Course 1273

Data Wrangling With Python

by

Logical Operations

Technical Editor:

Frank Schmidt

Packt

1273/CN/B.1/201/A.1

Copyright

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc. Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

Introduction and Overview

Course Objectives

- Extract and parse data from various sources
- Transform and clean data using Numpy and Pandas
- Summarize and visualize data with Matplotlib
- Read HTML, XML, and JSON data from internet resources
- Search and filter data sets
- Apply Python tools and techniques to process data sets efficiently

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Course Contents

	Introduction and Overview
Lesson 1	Introduction to Data Wrangling With Python
Lesson 2	Advanced Data Structures and File Handling
Lesson 3	Introduction to NumPy, Pandas, and Matplotlib
Lesson 4	A Deep Dive Into Data Wrangling With Python
Lesson 5	Getting Comfortable With Different Kinds of Data Sources
Lesson 6	Learning the Hidden Secrets of Data Wrangling
Lesson 7	Advanced Web Scraping and Data Gathering
Lesson 8	RDBMS And SQL
Lesson 9	Application of Data Wrangling in Real Life
Lesson 10	Course Summary
	Next Steps

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Packt

Data Wrangling With Python

Create actionable data from raw sources

Lesson 1:
Introduction to Data Wrangling With Python

Time: 1hr

Lesson Objectives

By the end of this lesson, you will be able to:

- Define the importance of data wrangling in data science
- Manipulate the data structures available in Python
- Compare the different implementations of the inbuilt Python data structures

Introduction

- Data wrangling is the process that ensures that the data is in a format that is clean, accurate, formatted, and ready to be used for data analysis.
- Real-life example: At Supercomputer Center of University of California San Diego (UCSD), there was a research on occurrence of wildfires in California.
- Data scientists at UCSD Supercomputer Center gathered data to predict the nature and spread direction of the fire. This data comes from diverse sources such as weather stations, sensors in the forest, fire stations, satellite imagery, and Twitter feeds and had incomplete or missing data.
- This data needs to be cleaned and formatted such that it can be used to predict future occurrences of wildfire. This cleaning and formatting is data wrangling

Discuss

- Can you think of another real-life situation where data wrangling knowledge would be helpful?

The Importance of Data Wrangling

The advantages of data wrangling can be summarized as:

- It enhances the credibility of data
- It simplifies the data science pipeline
- It reduces the overall cost to the organization
- It allows you to integrate data from different types of sources
- It allows the reuse of data transformation
- It allows you to easily integrate data
- It supports scalability

Data Wrangling is Refinement

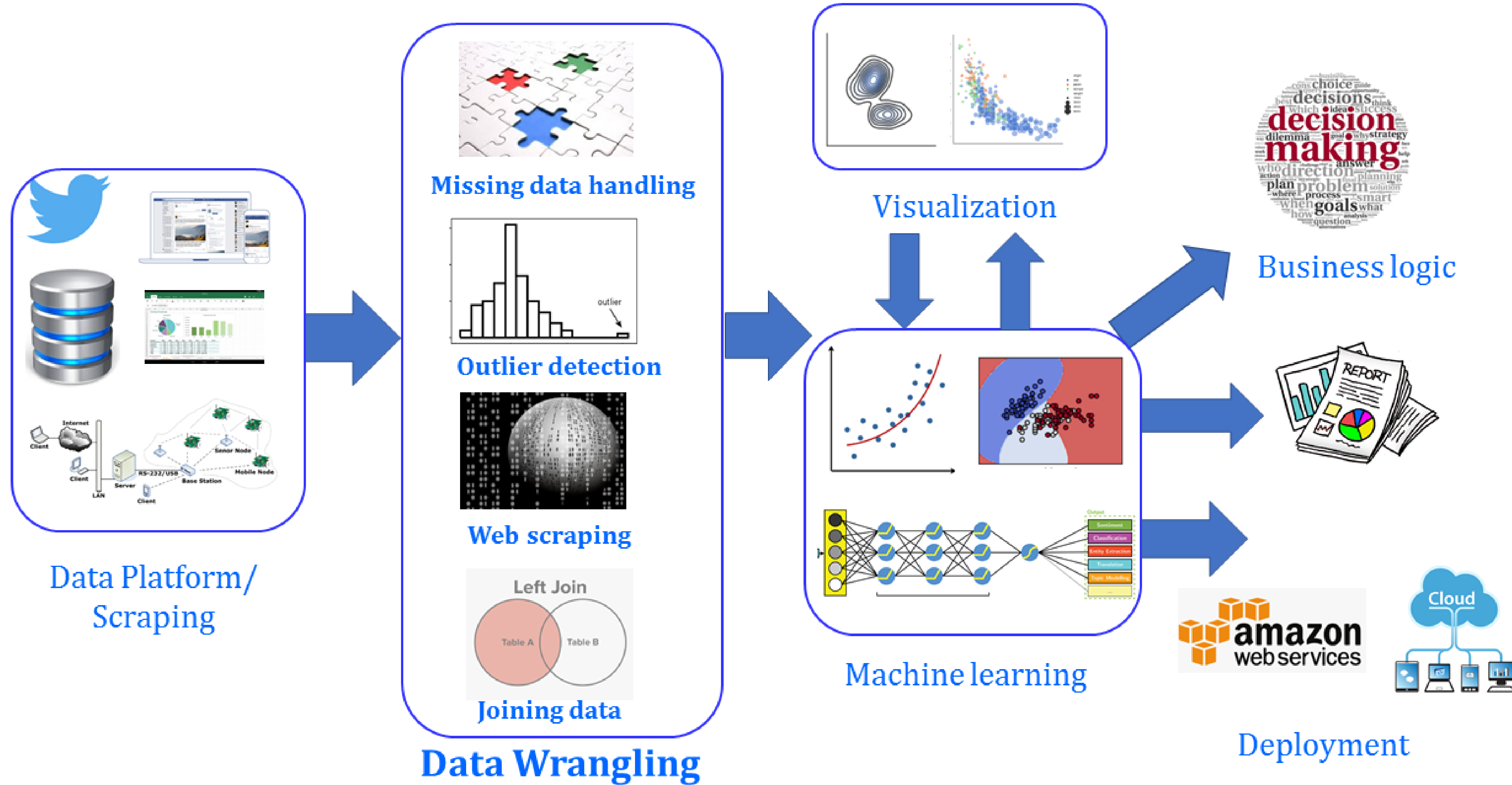
- Oil needs refinement and cannot be used in its crude form, and the same can be said for data.
- Data needs to be curated, massaged, and refined to be used in intelligent algorithms and consumer products.
- Mostly, we hear about two aspects of data science:
    - ➢ The cool visualization of a dataset, showing interesting trends and patterns
    - ➢ The amazing predictive power of a machine learning model that is applied to a dataset to mine for insights

Data Wrangling Tasks

Data wrangling tasks can be summarized as follows:

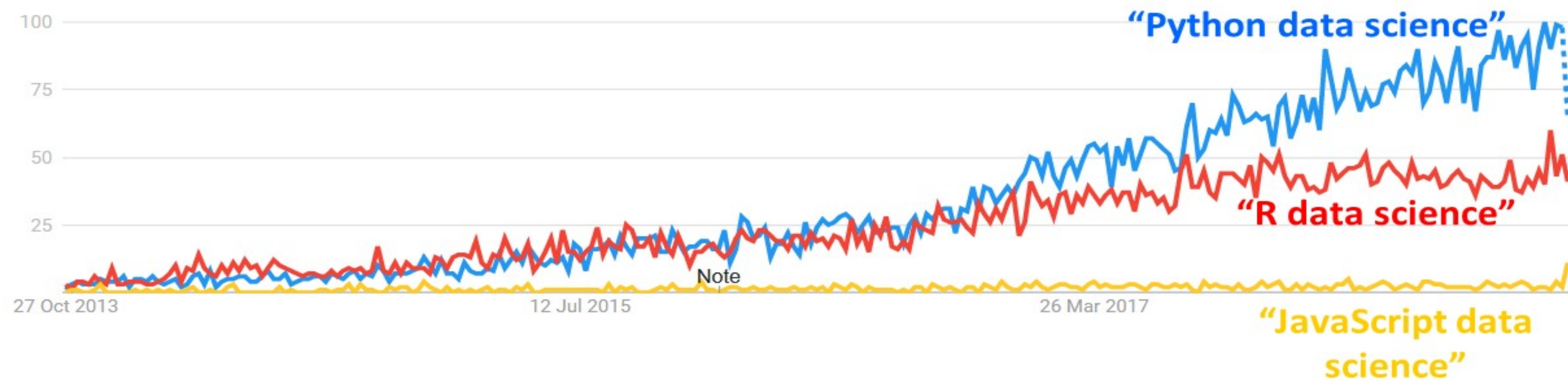
- Scraping raw data from multiple sources (including web and database tables)
- Imputing, formatting, and transforming – basically making it ready to be used in the modeling process (such as advanced machine learning)
- Handling read/write errors
- Detecting outliers
- Performing quick visualizations (plotting) and basic statistical analysis to judge the quality of your formatted data

Process of Data Wrangling



Python for Data Wrangling

- Google trend worldwide over the last 5 years:



Google Trend Worldwide over last 5 years. Clearly Python is among the favorite programming languages in the domain of Data Science.

Fundamental Python Data Structures

The fundamental Python data structures are as follows:

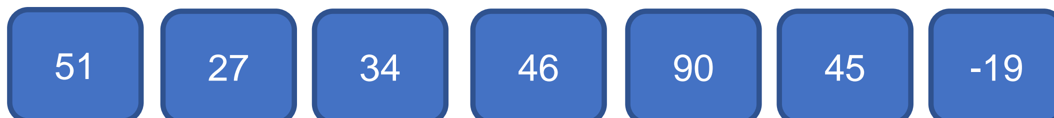
- List
- Set
- Dictionary
- Tuple
- String

List

- Have continuous memory locations
- Can host different data types
- Can be accessed by the index

Example:

- `list_example = [51, 27, 34, 46, 90, 45, -19]`



- `list_example2 = [15, "Yellow car", True, 9.456, [12, "Hello"]]`

Exercise 1: Accessing the List Members

List_1

	34	12	89	1
Indices (Forward)	0	1	2	3
Indices (Backward)	-4	-3	-2	-1

```
list_1 = [34, 12, 89, 1]
list_1[0]
=> 34
list_1[3]
=> 1
list_1[len(list_1) - 1]
=> 1
list_1[-1]
=> 1
list_1[-4]
=> 34
list_1[1:3]
=> [12, 89]
```

(It is also a list)

Exercise 2: Generating a List

- Create a list

```
list_1 = [x for x in range(0, 100)]
list_1.append(x)
```

- Generate a list of 100 elements:

```
list_1[0]
=> 0
list_1[99]
=> 99
```

- Generate a list of numbers between 0 and 100 that are divisible by 5

```
list_1 = [x for x in range(0, 100) if x % 5 == 0]
```

Discuss:

Can you check whether the simple addition and extend methods work if we have three or more lists to concatenate?

Exercise 3: Iterating over a List and Checking Membership


```
list_1 = [x for x in range(0, 100)]
```

- If we check for **25** in list_1, the response will be True
- However, if we check for **-45** in list_1, the response will be False

Exercise 4: Sorting a List

- Create a list

```
list_1 = [2, 45, 1, -9, 10]
```

- Then, sort it using the sort function:

```
list_1.sort(reverse=True)
 [45, 10, 2, 1, -9]
```

- Also, when we use the reverse function:

```
list_1.reverse()
 [45, 10, 2, 1, -9]
```

Caution: They change the list "in-place"

Exercise 5: Generating a Random List

- Use the randint function

```
import random

list_1 = [random.randint(0, 30) for x in range (0, 100)]
```

Note: You need to import the random function

Activity 1: Handling List

Aim: To create and handle lists gracefully

Scenario: Generate a list of random numbers and then generate another list from the first one that only contains numbers that are divisible by three. Repeat the experiment a few times. What is the average difference in length between the two lists?

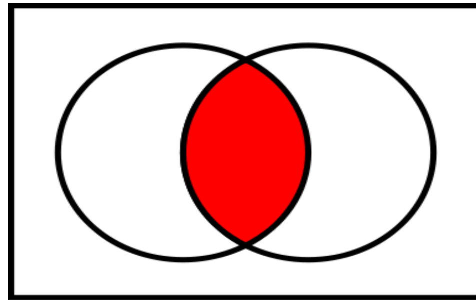
Set

- A collection of well-defined distinct objects
- Example sets:

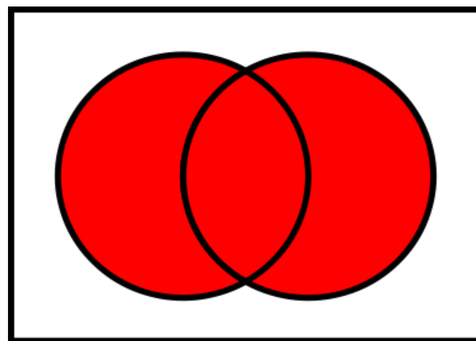
```
set1 = {"Apple", "Orange", "Banana"}
set2 = {"Pear", "Peach", "Mango", "Banana"}

set1 = set(["Apple", "Orange", "Banana"])
set2 = set(["Pear", "Peach", "Mango", "Banana"])
```

Union and Intersection of Sets


```
set1 = {"Apple", "Orange", "Banana"}
set2 = {"Pear", "Peach", "Mango", "Banana"}
```



Intersection - `set1 & set2-> {'Banana’}`



Union - `set1 | set2 -> {'Apple', 'Banana', 'Mango', 'Orange', 'Peach', 'Pear'}`

Creating Null Sets

- You can create a null set by creating a set containing no elements. You can do this
- by using the following code:

```
my_null_set = set({})
```

Discuss:

Can you use the built-in function "type" to check the data type in both of the preceding cases?

Dictionary

- Like a list
- Collection of several elements
- key: value pair

Example:

- `dict_1 = {"key1": "value1", "key2": "value2"}`
- `dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3", "key4": {"subkey1": "v1"}, "key5": 4.5}`

Exercise 6: Accessing and Setting Values in a Dictionary

- Access a particular key in a dictionary:

```
dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
"key4": {"subkey1": "v1"}, "key5": 4.5}
```

```
dict_1["key2"]
```

```
=> ["list_element1", 34]
```

- Assign a new value to the key:

```
dict_1["key2"] = "My new Value"
```

Exercise 7: Iterating over a Dictionary

- Create a dictionary

```
dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
"key4": {"subkey1": "v1"}, "key5": 4.5}
```

- Then:

```
for k, v in dict_1.items():
    print("{} - {}".format(k, v))
```

Discuss:

Try to iterate over a dictionary in the same way as we did on a list and see what happens.

Exercise 8: Revisiting the Unique Valued List Problem

- Generate a random list with duplicate values:

fromkeys and keys method

```
key_list = ['a', 'e', 'i', 'o', 'u' ]
```

- Create a unique valued list from **list_1**:

fromkeys -

```
vowels = dict.fromkeys(key_list)
=> {'a': None, 'e': None, 'i': None, 'o': None, 'u': None}
```

keys -

```
vowels.keys()
=> dict_keys(['a', 'e', 'i', 'o', 'u'])
```

Exercise 9: Delete Value from a Dict

- Create list_1 with five elements:

```
dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
"key4": {"subkey1": "v1"}, "key5": 4.5}
```

- We will use the **del** function and specify the element:

```
del dict_1["key2"]
```

dict_1 won’t have "key2" anymore.

Exercise 10: Dict Comprehension

Dict is very similar to list comprehension

- Using a list:

```
list_1 = [x for x in range(0, 10)]
dict_1 = {x : x**2 for x in list_1}
```

- Not using a list:

```
dict_1 = {x: x**2 for x in range(0, 10)}
```

Discuss:

What do you think will happen if you try to access an index outside the length of a dict?

Tuples

A sequence data type similar to list

Example:

```
my_tuple = ()
```

And this is how we create a tuple with only one value:

```
my_tuple = "Hello",
```

We can nest tuples, similar to list and dict:

```
my_tuple = "hello", "there"
my_tuple2 = my_tuple, 45, "guido"
```

Discuss

- `tuple_1 = "hello", "there"`
- `tuple_12 = tuple_1, 45, "Sam"`
- *What do you think the output will be if we print tuple_12?*

Exercise 11: Handling Tuples

Create a tuple to demonstrate how tuples are not immutable. Unpack it to read all of the elements.

Try to override a variable from the tuple.

Try to assign a series to the tuple.

Strings

This is a string:

```
my_string = 'Hello World!'
```

This is also a string:

```
my_string = "Hello World 2!"
```

Exercise 12: Accessing Strings

- Define the string

```
my_str = "Hello World!"
```

- All of these are valid code once you have a string defined

```
my_str[0]

my_str[4]

my_str[len(my_str) - 1]

my_str[-1]
```

Exercise 13: String Slices


```
my_str = "Hello World! I am learning data wrangling"
```

- Specify the slicing values and slice the string:

```
my_str[2:10]
=> 'llo Worl'
```

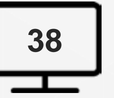
- Slice a string by skipping a slice value:

```
 my_str[-31]
=> 'd'
```

- Use negative numbers to slice the string:

```
my_str[-10:-5]
=> ' wran'
```

String Functions: len, lower, upper, find, replace



len, lower, upper, and find

```
my_str = "Hello World! I am learning data wrangling"

len(my_str)

=> 41

my_str = "A COMPLETE UPPER CASE STRING"

my_str.lower()

=> 'a complete upper case string'

 my_str.upper()

=> 'A COMPLETE UPPER CASE STRING'

my_str = "A complicated string looks like this"

my_str.find("complicated")

=> 2

my_str.find("hello")

=> -1
```

Exercise 14: Split and Join

- Create a string and add it to a list using the **split** method:

```
my_str = "Name, Age, Sex, Address"


list_1 = my_str.split(",")
```

- The preceding code will give you a list similar to the following:

```
["Name", "Age", "Sex", "Address"]
```

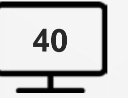
- We can combine this list with another string using the join method:

```
" | ".join(list_1)
```

- The preceding code will give you a string like the following:

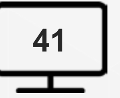
```
"Name | Age | Sex | Address"
```

Activity 2: Analyze a Multiline String and Generate the Unique Word Count



1. Get a multiline text and save it in a Python variable
2. Get rid of all new lines in it using string methods
3. Get all the unique words and their occurrences from the string
4. Repeat the step to find all unique words and occurrences, without considering case sensitivity

Discuss



- Can you think of another way to remove new lines and symbols? Rate the two methods on their effectiveness.

Summary

- In this lesson, we have:
  - Learned what the term *data wrangling* means, along with examples from real-life situations.
  - Learned about different built-in data structures in Python
  - Explored Lists, Sets, Dictionaries, Tuples, and Strings.
  - Practiced short hands-on exercises and activities on those built-in functions.

Practice Questions

Questions

1. What is the difference between an array from a statically typed language (such as C) and a Python list?

a) A C array has fixed objects and a Python list can have any objects
b) A C array is not resizable; Python lists are resizable
c) A C array has contiguous memory allocation
d) Python lists have contiguous allocations of references in memory
e) All of above

2. T/F Is a set immutable? – **FALSE**

3. Apart from the uniqueness of elements, list other differences there are between a set and a list?

4. Can you delete an element from a set? – **Yes**

5. T/F Handling string is easier in Python than in Java.

Lesson 2:
Advanced Data Structures and File Handling

Time: 2hrs

Lesson Objectives

By the end of this lesson, you will be able to:

- Compare Python's advanced data structures
- Utilize data structures to solve real-world problems
- Make use of OS file-handling operations

Introduction

In this lesson, we will cover the following:

- Advanced data structures : Iterator, Stacks, Queue
- Operations and manipulations on Files
- Use fundamental data structures to represent more complex, higher-level data structures
- Open a file, read data from it, write data to it, and safely close a file in Python.

Discuss:

Data wrangling consists of different types of data. What operations do you think an advanced data type should have?

Advanced Data Structures

30 mins

Iterator

An `iterator` in Python is:

- An object that implements the `next` method
- It remembers state of an object
- Once an item has been consumed, cannot go back to it
- Once all the elements in its scope have been exhausted, it raises a `StopIteration` exception
- A lot of useful iterators are already defined for us in the `itertools` module of Python

Exercise 15: Introduction to the Iterator

- Use an iterator to reduce memory utilization:

```
from itertools import repeat
small_list_of_numbers = repeat(1, times=10000000)
getsizeof(small_list_of_numbers)
```

- Loop over the newly generated iterator:

```
for i, x in enumerate(small_list_of_numbers):
    print(x)
    if i > 10:
        break
```

Discuss:

Why do you think the file object is implemented as an iterator in Python?

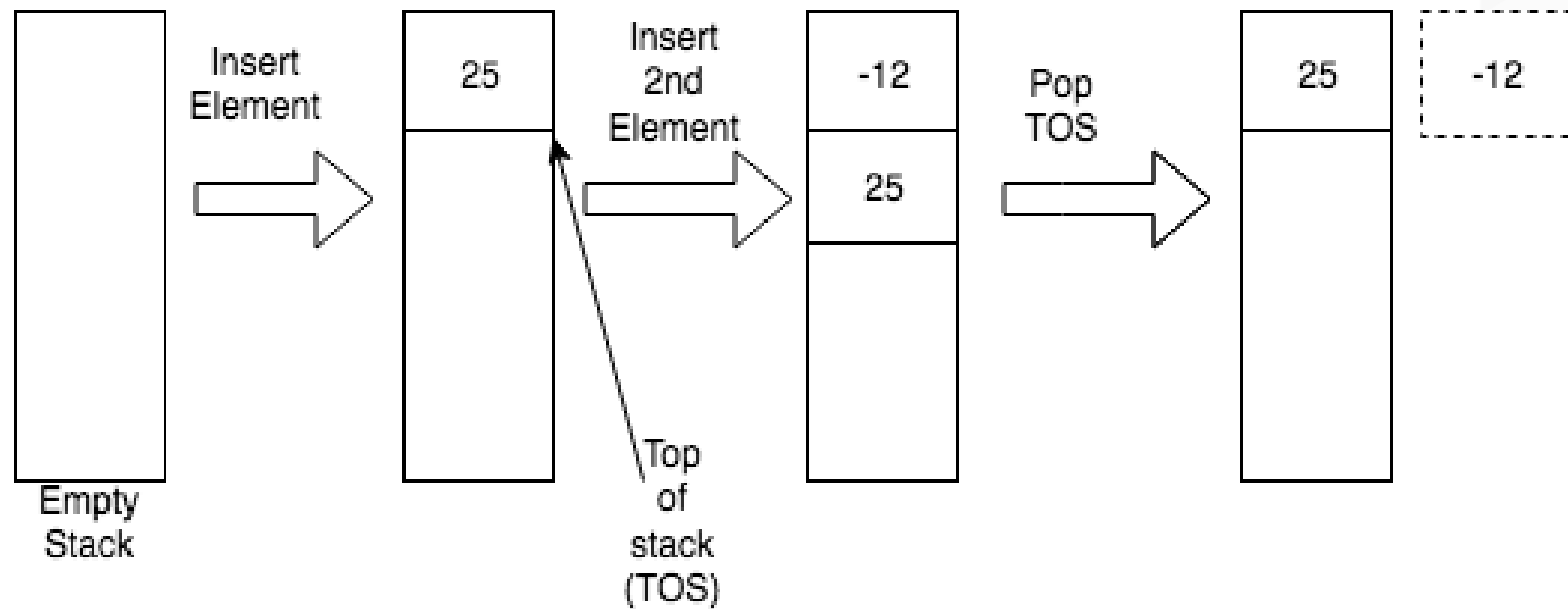
Hint: Think about how to read really large files and the benefit an iterator can offer.

Imagine scenarios where you will use permutation, combination, and dropwhile.

Stack

- A stack is a list-like data structure with a restriction on how you insert and read data

- The rule is **L**ast **I**n **F**irst **O**ut (LIFO)



Exercise 16: Implementing a Stack in Python

- Initializing an empty stack

```
stack = []
```

- Adding the first element into the stack

```
stack.append(25)
```

- Adding the second element into the stack

```
stack.append(-12)
```

- Checking elements in the stack

```
print(stack)
```

- Popping the last element from the stack

```
tos = stack.pop()
```

Practical uses for Stacks:

The following are few of the practical uses of stack:

- To scrape webpages: You can stack up all the links that you encounter in a web page and then pop them one by one to scrape them.

- To execute programs: Every single program execution on the computer uses a stack inside the processing space. It is an essential data structure for every piece of hardware.

