2 characters in a tweet x, use x[0:2]. To check equality, use a Boolean filter with ==.
result = filter(lambda x: x[0,2]=='RT', tweets_df['text'])

Chapter 4. Python Data Science Toolbox (Part 2)

Iterators and iterables

Iterable

・Examples: lists, strings, dictionaries, file connections
・Definition: An object with an associated iter() method
・Applying iter() to an iterable creates an iterator
Iterator
・Produces next value with next()
Once we have the iterator defined, we pass it to the function next() and this returns the first value

```
Word = ‘mom’
It=iter(word)
Next(it)
‘m’
```

Calling next again on the iterator returns the value until there are no values left to return and then it throws us a StopIteration error.

```
Next(it)
‘o’
Next(it)
‘m’
```

*Iterating at once with **

```
Word=’Data’
It=iter(word)
Print(*it)
D a t a
```

This star operator unpacks all elements of an iterator or an iterable

Iterating over dictionaries

```
Datascientists={‘john:brown’, ‘lisa’:stewart’}
For key, value in datascientists.items():
Print(key, value)

john brown
lisa stewart
```

Iterating over file connections

```
file=open(‘file.txt’)
```

```
it=iter(file)
print(next(it))
```

```
fist line
```

```
print(next(it))
second line
```

Exercises

1. Create a for loop to loop over flash and print the values in the list.

Use person as the loop variable.
Print each list item in flash using a for loop

```
for person in flash:
   print(person)
```

Not all iterables are actual lists. A couple of examples that we looked at are strings and use the range() function. Now, Let’s focus on the range() function.
You can use range() in a for loop as if it's a list to be iterated over:

```
for i in range(5):
   print(i)
```

Recall that range() doesn't actually create the list; instead, it creates a range object with an iterator that produces the values until it reaches the limit.

Using enumerate()

A function that takes any iterable as an argument, such as a list, and returns an enumerate object, produces a sequence of tuples, and each of the tuples is an index-value pair.

```
fruits = [‘apple’, ‘peach’, ‘pear’, ‘grape’]
e=enumerate(fruits)
print(type(e))
 <class ‘enumerate’>
```

*use the function list to turn this ‘enumerate’ object into a list of tuples and print it to see what it contains.

```
e_list=list(e)
print(e_list)
```

```
[(0, ‘apple’), (1, ‘peach’), (2, ‘pear’), (3, ‘grape)]
```

The enumerate object itself is also an iterable and we can loop over it while unpacking its elements using the clause: for intex, value in enumerate(fruits).

```
fruits = [‘apple’, ‘peach’, ‘pear’, ‘grape’]
for index, value in enumerate(fruits):
```

```
print(index, value)

0 apple
1 peach
2 pear
3 grape
```

It is default behavior of enumerate to begin indexing at 0. However, you can alter this with a second argument: start

```
for index, value in enumerate(fruits, start=10):
10 apple
11 peach
12 pear
13 grape
```

Using zip()

Accepts any number of iterables and returns a zip object-an iterator of tuples

```
fruits = [‘apple’, ‘peach’, pear’, ‘grape’]
usage = [‘juice’, ‘juice’, ‘dryfruit’, ‘wine’]
z=zip(fruits, usage)
print(type(z))
```

<class ‘zip’> _is a zip object which is an iterator of tuples. We can turn this zip object into a list and print the list

```
z_list=list(z)
print(z_list)
[(‘apple’, ‘juice’), (‘peach’, ‘juice), (‘pear’, ‘dryfruit’), (‘grape’, ‘wine’)] – List of tuples
```

The first element is a tuple containing the first elements of each list zipped.

We can also use a for loop to iterate over the zip object and print the tuples

```
fruits = [‘apple’, ‘peach’, pear’, ‘grape’]
usage = [‘juice’, ‘juice’, ‘dryfruit’, ‘wine’]
for z1, z2 in zip(fruits, usage)
print(z1, z2)

apple juice
peach juice
pear dry fruit
grape wine
```

We could have also used the splat operator to print all the elements

```
fruits = [‘apple’, ‘peach’, pear’, ‘grape’]
```

```
usage = [‘juice’, ‘juice’, ‘dryfruit’, ‘wine’]
z=zip(fruits, usage)
print(*z)
```

Populate a list with a for loop

	nums=[12, 8, 21, 3, 16]
1. initialize a new empty list	new_nums=[]
2. loop through the old list	for num in nums:
3. add 1 to each entry and append to the new list	new_nums.append(num+1)
	print(new_nums)
	[13, 9, 22, 4, 17]

Or, you can simply

Populate a list with a List comprehension

```
nums=[12, 8, 21, 3, 16]
new_nums=[num+1 for num in nums]
print(new_nums)
[13, 9, 22, 4, 17]
```

Nested loops

```
pairs_1=[]
for num1 in range (0, 2):
    for num2 in range (6,8):
        pairs_1.append(num1, num2)
print(pairs_1)
[(0, 6), (0, 7), (1, 6), (1, 7)]
```

Nested loops with comprehension

Within the square brackets, place the desired output expression (num1, num2), followed by the two required for loop clauses.

```
pairs2=[(num1, num2) for num1 in range (0,2) for num2 in range (6,8)]
```

・3 methods of calculating square in Python

multiplying numbers two times: (number*number)
using Exponent Operator (**): (number**2)
using math.pow() method: (math.pow(number,2)

In this exercise, you will be writing a list comprehension within another list comprehension, or nested list comprehensions. One of the ways in which lists can be used are in representing multidimensional objects such as matrices. Matrices can be represented as a list of lists in Python. For example, a 5 x 5 matrix with values 0 to 4 in each row can be written as:

```
matrix = [[0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4]]
```

Your task is to recreate this matrix by using nested list comprehensions. Recall that you can create one of the rows of the matrix with a single list comprehension. To create the list of lists, you simply need to supply the list comprehension as the output expression of the overall list comprehension:

[[output expression] for *iterator* **variable in** *iterable***]**

Note that here, the output expression is itself a list comprehension.

In the inner list comprehension - that is, the output expression of the nested list comprehension - create a list of values from 0 to 4 using range(). Use col as the iterator variable.

In the iterable part of your nested list comprehension, use range() to count 5 rows - that is, create a list of values from 0 to 4. Use row as the iterator variable; note that you won't be needing this to create values in the list of lists.

```
# Create a 5 x 5 matrix using a list of lists: matrix
matrix = [[col for col in range(5)] for row in range(5)]

# Print the matrix
for row in matrix:
   print(row)
   [0, 1, 2, 3, 4]
   [0, 1, 2, 3, 4]
   [0, 1, 2, 3, 4]
   [0, 1, 2, 3, 4]
   [0, 1, 2, 3, 4]
```

Advanced comprehensions

```
[num**2 for num in range(10) if num % 2==0]
[0, 4, 16, 36, 64]
```

The resulting list is the square of values in range(10) under the condition that (if) the value itself is even.

%-modulo operator-produces the remainder from the division of the first argument by the second. Thus, the integer modulo two is equal to zero if and only if the integer is even.

Conditionals in comprehensions

[output expression for iterator variable in iterable if predicate expression]

[num**2 if num%2 ==0 else 0 for num in range(10)]

(For an even integer we output its square, in any other case signified by else clause (for odd integers) we output 0)

[0, 0, 4, 0, 16, 0, 36, 0, 64, 0]

Comprehensions can include conditionals on the output expressions and/or conditionals on the interable.

1. basic conditional [output expression for iterator in the iterable]
2. advanced conditional [output expression + conditional on output for iterator variable + conditional on iterable]

Dict comprehensions

Use curly braces {}
The key and value are separated by a colon in the output expression (num:-num)

```
pos_neg={num: -num for num in range (9)}
print(pos_neg)
```

In this example, we are creating a dictionary with keys positive integers and corresponding values the respective negative integers.

```
{0: 0, 1:-1, 2:-2, 3:-3, 4:-4, 5:-5, 6:-6, 7:-7, 8:-8}
```

Generator functions

Produces a generator object and we can iterate over it with a for loop to print the values it yields.

```
sequence.py
def num_sequence(n) :
'Generate values from 0 to n.'
i=0
while i < n
    yield i
    i+1
```

(Adding 1 to the initial 0 value of i, until i equals n, and then the generator ceases to yield values.)