Can you think of any other use of stacks.

Exercise 17: Implementing a Stack Using User-Defined Methods

12

- Define functions `stack_push` and `stack_pop`.

```
def stack_push(s, value):
    return s + [value]

def stack_pop(s):
   tos = s[-1]
   del s[-1]
   return tos
```

- Create a stack

```
stack = [3, 89, 12]
```

- Pop a element from the stack

```
stack_pop(stack)
```

- After the last line, the original stack will become `[3, 89]`

Lambda Expression

The features of lambda expression are:

- One-liner nameless functions
- By convention, side effect free
- Do not need a return statement
- `lambda x: x`

=> will return the value passed to it, unchanged

- `lambda x : int(x)`

=> will try to cast the incoming value to int and return that

Exercise 18: Lambda Expression

- Import the `math` package:

```
import math
```

- Define two functions, `my_sine` and `my_cosine`.

```
def my_sine():
    return lambda x: math.sin(math.radians(x))
def my_cosine():
    return lambda x: math.cos(math.radians(x))
```

- Define `sine` and `cosine` for our purpose:

```
sine = my_sine()
cosine = my_cosine()
math.pow(sine(30), 2) + math.pow(cosine(30), 2)
```

Exercise 19: Lambda Expression for Sorting

- Suppose you have these tuples:

```
capitals = [("USA", "Washington"), ("India", "Delhi"), ("France", "Paris"),
("UK", "London")]
```

- Sort this list by the name of the capitals, using a simple lambda expression

```
capitals.sort(key=lambda item: item[1])
```

Discuss

- Can we write a multi-instruction lambda expression? If so, how?
- When should we make a real function instead of a lambda function?

Exercise 20: Multi-Element Membership checking

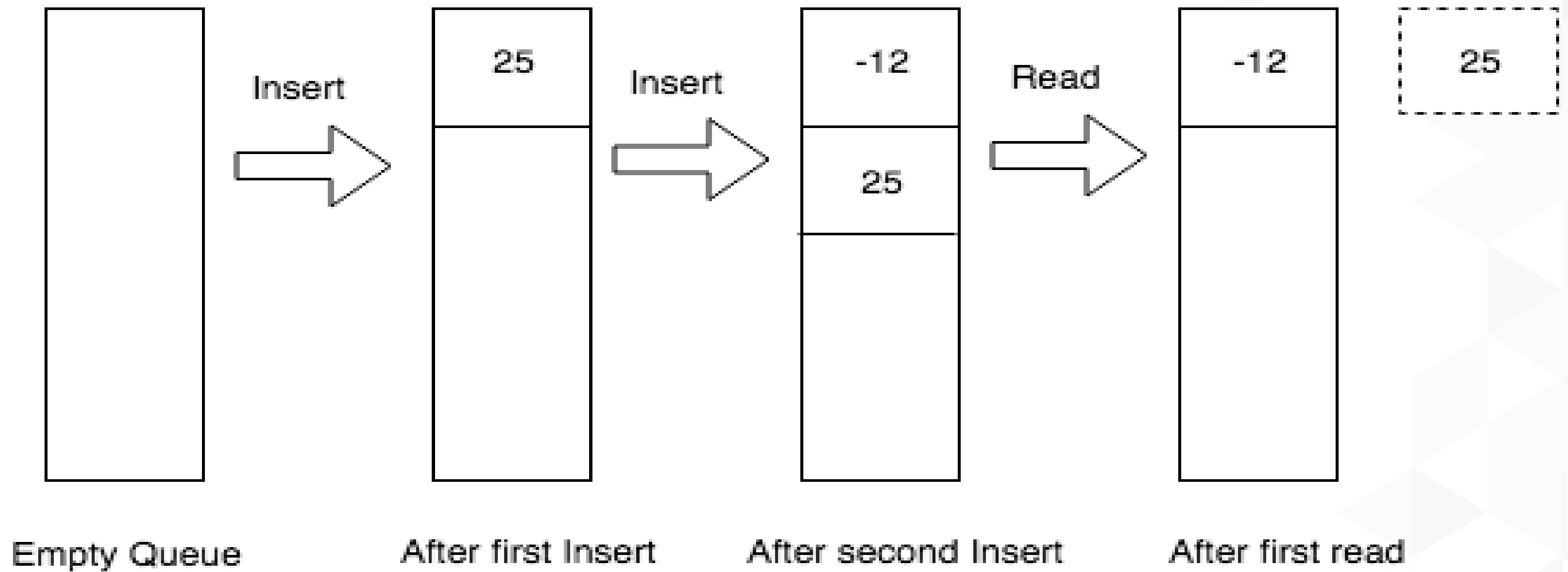
- Create a list of words:
  ```
  list_of_words = ["Hello", "there.", "How", "are", "you", "doing?"]
  ```
- Check for certain words:
  ```
  check_for = ["How", "are"]

  all(w in list_of_words for w in check_for)
  ```
- The last statement will return True or False based on whether all the elements from `check_for` are present in `list_of_words` or not.

Discuss

- Can you write code to do the same thing, but this time respecting the order, meaning ["How", "are"] is a match, whereas ["are", "How"] is not?

Queue

List-like data structure with the restriction of FIFO – **F**irst **I**n **F**irst **O**ut:



Exercise 21: Implementing a Queue in Python

- Create a Python queue with the vanilla list methods:

```
queue = []
for i in range(0, 100000):
   queue.append(i)
print("Queue created")
```

- Implement the same using dequeue

```
from collections import deque
queue2 = deque()
for i in range(0, 100000):
   queue2.append(i)
print("Queue created")
```

Discuss

Unlike stack, we do not use vanilla lists from Python.

Activity 3: Permutation, Iterator, Lambda, List

1. Look up the definition of `permutations` and `dropwhile` in the itertools documentation.
2. Using permutations, generate every possible three-digit number using 0, 1, and 2.
3. Loop over this iterator, print them, and also use `isinstance` and `assert` to make sure that the return types are tuples.
4. Use a single line of code to convert all of the tuples to lists
5. Write a function that takes a list as above and returns the actual number contained in it.

Exercise 23: Operating System File Operations

- An OS has a lot of environment variables. They are necessary for an OS to function.
- 12-factor app design (and similar other best practices) indicates that we should design apps that store and read configurations from environment variables.
- Import OS and OS environment

```
import os
os.environ['MY_KEY'] = "MY_VAL"
os.getenv('MY_KEY')
print(os.getenv('MY_KEY_NOT_SET'))
```

Discuss

- What will be printed with print(os.environ)?
- How will you make sure that an environment variable exists?
- Can you design a small function that takes the name of an environment variable and returns True if it exists and has a value, and False otherwise?

File Handling

- Use the built-in open function.
- It returns a "File Handler" which we use to operate on files.
- There are several file opening modes. They dictate what operations are allowed.

Character	Meaning of the character
‘r’	Open for reading
‘w’	Open for writing
‘x’	Create a new file and open it for writing
‘a’	Open for writing in append mode, if it exists
‘b’	Binary mode
‘t’	Text mode (default)
‘+’	Update mode (both write and read)

Discuss

- What will happen if we try to open a file that does not exist in all the different modes mentioned?

Exercise 23: Opening and Closing a File

- Open a file in `rt` mode

```
fd = open("Alice’s Adventures in Wonderland, by Lewis Carroll")
```

- Open the file in binary mode

```
fd = open("Alice’s Adventures in Wonderland, by Lewis Carroll", "rb")
```

- **Always** close open files. Otherwise, very strange, hard-to-track bugs may occur
- To close a file, just call
- `fd.close()`

Discuss

- What are the different exceptions that an open call can raise?
- What will happen if we call `fd.close()` a second time after it has already been called once?

The with Statement

The with statement is:

- A compound statement
- Wraps a block of code in the scope of a Context Manager
- Cleanest way to handle files

Example:

```
with open("Alice’s Adventures in Wonderland, by Lewis Carroll") as fd:
    print(fd.closed)

print(fd.closed)
```

Discuss

- Encourage them to write the code using two dummy small files and check what happens in each case
- Do you think "with" statements can be nested? If so, what happens when the control goes out of the inner with block but not the outer with block? You should try to experiment with it by opening two files by nested "with" statement and checking what happens.

Exercise 24: Reading Files

- Open a file and then read the file line by and print it as we read it:

```
with open("Alice’s Adventures in Wonderland, by Lewis Carroll") as fd:
    for line in fd:
        print(line)
```

- Duplicate the same for loop, just after the first one:

```
with open("Alice's Adventures in Wonderland, by Lewis Carroll",
          encoding="utf8") as fd:
    for line in fd:
        print(line)

print("Ended first loop")
for line in fd:
    print(line)
```

Discuss

Can you change the preceding code to use that method instead?

What are the main differences between these two ways?

Exercise 25: Writing to File (Method 1)

- Open a file and write into it

```
data_dict = {"India": "Delhi", "France": "Paris", "UK": "London", "USA": "Washington"}
with open("data_temporary_files.txt", "w") as fd:
   for country, capital in data_dict.items():
          fd.write("The capital of {} is {}\n".format(country, capital))
```

Exercise 25: Writing to Files (Method 2)

- Create a data_dict

```
data_dict = {"India": "Delhi", "France": "Paris", "UK": "London",
"USA": "Washington"}
```

- Open a temporary file and write in it.

```
with open("data_temporary_files.txt", "w") as fd:
   for country, capital in data_dict.items():
          print("The capital of {} is {}".format(country, capital), file=fd)
```

- This method requires Python 3 (or Python 2, but then you have to import the print_method from the __future__ module)

Activity 4: Design Your Own CSV Parser

In this activity, you will be given a CSV file. A CSV file is a simple text file that has a heading row, where we will have comma-separated (",") strings to tell what this column represents, such as Age, Location, Name, and so on. Then, we will have rows where each row represents one data point.

In this activity, we will design a small function/class (as you prefer) to perform some basic operations on CSV files.

1. Import **zip_longest** from **itertools**
2. Open the **sales_record.csv** file from the GitHub link and read the file
3. Read each line and pass that line to a function along with the list of the headers.

Summary

- In this lesson, we learned about the workings of advanced data structures such as stacks. We implemented and manipulated a stack. We then focused on different functional programming methods, including iterators. We also combined lists and functions. Then, we looked at queues and implemented and compared different queue operation methods.
- We then looked at OS-level functions and the management of environment variables and files. We also examined a clean way to deal with files and we created our own CSV parser in the final activity.
- In the next chapter, we will be dealing with the three most important libraries: NumPy, Pandas, and Matplotlib.

Practice Questions

1. What is the purpose of the "yield" statement?
2. Can you use "break" to break from a "with" statement?
3. What are some other advanced data structures apart from stack and queue?
4. Can you name some applications of a few of the advanced data structures?
5. Why is it important to write side effect-free functions?
6. Can you name some of the common data file formats (for example, CSV)?

Packt

Lesson 3:
Introduction to NumPy, Pandas, and Matplotlib

Time: 2hrs

Lesson Objectives

By the end of this lesson, you will be able to:

- Create and manipulate one-dimensional and multi-dimensional arrays
- Create and manipulate pandas DataFrames and series objects
- Create DataFrames from common file types and plot them
- Use matplotlib, NumPy, and pandas to calculate descriptive statistics from a DataFrame/matrix

Introduction

In this lesson, we will cover the following:

- NumPy arrays
- Stack
- Queue
- Iterator
- File operations in Python

Packt

NumPy Array Fundamentals

20 mins

The Importance of Numeric Arrays

- Arrays can contain integers, floating-point numbers, Booleans, strings, or even mixed types.
- Reading and manipulating arrays take 80% of the time of a data analysts job.
- Example of numeric arrays in real-life:
  - Phone numbers
  - Postal codes
  - Statistical simulations
  - Financial and transactional data
  - Stock price prediction

The Importance of NumPy Arrays

- NumPy is the base package on top of which most of the other libraries are built.
- NumPy offers the *ndarray* data structure.
- *Nddata* is essential for performing operations such as sorting, searching, mathematical manipulation, statistical modeling, and linear algebra.
- Pandas uses NumPy arrays as its base to build two-dimensional tables .
- A NumPy array can act as an in-memory database and model a large type of data sources.

NumPy Array Features

- Provides common mathematical and numerical routines in pre-compiled, fast functions.
- Contains single- or multi-dimensional arrays.
- Is at the heart of the NumPy package.
- Serves as the fundamental building block of advanced classes such as the pandas DataFrame.
- Built for vectorized operations. They can process a lot of numeric data with a single line of code.
- Contains many built-in mathematical functions written in low-level languages for fast execution.

Exercise 26: Creating a NumPy Array (from a List)

- Import the library, giving it a short name, np:

```
import numpy as np
```

- Create a NumPy array from a normal list:

```
lst1 = [1, 2, 3]
array1 = np.array(lst1)
```

- The type of a NumPy array is different from a normal Python list. You can check that with this:

```
type(array1)
>> numpy.ndarray
type (lst1)
>> list
```

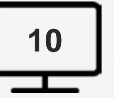
Exercise 27: Adding Two NumPy Arrays

- Notice the key difference between regular Python list/array and a NumPy array:

```
lst2 = lst1 + lst1

print(lst2)

>>[1, 2, 3, 1, 2, 3]

array2 = array1 + array1

print(array2)

>> [2, 4, 6]
```

- NumPy arrays are like **vectors -** built for element-wise operation.

Exercise 28: Mathematical Operations on NumPy Arrays



The mathematical operations that can be performed on arrays are:

- Multiplication:

```
print("array1 multiplied by array1: ", array1*array1)
```

- Division:

```
print("array1 divided by array1: ", array1/array1)
```

- Raising an array to the power of another array:

```
print("array1 raised to the power of array1: ", array1**array1)
```

Exercise 29: Advanced Mathematical Operations on NumPy Arrays

Advanced mathematical operations on arrays are:

```
lst_5=[i for i in range(1,6)]
array_5=np.array(lst_5)
```

- Sine operation:

```
print("Sine: ", np.sin(array_5))
```

- Logarithm operation:

```
print("Natural logarithm: ", np.log(array_5))
print("Base-10 logarithm: ", np.log10(array_5))
print("Base-2 logarithm: ", np.log2(array_5))
```

- Exponential operation:

```
print("Exponential: ", np.exp(array_5))
```

Exercise 30: Generating Arrays Using arange, and linspace

- The `arange` function creates a series of numbers based on the minimum and maximum bounds you give and the step size you specify.
- The `linspace` function creates a series of fixed number of intermediate points between two extremes.

```
print("A series of numbers:", np.arange(5,16))
print("Numbers spaced apart by 2: ", np.arange(0,11,2))
print("Numbers spaced apart by a floating point number: ",
np.arange(0,11,2.5))
print("Every 5th number from 30 in reverse order\n", np.arange(30,-1,-5))
print("11 linearly spaced numbers between 1 and 5: ", np.linspace(1,5,11))
```

Exercise 31: Creating Multi-Dimensional arrays

- Let’s create multi-dimensional arrays (such as a matrix in linear algebra).
- Just like we created the one-dimensional array from a simple flat list, we can create a two-dimensional array from a **list of lists**:

```
list_2D = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
mat1 = np.array(list_2D)
print("Type/Class of this object:", type(mat1))
print("Here is the matrix\n----------\n", mat1, "\n----------")
```

- Even **tuples** can be converted to multi-dimensional arrays:

```
tuple_2D = np.array([(1.5, 2, 3), (4, 5, 6)])
mat_tuple = np.array(tuple_2D)
print (mat_tuple)
```

Exercise 32: Dimension, Shape, Size, and Data Type of the 2D Array

- The following methods let you check the dimension, shape, and size of the array. Note that if it's a 3x2 matrix, that is, three rows and two columns, then the shape will be (3,2), but the size will be 6, as 6 = 3x2.
- Print the dimension of the matrix using `ndim`:

```
print("Dimension of this matrix: ", mat1.ndim,sep=‘’)
```

- Print the size of the matrix using `size`:

```
print("Size of this matrix: ", mat1.size,sep=‘’)
```

- Print the shape of the matrix using `shape`:

```
print("Shape of this matrix: ", mat1.shape,sep=‘’)
```

- Print the data type of the matrix using `dtype`:

```
print("Data type of this matrix: ", mat1.dtype,sep='')
```

Discuss

- How are NumPy arrays different from regular list objects?

Exercise 33: Zeros, Ones, Random, Identity Matrices, and Vectors

- Often, you may have to create matrices filled with zeroes, ones, random numbers, or ones in the diagonal:

```
print("Vector of zeros: ", np.zeros(5))
print("Matrix of zeros: ", np.zeros((3,4)))
print("Vector of ones: ", np.ones(4))
print("Matrix of ones: ", np.ones((4,2)))
print("Matrix of 5’s: ", 5*np.ones((3,3)))
print("Identity matrix of dimension 2:", np.eye(2))
print("Identity matrix of dimension 4:", np.eye(4))
print("Random matrix of shape (4,3):\n",np.random.randint(low=1, high=10,
size=(4,3)))
```

Exercise 34: Reshaping, Ravel, Min, Max, and Sorting

- **Reshaping** an array is a very useful operation. So is finding the minimum and maximum elements, and sorting.
- The following functions accomplish those tasks. Execute the following code:

```
a = np.random.randint(1, 100, 30)
b = a.reshape(2, 3, 5)
c = a.reshape(6, 5)
print ("Shape of a:", a.shape)
print ("Shape of b:", b.shape)
print ("Shape of c:", c.shape)
print("\na looks like\n", a)
print("\nb looks like\n", b)
print("\nc looks like\n", c)
```

- The opposite to reshape is the ravel function, which flattens any given array into a one-dimensional array:

```
b_flat = b.ravel()
print(b_flat)
```

Exercise 35: Indexing and Slicing

- Indexing and slicing of NumPy arrays is very similar to regular list indexing.
- Additionally, you can pass a list as the argument to select specific elements:

```
arr = np.arange(0,11)
print("Array:",arr)
print("Element at 7th index is:", arr[7])
print("Elements from 3rd to 5th index are:", arr[3:6])
print("Elements up to 4th index are:", arr[:4])
print("Elements from last backwards are:", arr[-1::-1])
print("3 Elements from last backwards are:", arr[-1:-6:-2])
```

- More code examples for multi-dimensional array indexing and slicing are given in the Instructor’s and Student’s guides.

Conditional Subsetting

- Conditional subsetting is a great way to select specific elements based on some numeric condition.
- It is almost like a short version of a SQL query to subset elements:

```
mat = np.array(ri(10, 100, 15)).reshape(3, 5)
print("Matrix of random 2-digit numbers\n", mat)
print ("\nElements greater than 50\n", mat[mat>50])
```

- We will explore conditional subsetting in our activity and in much more depth in the context of pandas DataFrames in the coming lessons.

Array Operations (array-array, array-scalar, universal functions)

- NumPy arrays operate just like mathematical matrices, and the operations are performed element-wise:

```
mat1 = np.random.randint(1, 10, 9).reshape(3, 3)
mat2 = np.random.randint(1, 10, 9).reshape(3, 3)
print("\n1st Matrix of random single-digit numbers\n", mat1)
print("\n2nd Matrix of random single-digit numbers\n", mat2)
print("\nAddition\n", mat1 + mat2)
print("\nMultiplication\n", mat1 * mat2)
print("\nDivision\n", mat1 / mat2)
print("\nLineaer combination: 3*A - 2*B\n", 3 * mat1 – 2 *mat2)
print("\nAddition of a scalar (100)\n", 100 + mat1)
print("\nLinear Exponentiation, matrix cubed here\n", mat1**3)
print("\nExponentiation, square root using 'pow' function\n", pow(mat1, 0.5))
```

Stacking Arrays

- Stacking arrays on top of each other (or side by side) is a useful operation for data wrangling:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print("Matrix a\n", a)
print("Matrix b\n", b)
print("Vertical stacking\n", np.vstack((a, b)))
print("Horizontal stacking\n", np.hstack((a, b)))
```

Other advanced features

- NumPy has many other advanced features, mainly related to statistics and linear algebra functions, which are used extensively in machine learning and data science tasks.
- However, not all of that is directly useful for beginner level data wrangling, so we won't cover it here.

Discuss

- Compare NumPy arrays to arrays in other languages
- Compare NumPy arrays with Python Lists

Pandas DataFrames

30 mins

What Is a DataFrame and How Does It Work?

- Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with relational or labeled data both easy and intuitive.
- It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python.
- The two primary data structures of pandas, series (one-dimensional) and DataFrame (two-dimensional), handle the vast majority of typical use cases.
- Pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other third-party libraries.

Exercise 37: Creating a Pandas Series

- If you have imported pandas as ***pd***, then the function to create series is simply **pd.Series**.
- First, create panda series from the list:

```
pd.Series(data=my_data)
```

- Then, create from the list along with the labels:

```
pd.Series(data=my_data, index = labels)
```

- Then, create a series from the NumPy array:

```
pd.Series(arr,labels)
```

- Lastly, create a series from the dictionary:

```
pd.Series(d)
```

Exercise 38: Pandas Series and Data Handling

- The pandas series object can hold many types of data.
- This is the key to constructing a bigger table where multiple series objects are stacked together to create a database-like entity:

```
print ("\nHolding numerical data\n", '-'*25, sep='')
print(pd.Series(arr))
print ("\nHolding text labels\n", '-'*20, sep='')
print(pd.Series(labels))
print ("\nHolding functions\n", '-'*20, sep='')
print(pd.Series(data=[sum, print, len]))
print ("\nHolding objects from a dictionary\n", '-'*40, sep='')
print(pd.Series(data=[d.keys, d.items, d.values]))
```

Exercise 39: Creating Pandas DataFrames

- Pandas DataFrame can be thought of as similar to an Excel table or relational database (SQL) table. It consists of three main components: the data, the index (or rows), and the columns.
- Under the hood, it is a stack of pandas series objects that are themselves built on top of NumPy arrays.
- The following code draws 20 random integers from the uniform distribution. Then reshape it into a (5,4) NumPy array – five rows and four columns. Then the code defines row names ('A','B','C','D','E') and column labels ('W','X','Y'):

```
matrix_data = np.random.randint(5, 4)
row_labels = ['A', 'B', 'C', 'D', 'E']
column_headings = ['W', 'X', 'Y', 'Z']
df = pd.DataFrame(data=matrix_data, index=row_labels, columns=column_headings)
print(df)
```

Exercise 40: Viewing a DataFrame Partially

- The most common way you will encounter to create a Pandas DataFrame will be to read tabular data from a file on local disk or over the internet – CSV, text, JSON, HTML, Excel, and so on.
- Pandas supports a myriad of ways to read files with simple function calls.
- We will see one example in the final Activity of this Lesson.

Viewing a DataFrame Partially

- In the preceding section, we typed code such as `print(df)` to print the whole DataFrame.
- But often, that is not required. For a large dataset, you don't want to print out or display the whole dataset at once.
- So how do we check only the first few rows of a DataFrame?
- To view only the first five rows of the DataFrame:

```
df.head()
```

- To view only the first ten rows of the DataFrame:

```
df.head(10)
```

- To view only the last eight rows of the DataFrame:

```
df.tail(8)
```

Indexing and Slicing Columns

- There are two methods for indexing and slicing columns from a DataFrame:
  - ❑ DOT method (not recommended)
  - ❑ Bracket method
- The bracket method is intuitive and easy to follow. In this method, you can access the data by the generic name/header of the column.
- Note: For more than one column, the object turns into a DataFrame. But for a single column, it becomes a Pandas Series object.

Indexing and Slicing Rows

- Indexing and slicing rows in a DataFrame can also be done using two methods:
  - ❑ Label-based loc method
  - ❑ Index-based iloc method
- The loc method is intuitive and easy to follow. In this method, you can access the data by the generic name of the row.
- On the other hand, the iloc method allows you to access the rows by numeric index. It can be very useful too for a large table with thousands of rows, especially when you want to iterate over the table in a loop with a numeric counter.

Exercise 41: Creating and Deleting a New Column or Row

- One of the most common tasks in data wrangling is creating or deleting columns or rows of data from your DataFrame.
- Sometimes, you want to create a new column based on some mathematical operation or transformation involving the existing columns.
- This is similar to manipulating database records and inserting a new column based on simple transformations.
- Note that all the normal operations are NOT in-place, that is, they don't impact the original DataFrame object but return a copy of the original with the addition (or deletion).
- The last bit of code in Exercise 21 shows how to make a change in the existing DataFrame with the '**inplace=True**' argument. **But note this change is irreversible and should be used with caution.**

Packt

Statistics and Visualization with NumPy and Pandas

15 mins

Refresher of Basic Descriptive Statistics

- Descriptive statistics provide simple summaries about the sample and the measures.
- These plots are often the first step in identifying fundamental patterns as well as oddities (if present) in the data.
- There are two broad approaches for descriptive statistical analysis:
  - ➢ Graphical techniques: bar plots, scatter plots, line charts, box plots, histograms, and so on.
  - ➢ Calculation of central tendency and spread: mean, median, mode, variance, standard deviation, range, and so on.

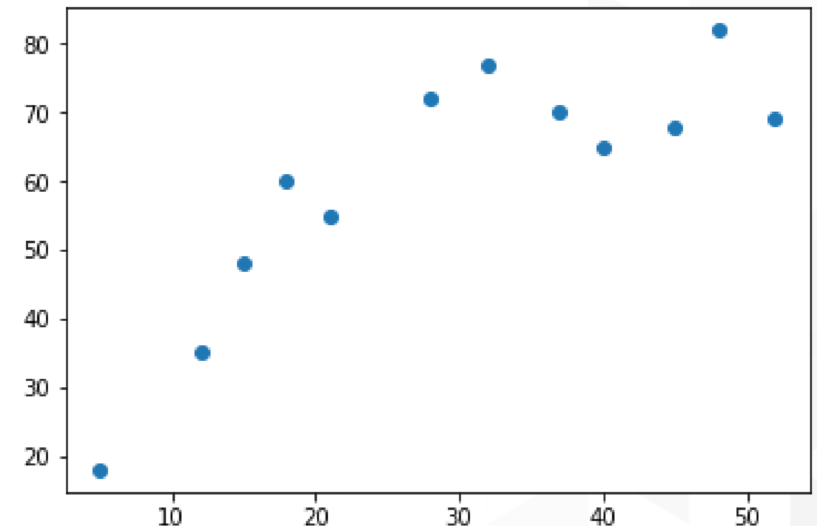
Exercise 42: Introduction to Matplotlib through a Scatter Plot

- We will demonstrate the power and simplicity of Matplotlib by creating a simple scatter plot from some data about the age, weight, and height of few people:

```
people = ['Ann', 'Brandon', 'Chen', 'David', 'Emily', 'Farook',
          'Gagan', 'Hamish', 'Imran', 'Joseph', 'Katherine', 'Lily']
age = [21, 12, 32, 45, 37, 18, 28, 52, 5, 40, 48, 15]
weight = [55, 35, 77, 68, 70, 60, 72, 69, 18, 65, 82, 48]
height = [160, 135, 170, 165, 173, 168, 175, 159, 105, 171,155, 158]
```

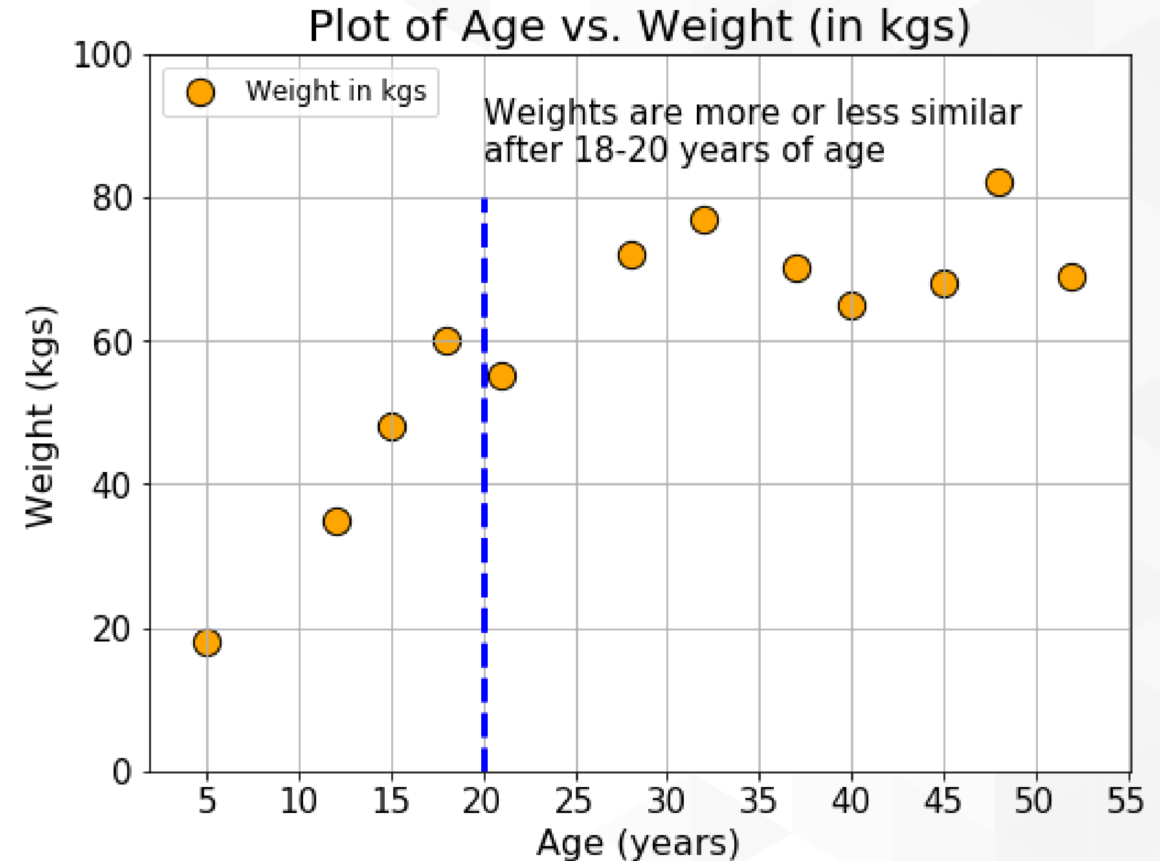
- Next, we create simple scatter plots of age versus weight:

```
plt.scatter(age,weight)
plt.show()
```



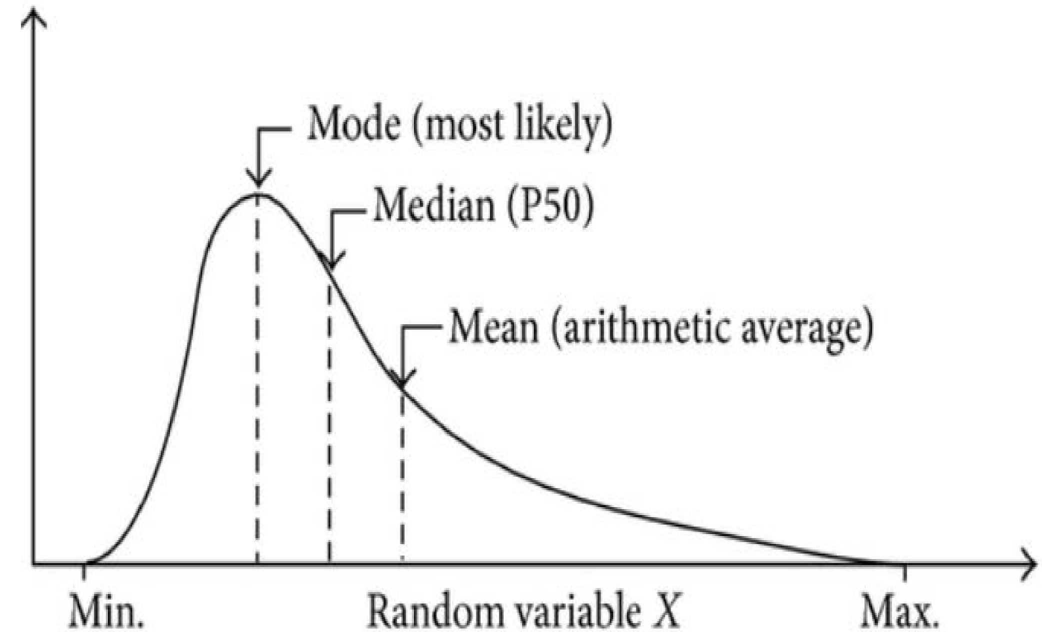
Creating a nicer plot!

- We add few embellishments to the plot:
  - ✓ Figure size enlarged and aspect ratio customized
  - ✓ Main title with large font size
  - ✓ X-axis label and Y-axis label with customized font size
  - ✓ Grid lines
  - ✓ X and Y tick marks' font size enlarged
  - ✓ Scatter plot point color is customized to orange, and the dots have been enlarged
  - ✓ A text annotation is added at a pre-defined place inside the plot
  - ✓ A vertical dashed line is drawn on top of the main scatter plot to demarcate the regions separated by the *x*=20 years
  - ✓ A legend is added



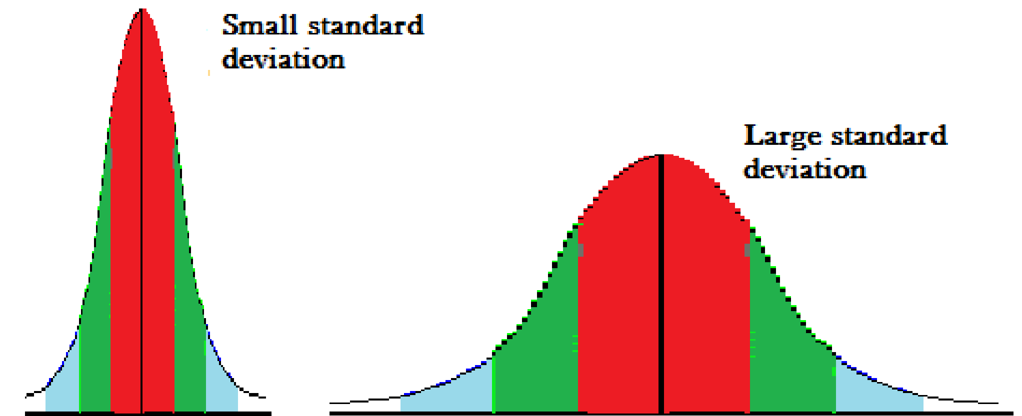
Definition of Statistical Measures: ***Central Tendency***

- A measure of ***central tendency*** is a single value that attempts to describe a set of data by identifying the central position within that set of data.
- **Mean**: Mean is equal to the *sum of all the values in the data set divided by the number of values* in the dataset.
- **Median**: The median is the middle value *that splits dataset in half*.
- **Mode**: The mode is the value *that occurs the most frequently* in your dataset. On a bar chart, the mode is the highest bar.



Definition of Statistical Measures: ***Spread***

- The spread of the data is a measure of how much the values in the dataset are likely to differ from the mean of the values.
- **Variance**: This is the most common measure of spread. Variance is the *average of the squares of the deviations from the mean*.
- **Standard Deviation**: Because variance is produced by squaring the distance from the mean, its unit does not match that of the original data. Standard deviation is just a mathematical trick to bring back the parity. It is the *positive square root of the variance*.



Random Variables

- A *random variable* is defined as the value of a given variable that represents the outcome of a statistical experiment or process. Although it sounds very formal, pretty much everything around us that we can measure can be thought as a random variable.
- The reason behind that is **almost all natural, social, biological, and physical processes are the final outcome of a large number of complex processes**, and we cannot know the details of those fundamental processes. All we can do is to observe and measure the final outcome. Typical examples around us are:
  - Economic output of a nation
  - Blood pressure of a patient
  - Temperature of a chemical process in a factory
  - Number of friends of a person on Facebook
  - Stock market price of a company

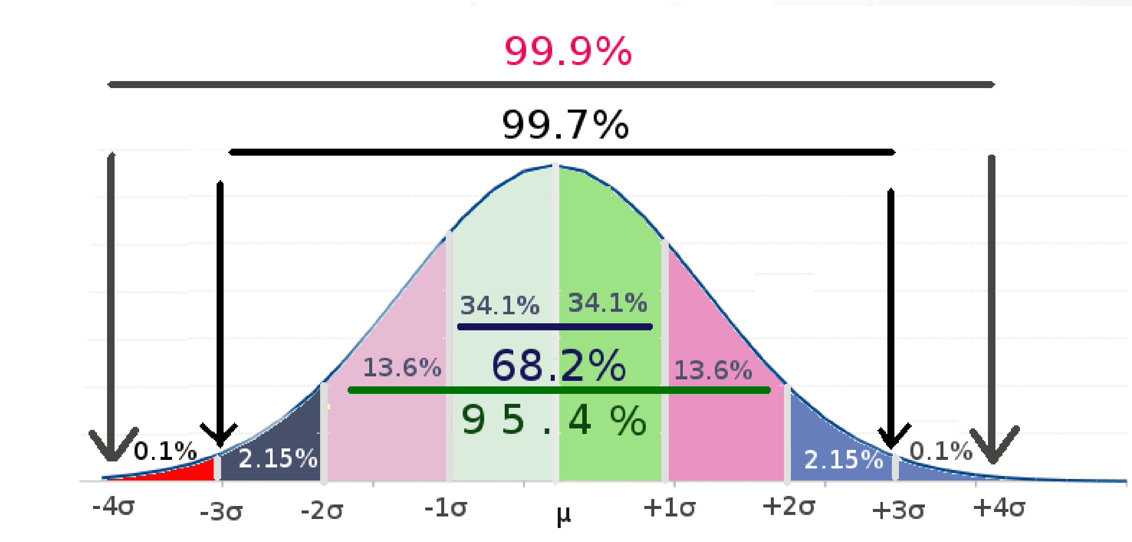
Probability Distributions

- A probability distribution is a function that describes the **likelihood of obtaining the possible values that a random variable can assume**. In other words, the values of the variable vary based on the underlying probability distribution.
- Suppose you go to a school and measure the heights of the students selected randomly. Height is an example of a random variable here. As you measure heights, you can create a distribution of heights.

Statistically distribution of heights

Continuous Distributions

- **Continuous distribution:** Continuous probability functions are also known as **probability density functions**. You have a continuous distribution *if the variable can assume an infinite number of values between any two values*. Continuous variables are often measurements on a real number scale, such as height, weight, and temperature.
- The most well-known continuous distribution is the ***Normal distribution***, which is also known as the ***Gaussian distribution*** or the "bell curve." This symmetric distribution fits a wide variety of phenomena, such as human height and IQ scores.



Random Number Generation Using NumPy

- NumPy offers an array of random number generation utility functions corresponding to various statistical distributions, such as uniform, binomial, Gaussian normal, Beta/Gamma, and chi-square.
- Most of these functions are extremely useful and appear countless times in advanced statistical data mining and machine learning tasks. **Having a solid knowledge of them is strongly encouraged for all the students on this course**.
- However, for the sake of brevity, we will not cover them in great detail.
- We will discuss only the three most important of these distributions that may come handy for data wrangling tasks – uniform, binomial, and Gaussian Normal.

Exercise 43: Generating Random Numbers from a Uniform Distribution

- Generate a random integer between 1 and 10:

```
x = np.random.randint(1,10)
print(x)
```

- Generate a random integer between 1 and 10 but with size=1 as an argument. This time it generates a NumPy array of size 1:

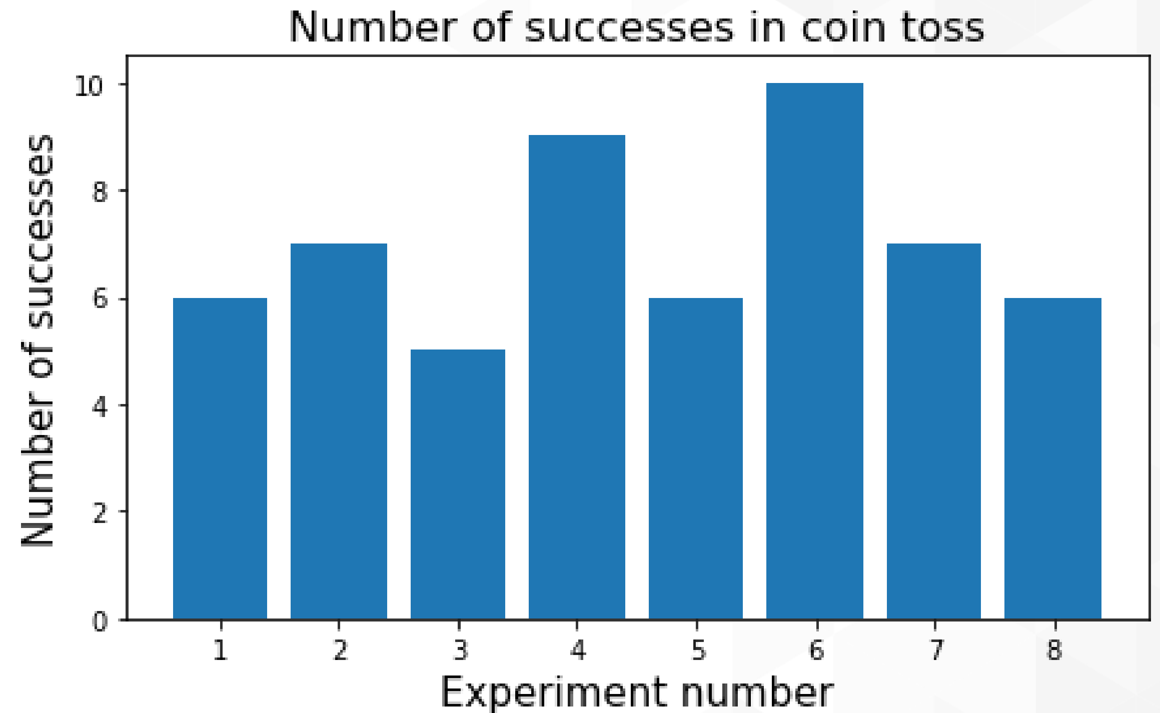
```
x = np.random.randint(1,10,size=1)
print(x)
```

- Therefore, we can easily write the code now to generate the outcome of a dice throw (normal 6-sided dice) for 10 trials. **Isn't it easy to simulate real-life probabilistic events such as throwing a dice using Python and NumPy?**

```
x = np.random.randint(1, 7, size=10)
print(x)
```

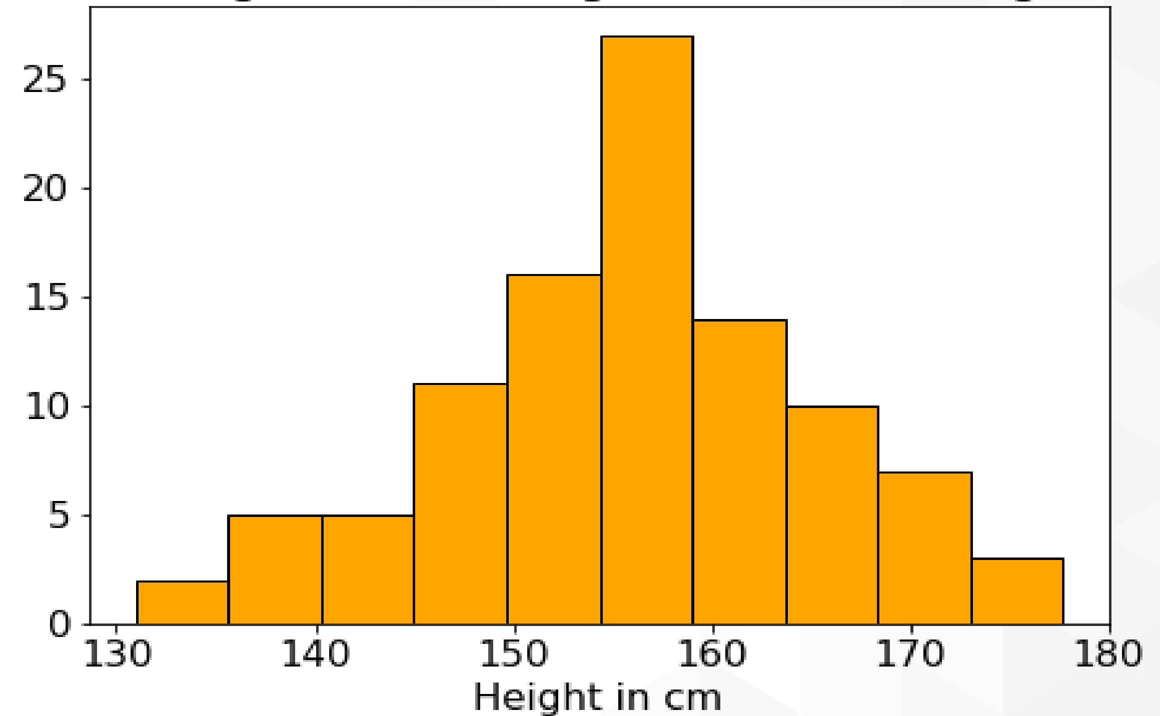
Exercise 44: Generating Random Numbers from a Binomial Distribution

- The Binomial distribution is the probability distribution of getting a specific number of success in a specific number of trials of an event with a pre-determined chance or probability.
- Example: Tossing a coin
- Plot a binomial distribution with a success probability of 0.6. Toss the coin 10 times and repeat the 10 tosses 8 times.



Exercise 45:Generating Random Numbers from a Normal Distribution

- The normal distribution is the most important probability distribution because many natural, social, and biological data follows this pattern.
- The NumPy function for a normal distribution is `normal`.
- Example: Plot the heights of the teenage (12-16 years) students distributed normally with a mean height of 155 cm and a standard deviation of 10 cm.



Exercise 46: Calculation of Descriptive Statistics from a DataFrame

Calculate statistical functions directly from a DataFrame object:

1. First, find out the number of rows and columns of the DataFrame:

```
print(people_df.shape)
```

2. Obtain a simple count (any column can be used for this purpose):

```
print(people_df['Age'].count())
```

3. Calculate the sum/mean total of age:

```
print(people_df['Age'].sum())
print(people_df['Age'].mean())
```

4. Calculate the median weight:

```
print(people_df['Weight'].median())
```

Exercise 47: Built-in Plotting Utilities

- This is the histogram function: `hist()`
- To create histograms of single columns, use this code:

```
people_df['Weight'].hist()
plt.show()
```

- The scatter function: `scatter()`
- To plot the relationship between two columns

```
people_df.plot.scatter('Weight', 'Height', s=150, c='orange', edgecolor='k')

plt.title("Weight vs. Height scatter plot", fontsize=18)

plt.xlabel("Weight (in kg)", fontsize=15)

plt.ylabel("Height (in cm)", fontsize=15)

plt.show()
```

Activity 5: Generating Statistics from a CSV File

These are the steps that you need to follow to complete this activity:

1. Load necessary libraries and read in the Boston housing dataset (given as a .csv file).
2. Create a smaller DataFrame with columns that do not include CHAS, NOX, B, and LSTAT. Check the last seven records of the new DataFrame you just created.
3. Plot histograms of all the variables (columns) in the new DataFrame. Plot them all at once using a for loop. Create a scatter plot of crime rate versus price. To understand the relationship better, plot using *log10(crime) vs. Price*.
4. Calculate some useful statistics, such as mean rooms per dwelling, median age, mean distance to five Boston employment centers, the percentage of houses with a low price (< $20,000).

Summary

- In this Lesson, we covered the following topics:
  - ❑ Basics of NumPy arrays – creation and properties
  - ❑ Operations with NumPy arrays – indexing, slicing, filtering, and reshaping
  - ❑ Special arrays – zeros, ones, identity matrices, and random arrays
  - ❑ Basics of the pandas series object
  - ❑ Basics of the pandas DataFrame
  - ❑ Basic operations on DataFrames – indexing, subsetting, and row and column addition and deletion
  - ❑ Basics of plotting with Matplotlib
  - ❑ Refresher on basic descriptive statistics and probability distributions
- In the upcoming lesson, we will cover more advanced operation with pandas DataFrames.

Practice Questions

Practice Questions

1. Can we use list comprehension inside the .loc method?
2. What method can you use to detect columns that have identical data throughout ?
3. For Boolean filtering, should we filter by passing the entire DataFrame or specific columns?
4. Give an example of setting/resetting the index of a DataFrame.
5. What are the three classes of functions we can apply in GroupBy?
6. Is the GroupBy method similar to an Excel pivot table?
7. TRUE or FALSE? There is an universally accepted definition of an outlier.
8. Does the fillna method work only with a fixed value?
9. Merging DataFrames often leads to duplicate values due to same key. How do we handle this?
10. For random sampling from a DataFrame, can you only do a fixed amount of sampling?

Packt>

Lesson 4
A Deep Dive into Data Wrangling With Python

Time: 2 Hrs

Lesson Objectives

By the end of this lesson, you will be able to

- Perform slicing, indexing, grouping, and bucketing on pandas DataFrames
- Apply Boolean filtering and indexing from a DataFrame to choose specific elements
- Perform JOIN operations in pandas that are analogous to the SQL command
- Identify missing or corrupted data and choose to drop or apply imputation techniques on missing or corrupted data

Subsetting, Filtering, and Grouping

30 mins

Why Filter or Group Data

- One of the most important aspects of data wrangling is to carefully curate useful data.
- More data is not always a good thing.
- Often, data sources are biased or, occasionally corrupt the incoming data.
- Example: US state-level economic output data. One ML model may require data only for large and populous states (such as California and Texas). Another model may demand processed data only for small and sparsely-populated states (for example, Montana or North Dakota). As a data wrangler, you have to filter and group data accordingly (based on the population of the state) before processing it and produce separate datasets as the final output for separate ML models.

Exercise 48: Loading and Examining a Superstore's Sales Data from an Excel File

- First, we load the Excel file (provided in the GitHub link) by a simple Pandas method, *read_excel*:

```
df = pd.read_excel("Sample - Superstore.xls")
df.head()
```

- We examine the file and observe that the first column is called “Row ID,” which is not very useful. So, we drop this column altogether from the DataFrame using the *.drop* method:

```
df.drop('Row ID', axis=1, inplace=True)
```

- We can quickly check the number of rows and columns in the dataset. We should see that the dataset has 9,994 rows and 20 columns now.

Discuss

What is the difference between drop in DataFrames and dropna in pandas?

Subsetting the DataFrame

- Subsetting involves the extraction of partial data based on specific columns and rows as per the business need. Suppose we are interested only in the following information from this dataset: Customer ID, Customer Name, City, Postal Code, and Sales.
- How can we subset the DataFrame to extract only lines 5 to 19 using a single line of code?
- We use the `.loc` method to index the dataset by the name of the columns and the index of the rows:

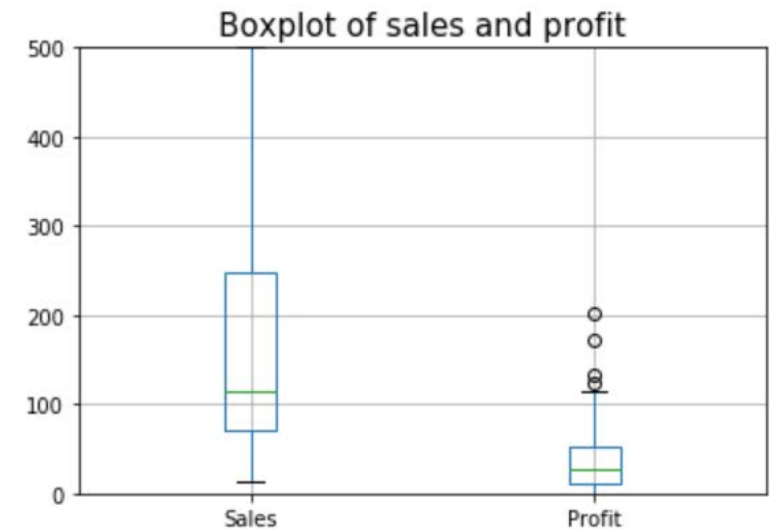
```
df_subset = df.loc[[i for i in range(5,10)], ['Customer ID',
'Customer Name', 'City', 'Postal Code', 'Sales']]
```

An Example Use Case – Determining Statistics on Sales and Profit for Records 100-199

- Calculate descriptive statistics of records 100-199 for sales and profit using subsetting:

```
df_subset = df.loc[[i for i in range(100,200)],
            ['Sales','Profit']]
df_subset.describe()
```

- **Why didn't we simply extract records 100-199 and run the "describe" function on it**? Because we are only interested in sales and profit numbers.
- For a real-life dataset, the number of rows and columns could be in the millions, and **we don't want to compute anything that is not asked for in the data wrangling task**.



Exercise 49: The unique Function

- This function is used to ***scan through data quickly and extract only unique values*** in a column or row.
- This is most useful when you want to know about the **variety of your data**.
- To find *how many countries/states/cities are present in the dataset:*

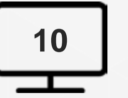
```
df['State'].unique()
```

- You can use the **nunique** method to count the number of unique values:

```
df['State'].nunique()
```

- This returns 49 for this dataset. So, one out of 50 states in the US does not appear in this dataset.

Conditional Selection and Boolean Filtering



- Often, we don't want to process a whole dataset and would like to select only a partial dataset whose contents satisfy a particular condition. **This is probably the most common use case of any data wrangling task.**
- In the context of our superstore sales dataset, think of these common questions that may arise from the daily activities of the business analytics team:
  - ➢ What are the average sales and profit figures in California
  - ➢ Which states have the highest and lowest total sales?
  - ➢ What consumer segment has most variance in sales/profit?
  - ➢ Among the top 5 states in terms of sales, which shipping mode and product category are the most popular choices?

Conditional Selection and Boolean Filtering (contd)

- If you have any prior experience with SQL, you will know that these kinds of questions require fairly complex SQL query writing. Remember the **WHERE** clause?
- We will show you how to use conditional subsetting and Boolean filtering to answer such questions.
- First, we need to understand the critical concept of Boolean indexing. This process essentially **accepts a conditional expression as an argument and returns a dataset of Booleans in which the TRUE value appears only in places where the condition was satisfied**.

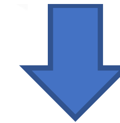
Conditional Selection and Boolean Filtering (contd)

- For demonstration purposes, we subset a small dataset of 10 records and 3 columns.
- Now, if we just want to know the records with sales higher than $100, then we can write:

```
df_subset['Sales'] > 100
```

- This produces the following Boolean DataFrame
- **Note the True and False entries** in the *Sales* column. Values in the *Ship Mode* and *State* columns were not impacted on by this code because the comparison was with a numerical quantity and the only numeric column in the original DataFrame was *Sales*.

	Ship Mode	State	Sales
0	Second Class	California	90.570
1	First Class	Minnesota	45.980
2	Standard Class	Delaware	45.000
3	First Class	New York	30.000
4	Standard Class	California	13.980
5	Standard Class	Minnesota	19.990
6	Standard Class	Illinois	95.976
7	Standard Class	Colorado	238.896
8	Standard Class	North Carolina	74.112
9	Standard Class	Illinois	339.960



	Ship Mode	State	Sales
0	True	True	False
1	True	True	False
2	True	True	False
3	True	True	False
4	True	True	False
5	True	True	False
6	True	True	False
7	True	True	True
8	True	True	False
9	True	True	True

Conditional Selection and Boolean Filtering (contd)

- Now, if we pass this Boolean DataFrame as an index to the original DataFrame, something funny happens:

```
df_subset[df_subset['Sales']>10]
```

- Where did the ***NaN*** values come from?
- They simply came from the fact that the preceding code tried to create a DataFrame with TRUE indices (in the Boolean DataFrame) only. So, for the first two columns, everything went OK and all the rows were kept. For the third column, however, only some values were TRUE in the Boolean DataFrame and, therefore, only those values were included in the final output DataFrame.
- But the first two columns were already set with all the rows, so the final DataFrame must contain all the rows. Therefore, the program inserted NaN values for the rows where data was not available.

Conditional Selection and Boolean Filtering (contd)

- We probably don't want to work with this resulting DataFrame with NaN values. We wanted a smaller DataFrame with only the rows where Sales > $100.
- We can achieve that simply by passing not the entire DataFrame in the conditional expression but only the Sales column:

```
df_subset[df_subset['Sales']>100]
```

- We are not limited to conditional expressions involving numeric quantities only; try:

```
df_subset[(df_subset['State']!='California') & (df_subset['Sales']>100)]
```

Exercise 50: Setting and Resetting the Index

- Sometimes, we may need to reset or eliminate the default index of a DataFrame and assign a new column as the index.
- The code is given in your notebook.
- The resulting DataFrames are shown here.

The DataFrame

	Age	Height	Weight
A	22	66	140
B	42	70	148
C	30	62	125
D	35	68	160
E	25	62	152

After resetting index

	index	Age	Height	Weight
0	A	22	66	140
1	B	42	70	148
2	C	30	62	125
3	D	35	68	160
4	E	25	62	152

After resetting index with 'drop' option TRUE

	Age	Height	Weight
0	22	66	140
1	42	70	148
2	30	62	125
3	35	68	160
4	25	62	152

Adding a new column 'Profession'

	Age	Height	Weight	Profession
A	22	66	140	Student
B	42	70	148	Teacher
C	30	62	125	Engineer
D	35	68	160	Doctor
E	25	62	152	Nurse

Setting 'Profession' column as index

Profession	Age	Height	Weight
Student	22	66	140
Teacher	42	70	148
Engineer	30	62	125
Doctor	35	68	160
Nurse	25	62	152

The GroupBy method

- By “group by” we are referring to a process involving one or more of the following steps:
  - ➢ Splitting the data into groups based on some criteria
  - ➢ Applying a function to each group independently
  - ➢ Combining the results into a data structure
- In some situations, we may wish to split the dataset into groups and do something with those groups.
- In the apply step, we might wish to do one of the following,
  - ➢ **Aggregation**: Compute a summary statistic (or statistics) for each group – sum, mean, and so on
  - ➢ **Transformation**: Perform some group-specific computations and return a like-indexed object – z-transformation or filling missing data with a value
  - ➢ **Filtration**: Discard some groups, according to a group-wise computation that evaluates TRUE or FALSE

Exercise 51: The GroupBy Method

- We start by creating the same 10-record subset as before
- The following code creates a pandas DataFrameGroupBy object:

```
byState = df_subset.groupby('State')
```

- We can now calculate the mean or total sales figure by state:

```
print("\nGrouping by 'State' column and listing mean sales\n",
'-'*50, sep='')
print(byState.mean())

print("\nGrouping by 'State' column and listing total sum of sales\n",
'-'*50, sep='')
print(byState.sum())
```

```
Grouping by 'State' column and listing mean sales
--------------------------------------------------
                 Sales
State
California  185.219333
Florida     489.972750
Kentucky    496.950000
```

```
Grouping by 'State' column and listing total sum of sales
--------------------------------------------------
                Sales
State
California  1111.3160
Florida      979.9455
Kentucky     993.9000
```

Exercise 51: The GroupBy method (contd)

- There is of course a describe method to this GroupBy object, which produces the summary statistics in the form of a DataFrame.
- We can subset that DataFrame for a particular state and show the statistics:

```
pd.DataFrame(byState.describe().loc['California'])
```

		California
Sales	count	2.000000
	mean	52.275000
	std	54.157308
	min	13.980000
	25%	33.127500
	50%	52.275000
	75%	71.422500
	max	90.570000

- Here's another similar summarization by the *Ship Mode* attribute:

```
df_subset.groupby('Ship Mode').describe().loc[['Second Class',
'Standard Class']]
```

	Sales							
	count	mean	std	min	25%	50%	75%	max
Ship Mode								
Second Class	3.0	336.173333	364.373037	14.62	138.290	261.96	496.950	731.9400
Standard Class	7.0	296.663071	435.947552	7.28	20.436	48.86	511.026	957.5775

Exercise 51: The GroupBy method (contd)

- GroupBy is not limited to a single variable.
- If you pass on multiple variables (as a list), then you will get back a **structure essentially similar to a Pivot Table (from Excel).**
- The following is an example where we group together all states and cities from the whole dataset (the snapshot is a partial view only).
- Note, how pandas grouped the data by *State* first and then by cities under each state.

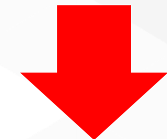
		count	mean	std	min	25%	50%	75%	max
State	**City**								
Alabama	**Auburn**	6.0	294.471667	361.914543	3.760	8.8050	182.0300	456.40750	900.080
	Decatur	13.0	259.601538	385.660903	14.940	23.9200	44.9500	239.92000	1215.920
	Florence	5.0	399.470000	796.488863	4.980	7.2700	12.4800	152.76000	1819.860
	Hoover	4.0	131.462500	230.646923	7.160	13.3925	20.7250	138.79500	477.240
	Huntsville	10.0	248.437000	419.576667	3.620	26.8700	81.9200	171.80750	1319.960
	Mobile	11.0	496.635455	914.087425	8.960	46.8600	70.9800	505.96500	3040.000
	Montgomery	10.0	372.273000	475.397645	10.160	21.7075	187.2150	499.05500	1394.950
	Tuscaloosa	2.0	87.850000	76.523096	33.740	60.7950	87.8500	114.90500	141.960
Arizona	**Avondale**	6.0	157.801333	288.247527	14.576	18.1480	35.5960	88.67800	742.336
	Bullhead City	2.0	11.144000	4.559425	7.920	9.5320	11.1440	12.75600	14.368
	Chandler	7.0	153.821000	305.283748	8.544	9.1200	49.7920	78.89750	842.376
	Gilbert	15.0	278.158800	346.945589	5.904	36.1240	82.3680	375.80700	1113.024
	Glendale	23.0	126.863696	225.003236	2.368	14.8760	42.9760	109.13200	933.536
	Mesa	28.0	144.205000	155.275947	4.368	31.7640	81.6515	202.90250	552.000

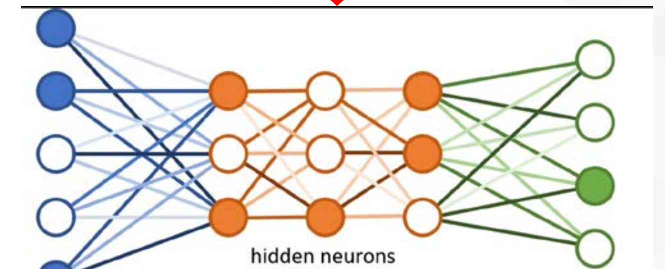
Detecting Outliers and Handling Missing Values

20 mins

Outlier Detection and Handling Missing Values

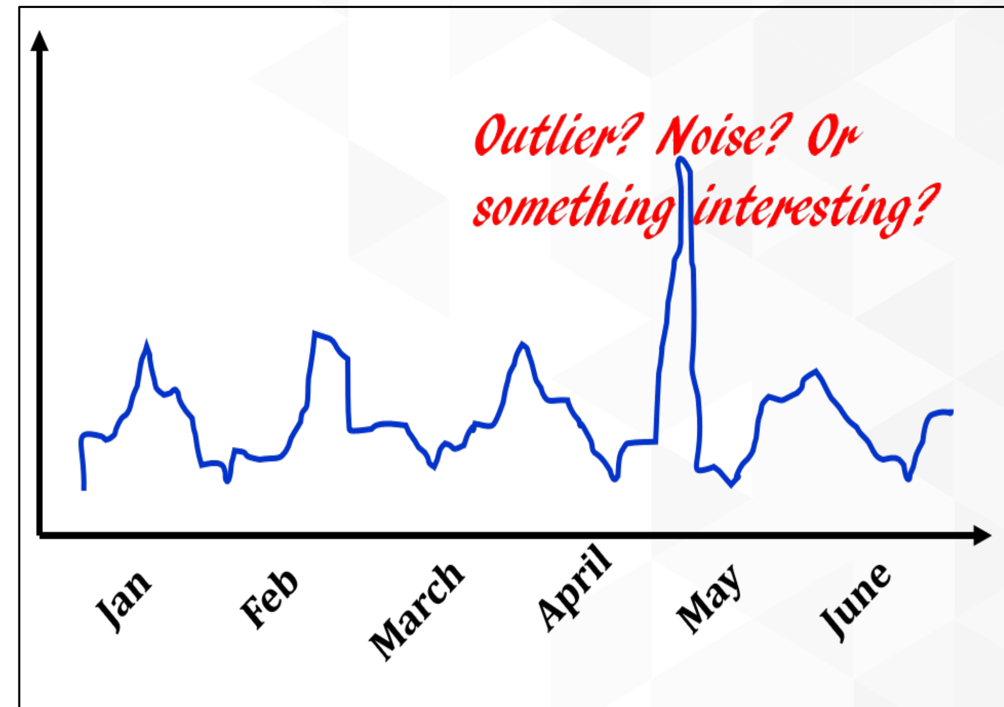
- A well-known adage in data science/simulation community is: "Garbage in, garbage out."
- A modeling or data mining process depends on the quality and consistency of the input data.
  - Some ML models, such as Bayesian learning, are inherently robust to outliers and missing data.
  - Decision Trees and Random Forest have issues with missing data.
  - Therefore, it is almost always imperative to impute missing data before handing it over to such an ML model.





An *Ambiguous* Example of an Outlier…

- Notice the unusual spike in the data around mid-April.
- Is it a noisy, bad measurement? Is it something real?
- A good data scientist or data wrangler should not delete it just because it falls outside the statistical range.
- But just because it was real does not mean it is useful.
- Therefore, the key to outliers is their systematic and timely detection in an incoming stream of millions of pieces of data or while reading data from a cloud-based storage.



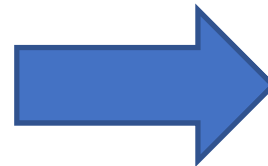
Example with the fictitious sales data of an American fast-food chain restaurant

Missing Values in Pandas

- One of the most useful functions for detecting missing values is **'isnull'**. Here, we show a snapshot of a DataFrame called *df_missing* with some missing values.
- If we simply run the following code, we get TRUE in the places where a NaN was encountered.
- Therefore, it is important to test for the presence of any NaN/missing values.

DataFrame with Missing values

	Customer	Product	Sales	Quantity	Discount	Profit
0	Leonard Middleton	NaN	1706.184	9.0	0.2	85.3092
1	Kean Nguyen	Phones	911.424	4.0	0.2	68.3568
2	Xylona Preis	Art	8.560	2.0	0.0	2.4824
3	NaN	Phones	NaN	3.0	0.2	16.0110
4	Jas O'Carroll	Binders	22.720	4.0	0.2	7.3840
5	Same Day	Binders	11.648	NaN	0.2	4.2224
6	Maris LaWare	Accessories	90.570	3.0	0.0	11.7741
7	Eileen Kiefer	NaN	77.880	2.0	0.0	NaN
8	NaN	Accessories	13.980	2.0	0.0	6.1512
9	Darrin Martin	Binders	25.824	6.0	0.2	9.3612
10	Chris Cortes	Paper	146.730	3.0	0.0	68.9631



	Customer	Product	Sales	Quantity	Discount	Profit
0	Leonard Middleton	NaN	1706.184	9.0	0.2	85.3092
1	Kean Nguyen	Phones	911.424	4.0	0.2	68.3568
2	Xylona Preis	Art	8.560	2.0	0.0	2.4824
3	NaN	Phones	NaN	3.0	0.2	16.0110
4	Jas O'Carroll	Binders	22.720	4.0	0.2	7.3840
5	Same Day	Binders	11.648	NaN	0.2	4.2224
6	Maris LaWare	Accessories	90.570	3.0	0.0	11.7741
7	Eileen Kiefer	NaN	77.880	2.0	0.0	NaN
8	NaN	Accessories	13.980	2.0	0.0	6.1512
9	Darrin Martin	Binders	25.824	6.0	0.2	9.3612
10	Chris Cortes	Paper	146.730	3.0	0.0	68.9631

Missing Values in Pandas

- If the result is > 0, then you know there are some TRUE values and missing values.

```
for c in df_missing.columns:
    miss = df_missing[c].isnull().sum()
    if miss > 0:
        print("{} has {} missing value(s)".format(c,miss))
    else:
        print("{} has NO missing value!".format(c))
```

Output:

```
Customer has 2 missing value(s)
Product has 2 missing value(s)
Sales has 1 missing value(s)
Quantity has 1 missing value(s)
Discount has NO missing value!
Profit has 1 missing value(s)
```

Exercise 52: Filling in Missing Values with `fillna()`

- To handle missing values, fill NaN values using the `fillna`, `bfill`, and `ffill` methods
- We can give a fixed replacement argument to fillna, for example:

```
df_missing.fillna('FILL')
```

- Mention the columns to be filled:

```
df_missing[['Customer','Product']].fillna('FILL')
```

- 'Pad' or 'ffill' is to forward fill the data, that is, copying from the preceding data in the series.

```
df_missing['Sales'].fillna(method='ffill')
```

- Backfill or 'bfill' is to backward fill, that is, copying from the next piece of data in the series.

```
df_missing['Sales'].fillna(method='bfill')
```

Exercise 9: Filling in Missing Values with **fillna()**

- We may want to ***fill in the missing values in Sales by the average sales amount***. Here is how we can do that:

```
df_missing['Sales'].fillna(df_missing.mean()['Sales'])
```

- Here's an example of forward fill, backward fill, and average:

```
0      1706.184
1       911.424
2         8.560
3         8.560
4        22.720
5        11.648
6        90.570
7        77.880
8        13.980
9        25.824
10      146.730
Name: Sales, dtype: float64
```

This data was filled forward i.e. copying the preceding value

```
0      1706.184
1       911.424
2         8.560
3        22.720
4        22.720
5        11.648
6        90.570
7        77.880
8        13.980
9        25.824
10      146.730
Name: Sales, dtype: float64
```

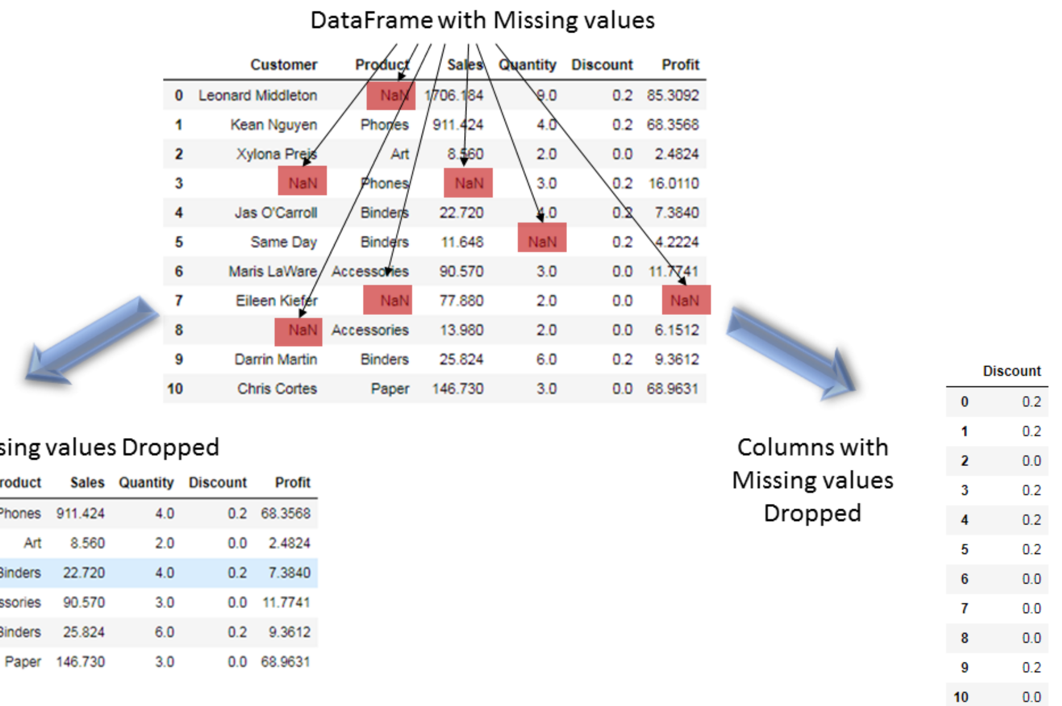
This data was filled backward i.e. copying the next value