Generator expressions basically have the same syntax as list comprehensions, except that it uses parentheses () instead of brackets []. Furthermore, if you have ever iterated over a dictionary with .items(), or used the range() function, for example, you have already encountered and used generators before, without knowing it! When you use these functions, Python creates generators for you behind the scenes.

# Open a connection to the file
with open(…) as …:

Chapter 5: Importing Data in Python (Part1)

Reading a text file

•assign the filename to a variable as a string
•pass the filename to the function open
•and also pass it to the argument, mode=‘r’ (which makes sure that we can only read it; we wouldn’t want to accidentally write to it!)

filename=’shakespeare’
file=open(filename, mode=’r’) #’r’ is to read
If you want to open a file in order to write to it, pass the argument, mode=’w’
file=open(filename, mode=’w’)
#assign the text from the file to a variable text by applying the method read to the connection to the file
text=file.read()
#After this, make sure that you close the connection to the file using the command
file.close()
#Finally you can print the file
print(text)

•You can avoid having to close the connection to the file by using ‘with’ statement.

```
with open(‘shakespeare’, ‘r’) as file:
print(file.read()
```

This allows you to create a context in which you can execute commands with the file open. Once out of this clause/context, the file is no longer open and, for this reason, ‘with’ is called Context manager.

Flat files

Are basic text files containing records, that is, table data, without structured relationships. Here records represent rows of fields or attributes, each of which contains at most one item of information.

File extension

.csv- comma separated values
.txt-text file
values can be separated by characters other than commas, such as a tab-tab-delimited file

Importing flat files

1. Using NumPy

If the files consist entirely of numbers and we want to store them as a numpy array we can use numpy. Here is why.

- Numpy arrays are the Python standard for storing numerical data. They are efficient, fast and clean.
- Numpy arrays are often essential for other packages (e.g. scikit learn).
- Has a number of built-in functions that make it easier and more efficient for us to import data as arrays, for example loadtxt(), genfromtxt(), etc.

Most of the time you will need to import datasets which have different data types in different columns; one column may contain strings and another floats, for example. The function np.loadtxt() will break at this. There is another function, np.genfromtxt(), which can handle such structures. Otherwise, we can pass dtype=None function, which will figure out what types each column should be.
There is also another function np.recfromcsv() that behaves similarly to np.genfromtxt(), except that its default ’dtype’ is None.

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

Customizing NumPy import

- skiprows is used when you want to skip some rows of the text
- if you want only the 1st and 3rd columns of the data, you can use usecols=[0,2]
- you can also import different data types into NumPy arrays: for example dtype=str will ensure that all entries are imported as strings
- delimiter changes the delimiter that loadtxt() is expecting.
  you can use ',' for comma-delimited. csv-comma separated values
  you can use '\t' for tab-delimited. tsv-tab separated values

Note that in Pandas we use sep for the delimiter

Recall how to print a scatter plot

plt.scatter(data_float[:, 0], data_float[:, 1]) *data is represented as floats (0.2, 0.4, 0.6…)*

plt.xlabel('…') *defining the name of the x axis*

plt.ylabel('…') *defining the name of the y axis*

plt.show()

2. Using pandas

Functionalities:

- Two-dimensional labeled data structures
- with columns of potentially different types
- manipulate, slice, reshape, groupby, join, merge
- perform statistics in a missing-value-friendly manner
- deal with time series data

```
import pandas as pd
filename='…'
data=pd.read_csv(filename)
data.head()
```

#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).

```
data=pd.read_csv(file, nrows=5, header=None)
```

#Build a numpy array from the resulting DataFrame in data and assign to data_array.

```
data_array =np.array(data)
```

· 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.

#Complete the sep (the pandas version of delimiter), comment and na_values arguments of pd.read_csv(). comment occur after the character '#' in the file. na_values takes a list of strings to recognize as NA/NaN, in this case the string 'Nothing'.

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

In computer science, in the context of data storage, **serialization** is the process of translating data structures or object state into a format that can be **stored** (for example, in a file or memory buffer) or **transmitted** (for example, across a network connection link) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously linked.

This process of serializing an object is also called marshaling an object in some situations. The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (also called unmarshalling).

- Given an Excel file imported into a variable spreadsheet, you can retrieve a list of the sheet names using the attribute spreadsheet.sheet_names.

```
# Print sheet names
print(xls.sheet_names)
   ['2002', '2004'] printed list of the sheets' names on the Excel file
```

Importing SAS/Stata files using pandas

SAS: Statistical Analysis System
Stata: 'Statistics' and 'Data'

The most common SAS files have the extension .sas7bdat (dataset files) and .sas7bcat (catalog files)

```
import pandas as pd
from sas7bdat import SAS7BDAT
with SAS7BDAT('urbanpop.sas7dat') as file:
     df_sas=file.to_data_frame()
```

In this case, you can bind the variable file to the file 'urbanpop.sas7bdat' in the context manager (with). Within this context, you can assign the result of applying the method, to_data_frame to file (file.to_data_frame()) to the variable df_sas.

Importing Stata files

Stata files have extension .dta and we can import them using pandas.
In this case, we don't need to initialize a context manager.

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

#Here, you'll import 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.

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.

```
# 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()
```

Importing HDF5 files

HDF5: Hierarchical Data Format version 5
Standard mechanism for storing large quantaties of numerical data

```
import h5py
filename='…'
data=h5py.File(filename, 'r') #'r' is to read only
print(type(data))
```

```
<class'h5py._hl.files.File'>
```

The structure of HDF5 files

```
for key in data.keys()
meta
quality
```

```
strain
```

- meta: contains metadata for the file
- quality: contains information about data quality
- strain: contains strain data from the interferometer; the main measurement performed by LIGO.

To find out what type of metadata there is, you can easily print out the keys

```
for keys in data['meta'].keys():
    print(key)
```

```
Description
Duration
Type
Detector
```

After knowing the keys, you can access any metadata of interest. If you are interested in 'Description' and 'Detector', you can print out the corresponding values:

```
print(data['meta']['Description'].value, data['meta']['Detector'].value)
```

```
b'Strain data time series from LIGO' b'H1'
```

The data in the file is 'Strain data time series from LIGO' and the detector used was 'H1'

Importing MATLAB files

MATLAB: Matrix Laboratory
Is a numerical computing environment that is an industry standard in the disciplines of engineering and science.
The data is saved in .mat format

In Python, the standard library 'scipy' has functions to read and write .mat files:
- scipy.io.loadmat() – read .mat files
- scipy.io.savemat() – write .mat files

```
import scipy.io
filename='…'
mat=scipy.io.loadmat(filename)
print(type(mat))
```

```
<class 'dict'>
```

How this resulting dictionary relates to the MATLAB workspace is straightforward: the keys of the Python dictionary are the MATLAB variable names and the values of the Python dictionary are objects that are assigned to the variables.

print(type(mat['x']))
mat['x'] is a numpy corresponding to the MATLAB array x in your MATLAB workplace.

Introduction to relational databases

Type of a database that is based upon the relational model of data
Codd's 12 Rules

- Consists of 13 rules (the first rule is zero-indexed)
- Describes what a Relational Database Management System should adhere in order to be considered a relational

Relational Database Management Systems

- PostgreSQL
- MySQL
- SQLite

All of them are using the SQL query language
SQL: Structured Query Language, which describes how you communicate with a database in order to both access and update the information it contains.
The term 'querying' is just a fancy way of saying getting data out from a database.

- Import the function create_engine from the module sqlalchemy and create an engine to connect to the SQLite database 'Chinook.sqlite', which is in your working directory.
  ```
  # Import necessary module
  from sqlalchemy import create_engine

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

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

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

Querying relational databases in Python

Basic SQL query

SELECT * FROM Table_Name
Returns all columns of all rows of the table of interest (*- after select means all columns)
'Table_name' is the name of any of the tables in the database.

Workflow of the SQL querying

1. Import packages and functions
2. Create the database engine
3. Connect to the engine
4. Query the database
5. Save query results to a DataFrame
6. Close the connection

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

Create the engine using the function create_engine

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

To connect to the database after creating the engine, you create an connection object con by applying the method connect to the engine, engine.connect()

```
con=engine.connect()
```

(To query the database, apply the method execute to the connection con and pass it a single argument, the relevant SQL query)

```
rs=con.execute("SELECT*FROM Orders")
```

(This creates a SQLAlchemy results object, which we assign to the variable rs)
To turn the results object rs into a dataframe, we apply the method fetchall to rs and save it as a dataframe using the pandas function DataFrame.