```
0      1706.184
1       911.424
2         8.560
3       301.552
4        22.720
5        11.648
6        90.570
7        77.880
8        13.980
9        25.824
10      146.730
Name: Sales, dtype: float64
```

This data was filled by the AVERAGE

Exercise 53: Dropping Missing Values with `dropna`

- This function is used to simply drop the rows or columns that contain NaN/missing values
- You can pass on an axis parameter, if axis = 0, then rows containing missing values are dropped and if axis = 1, then columns containing missing values are dropped



DataFrame with Missing values

	Customer	Product	Sales	Quantity	Discount	Profit
0	Leonard Middleton	NaN	1706.184	9.0	0.2	85.3092
1	Kean Nguyen	Phones	911.424	4.0	0.2	68.3568
2	Xylona Preis	Art	8.560	2.0	0.0	2.4824
3	NaN	Phones	NaN	3.0	0.2	16.0110
4	Jas O'Carroll	Binders	22.720	4.0	0.2	7.3840
5	Same Day	Binders	11.648	NaN	0.2	4.2224
6	Maris LaWare	Accessories	90.570	3.0	0.0	11.7741
7	Eileen Kiefer	NaN	77.880	2.0	0.0	NaN
8	NaN	Accessories	13.980	2.0	0.0	6.1512
9	Darrin Martin	Binders	25.824	6.0	0.2	9.3612
10	Chris Cortes	Paper	146.730	3.0	0.0	68.9631

Rows with Missing values Dropped

	Customer	Product	Sales	Quantity	Discount	Profit
1	Kean Nguyen	Phones	911.424	4.0	0.2	68.3568
2	Xylona Preis	Art	8.560	2.0	0.0	2.4824
4	Jas O'Carroll	Binders	22.720	4.0	0.2	7.3840
6	Maris LaWare	Accessories	90.570	3.0	0.0	11.7741
9	Darrin Martin	Binders	25.824	6.0	0.2	9.3612
10	Chris Cortes	Paper	146.730	3.0	0.0	68.9631

Columns with Missing values Dropped

	Discount
0	0.2
1	0.2
2	0.0
3	0.2
4	0.2
5	0.2
6	0.0
7	0.0
8	0.0
9	0.2
10	0.0

Exercise 53: Dropping Missing Values with **dropna()**

- Two more arguments are useful for the dropna() method
- '*how*' - Determine whether a row or column has been removed from a DataFrame, when we have at least one NaN or all NaNs.
- '*thresh*' - Requires that many non-NaN values to keep the row/column.
- These are useful if we want to drop a particular row/column if the NaN value does not exceed a certain percentage.
- Try this code to see whether you understand the concept:

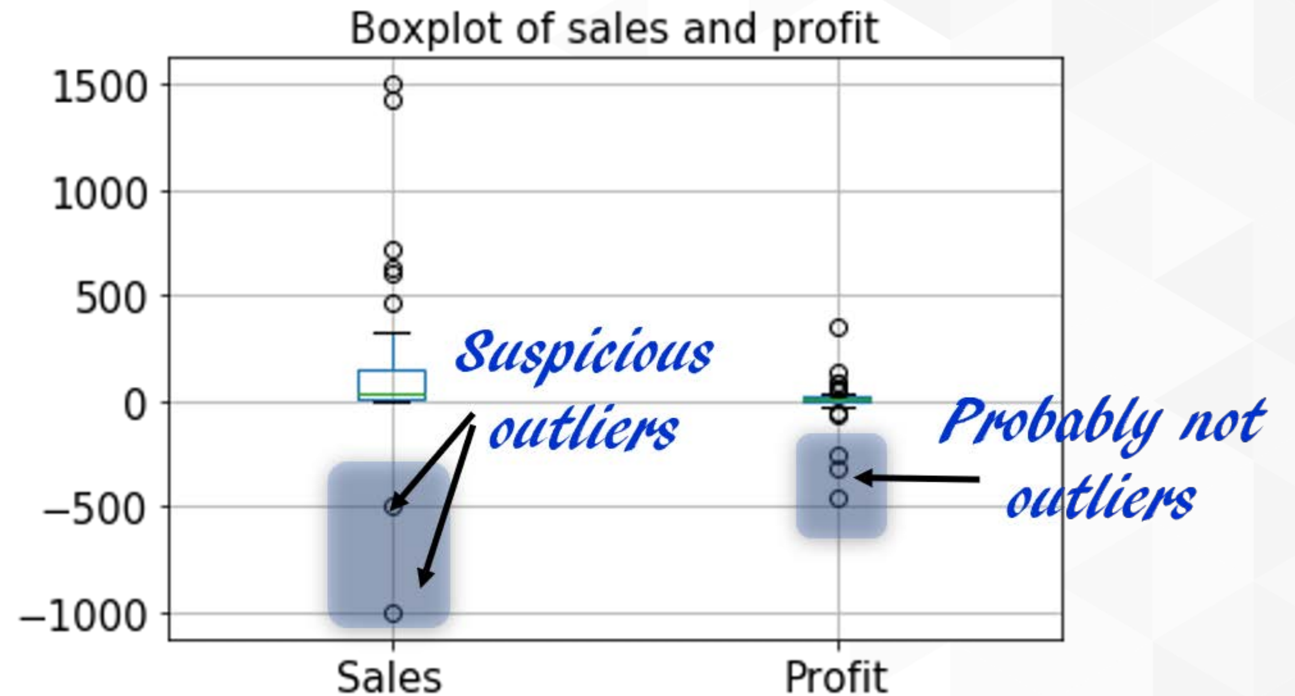
```
df_missing.dropna(axis=1, thresh=10)
```

Outlier Detection Using a Simple Statistical Test/Visualization

- As discussed, outliers in a dataset can be caused by many factors and in many ways:
  - ➢ Data entry errors
  - ➢ Experimental errors (data extraction-related)
  - ➢ Measurement errors due to noise or instrumental failure
  - ➢ Data processing errors (data manipulation or mutations due to coding errors)
  - ➢ Sampling errors (extracting or mixing data from incorrect or various sources)
- It is impossible to pinpoint one universal method for detecting outliers.
- So, we will cover some simple tricks for numeric data using standard statistical tests.

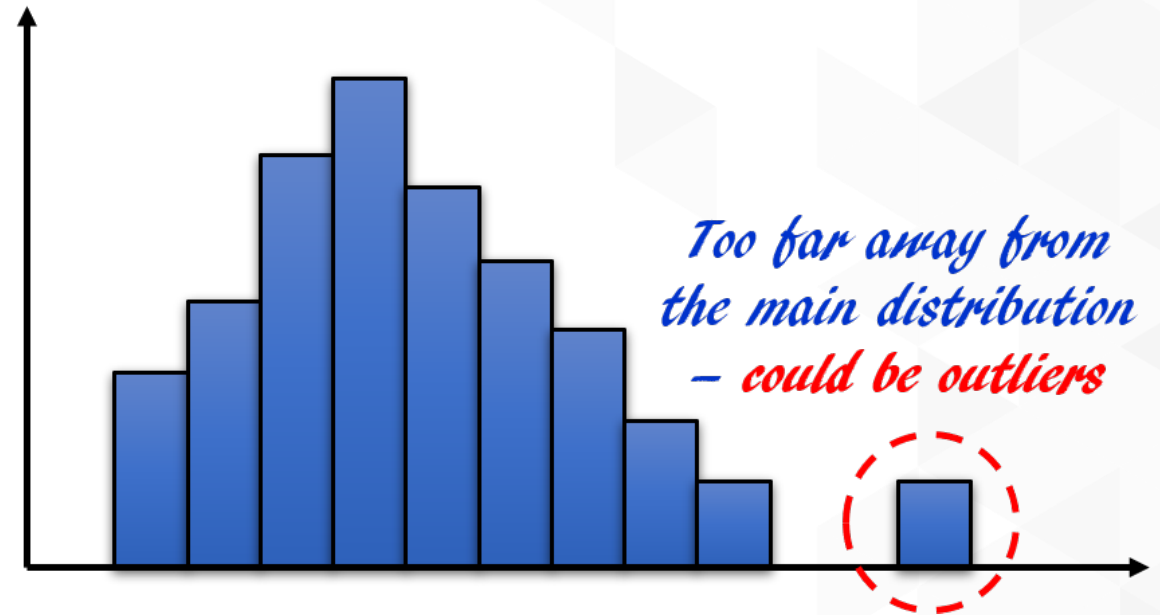
Outlier Detection Using a Simple Statistical Test/Visualization

- We can create simple boxplots to check for any unusual/nonsensical values. For example, in this exercise, we intentionally corrupted two sales values to negative and they are readily caught in a boxplot.
- *Note that profit may be negative, so those negative points are generally not suspicious. But sales cannot be negative in general, so they are detected as outliers.*



Outlier Detection Using a Simple Statistical Test/Visualization

- We can also create a distribution of a numerical quantity and check for values that lie at the extreme end to see whether they are truly part of the data or outliers.
- For example, if a distribution is almost normal, then any value of more than 4 or 5 standard deviations away may be suspect.
- Some distributions are heavy-tailed, so we need to have a good idea about the true distribution behind the data.



Concatenating, Merging, and Joining

25 mins

Where are Concatenation and Merging Used?

- Merging and joining tables or datasets is a highly common operation in the day-to-day jobs of data-wrangling professionals.
- These operations are akin to JOIN queries in SQL for relational database tables. Often, the same key data is present in multiple tables, and those records need to be brought into one combined table matching that common key.
- This is an extremely common operation in any type of sales or transactional data, and therefore, must be mastered by a data wrangler.
- Pandas offers nice and intuitive built-in methods to perform various types of JOIN involving multiple DataFrame objects.

Exercise 54: Concatenation

- We will start by showing how you can concatenate DataFrames along various axes (rows or columns). This is a very useful operation as it allows you to grow a DataFrame as new data comes in or new feature columns need to be inserted into the table.
- We first sample four records each to create three DataFrames at random from the original sales dataset we are working with:

```
df_1 = df[['Customer Name', 'State', 'Sales', 'Profit']].sample(n=4)
df_2 = df[['Customer Name', 'State', 'Sales', 'Profit']].sample(n=4)
df_3 = df[['Customer Name', 'State', 'Sales', 'Profit']].sample(n=4)
```

- Then, we create a combined DataFrame with all the rows concatenated. For that, we have to run the following code:

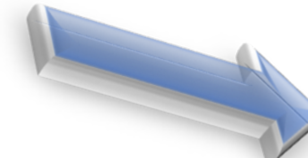
```
df_cat1 = pd.concat([df_1, df_2, df_3], axis=0)
```

Exercise 54: Concatenation

- The result of the concatenation done by the code on the previous slide is shown here. This gives you an idea about the whole process.

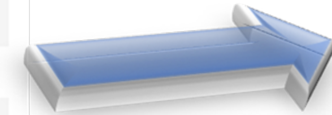
DataFrame-1

	Customer Name	State	Sales	Profit
576	Trudy Glocke	California	19.920	9.7608
6382	Troy Blackwell	Ohio	158.376	-36.9544
6883	Tamara Willingham	Washington	114.200	52.5320
6313	Nat Gilpin	Connecticut	14.990	7.3451



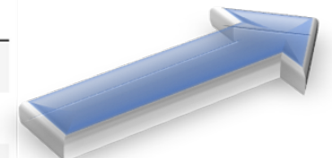
DataFrame-2

	Customer Name	State	Sales	Profit
693	Alan Barnes	California	33.400	16.0320
107	Janet Martin	North Carolina	27.992	2.0994
7706	Andrew Allen	Michigan	8.640	2.5056
803	Mike Caudle	Louisiana	12.960	6.2208



DataFrame-3

	Customer Name	State	Sales	Profit
3081	Dianna Wilson	Kentucky	58.340	28.0032
6208	Michelle Ellison	Ohio	51.168	-6.3960
9561	Susan MacKendrick	Ohio	431.976	-75.5958
4488	Cathy Armstrong	Texas	69.120	-14.6880



Concatenated DataFrame

	Customer Name	State	Sales	Profit
576	Trudy Glocke	California	19.920	9.7608
6382	Troy Blackwell	Ohio	158.376	-36.9544
6883	Tamara Willingham	Washington	114.200	52.5320
6313	Nat Gilpin	Connecticut	14.990	7.3451
693	Alan Barnes	California	33.400	16.0320
107	Janet Martin	North Carolina	27.992	2.0994
7706	Andrew Allen	Michigan	8.640	2.5056
803	Mike Caudle	Louisiana	12.960	6.2208
3081	Dianna Wilson	Kentucky	58.340	28.0032
6208	Michelle Ellison	Ohio	51.168	-6.3960
9561	Susan MacKendrick	Ohio	431.976	-75.5958
4488	Cathy Armstrong	Texas	69.120	-14.6880

Exercise 54: Concatenation

- You can also try concatenating along the columns, although that does not make any practical sense for this particular example. However, we're showing the result here just to give an idea how Pandas fill the unavailable values with NaN for that operation:

	Customer Name	State	Sales	Profit	Customer Name	State	Sales	Profit	Customer Name	State	Sales	Profit
107	NaN	NaN	NaN	NaN	Janet Martin	North Carolina	27.992	2.0994	NaN	NaN	NaN	NaN
576	Trudy Glocke	California	19.920	9.7608	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
693	NaN	NaN	NaN	NaN	Alan Barnes	California	33.400	16.0320	NaN	NaN	NaN	NaN
803	NaN	NaN	NaN	NaN	Mike Caudle	Louisiana	12.960	6.2208	NaN	NaN	NaN	NaN
3081	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Dianna Wilson	Kentucky	58.340	28.0032
4488	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Cathy Armstrong	Texas	69.120	-14.6880
6208	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Michelle Ellison	Ohio	51.168	-6.3960
6313	Nat Gilpin	Connecticut	14.990	7.3451	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6382	Troy Blackwell	Ohio	158.376	-36.9544	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6883	Tamara Willingham	Washington	114.200	52.5320	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7706	NaN	NaN	NaN	NaN	Andrew Allen	Michigan	8.640	2.5056	NaN	NaN	NaN	NaN
9561	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Susan MacKendrick	Ohio	431.976	-75.5958

Exercise 55: Merging by a Common Key

- Merging by a common key is an extremely common operation for data tables.
- The first step in building a large database for machine learning tasks where daily incoming data may be put into separate tables.
- At times, few tables needs to be merged with the master data table to be fed into the backend ML server. This will update the model and models prediction capacity.
- Here, we are showing a simple example of **inner join** with Customer Name as the key. One DataFrame `df_1` had shipping info associated with the customer name and another table `df_2` had the product information tabulated. The goal is to merge these tables into one DataFrame on the common customer name.

Exercise 55: Merging by a Common Key

- Start by subsetting a small dataset:

```
df_1=df[['Ship Date', 'Ship Mode', 'Customer Name']][0:4]
df_2=df[['Customer Name', 'Product Name', 'Quantity']][0:4]
```

- Now join these two tables with **inner join**:

```
pd.merge(df_1, df_2, on='Customer Name', how='inner')
```

	Customer Name	Ship Date	Ship Mode
0	Claire Gute	2016-11-11	Second Class
1	Claire Gute	2016-11-11	Second Class
2	Darrin Van Huff	2016-06-16	Second Class
3	Sean O'Donnell	2015-10-18	Standard Class

	Customer Name	Product Name	Quantity
0	Claire Gute	Bush Somerset Collection Bookcase	2
1	Claire Gute	Hon Deluxe Fabric Upholstered Stacking Chairs,...	3
2	Darrin Van Huff	Self-Adhesive Address Labels for Typewriters b...	2
3	Sean O'Donnell	Bretford CR4500 Series Slim Rectangular Table	5

	Customer Name	Ship Date	Ship Mode	Product Name	Quantity
0	Claire Gute	2016-11-11	Second Class	Bush Somerset Collection Bookcase	2
1	Claire Gute	2016-11-11	Second Class	Hon Deluxe Fabric Upholstered Stacking Chairs,...	3
2	Claire Gute	2016-11-11	Second Class	Bush Somerset Collection Bookcase	2
3	Claire Gute	2016-11-11	Second Class	Hon Deluxe Fabric Upholstered Stacking Chairs,...	3
4	Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2
5	Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5

Exercise 13: Merging by a Common Key

- Notice how some entries are duplicated due to multiple key values being similar. For some situations, this may be desirable, but in this case, you may want to **drop the duplicates**. To do so, you can use:

```
pd.merge(df_1, df_2, on='Customer Name', how='inner').drop_duplicates()
```

	Customer Name	Ship Date	Ship Mode	Product Name	Quantity
0	Claire Gute	2016-11-11	Second Class	Bush Somerset Collection Bookcase	2
1	Claire Gute	2016-11-11	Second Class	Hon Deluxe Fabric Upholstered Stacking Chairs,...	3
4	Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2
5	Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5

Exercise 55: Merging by a Common Key

- Next, we extract another small table, df_3, to show the concept of outer join:

```
df_3=df[['Customer Name', 'Product Name', 'Quantity']][2:6]
```

- Now, if we do inner join, we should get those elements from both tables where customer names have in common. For an outer join, we get all combination of all elements.

```
pd.merge(df_1, df_3, on='Customer Name', how='outer').drop_duplicates()
```

	Customer Name	Ship Date	Ship Mode	Product Name	Quantity
0	Claire Gute	2016-11-11	Second Class	NaN	NaN
2	Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2.0
3	Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5.0
4	Sean O'Donnell	2015-10-18	Standard Class	Eldon Fold 'N Roll Cart System	2.0
5	Brosina Hoffman	NaT	NaN	Eldon Expressions Wood and Plastic Desk Access...	7.0

Exercise 56: The join Method

- Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single DataFrame based on something called **'index keys'**.
- It follows the same core concept as in merge, but offers a faster way to accomplish merging by row indices.
- This is useful if the records in different tables are indexed differently but represent the same inherent data and you want to merge them into a single table.
- As an example, we created the following tables with customer name as the index.

	Ship Date	Ship Mode
Customer Name		
Claire Gute	2016-11-11	Second Class
Claire Gute	2016-11-11	Second Class
Darrin Van Huff	2016-06-16	Second Class
Sean O'Donnell	2015-10-18	Standard Class

	Product Name	Quantity
Customer Name		
Darrin Van Huff	Self-Adhesive Address Labels for Typewriters b...	2
Sean O'Donnell	Bretford CR4500 Series Slim Rectangular Table	5
Sean O'Donnell	Eldon Fold 'N Roll Cart System	2
Brosina Hoffman	Eldon Expressions Wood and Plastic Desk Access...	7

Exercise 56: The join Method

Left Join

Customer Name	Ship Date	Ship Mode	Product Name	Quantity
Claire Gute	2016-11-11	Second Class	NaN	NaN
Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2.0
Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5.0
Sean O'Donnell	2015-10-18	Standard Class	Eldon Fold 'N Roll Cart System	2.0

Right Join

Customer Name	Ship Date	Ship Mode	Product Name	Quantity
Brosina Hoffman	NaT	NaN	Eldon Expressions Wood and Plastic Desk Access...	7
Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2
Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5
Sean O'Donnell	2015-10-18	Standard Class	Eldon Fold 'N Roll Cart System	2

Inner Join

Customer Name	Ship Date	Ship Mode	Product Name	Quantity
Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2
Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5
Sean O'Donnell	2015-10-18	Standard Class	Eldon Fold 'N Roll Cart System	2

Outer Join

Customer Name	Ship Date	Ship Mode	Product Name	Quantity
Brosina Hoffman	NaT	NaN	Eldon Expressions Wood and Plastic Desk Access...	7.0
Claire Gute	2016-11-11	Second Class	NaN	NaN
Darrin Van Huff	2016-06-16	Second Class	Self-Adhesive Address Labels for Typewriters b...	2.0
Sean O'Donnell	2015-10-18	Standard Class	Bretford CR4500 Series Slim Rectangular Table	5.0
Sean O'Donnell	2015-10-18	Standard Class	Eldon Fold 'N Roll Cart System	2.0

Miscellaneous Useful Methods of Pandas

25 mins

What are the Miscellaneous Methods?

- In this topic, we'll discuss some small utility functions offered by pandas to work efficiently with DataFrames. They don't fall under any particular group of functions, so they are mentioned here under the miscellaneous category.

Exercise 57: Randomized Sampling

- Sampling a random fraction of a big DataFrame is often very useful. If you have a database table of 1 million records, then it is not computationally effective to run your test scripts on the full table.
- However, you may also not want to extract only the first 100 elements as the data may have been sorted by some particular key and you may get back an uninteresting table back, which may not represent the full statistical diversity of the parent database.
- In these situations, the 'sample' method comes in handy. You can choose either the number of samples you want:

```
df.sample(n=5)
```

- Or a definite fraction (percentage):

```
df.sample(frac=0.1)
```

The value_counts Method

- We discussed the unique method before, which finds and counts unique records from a DataFrame. Another useful function in the same vein is `value_counts`.
- This function returns an object containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element.
- Suppose, the business question is: which 10 customers' names occur most frequently in the sales table? You can imagine a complex SQL query for extracting this data, but in Pandas, this can be done in one simple function:

```
df['Customer Name'].value_counts()[:10]
```

Pivot Table Functionality

- Similar to group by, Pandas also offer pivot table functionality, which works the same as a pivot table in spreadsheet programs such as MS Excel.
- For example, in this sales database, you want to know the average sales, profit, and quantity sold by *Region* and *State* (two levels of index). We can easily extract the information using one simple line of code (we sample 100 records first to keep the computation fast and then apply the code):

```
df_sample.pivot_table(values=['Sales', 'Quantity',
'Profit'], index=['Region', 'State'], aggfunc='mean')
```

		Profit	Quantity	Sales
Region	**State**			
Central	**Illinois**	-13.383540	3.200000	115.384000
	Indiana	39.889350	7.000000	167.595000
	Iowa	11.847700	3.000000	25.615000
	Michigan	23.808000	4.000000	79.360000
	Minnesota	160.623000	3.000000	535.410000
	Nebraska	8.017800	3.000000	17.430000
	Texas	-18.250420	3.700000	237.700280
East	**Connecticut**	117.106650	3.000000	573.240000
	Delaware	63.476600	2.000000	214.830000
	Massachusetts	19.149300	3.666667	92.990000
	New Jersey	4.950000	3.000000	45.000000
	New York	16.665840	3.800000	115.241200
	Ohio	-11.668240	5.800000	359.931600

Exercise 58: Sorting by Column Values – sort_values

48

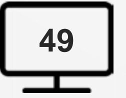
- Sorting a table by a particular column is one of the most frequently used operations in the daily work of an analyst. Not surprisingly, pandas provides a simple and intuitive method for accomplishing this – the **sort_values** method.

	Customer Name	State	Sales	Quantity
9664	Vincent Ferguson	California	12.960	2
6343	Trevor Keller	Delaware	30.440	2
2144	Kennedy Erickson	Pennsylvania	16.688	7
3919	Lisandra Harding	Virginia	61.680	4
6546	Oliver Daniels	Ohio	24.784	1
1866	Herman Macdonald	Nevada	196.450	5
462	Leo Cruz	Arizona	23.560	5
7615	Jamal Flores	Florida	6.642	9
3690	Serina Bush	Colorado	59.994	2
5511	Ahmed Stephenson	Oregon	32.896	4
2273	Karina Soto	California	1119.984	2
2642	Magee Cook	Illinois	186.048	6
3005	Regan Bryan	Wisconsin	91.680	3
8539	Jin Morgan	California	479.984	2
8684	Amir Michael	California	21.210	7

	Customer Name	State	Sales	Quantity
7615	Jamal Flores	Florida	6.642	9
9664	Vincent Ferguson	California	12.960	2
2144	Kennedy Erickson	Pennsylvania	16.688	7
8684	Amir Michael	California	21.210	7
462	Leo Cruz	Arizona	23.560	5
6546	Oliver Daniels	Ohio	24.784	1
6343	Trevor Keller	Delaware	30.440	2
5511	Ahmed Stephenson	Oregon	32.896	4
3690	Serina Bush	Colorado	59.994	2
3919	Lisandra Harding	Virginia	61.680	4
3005	Regan Bryan	Wisconsin	91.680	3
2642	Magee Cook	Illinois	186.048	6
1866	Herman Macdonald	Nevada	196.450	5
8539	Jin Morgan	California	479.984	2
2273	Karina Soto	California	1119.984	2

```
df_sample.sort_values(by='Sales')
df_sample.sort_values(by=['State', 'Sales'])
```

Exercise 59: User-Defined Functions with the ***apply*** Method



- Pandas provides great flexibility to work with user-defined functions of arbitrary complexity with the apply method.
- Much like the native Python apply function, this method accepts a user-defined function and additional arguments and returns a new column after applying the function on a particular column element-wise.
- Suppose **we want to create a column of categorical features such as high/medium/low-based on the sales price column**. Note that it is a conversion from a numeric value to a categorical factor (string) based on certain conditions (threshold values of sales). So, it is best accomplished by a simple user-defined function such as the following:

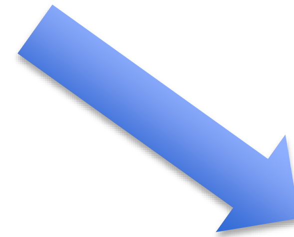
```
def categorize_sales(price):
    if price < 50:
        return "Low"
    elif price < 200:
        return "Medium"
    else:
        return "High"
```

Exercise 59: User-Defined Functions with the ***apply*** Method

50

- To illustrate the application of this function, we first sample 100 records randomly from the database.
- Next, we use the apply method to apply the categorization function to the *Sales* column. Note that we need to create a new column to store the category string values returned by the function:

	Customer Name	State	Sales
6566	Martina Sharp	Texas	243.992
3338	Margaret Everett	California	117.144
5734	Galena Payne	Connecticut	25.160
5951	Xander Giles	Mississippi	234.360
9444	Amir Mathis	Texas	95.840
7773	Brady Kirkland	Ohio	59.976
5520	Linda Howell	Pennsylvania	23.920
4025	Sierra Mosley	Texas	29.664
9511	Hamish Hurley	Illinois	9.264
6235	Russell Roach	Washington	79.960



The new column populated by the function

	Customer Name	State	Sales	Sales Price Category
6566	Martina Sharp	Texas	243.992	High
3338	Margaret Everett	California	117.144	Medium
5734	Galena Payne	Connecticut	25.160	Low
5951	Xander Giles	Mississippi	234.360	High
9444	Amir Mathis	Texas	95.840	Medium
7773	Brady Kirkland	Ohio	59.976	Medium
5520	Linda Howell	Pennsylvania	23.920	Low
4025	Sierra Mosley	Texas	29.664	Low
9511	Hamish Hurley	Illinois	9.264	Low
6235	Russell Roach	Washington	79.960	Medium

```
df_sample['Sales Price Category'] = df_sample['Sales'].apply(categorize_sales)
```

Exercise 59: User-Defined Functions with the ***apply*** Method

- The apply method also **works with built-in native Python functions**. Just for fun, if we want to create another column storing the length of the name of the customer, we can do that too using the familiar len function!