```
df=pd.DataFrame(rs.fetchall())
```

(Fetchall fetches all rows, as you would expect)
To close the connection execute con.close

```
con.close()
```

You can then print the head of the dataframe, as a sanity check

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

To correct the names of the columns, before closing the connection, you can set the dataframe's column names by executing df.columns = rs.keys()

Another way to implement the above mentioned functions with more specific commands, is as follows.

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

Filtering the database records using SQL's WHERE

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

e.g. SELECT * FROM Customer WHERE Country = 'Canada'

Ordering SQL records with ORDER BY

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

"SELECT * FROM Employee ORDER BY BirthDate"

Querying relational databases directly with pandas

4 lines of code were needed to get the results of any particluar line from the database engine.

1. connecting
2. executing a query
3. passing the results to a dataframe
4. naming the columns.

However, we can actually do it in 1 line, by utilizing the pandas function read_sql_query and passing it 2 arguments.
df=pd.read_sql_query('SELECT * FROM Orders', engine)

Advanced querying: The power of SQL is relationships between tables

Using the join query to join columns of different tables together. Specifically, it's an INNER JOIN.

from sqlalchemy import create_engine
import pandas as pd
engine=pd.read_sql_query('SELECT OrderID, CompanyName FROM Orders INNER JOIN Customers on Orders.CustomerID=Customers.CustomersID', engine)

Chapter 6: Importing Data in Python (Part2)

Importing flat files from the web

The urllib package
・Provides interface for fetching data across the web
・urlopen() - accepts URL (Universal Resource Locators) instead of dataframes

Import a function called urlretrieve from the request subpackage of the urllib package
Assign the relevant string to the variable url
We then use the urlretrieve function to write the contents of the url to a file 'wimeequality_white.csv'

```
from urllib.request import urlretrieve
url=('https://s3.amazonaws.com/assets.datacamp.com/production/course_1606/datasets/winequality-red.csv')
urlretrieve(url, 'winequality-red.csv')

#Read file into a DataFrame and print its head
df=pd.read_csv('winequality-red.csv', sep=';')
#Print out the head of the DataFrame
print(df.head())
```

・You have just imported a file from the web, saved it locally and loaded it into a DataFrame. If you just wanted to load a file from the web into a DataFrame without first saving it locally, you can do that easily using pandas. In particular, you can use the function pd.read_csv() with the URL as the first argument and the separator sep as the second argument.

Assign the URL of the file to the variable url.
Read file into a DataFrame df using pd.read_csv(), recalling that the separator in the file is ';'.
Print the head of the DataFrame df.
Execute the rest of the code to plot histogram of the first feature in the DataFrame df.

```
#Import packages
import matplotlib.pyplot as plt
import pandas as pd
url=('https://s3.amazonaws.com/assets.datacamp.com/production/course_1606/datasets/winequality-red.csv')
df=pd.read_csv(url, sep =';')
print(df.head())

# Plot first column of df
pd.DataFrame.hist(df.ix[:, 0:1])
plt.xlabel('fixed acidity (g(tartaric acid)/dm$^3$)')
plt.ylabel('count')
plt.show()
```

・Use pd.read_excel() to import an Excel spreadsheet.

- Assign the URL of the file to the variable url.
- Read the file in the url into a dictionary xls using pd.read_excel() recalling that, in order to import all sheets you need to pass None to the argument sheet_name.
- Print the names of the sheets in the Excel spreadsheet; these will be the keys of the dictionary xls. Note that the output of pd.read_excel() is a Python dictionary with sheet names as keys and corresponding DataFrames as values.
- Print the head of the first sheet using the sheet name, not the index of the sheet! The sheet name is '1700'

```
import pandas as pd
url=’http://s3.amazonaws.com/assets.datacamp.com/course/importing_data_into_r/latitude.xl
s'
xls = pd.read_excel(url, sheet_name=None)
print(xls.keys())
# Print the head of the first sheet (using its name, NOT its index)
print(xls['1700'].head())
```

HTTP requests to import files from the web

Focuses on URLs that are web addresses OR the locations of web sites.
Ingredients:
- protocol identifier http or https
- resource name such as datacamp.com

HTTP
• HyperText Transfer Protocol
• Foundation of data communication for the World Wide Web
• HTTPS – more secure form of HTTP
• Going to website = sending HTTP request
- Get request
• urlretrieve () performs GET request too and also saves the relevant data locally
HTML – HyperText Markup Language

GET requests using urllib

```
from urllib.request import urlopen, Request
url = ‘https://www.wikipedia.org/’
request=Request(url)
response=urlopen(request)
html=response.read()
response.close()
```

request the url,catch the response with the urlopen() function, extract the response using the response.read() function

GET requests using requests

One of the most downloaded Python packages
import requests
url='https://www.wikipedia.org/'
 # Package the request to the URL, send the request and catch the response with a single function requests.get()
r=requests.get(url)
 #return the HTML as a string
text=r.text
・Note that unlike the urllib, you don't have to close the connection when using requests!

Scraping the web in Python. BeautifulSoup

In the previous codes you retrieved the relevant HTML as a string. But generally HTML represents a mix of unstructured and structured data.
Thus, to turn HTML that you have scraped from the world wide web into useful data, you'll need to parse it and extract structured data from it.
BeautifulSoup
・Parse and extract structured data from HTML
Name: In web development, the term 'tagsoup' refers to structurally incorrect HTML code written for a web page. What Beautiful Soup does best is to make tag soup beautiful again.
from bs4 import BautifulSoup
import requests
url='https://www.crummy.com/software/BeautifulSoup/'
r=requests.get(url)
html_doc=r.text
soup=BautifulSoup(html_doc)
Exploring BeautifulSoup

・To extract title of the web page
print(soup.title)
・To extract the text
print(soup.get_text())
・find_all() extracts the URLs of all of the hyperlinks in the HTML.
Note that hyperlinks are defined by the HTML tag <a> but passed to the function find_all without angle brackets.

for link in soup.find_all('a')
print(link.get('href'))

Introduction to APIs and JSONs

APIs

・Application Programming Interface

・Protocols and routines for building and interacting with software applications

OMDb API　The open Movie Database

JSON

・Name: JavaScript Object Notation

・Real-time server-to-browser communication

・Human readable

・Resemble key-value pairs in a Python dictionary, that's why when loading JSONs into Python, it is natural to store them in a dict.

・The keys in JSONs will always be strings enclosed in quotation marks

・The values can be strings, integers, arrays or even objects

Loading JSONs in Python

Assuming that, JSON is stored in our working directory as 'snakes.json'. To load the JSON into my Python environment, we would execute the codes:

```
import  json
#Load the 'snakes.json' into the variable json_file
with open('snakes.json', 'r') as json_file:
      json_data=json.load(json_file)
 type(json_data)
dict
```

・To print key-value pairs to the console, we can then iterate over the key-value pairs using a for loop.

```
for key, value in json_data.items():
    print(key+';', value)
```

APIs and interacting with the world wide web

・Set of protocols and routines

・Bunch of codes allowing 2 software programs to communicate with each other

```
#Import request package
import requests
url='http://www.ombdapi.com/?t=hackers'
json_data=r.json()
for key, value in json_data.item():
    print(key+':', value)
```

- **r.json()-**Response objects, such as r, have an associated method json, which is a built-in JSON decoder for dealing with JSON data.
- This returns a dictionary and then we can print all the key-value pairs to check out what we pulled out from the OMBD API.

• What was the URL that was used above?
- http-making an HTTP request
- ?t=hackers

This string that begins with a question mark is called a Query String.
What follows the question mark in the Query String is the query we are making to the OMBD API. ‘t=hackers’ query means that we asked the API to return the data about the movie with the title Hackers.

The Twitter API and Authentication

Using Tweepy: Authentication handler

First off, it has a tw_auth.py handler which takes care of all the nasty stuff for you: all you need to do is to

```
import tweepy, json
access_token='…'
access_token_secret='…'
consumer_key='…'
consumer_secret='…'
auth=tweepy.0AuthHandler(consumer_key, consumer_secret)
auth.sec_access_token(access_token, access_token_secret)
```

After this, you’ll need to define your Twitter stream listener class
st_class.py

Here we define a Tweet listener that creates a file called ‘tweet.txt’, collects streaming tweets and writes and writes them to the file ‘tweets.txt’. Once 100 tweets have been streamed, the listener closes the file and stops listening.