```
df_sample['Customer Name Length'] = df_sample['Customer Name'].apply(len)
```

	Customer Name	State	Sales	Sales Price Category	Customer Name Length
6566	Martina Sharp	Texas	243.992	High	13
3338	Margaret Everett	California	117.144	Medium	16
5734	Galena Payne	Connecticut	25.160	Low	12
5951	Xander Giles	Mississippi	234.360	High	12
9444	Amir Mathis	Texas	95.840	Medium	11
7773	Brady Kirkland	Ohio	59.976	Medium	14
5520	Linda Howell	Pennsylvania	23.920	Low	12
4025	Sierra Mosley	Texas	29.664	Low	13
9511	Hamish Hurley	Illinois	9.264	Low	13
6235	Russell Roach	Washington	79.960	Medium	13

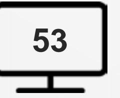
Exercise 59: User-Defined Functions with the ***apply*** Method

- Instead of writing out a separate function, we can even **insert lambda expressions directly into the apply method for short functions**. For example, let's say we are promoting our product and want to show the discounted sales price if the original price is > $200. We can do this using a lambda function and the apply method:

```
df_sample['Discounted Price'] = df_sample['Sales'].apply(lambda x:0.85*x if x>200 else x)
```

	Customer Name	State	Sales	Sales Price Category	Customer Name Length	Discounted Price
7154	Lena Hernandez	Georgia	275.880	High	14	234.4980
7627	Eudokia Martin	Arizona	23.344	Low	14	23.3440
6001	Randy Ferguson	Arizona	87.960	Medium	14	87.9600
6128	Aaron Hawkins	California	49.408	Low	13	49.4080
7289	Steven Cartwright	California	305.010	High	17	259.2585
3320	Tony Sayre	Tennessee	662.880	High	10	563.4480
8367	Lisa Ryan	California	20.560	Low	9	20.5600
3973	Janet Martin	New York	29.800	Low	12	29.8000
7380	Alejandro Grove	Nebraska	14.400	Low	15	14.4000
8915	Cynthia Voltz	Texas	13.184	Low	13	13.1840

Activity 6: Working with the Adult Income Dataset



Follow these steps to complete this activity:

- Load the necessary libraries and read the Adult Income Dataset.
- Create a script that will read a text file line by line and find the missing values.
- Create a DataFrame with age, education, and occupation by using subsetting and plot a histogram for age with a bin size of 20.
- Create a function to strip the whitespace characters. Use the apply method to apply this function to all the columns with string values, create a new column, copy the values from this new column to the old column, and drop the new column.
- Use subset and groupby to find outliers.
- Merge the data using common keys.

Summary

In this Lesson, we covered the following topics:

- Advanced subsetting and filtering on DataFrames
- Boolean indexing and the conditional selection of a subset of data
- Setting and resetting the index and Group by method of a DataFrame
- Handling missing data using various imputation techniques by dropping
- Outlier detection methods using simple statistical property checks
- Methods and a usage example of the concatenation of DataFrame objects
- Merging DataFrames using a common key
- The join method and how it compares to a similar operation in SQL
- Miscellaneous useful methods for DataFrames – randomized sampling, unique, value_count, sort_values, etc.
- The pivot table functionality of a DataFrame
- An example of running an arbitrary user-defined function on a DataFrame using the 'Apply' method

Practice Questions

Practice Questions

1. Which method can detect columns in a dataset that have identical data?
2. For Boolean filtering, should we filter by passing the entire DataFrame or only the columns that we are interested in?
3. Give a usage example of setting/resetting the index of a DataFrame.
4. What are the three classes of functions we can apply in GroupBy?

6. Can the GroupBy method be thought of as being similar to an Excel pivot table?

Practice Questions

6. True or false? There is a universally accepted definition of outlier.
7. Does the fillna method work only with a fixed value (that we can insert in place of the missing value)?
8. Merging DataFrames often leads to duplicate values due to the same key. How can this be handled?

Packt

5
Getting Comfortable With Different Kinds of Data Sources

2 Hrs

Lesson Objectives

By the end of this lesson, you will be able to:

- Read CSV, Excel, and JSON files into pandas DataFrames
- Read PDF documents and HTML tables into pandas DataFrames
- Perform basic web scraping using powerful yet easy to use libraries such as BeautifulSoup
- Extract structured and textual information from portals

Introduction

- Now, let's learn about various techniques by which we can read data into a DataFrame from external sources.
- Some of those sources could be text-based (CSV, HTML, JSON, and so on), whereas some other could be binary (Excel, PDF, and so on), that is, not in ASCII format.
- In this lesson, we will learn how to deal with data that is present in web pages and HTML documents. This holds very high importance in the work of a data practitioner.

Packt

Reading Data from Different Text-Based (And Non-Text-Based) Sources

Why do you need to read text and non-text data?

- One of the most valued and widely used skills of a data wrangling professional is the ability to extract and read data from a diverse array of sources into a structured format.
- Modern analytics pipelines depend on their ability to scan and absorb a variety of data sources to build and analyze a pattern-rich model.
- Such a feature-rich, multi-dimensional model will have high predictive and generalization accuracy and will be valued by stakeholders and end users alike for any data-driven product.

Exercise 60: Reading Data from a CSV File Where Headers Are Missing

- Execute the following command(making sure that you have the required file in the same local directory):

```
df1 = pd.read_csv("CSV_EX_1.csv")
```

- But if you just depend on the default function, then you will get an unexpected result. For example, if you execute the following, you may get the following DataFrame:

```
df2 = pd.read_csv("CSV_EX_2.csv")
```

	2	1500	Good	300000
0	3	1300	Fair	240000
1	3	1900	Very good	450000
2	3	1850	Bad	280000
3	2	1640	Good	310000

- Certainly, the top data row has been mistakenly read as the column header. You can specify header=None to avoid this.

```
df2 = pd.read_csv("CSV_EX_2.csv",header=None)
```

	0	1	2	3
0	2	1500	Good	300000
1	3	1300	Fair	240000
2	3	1900	Very good	450000
3	3	1850	Bad	280000
4	2	1640	Good	310000

Reading Data from a CSV File Where Headers Are Missing (Continued…)

- Without any header information, you will get back the following. The default headers will be just some default numeric indices starting from 0.

	0	1	2	3
0	2	1500	Good	300000
1	3	1300	Fair	240000
2	3	1900	Very good	450000
3	3	1850	Bad	280000
4	2	1640	Good	310000

- This may be fine for data analysis purposes, but if you want the DataFrame to truly reflect the proper headers, then in this case, you will have to supply them using the **names** argument:

```
df2 = pd.read_csv("CSV_EX_2.csv", header=None,
names=['Bedroom', 'Sq.ft', 'Locality',
'Price($)'])
```

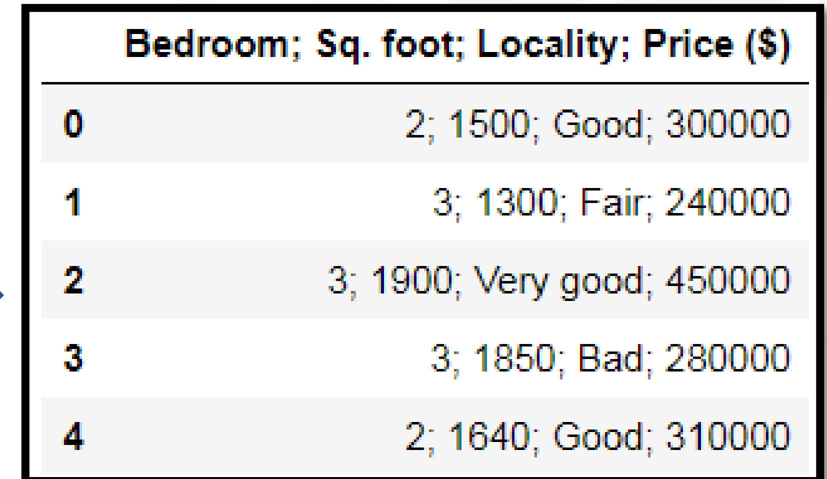
	Bedroom	Sq.ft	Locality	Price($)
0	2	1500	Good	300000
1	3	1300	Fair	240000
2	3	1900	Very good	450000
3	3	1850	Bad	280000
4	2	1640	Good	310000

Exercise 61: Reading data from a CSV where delimiters/separators are not commas

8

- Although CSV stands for comma-separated-values, it is fairly common to encounter raw data files where the separator/delimiter is a character other than a comma. If you execute the following, you will see the following DataFrame:

```
df3 = pd.read_csv("CSV_EX_3.csv")
```



	Bedroom; Sq. foot; Locality; Price ($)
0	2; 1500; Good; 300000
1	3; 1300; Fair; 240000
2	3; 1900; Very good; 450000
3	3; 1850; Bad; 280000
4	2; 1640; Good; 310000

- Clearly, the ; separator was not expected and the reading is flawed. A simple workaround is to specify the separator/delimiter explicitly in the read function:

```
df3 = pd.read_csv("CSV_EX_3.csv", sep=';')
```

Exercise 62: Bypassing the Headers of a CSV file

- If your CSV file already comes with headers but you want to bypass them and put your own, you have to specifically set **header = 0** to make it happen. If you try to set names variable to your header list, unexpected thing can happen:

```
df4 = pd.read_csv("CSV_EX_1.csv",
names=['A','B','C','D'])
```

	A	B	C	D
0	Bedroom	Sq. foot	Locality	Price ($)
1	2	1500	Good	300000
2	3	1300	Fair	240000
3	3	1900	Very good	450000
4	3	1850	Bad	280000
5	2	1640	Good	310000

- To avoid this, set **header** to zero and provide a names list:

```
df4 = pd.read_csv("CSV_EX_1.csv", header=0,
names=['A','B','C','D'])
```

Exercise 63: Skipping initial rows

10

- Skipping initial rows is a widely useful method because, most of the time, the first few rows of a CSV data file are metadata about the data source or similar information, which is not read into the table. Note that the first two lines in the CSV file are irrelevant data.

Filetype: CSV			
	Info about some houses		
Bedroom	Sq. foot	Locality	Price ($)
2	1500	Good	300000
3	1300	Fair	240000
3	1900	Very good	450000
3	1850	Bad	280000
2	1640	Good	310000

- Now, if you try the default code on this CSV, then you will get an unexpected result in the DataFrame:

```
df5 = pd.read_csv("CSV_EX_skiprows.csv")
```

	Filetype: CSV	Unnamed: 1	Unnamed: 2	Unnamed: 3
0	NaN	Info about some houses	NaN	NaN
1	Bedroom	Sq. foot	Locality	Price ($)
2	2	1500	Good	300000
3	3	1300	Fair	240000
4	3	1900	Very good	450000
5	3	1850	Bad	280000
6	2	1640	Good	310000

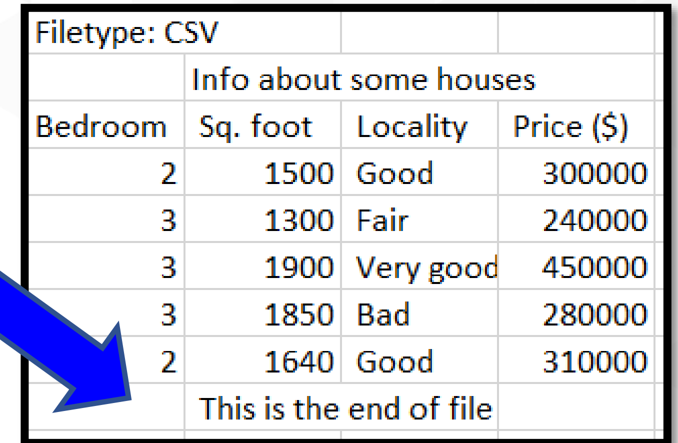
- Instead, you have to examine the file and understand that the first two rows are to be skipped. Therefore, the following code will result in the expected DataFrame:

```
df5 = pd.read_csv("CSV_EX_skiprows.csv", skiprows=2)
```

Skipping Footers

- Similar to skipping initial rows, it may be necessary to skip the footer of a file. For example, we do not want to read the data at the end of the following file:
- We have to use **skipfooter** and the **engine='python'** option to enable this. There are two engines for these CSV reader functions – based on C or Python, of which only the Python engine supports the skipfooter option:

```
df6 = pd.read_csv("CSV_EX_skipfooter.csv",
skiprows=2, skipfooter=1,engine='python')
```



Filetype: CSV			
	Info about some houses		
Bedroom	Sq. foot	Locality	Price ($)
2	1500	Good	300000
3	1300	Fair	240000
3	1900	Very good	450000
3	1850	Bad	280000
2	1640	Good	310000
	This is the end of file		

Reading Only the First *n* Rows (Especially Useful for Large Files)

- In many situations, we may not want to read a whole data file but only the first few rows.
- This is particularly useful for extremely large data files, where we may just want to read the first couple of hundred rows to check an initial pattern and then decide to read the whole data later on.
- Otherwise, the reading process can take a long time and slow down the entire data wrangling pipeline. A simple option, **nrows,** in the **read_csv** function enables us to do just that:

```
df7 = pd.read_csv("CSV_EX_1.csv", nrows=2)
```

Exercise 64: Combining Skiprows and Nrows to Read Data in Small Chunks

13

- Loop over the CSV file to read only a fixed number of rows at a time:

```
for i in range(0, num_chunks*rows_in_a_chunk, rows_in_a_chunk):
    df = pd.read_csv("Boston_housing.csv", header=0,
                     skiprows=i, nrows=rows_in_a_chunk, names=colnames)
    list_of_dataframe.append(df)
```

Setting the ***skip_blank_lines*** Option

- By default, **read_csv** ignores blank lines.
- But sometimes, you may want to read them in as **NaN** so that you can count how many such blank entries was present in the raw data file.
- In some situations, this is an indicator of the default data streaming quality and consistency. For this, you have to disable the **skip_blank_lines** option using the following command:

```
df9 = pd.read_csv("CSV_EX_blankline.csv", skip_blank_lines=False)
```

Reading a CSV from inside a Compressed (.zip/.gz/.bz2/.xz) File

- This is an awesome feature of Pandas, in that it allows you to read directly from a compressed file such as .zip, .gz, .bz2, or .xz.
- The only requirement is that the intended data file (CSV) should be the only file inside the compressed file.
- In this example, we compressed the example CSV file with 7-zip program and read directly using the read_csv method:

```
df10 = pd.read_csv('CSV_EX_1.zip')
```

Discuss

Try out the other compressed files. What are the difficulties you face?

Reading from an Excel File Using sheet_name and Handling a Distinct sheet_name

- For example, in the associated data file, Housing_data.xlsx, we have three tabs and the following code reads them one by one in three separate DataFrames:

```
df11_1 = pd.read_excel("Housing_data.xlsx", sheet_name='Data_Tab_1')

df11_2 = pd.read_excel("Housing_data.xlsx", sheet_name='Data_Tab_2')

df11_3 = pd.read_excel("Housing_data.xlsx", sheet_name='Data_Tab_3’)
```

- Let’s consider the following example:

```
dict_df = pd.read_excel("Housing_data.xlsx", sheet_name=None)

dict_df.keys()
```

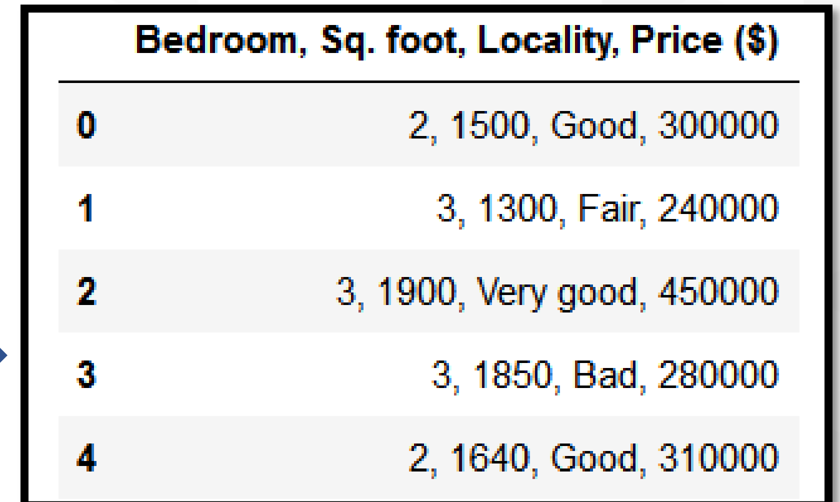
Discuss

Is sheet_name case-sensitive?

Exercise 65: Reading a General Delimited Text File

- General text files can be read as easily as CSV files. However, you have to pass on the proper separator if it is anything other than whitespace or tab.
- For example, a comma-separated file, saved with the .txt extension, will result in following DataFrame if read without explicitly setting the separator:

```
df13 = pd.read_table("Table_EX_1.txt")
```



	Bedroom, Sq. foot, Locality, Price ($)
0	2, 1500, Good, 300000
1	3, 1300, Fair, 240000
2	3, 1900, Very good, 450000
3	3, 1850, Bad, 280000
4	2, 1640, Good, 310000

- In this case, we have to set the separator explicitly:

```
df13 = pd.read_table("Table_EX_1.txt", sep=',')
```

Reading HTML Tables Directly from a URL

- Pandas allows us to read HTML tables directly from a URL.
- That means they already have some kind of **built-in HTML parser** that processes the HTML content of a given page and tries to extract various tables in the page.
- Note, that the **read_html** method returns a **list of DataFrames** (even if the page has a single DataFrame) and you have to extract the relevant tables from the list:

```
url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'
list_of_df = pd.read_html(url)
df14 = list_of_df[0]
df14.head()
```

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0	Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	December 15, 2017	February 21, 2018
1	The Farmers and Merchants State Bank of Argonia	Argonia	KS	17719	Conway Bank	October 13, 2017	February 21, 2018
2	Fayette County Bank	Saint Elmo	IL	1802	United Fidelity Bank, fsb	May 26, 2017	July 26, 2017
3	Guaranty Bank, (d/b/a BestBank in Georgia & Mi...	Milwaukee	WI	30003	First-Citizens Bank & Trust Company	May 5, 2017	March 22, 2018
4	First NBC Bank	New Orleans	LA	58302	Whitney Bank	April 28, 2017	December 5, 2017

Exercise 66: Further Wrangling to Get the Desired Data

- For example, if we want to get the table of the **2016 summer Olympics medal tally** (by nation), we can easily search to get a page on **Wikipedia** that we can pass on to Pandas.
- To look for the proper table, we can run a simple loop:

```
for t in list_of_df:
    print(t.shape)
>> (1, 1), (87, 6), (10, 8), (0, 2), (1, 1), (4, 2)
```

- It looks like the second element in this list is the table we are looking for:

```
df15 = list_of_df[1]
df15.head()
```

	Rank	NOC	Gold	Silver	Bronze	Total
0	1	United States (USA)	46	37	38	121.0
1	2	Great Britain (GBR)	27	23	17	67.0
2	3	China (CHN)	26	18	26	70.0
3	4	Russia (RUS)	19	17	20	56.0
4	5	Germany (GER)	17	10	15	42.0

Exercise 67: Reading from a JSON File

- Suppose, we want to extract the cast list for the 2012 Avengers movie (from Marvel comics):

```
df16 = pd.read_json("movies.json")
df16.head()
```

- To look for the cast where the title is 'Avengers' we can use filtering:

```
cast_of_avengers=df16[(df16['title']=="The Avengers") &
(df16['year']==2012)]['cast']
print(list(cast_of_avengers))

>> [['Robert Downey, Jr.', 'Chris Evans', 'Mark Ruffalo', 'Chris Hemsworth',
'Scarlett Johansson', 'Jeremy Renner', 'Tom Hiddleston', 'Clark Gregg', 'Cobie
Smulders', 'Stellan SkarsgÃ¥rd', 'Samuel L. Jackson']]
```

Reading a Stata File (.dta)

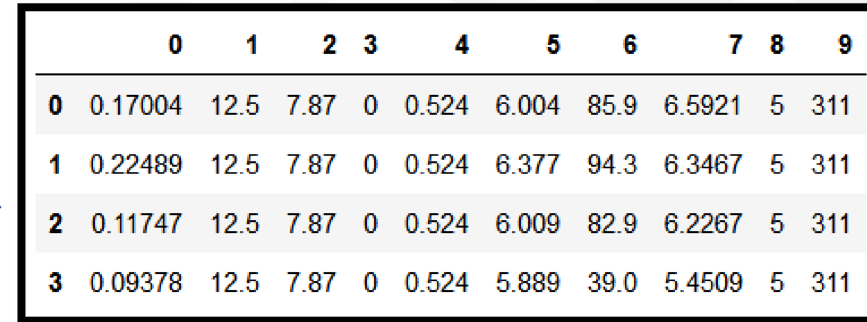
- Pandas provides a direct reading function for Stata files too.
- Stata is a popular statistical modeling platform used in many governmental and research organizations, especially by economists and social scientists. The simple code to read in a Stata file (.dta format) is as follows:

```
df17 = pd.read_stata("rscfp2016.dta")
```

Exercise 68: Reading Tabular Data from a PDF File

- The PDF format is probably the most difficult to parse in general.
- The following code retrieves the tables from two pages and joins them to make one table:

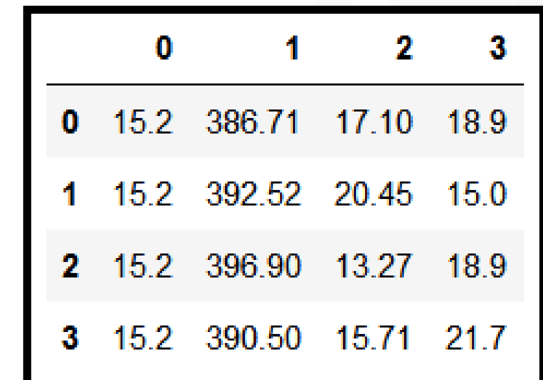
```
from tabula import read_pdf
df18_1 = read_pdf('Housing_data.pdf', pages=[1],
pandas_options={'header': None})
```



	0	1	2	3	4	5	6	7	8	9
0	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311
1	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311
2	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311
3	0.09378	12.5	7.87	0	0.524	5.889	39.0	5.4509	5	311

- For the second page:

```
df18_2 = read_pdf('Housing_data.pdf', pages=[2],
pandas_options={'header': None})
```



	0	1	2	3
0	15.2	386.71	17.10	18.9
1	15.2	392.52	20.45	15.0
2	15.2	396.90	13.27	18.9
3	15.2	390.50	15.71	21.7

Exercise 68: Reading Tabular Data from a PDF File (Continued…)

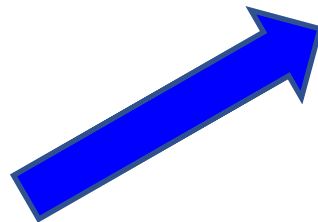
25

- Now we concatenate:

```
df18=pd.concat([df18_1, df18_2], axis=1)
```

	0	1	2	3	4	5	6	7	8	9	0	1	2	3
0	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.10	18.9
1	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311	15.2	392.52	20.45	15.0
2	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311	15.2	396.90	13.27	18.9
3	0.09378	12.5	7.87	0	0.524	5.889	39.0	5.4509	5	311	15.2	390.50	15.71	21.7

- With PDF extraction, most of the time, headers will be difficult to extract automatically. You have to pass on the list of headers as the **names** argument in the read_pdf function as **pandas_option:**



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.10	18.9
1	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311	15.2	392.52	20.45	15.0
2	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311	15.2	396.90	13.27	18.9
3	0.09378	12.5	7.87	0	0.524	5.889	39.0	5.4509	5	311	15.2	390.50	15.71	21.7