```
class MyStreamListener(tweepy.StreamListener) :
    def _ _init_ _(self.api=None) :
    super(MyStreamListener, self)._ _init_ _()
    self.num_tweets=0
    self.file=open(‘tweets.txt’, ‘w’)
    def on_status(self, status) :
    tweet=status._json
    self.file.write(json.dumps(tweet)+’\\n’)
    tweet_list.append(status)
    self.num_tweets+=1
    if self.num_tweets < 100 :
        return True
    else:
        return False
    self.file.close()
```

Now that we have written our Twitter Stream Listener Class, all we need to do is to create an instance of it and authenticate it.

```
l=MyStreamListener() #l=listener
stream=tweepy.Stream(auth, l)
```

We can then stream tweets that contain keywords of choice by applying the *filter* method to the object *stream*.

```
stream.filter(track=['apples', 'oranges']) #'track' is an argument
```

・In this exercise, you'll read the Twitter data into a list: tweets_data
Import json package. Assign the file 'tweets.txt' to the variable tweets_data_path. Initialize tweets_data as an empty list to store the tweets in. Open connection to file. Within the for loop initiated by for line in tweets_file:, load each tweet into a variable, tweet, using json.loads(), then append tweet to tweets_data using the append() method. Close connection to file. Print the keys of the first tweet dict.

```
import json
tweets_data_path='tweets.txt'
tweets_data=[]
tweets_file=open(tweets_data_path, 'r')
for line in tweets_file :
    tweet=json.loads(line)
    tweets_data.append(tweet)
tweets_file.close()
print(tweets_data[0].keys())
```

・Now you have the Twitter data in a list of dictionaries, tweets_data, where each dictionary corresponds to a single tweet. Next, you're going to extract the text and language of each tweet. Your task is to build a DataFrame in which each row is a tweet and the columns are 'text' and 'lang'.

```
import pandas as pd
df=pd.DataFrame(tweets_data, columns=['text', 'lang'])
print(df.head())
```

・Now that you have your DataFrame of tweets set up, you're going to do a bit of text analysis to count how many tweets contain the words 'clinton', 'trump', 'sanders' and 'cruz'. In the pre-exercise code, we have defined the following function word_in_text(), which will tell you whether the first argument (a word/candidate) occurs within the 2nd argument (a tweet).

```
import re

def word_in_text(word, text):
    word = word.lower()
    text = text.lower()
    match = re.search(word, text)

    if match:
        return True
    return False
```

You're going to iterate over the rows /tweets/ of the DataFrame and calculate how many tweets contain each of our keywords. The list of objects for each candidate has been initialized to 0.

```
[clinton, trump, sanders, cruz] = [0, 0, 0, 0]
for index, row in df.iterrows():
    clinton += word_in_text('clinton', row['text'])
    trump += word_in_text('trump', row['text'])
    sanders += word_in_text('sanders', row['text'])
    cruz += word_in_text('cruz', row['text'])
```

・Now that you have the number of tweets that each candidate was mentioned in, you can plot a bar chart of this data. You'll use the statistical data visualization library seaborn.You'll first import matplotlib.pyplot and seaborn as sns using the aliases plt and sns, respectively. Create a list of labels, cd. You'll then construct a barplot of the data using sns.barplot. The first argument should be the list of labels to appear on the x-axis (created in the previous step). The second argument should be a list of the variables you wish to plot (i.e. a list containing clinton, trump, etc).

```
import matplotlib.pyplot as plt
import seaborn as sns
# Set seaborn style
sns.set(color_codes=True)
cd = ['clinton', 'trump', 'sanders', 'cruz']
ax = sns.barplot(cd, [clinton, trump, sanders, cruz])
ax.set(ylabel="count") #name of the y axis
plt.show()
```

STOP WORDS (quanteda)

In computing, stop words are words which are filtered out before processing of natural language data. Stop words are generally the most common words in a language; there is no single universal list of stop words used by all the natural language processing tools, and indeed not all tools even use such a list.

RegEx Regular Expressions

'^\\p{L}+$'
\p{Ll} or \p{Lowercase_Letter}: a lowercase letter that has an uppercase variant.

Basic R objects and commands
R has three types of objects: vector, data frame and matrix
1. R's most basic objects are vectors. Vectors contain a set of values, e.g. vec_num is a numeric vector, while vec_char is a character vector.
Once a vector is created you can extract elements with [] operator and index numbers of desired elements.
print(vec_num[1])
print(vec_char[c(1, 3)])
2. A data frame combines multiple vectors to construct a dataset.
You can combine vectors into a data frame only if they have the same lengths.
You can use subset() to select records in the data frame.
3. Matrices
Similar to a dataframe, a matrix contains multi-dimensional data. In contrast to a data frame, its values must all be the same type.
mat <- matrix(c(1, 3, 6, 8, 3, 5, 2, 7), nrow = 2)
print(mat)
```
##        [,1] [,2] [,3] [,4]
## [1,]   1      6    3    2
## [2,]   3      8    5    7
```

You can extract the size of a matrix by dim() that returns a two-element numeric vector (number of rows and columns).
print(dim(mat))
## [1] 2, 4

Test statistic

A single number than can be computed from observed data (Point 1) and from data you *simulate under null hypothesis (Point 2)*
It serves as a basis of comparison between what the hypothesis predicts and what we observed.

The Null hypothesis

You should choose your test statistic to be something pertinent to the question you are trying to answer with your hypothesis test, in this case, are the two states' electoral outcome different (assuming, that sample A is the electoral outcome of one state, and B – another). It is possible that the observed mean difference of 1.16 was by chance. You will compute the probability of getting at least a 1.16 difference in mean electoral outcome under the hypothesis that the distribution of electoral outcomes for 2 states are identical.

If they are identical, they should have the **same/equal mean vote share**.

So, the difference between mean vote share should be zero (in order to be zero, they need to be equal)

*np.mean(sample_A)–np.mean(sample_B)=np.mean(perm_sample_A)–np.mean(perm_sample_B)

We use a permutation test with a test statistic of the difference means to test this hypothesis. First, we need to have the mean difference in the observed data. Second, the mean difference of permutation samples related to the electoral outcome of the two states.

1. The value of the test statistic computed from the original data

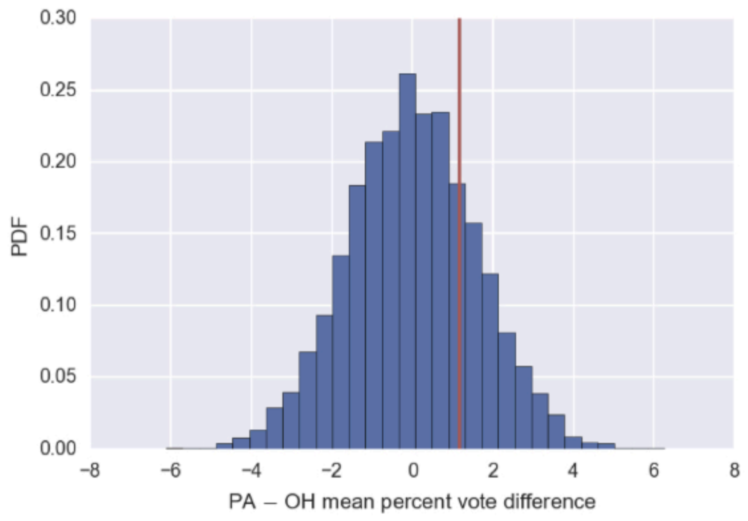
np.mean(sample_A) – np.mean(sample_B)=**1.16**

2. The value of a test statistic computed from permutation samples

np.mean(perm_sample_A) – np.mean(perm_sample_B)=**1.12**

So, for this permutation we didn't quite get as big of a difference in means than what was observed in the original data.

3. Repeat permutation of the Point 2 10.00 times under the null hypothesis (by 'for loop')



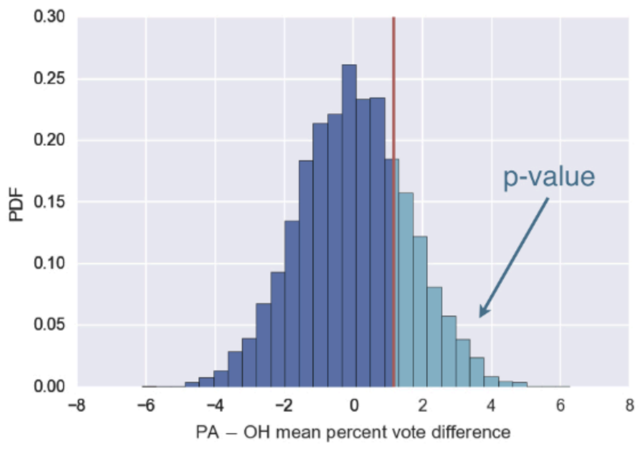
histogram of all the permutation replicates (***the difference of means*** *simulated under the null hypothesis lies somewhere between -4 and 4%)*

However, the actual mean percent vote difference (Point 1) was 1.16% (displayed by the red line)

4. If we tally up the area of the histogram that is to the right of the red line, we get that about **'23 %' of the simulated elections** had at least a 1.16% difference or greater

(the t statistic about the electoral outcome of the states under the null hypotheis

which is equal or greater than the t statistic of the original data)



This value, of point 23, is called a p value.
The probability of observing
test statistic that is
equal or more extreme than
the one you observed.
(given that the null hypothesis is true)
The p-value is the fraction of your simulated data set for which the test statistic is as extreme as for the real data.
It is the probability of getting at least 1.16% difference in the mean vote share assuming the states have identically distributed voting (*).
So is it probable that we would actually observe the outcome we got if the states had identically distributed voting? Sure, it happened 23% of time under the null hypothesis.
When the p-value is small, it is often said that the data are statistically significantly different from what we would observe under the null hypothesis.
How to code
As discussed in the video, a permutation replicate is a single value of a statistic computed from a permutation sample. The function to write permutation replicates is:
draw_perm_reps()
The function has call a signature:
draw_perm_reps(data1, data2, func, size=1)
Importantly, func must be a function that takes two arrays as arguments.

```
def draw_perm_reps(data_1, data_2, func, size=1):
"""Generate multiple permutation replicates."""
   # Initialize array of replicates: perm_replicates
perm_replicates = np.empty(size)
   for i in range(size):
      # Generate permutation sample
      perm_sample_1, perm_sample_2 = permutation_sample(data_1, data_2)
      # Compute the test statistic
      perm_replicates[i] = func(perm_sample_1, perm_sample_2)
   return perm_replicates
```

Definition of p Value

P value is 'The Probability' for the 'Null Hypothesis' to be 'True'

Null Hypothesis

Treats

Everything same

Everything equal

If the p value > level of significance, Null hypothesis Accepted

If the p value < level of significance, Null hypothesis is Rejected

Pearson correlation coefficient

is a measure of how much of the variability in two variables is due to them being correlated.

Bootstrapping

is the use of resampled data to perform statistical inference

---

Using a Classifier. Supervised learning

Split your data into 2 sets, **training set** and **test set**

1. You train or fit the classifier on the training data
2. Then you make predictions on the test data
3. Compare these predictions with the known labels
4. Compute the accuracy of predictions

train_test_split = returns 4 arrays:

training data - X train

test data – X test

training labels – y train

test labels – y test

Then, we fit the classifier to the training data and make predictions on the test data

```
knn = KNeighborsClassifier(n_neighbors=8)
knn.fit(X_train, y_train) # fit both X and Y
y_pred = knn.predict(X_test) #predict Y using the X test data
```

Ways to normalize the data

Standardization: Subtract each feature by that feature mean and divide by its standard deviation. The result is that all features are ***centered around zero*** and have ***variance of one***. You can also subtract the minimum and divide by range. So that the normalized dataset has *minimum zero* and *maximum one.*

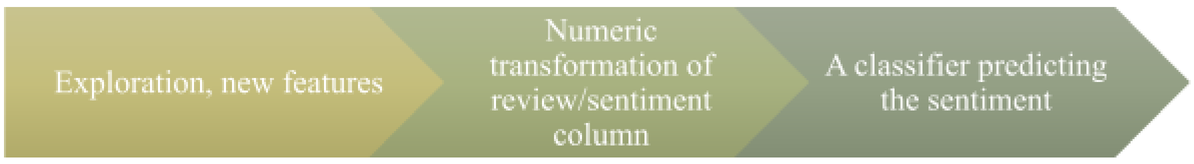
Can also normalize so that data ranges from -1 to 1.

```
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
            ]
        )),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Unsupervised learning in Python / Dendrogram

In complete linkage, the distance between clusters is the distance between the furthest points of the clusters.
In single linkage, the distance between clusters is the distance between the closest points of the clusters.

Sentiment Analysis in Python


Exploration, new features
Numeric transformation of review/sentiment column
A classifier predicting the sentiment

The algorithms used for sentiment analysis could be split into 2 main categories.
・The first rule is lexicon based.
Such methods most commonly have a predefined list of words with a valence score. For example, nice could be +2, good +1, terrible -1, and so on. The algorithm then matches the words from the lexicon to the words in the text and either sums or averages the scores in some way.

e.g. 'Today was a good day.'

Each word gets a score.
Today:0, was:0, good:+1, day:0
To get the total valence we sum the score.

Total valance: +1

In this case, we have a positive sentence.

・A second category is automated systems, which are based on **machine learning**. The task is usually modeled as a classification problem where using some **historical data** with known sentiment, we need to predict the sentiment of a new piece of text.
We can calculate the valence of a text, using Python's '**textblob' library**.

```
text = 'Today was a good day.'
from textblob import TextBlob
TextBlob(text)
```

Bag of words

A bag-of-words is an approach to transform text to **numeric form**. It basically builds a vocabulary of all the words occurring in the document and keeps track of their frequencies. In this case, we lose the word order and grammar rules, that's why this approach is called a 'bag of words'- resembling dropping a bunch of items in a bag and losing any sense of their order.

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer(max_features=1000)
vect.fit(data.review)
X = vect.transform(data.review)
```

The result is sparse matrix, which stores only the entities that are non-zero, where the rows correspond to the number of the rows in the data set, and columns to the BOW vocabulary.

To look at its contents, we perform
```
my_array = X.toarray()
```
to transform it back to 'NumPy array'

We can build a pandas DataFrame from the array, where the columns' names are obtained from the get_feature_names() method of the vectorizer.
```
X_df = pd.DataFrame(my_array, columns=vect.get_feature_names())
```
this returns a list where every entry corresponds to one feature

We can also specify.
1. token sequence with ngrams_range=(1_{unigram}, 1_{unigram}) or (1_{unigram}, 2_{bigram})
2. size of the vocabulary of the CountVectorizer(max_feature, max_df, min_df)

Stopwords
Words that occur too frequently and are not considered informative

1. grammatical words (conjunctions, prepositions)
2. **topic of the issue** mentioned too frequently

TfIdf
Term frequency inverse document frequency
The Tfidf vectorizer is another way to construct a vocabulary from our sentiment column.
It tells us how often a given word appears within a document in the corpus.

TF : The term frequency is the log-ratio between the total number of documents and the number of documents that contain a specific word
(log-ratio = absolute differences being divided by the logarithmic mean)

IDF : What inverse document frequency means is that *rare words will have a high inverse document frequency*. – Used to calculate the weight of words that do not occur frequently.

TfIdf = term frequency * inverse document frequency

TfiDF will also highlight words that are more interesting, i.e. words that are common in a specific document but not across all documents. The Tfidfvectorizer also returns a sparse matrix.
Sparse matrix is a matrix with mostly zero values, storing only the non-zero values.
We need to implement this step, in order to apply a supervised machine learning model to a sentiment analysis.

Regularization in logistic regression

Regularization is applied by default in the logistic regression function from the 'sklearn' package.
It uses the so-called L2 penalty, which shrinks all the coefficients towards zero, effectively reducing the impact of each feature.
The parameter that determines the strength of regularization is given by C, which takes a default value of 1.
Higher values of C correspond to low regularization, in other words, the model will try to fit the data as best as possible > *higher accuracy*
Lower values of C correspond to higher penalization (or regularization), meaning that the coefficients of the logistic regression will be closer to 0. The model will be less flexible, because it will not fit the training data so well > *lower accuracy*
We often sacrifice some accuracy when we regularize a model but the benefit is lower complexity and lower chance of overfitting.

Predicting a probability vs. predicting a class

During training a logistic regression model, we applied the predict function to the test set to predict the labels.