```
names=['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'PRICE']
df18_1 = read_pdf('Housing_data.pdf', pages=[1], pandas_options={'header': None,'names': names[:10]})
```

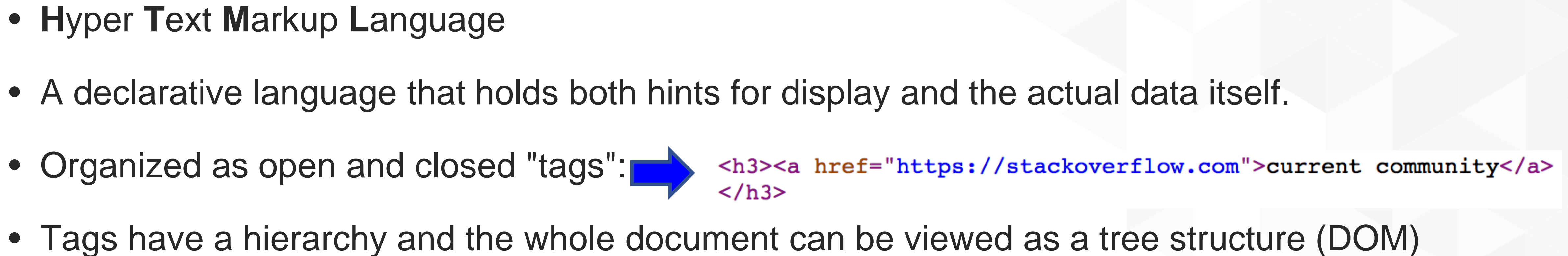
Packt

Introduction to Beautiful Soup 4 and Web Page Parsing

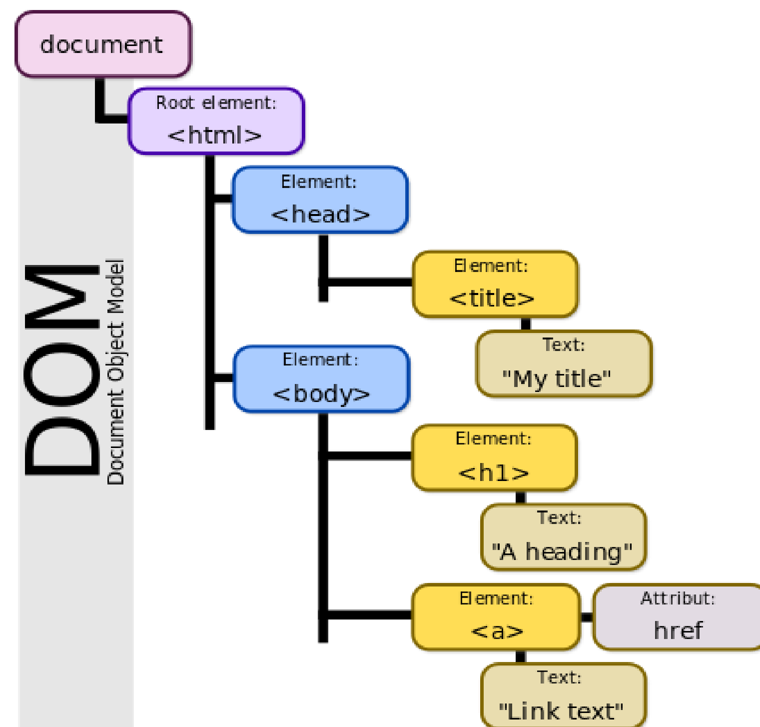
30 mins

Structure of HTML

- **H**yper **T**ext **M**arkup **L**anguage
- A declarative language that holds both hints for display and the actual data itself.
- Organized as open and closed "tags":



- Tags have a hierarchy and the whole document can be viewed as a tree structure (DOM)



What is beautiful soup?

- It’s a very mature Python library for reading and parsing HTML documents.
- It provides a lot of functionalities out of the box.
- It doesn’t include any parser by itself
- According to its website, "It commonly saves programmers hours or days of work."
- It’s often abbreviated as bs4 (the present version being 4)

Exercise 69: Reading an HTML File and Extracting Contents Using BeautifulSoup

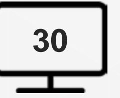
- Import the bs4 library:

```
from bs4 import BeautifulSoup

with open("test.html", "r") as fd:

    soup = BeautifulSoup(fd)

    print(type(soup))
```

- Print the contents of the file in a nice way by using the prettify method from the class like this:

```
print(soup.prettify())

with open("test.html", "r") as fd:

    soup = BeautifulSoup(fd)

    print(soup.p)
```

Exercise 69: Reading an HTML File and Extracting Contents Using BeautifulSoup (continued…)



- We the "find_all" function, provided by bs4 to get the contents of all the same tags (all of the "p"s or all of the "img"s, and so on)
- It returns a list where we can access all the instances of the similar tag in the present document.

```
with open("test.html", "r")as fd:
    soup = BeautifulSoup(fd)
    all_ps = soup.find_all('p')
    print("Total number of <p>  --- {}".format(len(all_ps)))
```

- We will see that the length is 6, which is consistent with the number of <p> tags we had.

Exercise 69: Reading an HTML File and Extracting Contents Using BeautifulSoup (Continued…)

- After getting a tag using "dot" dereferencing, we will use the exact notation to get the "contents" of the tag:

```
with open("test.html", "r")as fd:
    soup = BeautifulSoup(fd)
    table = soup.table
    print(table.contents)
```

Exercise 69: Reading an HTML File and Extracting Contents Using BeautifulSoup (Continued…)

32

- We saw that HTML is structured as a tree, thus we can get all the child nodes of a parent node.
- bs4 gives us an iterator called "children" for each tag, which we can use to achieve extraction of text in child elements:

```
with open("test.html", "r")as fd:
    soup = BeautifulSoup(fd)
    table = soup.table
    for child in table.children:
        print(child)
        print("*****")
```

```
*****
<tbody><tr>
    <th>Entry Header 1</th>
    <th>Entry Header 2</th>
    <th>Entry Header 3</th>
    <th>Entry Header 4</th>
  </tr>
  <tr>
    <td>Entry First Line 1</td>
    <td>Entry First Line 2</td>
    <td>Entry First Line 3</td>
    <td>Entry First Line 4</td>
  </tr>
  <tr>
    <td>Entry Line 1</td>
    <td>Entry Line 2</td>
    <td>Entry Line 3</td>
    <td>Entry Line 4</td>
  </tr>
  <tr>
    <td>Entry Last Line 1</td>
    <td>Entry Last Line 2</td>
    <td>Entry Last Line 3</td>
    <td>Entry Last Line 4</td>
  </tr>
</tbody>
*****
```

Exercise 69: Reading an HTML File and Extracting Contents Using BeautifulSoup (Continued…)

- Children gives only the the immediate children of a tag
- We can use the "descendants" generator to iterate over each of them, maintaining the hierarchy:

```
with open("test.html", "r")as fd:
    soup = BeautifulSoup(fd)
    table = soup.table
    children = table.children
    des = table.descendants
    print(len(list(children)), len(list(des)))
```

- print shows 2 versus 62! This means "children" gave us just the immediate children of table, packing all the contents into it, whereas "descendant" gave all, maintaining the order.

Exercise 70: DataFrames and BeautifulSoup

- Tables in HTML are the closest thing we can get to traditional structured data.
- We need to be able to navigate the structure of HTML well and be able to import data as a pandas DataFrame.
- To do this properly, we need to have knowledge about the structure of the source HTML file.
- We know that the page has only one table, so the following is fine to get the data from all of the rows:

```
data = soup.findAll('tr')
```

- Then, separate the headers and the data:

```
data_without_header = data[1:]

headers = data[0]
```

Exercise 70: DataFrames and BeautifulSoup(contd)

- Prepare the headers for the dataframe:

```
col_headers = [th.getText() for th in headers.findAll('th')]
```

- Prepare the data:

```
df_data = [[td.getText() for td in tr.findAll('td')] for tr in data_without_header]
```

- Create a DataFrame:

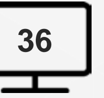
```
df = pd.DataFrame(df_data, columns=col_headers)
```

- Show the "head":

```
df.head()
```

	Entry Header 1	Entry Header 2	Entry Header 3	Entry Header 4
0	Entry First Line 1	Entry First Line 2	Entry First Line 3	Entry First Line 4
1	Entry Line 1	Entry Line 2	Entry Line 3	Entry Line 4
2	Entry Last Line 1	Entry Last Line 2	Entry Last Line 3	Entry Last Line 4

Exercise 71: Exporting a DataFrame as an Excel File



- Install "openpyxl":

```
!pip install openpyxl (from Jupyter Notebook)
```

- Create a writer:

```
writer = pd.ExcelWriter('test_output.xlsx')
```

- Export to Excel:

```
df.to_excel(writer, "Sheet1")
```

- Save the file:

```
writer.save()
```

Exercise 72: Stacking URLs from a Document Using bs4

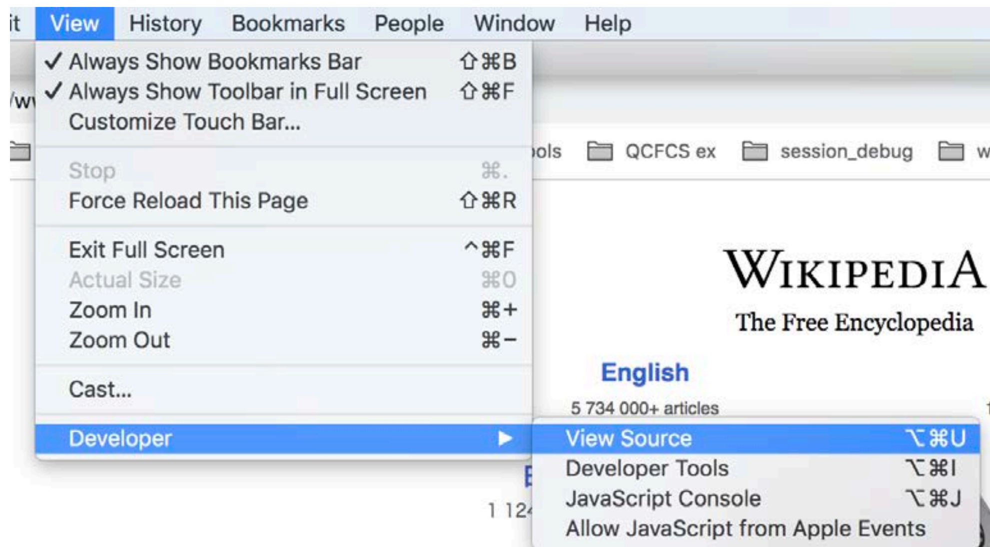
- Using a previously discussed data structure (a stack), we read all the URLs from a document and then we stack them
- We use knowledge about the source structure of the document:

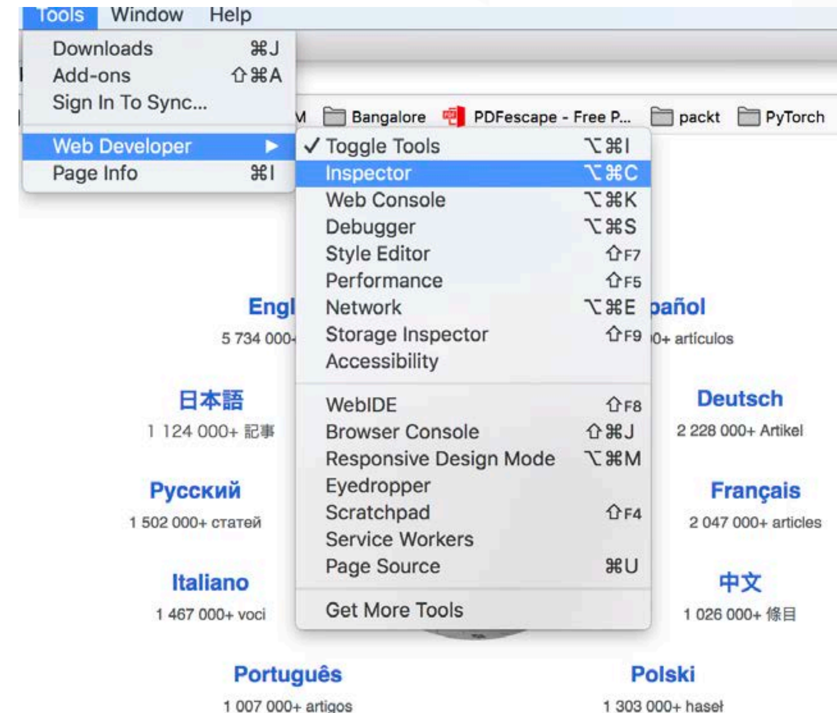
```
lis = soup.find('ul').findAll('li')
stack = []
for li in lis:
    a = li.find('a', href=True)
    stack.append(a)
```

- Here, again, we exploited knowledge about the structure of the document.

Important Takeaway

- Always, always read the source of the HTML very carefully.
- You can use developer tools from Chrome or Firefox to do that.





Activity 7: Reading Tabular Data from a Web Page and Creating DataFrames

You will have to:

- Open the page in a separate Chrome/Firefox tab and use something like an inspect element tool to see the source HTML and understand its structure
- Read the page using bs4 and find the table structure you will need to deal with (how many tables are there?)
- Find the right table using bs4 and separate the source names and their corresponding data
- Get the source names from the list of sources you have created
- Separate the header and data from the data that you separated before for the first source only, and then create a DataFrame using that.
- Repeat the last task for the other two data sources.

Summary

In this lesson, we have:

- Successfully read from CSV, Excel, and JSON files into pandas DataFrames
- Successfully read from PDF documents and HTML tables into pandas DataFrames
- Performed basic web scraping using BeautifulSoup
- Extracted structured and textual information

Practice Questions

Practice Questions

- What kind of Language is HTML? How does it compare with other languages, such as Python?
- What parsers are supported by bs4? How do they compare with each other? Which one will you choose for production use?
- What is the importance of looking into the HTML source of the web page we are scraping?
- Can you look up the definition of "find_all" in the official bs4 documentation and suggest any other ways of getting all the "<p>" tags?
- What is the role of openpyxl?

6
Learning the Hidden Secrets of Data Wrangling

1 hour 15 mins

Lesson Objectives

By the end of this lesson, you will be able to:

- Clean real-life messy data
- Prepare data for data analysis by formatting data in the format required by downstream systems
- Identify and remove outliers from data.

Introduction

- Imagine you have a database of patients who have heart diseases, and like any survey data is missing, incorrect, or has outliers.
- Outliers are values that are abnormal and tend to be far away from the central tendency, and thus including it into your fancy machine learning model may introduce a terrible bias that we need to avoid. Often, these problems can cause a huge difference in terms of money, man-hours, and other organizational resources.
- The code for this exercise depends on two additional libraries, SciPy and python-Levenshtein. To install the libraries, type the following command:

```
!pip install scipy python-Levenshtein
```

Packt

4

Advanced List Comprehension and the zip Function

30 mins

Recap: List Comprehension

- Basic list comprehension:

```
list_1 = [x for x in range(10)]
```

- List comprehension with basic conditionals:

```
list_2 = [x for x in range(10) if x % 2 == 0]
```

Exercise 73: Generator Expressions

- Create list using the following command:

```
odd_numbers2 = [x for x in range(100000) if x % 2 != 0]
```

- Use getsizeof from sys using the following code:

```
from sys import getsizeof

getsizeof(odd_numbers2)
```

- The output is 406496. This takes a lot of memory and time.
- Turn it into generator expression like so:

```
odd_numbers2 = (x for x in range(100000) if x % 2 != 0)
```

- *Notice the change of square brackets "[" to normal ones "("*

Discuss

Can we use if-else instead of just an if as conditional in a list comprehension?

Exercise 74: One-Liner Generator Expression

- Write a generator expression that helps to clean the text:

```
Given - words = ["Hello\n", "My name", "is\n", "Bob", "How are you", "doing\n"]
```

- The generator splits the words using the following command:

```
modified_words = (word.strip().lower() for word in words)
```

- Equivalent explicit for loop:

```
final_list_of_word = [word for word in modified_words]

final_list_of_word
```

Discuss

What will happen if we repeat the list comprehension a second time on final_list_of_word?

Exercise 75: Extracting a List with Single Words

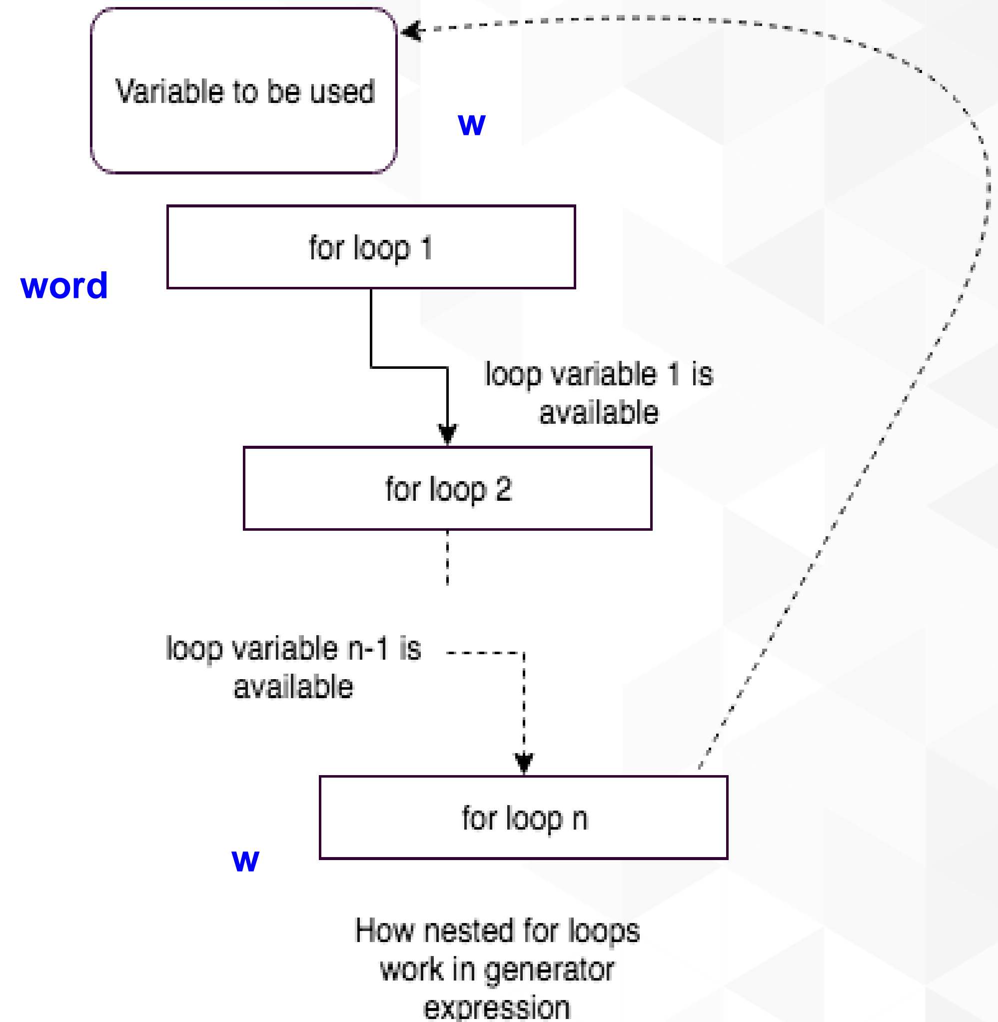
10

- The previous output had multiple words stuck together separated by a space. We can improve on that:

```
modified_words2 = (w.strip().lower() for
word in words for w in word.split(" "))
```

- This is an equivalent explicit for loop:

```
modified_words3 = []
for word in words:
    for w in word.split(" "):

modified_words3.append(w.strip().lower())
```



How nested for loops work in generator expression

Discuss

- Can we use an if statement for nested for loops? How can we convert Bob to lowercase?

Nested Loop Illustration

- Create the following two lists:

```
marbles = ["RED", "BLUE", "GREEN"]
counts = [1, 5, 13]
marble_with_count = ((m, c) for m in marbles for c in counts)
```

- This generator expression creates a tuple in each iteration of the simultaneous for loops:

```
marble_with_count_as_list_2 = []
for m in marbles:
    for c in counts:
        marble_with_count_as_list_2.append((m, c))
marble_with_count_as_list_2
```

Discuss

What changes will you do to the code to change the red color of a marble from 13 to 14?

Exercise 76: The zip Function

- Create the following two lists:

```
countries = ["India", "USA", "France", "UK"]
capitals = ["Delhi", "Washington", "Paris", "London"]
```

- To generate a list of tuples where each tuple is a pair of (country, capital)

```
countries_and_capitals = [t for t in zip(countries, capitals)]
```

Discuss

What data types are allowed for the parameter for the zip function? Can we use sets, tuples, dicts?

Exercise 77: Handling Messy Data

- Create two lists of unequal length:

```
countries = ["India", "USA", "France", "UK", "Brasil", "Japan"]
capitals = ["Delhi", "Washington", "Paris", "London"]
```

- We should not use zip. Use this instead:

```
itertools.zip_longest instead

from itertools import zip_longest
countries_and_capitals_as_dict_2 = dict(zip_longest(countries, capitals))
```

Packt

17

Data Formatting

15 mins

The % operator

- Load the data from the raw_data CSV file.
- Use the % operator to create meaningful statements:

```
for data in raw_data:
    report_str = """%s is %s years old and is %s meter tall weighing about %s kg.\n
    Has a history of family illness: %s.\n
    Presently suffering from a heart disease: %s
    """ % (data["Name"], data["Age"], data["Height"], data["Weight"],
    data["Disease_history"], data["Heart_problem"])
    print(report_str)
```

```
In [19]: raw_data

Out[19]: [{'Name': 'Bob',
           'Age': '23.0',
           'Height': '1.7',
           'Weight': '70',
           'Disease_history': 'N',
           'Heart_problem': 'N'},
          {'Name': 'Alex',
           'Age': '45',
           'Height': '1.61',
           'Weight': '61',
           'Disease_history': 'Y',
           'Heart_problem': 'N'},
          {'Name': 'George',
           'Age': '12.5',
           'Height': '1.4',
           'Weight': '40',
           'Disease_history': 'N',
           'Heart_problem': ''},
          {'Name': 'Alice',
           'Age': '34',
           'Height': '1.56',
           'Weight': '51',
           'Disease_history': 'N',
           'Heart_problem': 'Y'}]
```

The Advanced Approach Using the format Function

You can use named variables inside the string and then use them as key=value in the format statement.