```
log_reg = LogisticRegression.fit(X_train, y_train)
y_predicted = log_reg.predict(X_test)
```

The *predict* function predicts a class: 0 or 1, if we are working with a binary classifier.

However, instead of a class we can predict a probability using the *predict_proba* function. We again pass as an argument the test dataset.

```
y_probab = log_reg.predict_proba(X_test)
```

This returns an array of probabilities, ordered by the label of classes – first the class 0, then the class 1.
The probabilities of each observation are displayed on a separate row.
The first value is the probability that the instance is of class 0, and the second of a class 1.

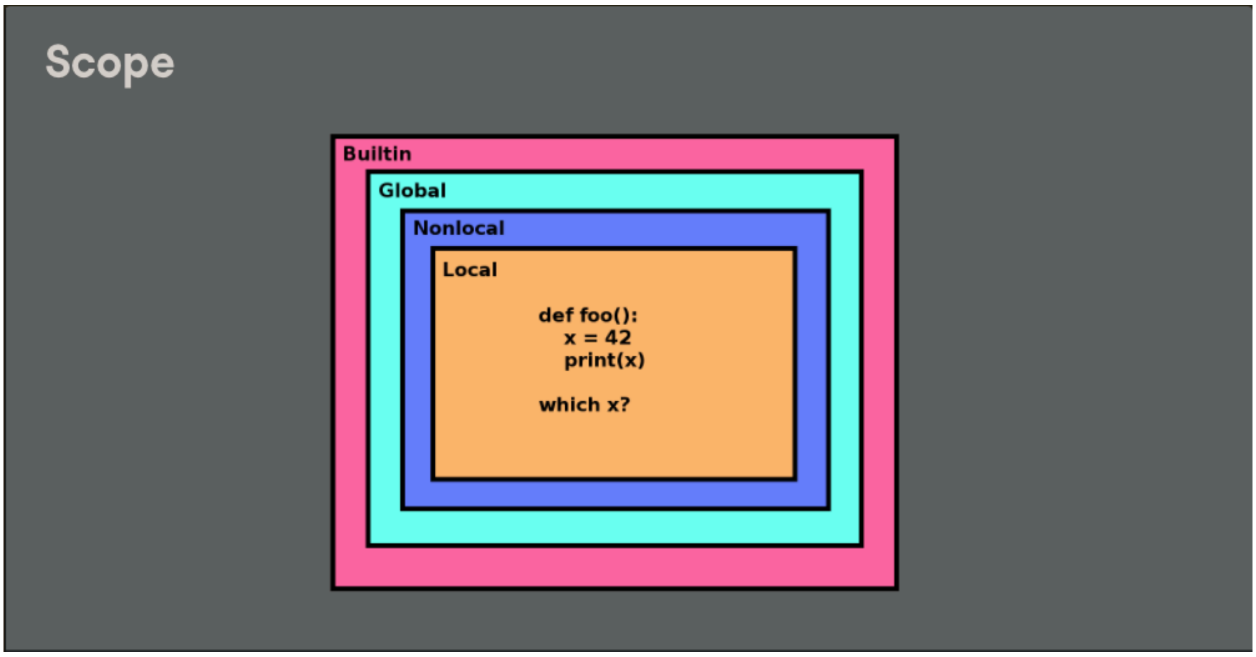
```
y_probab
array([[0.5, 0.4],
       [0.4, 0.5],
       ….,
       [0.7, 0.2]])
```

Therefore, it is common when predicting the probabilities to specify already that we want to extract the probabilities of the class 1.

```
y_proba = log_reg.predict_proba(X_test)[:, 1]
```

However, in order to apply an accuracy score or confusion matrix to the predicted probabilities, we need to encode them as classes.
The default is that any probability higher or equal to 0.5, is translated to class 1, otherwise to class 0.

First, the interpreter looks in the **local scope**. When you are inside a function, the local scope is made up of the arguments and any variables defined inside the function.
If the interpreter can't find the variable in the local scope, it expands its search to the **global scope**. These are the things defined outside the function.
Finally, if it can't find the thing it is looking for in the global scope, the interpreter checks the **builtin scope**. These are things that are always available in Python. For instance, the print() function is in the builtin scope, which is why we are able to use it in our foo() function.
In the case of nested functions, where one function is defined inside another function, Python will check the scope of the parent function before checking the global scope. This is called the **nonlocal scope** to show that it is not the local scope of the child function and not the global scope.



```
def foo():
    x = 42
    print(x)

which x?
```

A nonlocal variable is any variable that gets defined in the parent function's scope, that gets used by the child function.

A closure is Python's way of attaching nonlocal variables to a returned function so that the function can operate even when it is called outside of its parent's scope.

What four values does this script print?

```
x = 50

def one():
  x = 10

def two():
  global x
  x = 30

def three():
  x = 100
  print(x)

for func in [one, two, three]:
  func()
  print(x)
```

50, 30, 100, 30

Good job! `one()` doesn't change the global `x`, so the first `print()` statement prints `50`.

`two()` *does* change the global `x` so the second `print()` statement prints `30`.

The `print()` statement inside the function `three()` is referencing the `x` value that is local to `three()`, so it prints `100`.

But `three()` does not change the global `x` value so the last `print()` statement prints `30` again.

Decorators

Note that
@print_args before the definition of my_function

```
@print_args
def my_function(a, b, c):
  print(a + b + c)

my_function(1, 2, 3)
```

is exactly equivalent to
my_function = print_args(my_function).
Remember, even though decorators are functions themselves, when you use decorator syntax with the @ symbol you do not include the parentheses after the decorator name.

Exploratory data analysis

Simple regression

Compute the linear regression between the dependent and the independent variables
res = linregress(xs, ys)

Compute the line of best fit
fx = np.array([**xs.min**(), **xs.max**()])
fy = fx * **res**.slope + **res**.intercept
plt.plot(fx, fy, '-')

Correlation and simple regression can't measure non-linear relationships.
But multiple regression can!

Multiple regression

Y=real income, x = education, age
*Results = smf.ols('**realinc ~ educ + age**', data = data).fit()*
Results.params

-slope
-intercept

Since the relationship of age with the income was not linear, the slope that the model generated for the 2 variables was very low.
In this case, to describe a nonlinear relationship, one option is to add a new variable that is a non-linear combination of other variables.

Eg.

*age2 = age**2*
*model = smf.ols('**realinc ~ educ + age + age2**', data = data)*
results=model.fit()
results.params

Here, the slope for the income and age becomes substantially higher.

Cluster Analysis in Python

Common unsupervised learning algorithms are

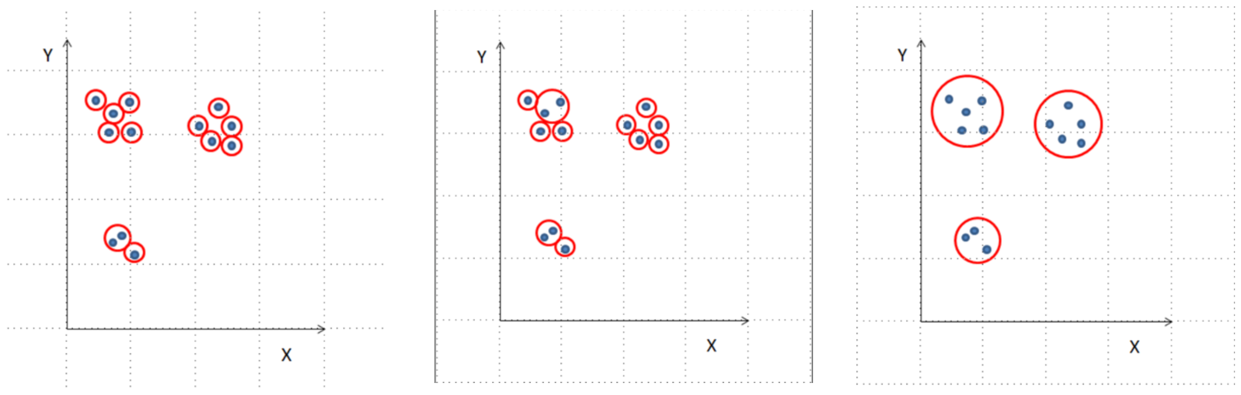
- clustering,
- anomaly detections, and
- neural networks.

Clustering is used to group similar data points together.

Clustering falls under the group of unsupervised learning algorithms as the data is not labeled, grouped or characterized beforehand.

As the training data has no labels, an unsupervised algorithm needs to be used to understand patterns in the data.

1. *A **cluster center** will have 2 attributes – the mean of x and y coordinates.*
2. *Next, the distances between all pairs of cluster centers are computed and the 2 closest clusters are merged.*
3. *The cluster center of the merged cluster is then recomputed.*
4. *In the example, 2 clusters on the bottom left have been merged, so we are left with 12 clusters.*
5. *In the second step, the clusters with the closest centers are merged on the top left.*
6. *At every step, the number of clusters reduces one by one.*
7. *Finally, these are the clusters that you arrive at after the algorithm has run.*



Times Series Data in Python

To convert the string, represented as dtype objects, to the correct data type
Data.column=**pd.to_datetime**(data.column)
Or.
Data=pd.read_csv('data.csv', parse_dates['date'], index_col='date')

Setting the date column as indexes, and not making a copy of the data
Data.set_index('date', inplace=True)

The resulting DateTimeIndex lets you treat the entire DataFrame as time series data.

Normalize the stock data

Divide data by the first value for each series, multiply by 100 and plot the result.
data.div(data.iloc[0]).mul(100).plot()
plt.show()

Value weighted index

Each stock is weighted by the value of the company on the stock market.
As a result, larger companies receive a larger weight.

```
# Select largest company for each sector
components = listings.groupby(['Sector'])['Market Capitalization'].nlargest(1)

# Print components, sorted by market cap
print(components.sort_values(ascending=False))

# Select stock symbols and print the result
tickers = components.index.get_level_values('Stock Symbol')
print(tickers)

# Print company name, market cap, and last price for each component
info_cols = ['Company Name', 'Market Capitalization', 'Last Sale']
print(listings.loc[tickers, info_cols].sort_values('Market Capitalization', ascending=False))
```

Save your analysis to multiple excel worksheets

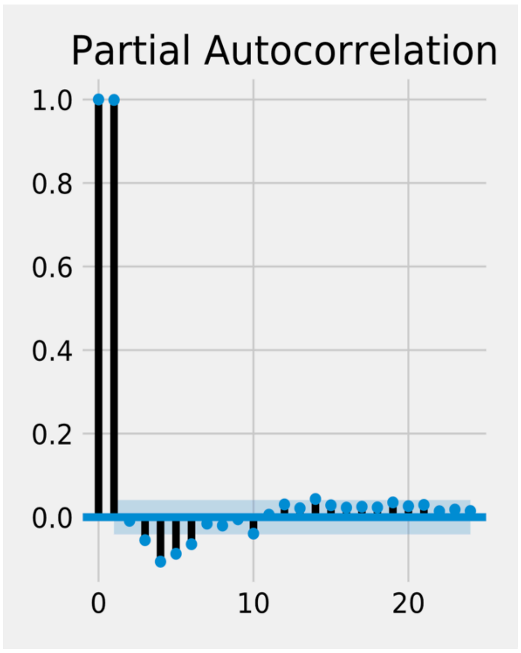
```
# Export data and data as returns to excel
with pd.ExcelWriter('data.xls') as writer:
   data.to_excel(writer, sheet_name='data')
   returns.to_excel(writer, sheet_name='returns')
```

Autoregression (AR)

Moving averages (MA)

Also known as rolling mean, is a commonly used technique in the field of time series analysis.

It can be used to smooth out short-term fluctuations, remove outliers, and highlight long-term trends or cycles.



Taking the rolling mean of your time series is equivalent to 'smoothing' your time series data.
Return = mean + noise + fraction theta of yesterday's noise.

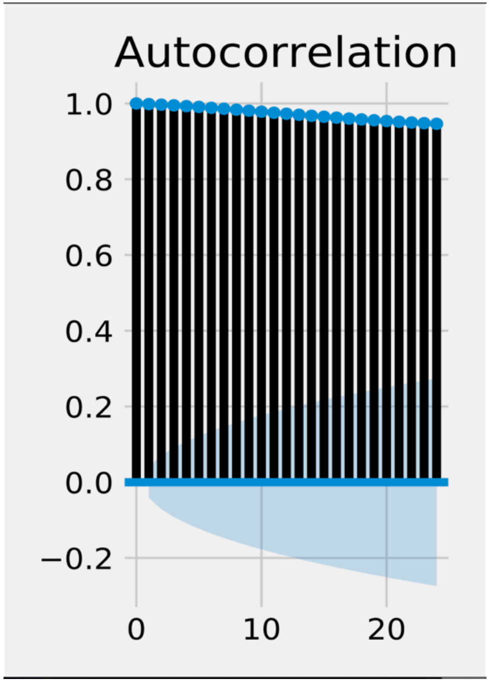
$$R_t = \mu + \epsilon_t + \theta_{\epsilon t-1}$$

Autocorrelation

Measure of the correlation between your time series and a delayed copy of itself.
E.g. an autocorrelation of order 3 returns the correlation between a time series at points (t_1, t_2, t_3…) and its own values lagged by 3 time periods, i.e. (t_4, t_5, t_6, ...).
Autocorrelation is used to find repeating patterns or periodic signals in time series data. It is also called autocovariance.
Because autocorrelation is a correlation measure, the autocorrelation coefficient can only take values between -1 and 1.
If the values are small and inside the blue shaded region, they are not statistically significant.

Partial autocorrelation

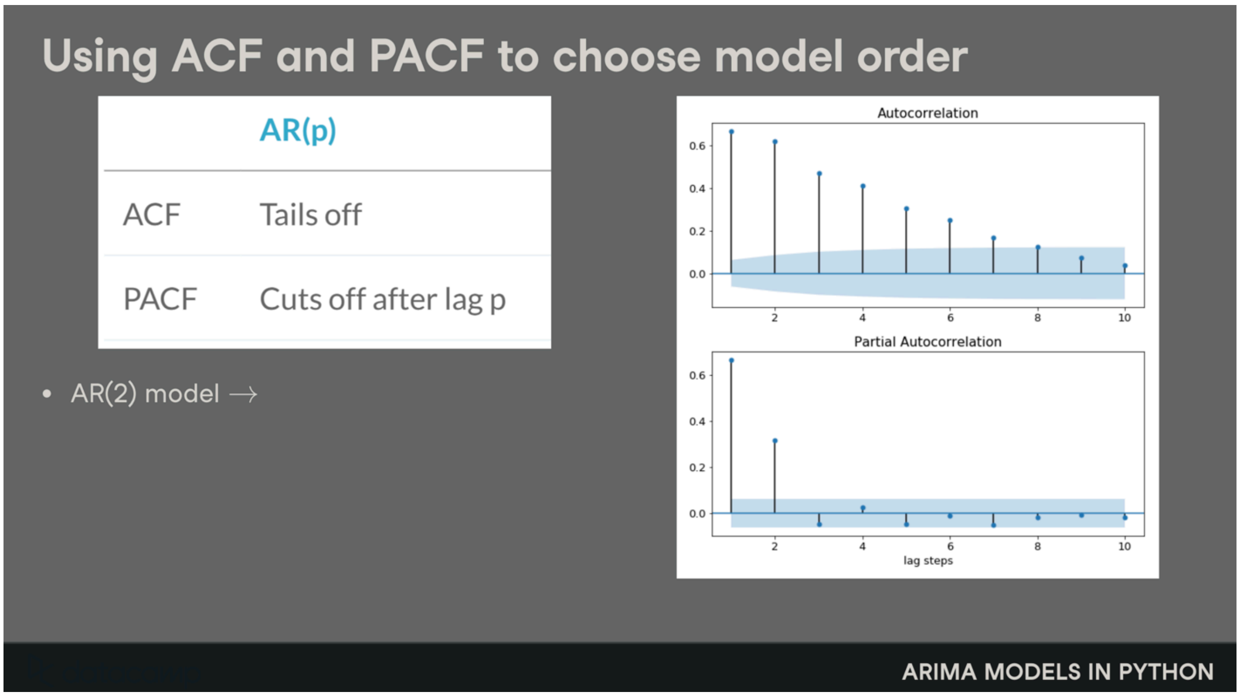
Like autocorrelation, the partial autocorrelation function (PACF) measures the correlation coefficient between a time-series and lagged versions of itself. However, it extends upon this idea by also removing the effect of previous time points. For example, a partial autocorrelation function of order 3 returns the correlation between our time series (t_1, t_2, t_3, …) and its own values lagged by 3 time points (t_4, t_5, t_6, …), but only after removing all effects attributable to lags 1 and 2, meaning that we subtract the effect of correlation at smaller lags. So, it is the correlation associated with just that particular lag.



The .plot_pacf() function also returns confidence intervals, which are represented as blue shaded regions. If partial autocorrelation values are beyond these confidence interval regions, then you can assume that the observed partial autocorrelation values are statistically significant.

By comparing the ACF and PACF for a time series we can deduce the model order for AR, MA or ARMA (AR & MA).

If the amplitude of the ACF tails off with increasing lag and the **PACF cuts off** after some lag p, then we have an AR (p) model. This plot is an AR (2) model.

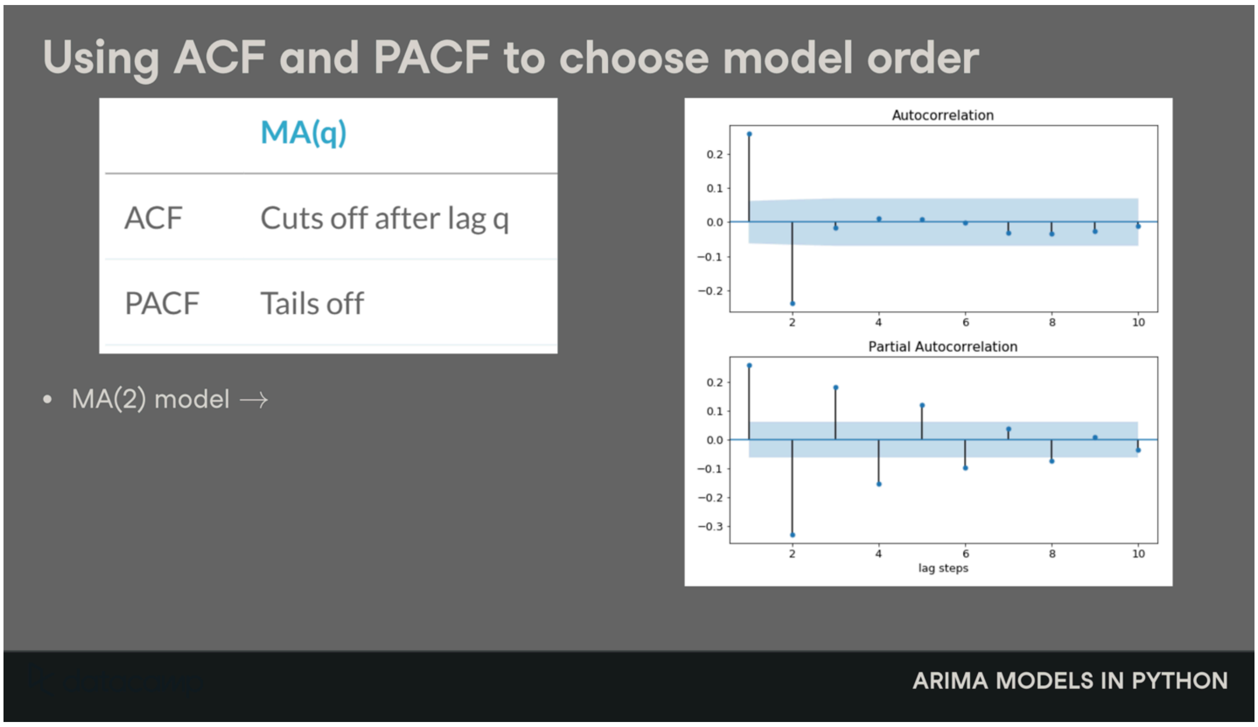


Using ACF and PACF to choose model order

	AR(p)
ACF	Tails off
PACF	Cuts off after lag p

- AR(2) model →

If the amplitude of the ACF cuts off after some lag q and the amplitude of PACF tails off then we have a MA (q) model. This is an MA (2) model.



Using ACF and PACF to choose model order

	MA(q)
ACF	Cuts off after lag q
PACF	Tails off

- MA(2) model →

If both ACF and PACF tail off then we have an ARMA model. In this case we can't deduce the model order of p and q from the plot.

Data visualization commands

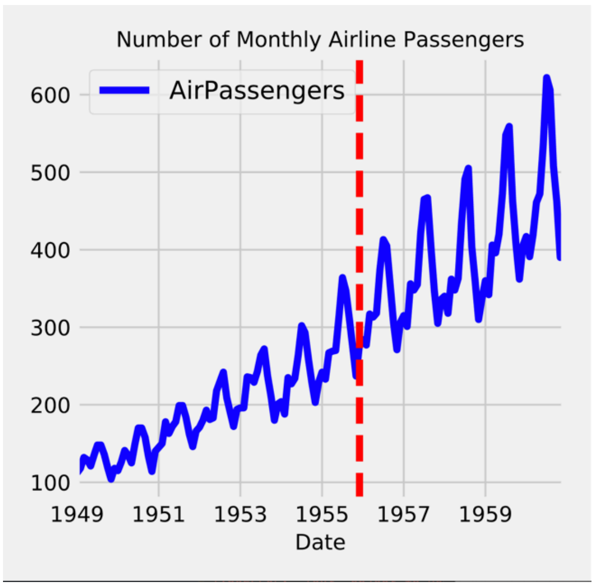
```
# Print the data type of each column in discoveries
print(discoveries.dtypes)
# Convert the date column to a timestamp type
discoveries['date'] = pd.to_datetime(discoveries['date'])
```

Add a red vertical line on the plot

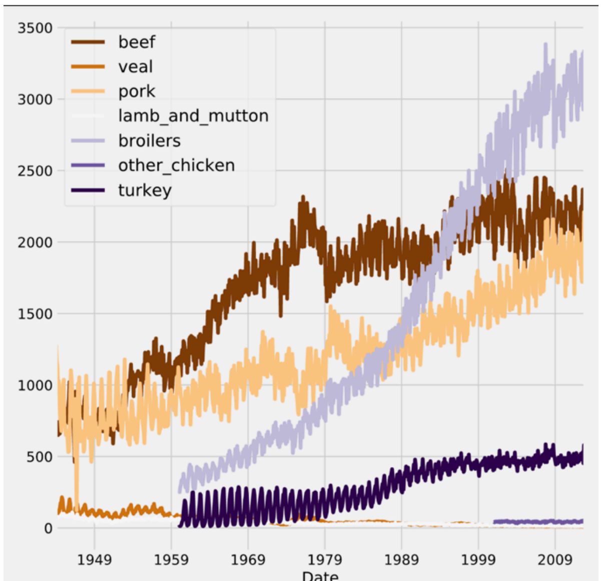
```
# Plot the time series in your dataframe
ax = airline.plot(color='blue', fontsize=12)

# Add a red vertical line at the date 1955-12-01
ax.axvline('1955-12-01', color='red', linestyle='--')

# Specify the labels in your plot
ax.set_xlabel('Date', fontsize=12)
ax.set_title('Number of Monthly Airline Passengers', fontsize=12)
plt.show()
```



```
# Plot time series dataset using the cubehelix color palette
ax = meat.plot(colormap='PuOr', fontsize=15)
# Additional customizations
ax.set_xlabel('Date')
ax.legend(fontsize=18)
# Show plot
plt.show()
```


beef
veal
pork
lamb_and_mutton
broilers
other_chicken
turkey
Date