```
for data in raw_data:
    report_str = """{} is {} years old and is {} meter tall weighing about {} kg.\n
    Has a history of family illness: {}.\n
    Presently suffering from a heart disease: {}
    """.format(data["Name"], data["Age"], data["Height"], data["Weight"],
    data["Disease_history"], data["Heart_problem"])
    print(report_str)
```

Exercise 78: Data Representation Using {}

- Change a decimal to binary using format:

```
original_number = 42
print("The binary representation of 42 is - {0:b}".format(original_number))
```

- Printing a string that's center oriented (that is, put 42 spaces before printing):

```
print("{:^42}".format("I am at the center"))
```

- Printing a string that's center oriented, but this time with padding on both sides:

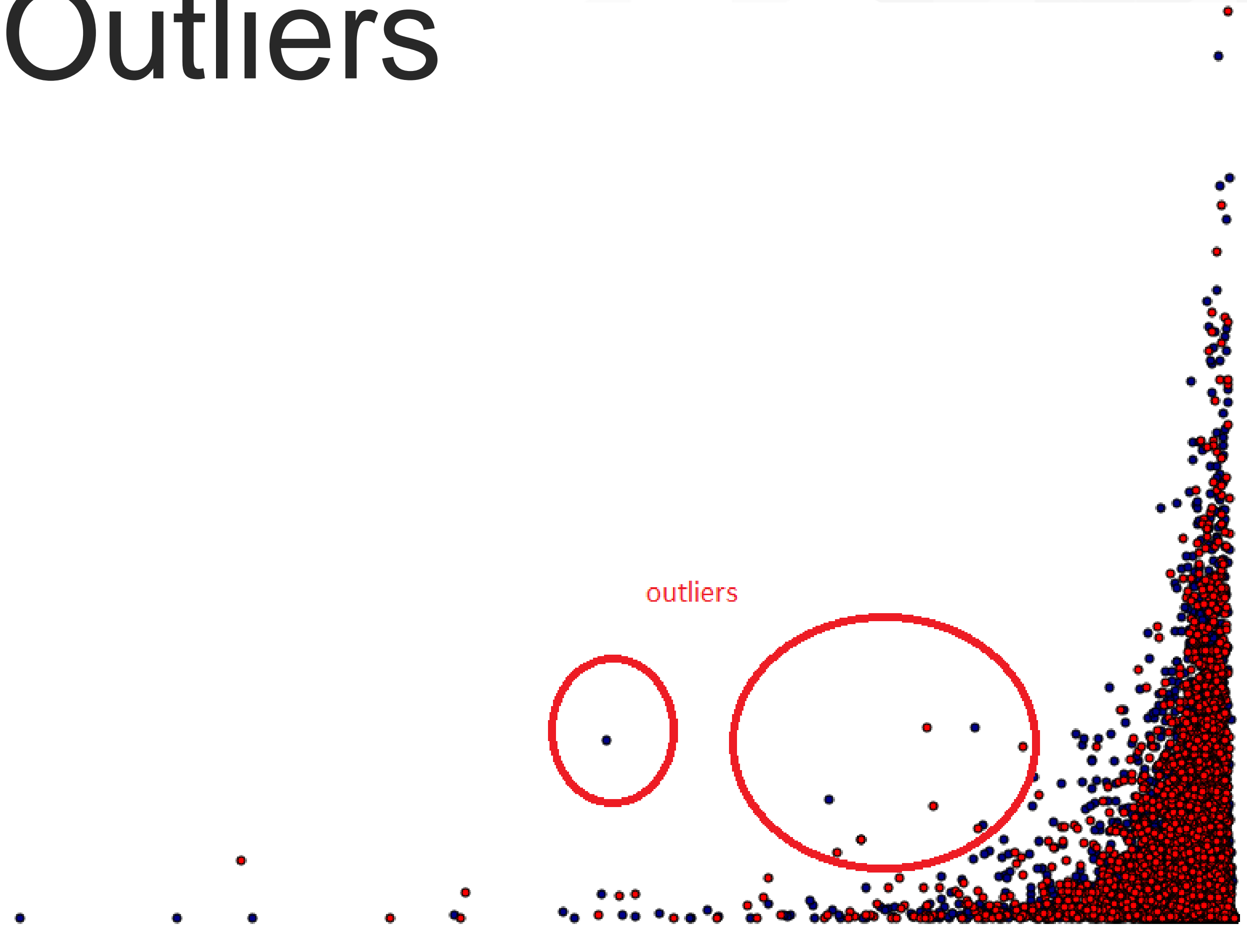
```
print("{:=^42}".format("I am at the center"))
```

- Date formatting is one of the main applications of formatting:

```
from datetime import datetime
print("The present datetime is {:%Y-%m-%d %H:%M:%S}".format(datetime.utcnow()))
```

Identify and Clean Outliers

30 mins



Exercise 79: Outliers in Numerical Data

- Box plots are very useful for detecting if there is an outlier.
- They are visual proof and thus should be a part of any data dashboard you are building.
- To understand outliers, construct a cosine curve and introduce an outlier:

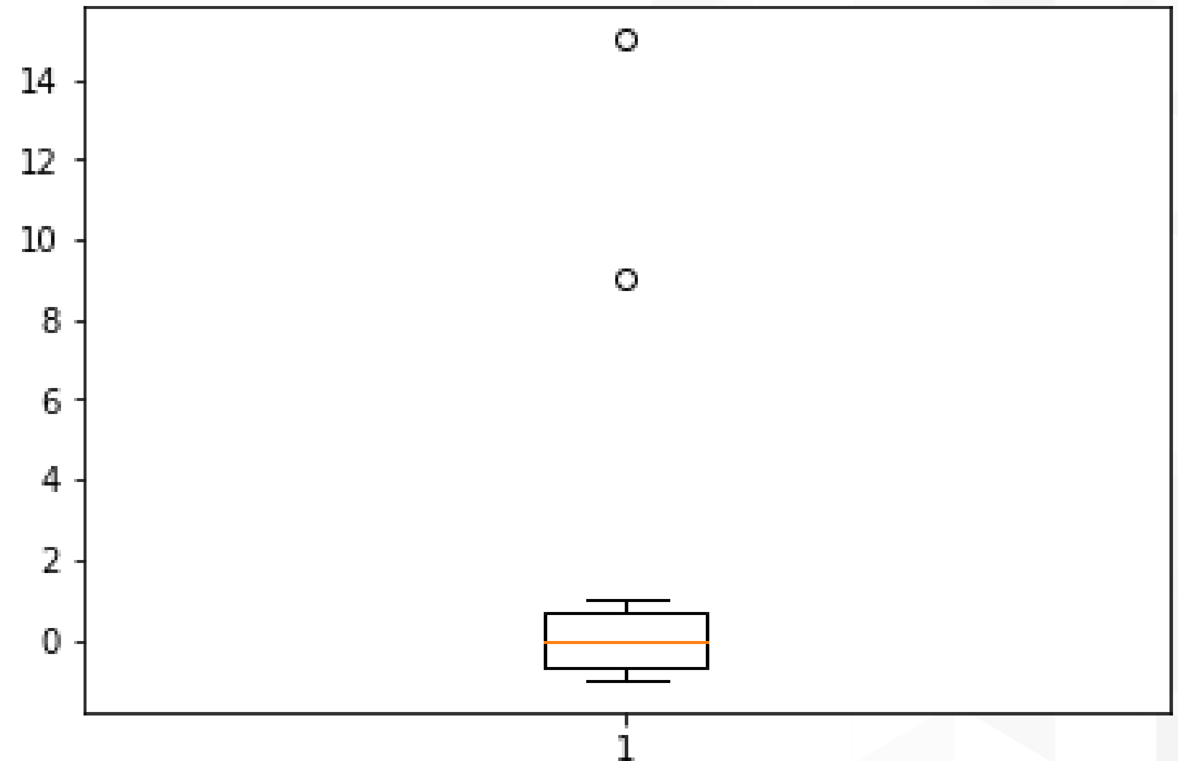
```
from math import cos, pi
ys = [cos(i*(pi/4)) for i in range(50)]
ys[4] = ys[4] + 5.0 # Outliers
ys[20] = ys[20] + 8.0 #Outliers
```

We can see that we have successfully introduced two values in the curve that broke the smoothness and hence can be considered as outliers.

Exercise 79: Outliers in numerical data (contd)

Given that ys represents a sample dataset, we can draw a boxplot as follows:

```
import matplotlib.pyplot as plt

plt.boxplot(ys)
```

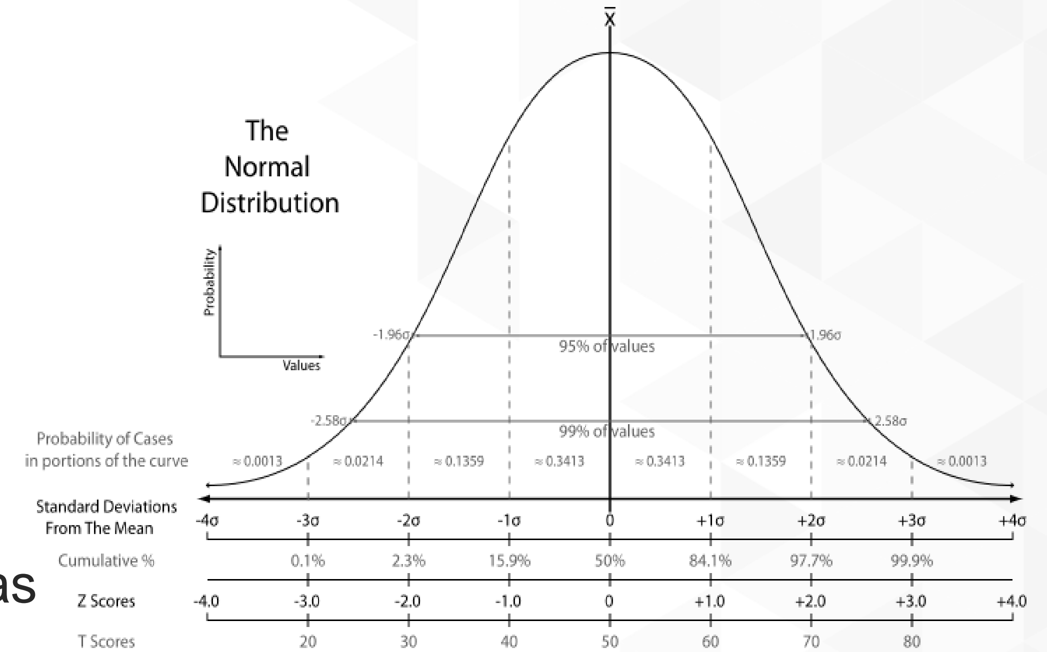


Z-Score

- A Z-Score is a measure on a dataset that gives you a value for each data point of how much that data point is spread out with respect to the standard deviation and mean of the dataset.
- Calculate the Z-Score:

```
from scipy import stats
cos_arr_z_score = stats.zscore(ys)
```

- Anything beyond +3 and -3 are usually considered as outliers.



Exercise 80: The Z-Score Value to Remove Outliers

25

- Import pandas and DataFrames:

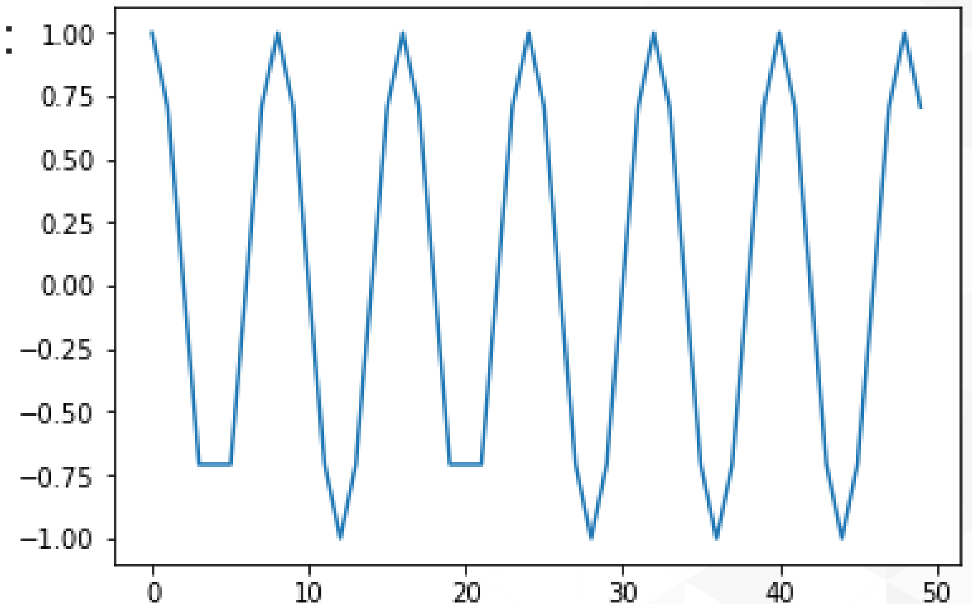
```
import pandas as pd
df_original = pd.DataFrame(ys)
```

- Assign outliers with a Z-Score less than 3:

```
cos_arr_without_outliers = df_original[(cos_arr_z_score < 3)]
```

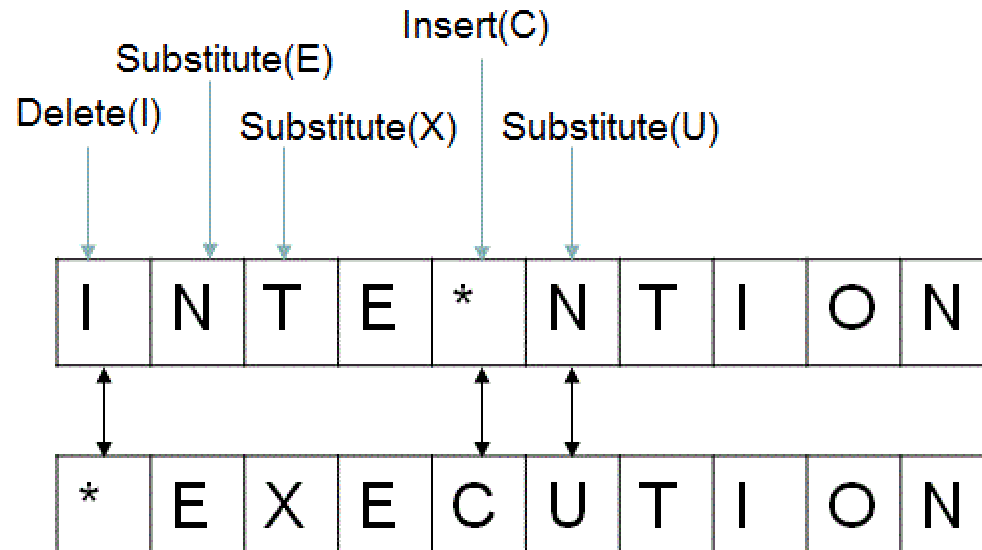
- Use the print function to print the new and old shape:

```
print(cos_arr_without_outliers.shape)
```



Exercise 81: Fuzzy Matching of Strings

Minimum “Edit” distance between two strings



Exercise 81: Fuzzy Matching of Strings (contd)

- Import the Levenshtein module:

```
from Levenshtein import distance
```

- Create two strings:

```
str1 = "Hello"
str2 = "Helol"
```

- Calculate the distance between the two strings:

```
distance(str1, str2)
```

- Consider this data and find the different Sea Princess using fuzzy matching.

```
ship_data = {"Sea Princess": {"date": "12/08/18", "load": 40000},
             "Sea Pincess": {"date": "10/06/18", "load": 30000},
             "Sea Princes": {"date": "12/04/18", "load": 30000},
             }
```

Activity 8: Handling Outliers and Missing Data

The steps that will help you solve this activity are:

1. Read the visit_data.csv file.
2. Check for duplicates
3. Check if any essential column contains NaN.
4. Get rid of the outliers.
5. Report the size difference.
6. Create a box plot visit to further check any outliers.
7. Get rid of any outliers.

Discuss

- What other methods are there for detecting and getting rid of outliers?

Summary

This lesson covered the following:

- Listing data using a generator expression
- Different ways to format data
- Methods to identify and remove outliers
- Rules for identifying outliers

Packt

Practice Questions

Practice Questions

1. How can you find outliers?
2. What methods are permitted for the zip function?
3. Can a list comprehension have if, if-else, and while loop constructs?
4. Are all missing values denoted as NaN?
5. How do you identify outliers without creating graphs?

Packt

1

Advanced Web Scraping and Data Gathering

35 mins

Lesson Objectives

By the end of this lesson, you will be able to:

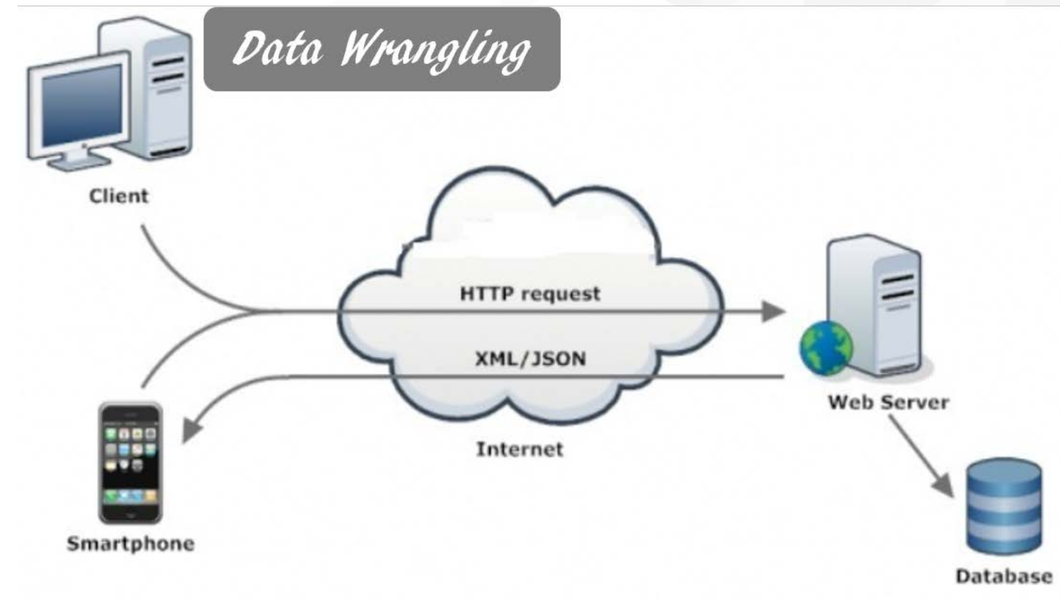
- Make use of requests and BeautifulSoup to read various different web pages and gather data from them
- Perform read operation on XML files and the web using an Application Program Interface (API).
- Make use of regex techniques to scrape useful information from a large and messy text corpus.

Basics of Web Scraping and the Beautiful Soup Library

35 mins

Need to Scrape Data From the web?

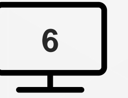
- The most valued and widely used skill for a data wrangling professional is the ability to extract and read data from web pages and databases hosted on the web.
- Organizations are hosting more and more data on the cloud (public or private), and the majority of web microservices these days provide some kind of API for external users to accessing data.
- Therefore, it is absolutely critical for a data wrangling engineer to know the basics about the structure of web pages and how to use Python libraries to scrape the required data.



Libraries Used in This Lesson: **Requests and BeautifulSoup**

- The **Requests** library is an API built on top of pure Python web utility libraries, which makes placing HTTP requests easy and intuitive.
- **BeautifulSoup** parses the HTML content you pass on and builds a detailed tree of all tags and markups within the page for easy and intuitive traversal. This tree can be used by a programmer to look for certain markup elements (for example, a table, or a hyperlink, or a blob of text within a particular div ID) to scrape useful data.

Exercise 81: Using the Requests Library to Get a Response from the Wikipedia Home Page



- Wikipedia is one of the most popular web portals in the whole world. Wouldn't it be cool to read the home page of Wikipedia and extract some useful textual information?
- To do this, we need to peel off the layers of HTML/CSS/JS and extract the selected sections.
- First, assign the home page URL to a variable:

```
wiki_home = https://en.wikipedia.org/wiki/main_page
```

- Then, use the get method from the requests library to get a response from this page:

```
response = requests.get(wiki_home)
```

- What is the response object? It is a model data structure defined in the requests library:

```
type(response)
>> requests.models.Response
```

Exercise 82: Checking the Status of the Web Request

- A web page request generally comes back with various codes.
- Write a function to check the status of the web request:

```
def status_check(r):
    if r.status_code==200:
        print("Success!")
        return 1
    else:
        print("Failed!")
        return -1
```

- Note that, along with printing an appropriate message, we are returning either 1 or -1 from this function.
- It is important to check this value using a conditional statement and then execute the subsequent code based on that.

Code	Description	Code	Description
200	OK	400	Bad Request
201	Created	401	Unauthorized
202	Accepted	403	Forbidden
301	Moved Permanently	404	Not Found
303	See Other	410	Gone
304	Not Modified	500	Internal Server Error
307	Temporary Redirect	503	Service Unavailable

Checking the Encoding of the Web Page

- We can also write a utility function to check the encoding of the web page.
- There are various encodings possible with any HTML document, although the most popular is UTF-8. When we run this function on the Wikipedia home page, we get back this encoding.
- This function, like the previous one, takes the requests response object as an argument and returns a value:

```
def encoding_check(r):
    return (r.encoding)

encoding_check(response)
>> 'UTF-8'
```

Exercise 83: Creating a Function to Decode the Contents of the Response and Check its Length

- The aim is to get a page's contents as a blob of text or as a string object.
- Write a function to decode the contents of the response:

```
def decode_content(r,encoding):
    return (r.content.decode(encoding))

contents = decode_content(response,encoding_check(response))
```

- Check the length of the object and try printing some of it:

```
len(contents)
```

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Wikipedia, the free encyclopedia</title>
<script>document.documentElement.className = document.documentElement.className.replace( /(^|\s)client-nojs(\s|$)/, "
$1client-js$2" );</script>
<script>(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamespace":"","wgCanonicalSpecialPageN
ame":false,"wgNamespaceNumber":0,"wgPageName":"Main_Page","wgTitle":"Main Page","wgCurRevisionId":865422981,"wgRevisi
onId":865422981,"wgArticleId":15580374,"wgIsArticle":true,"wgIsRedirect":false,"wgAction":"view","wgUserName":null,"w
gUserGroups":["*"],"wgCategories":[],"wgBreakFrames":false,"wgPageContentLanguage":"en","wgPageContentModel":"wikitex
t","wgSeparatorTransformTable":["",""],"wgDigitTransformTable":["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":[""
,"January","February","March","April","May","June","July","August","September","October","November","December"],"wgMo
nthNamesShor
```

Exercise 84: Extracting Human-Readable Text from a BeautifulSoup Object

10

- Import the package and then pass on the whole string (HTML content) to a method for parsing:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(contents, 'html.parser')
```

- Execute the following code in your Notebook and you will see similar output:

```
txt_dump=soup.text
Find the type of the txt_dmp
type(txt_dump)
```

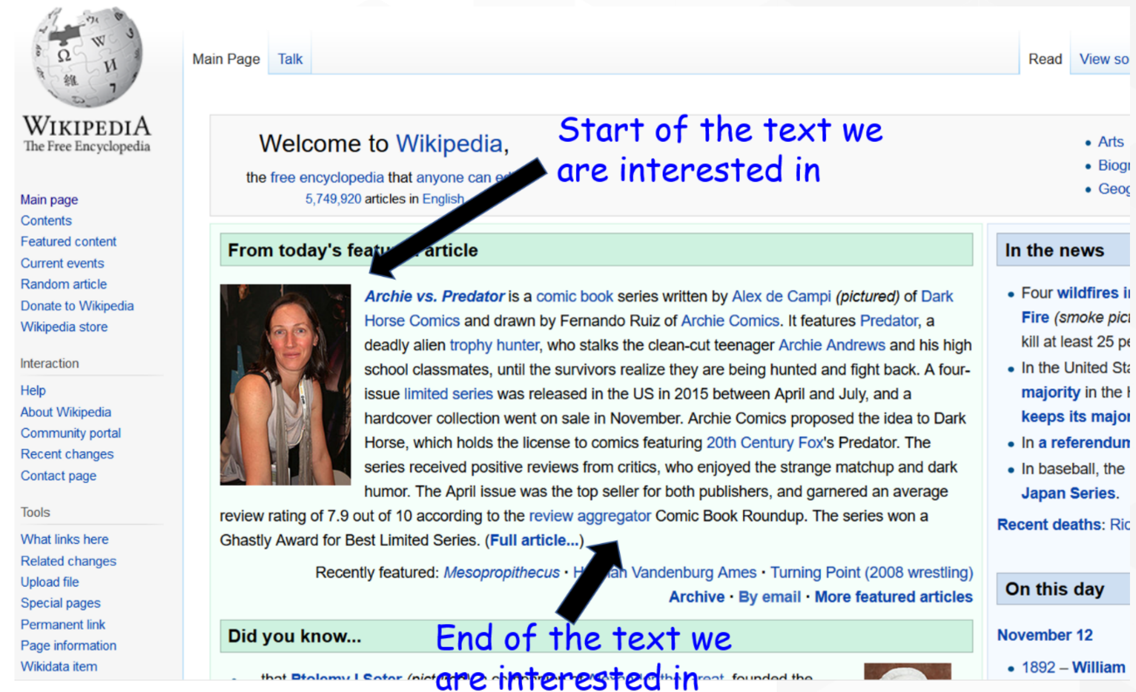
- Find the length of the txt_dmp:

```
len(txt_dump)
```

Extracting Text from a Section

- Identify two indices – the **start** and **end** indices of the string, which demarcate the start and end of the text we are interested in.
- The following code accomplishes the extraction:

```
idx1=txt_dump.find("From today's featured article")
idx2=txt_dump.find("Recently featured")
print(txt_dump[idx1+len("From today's featured article"):idx2])
```



- Note how we add the length of the “From today’s featured article” string to ***idx1*** and then pass that as the starting index.

Archie vs. Predator is a comic book series written by Alex de Campi (pictured) of Dark Horse Comics and drawn by Fernando Ruiz of Archie Comics. It features Predator, a deadly alien trophy hunter, who stalks the clean-cut teenager Archie Andrews and his high school classmates, until the survivors realize they are being hunted and fight back. A four-issue limited series was released in the US in 2015 between April and July, and a hardcover collection went on sale in November. Archie Comics proposed the idea to Dark Horse,

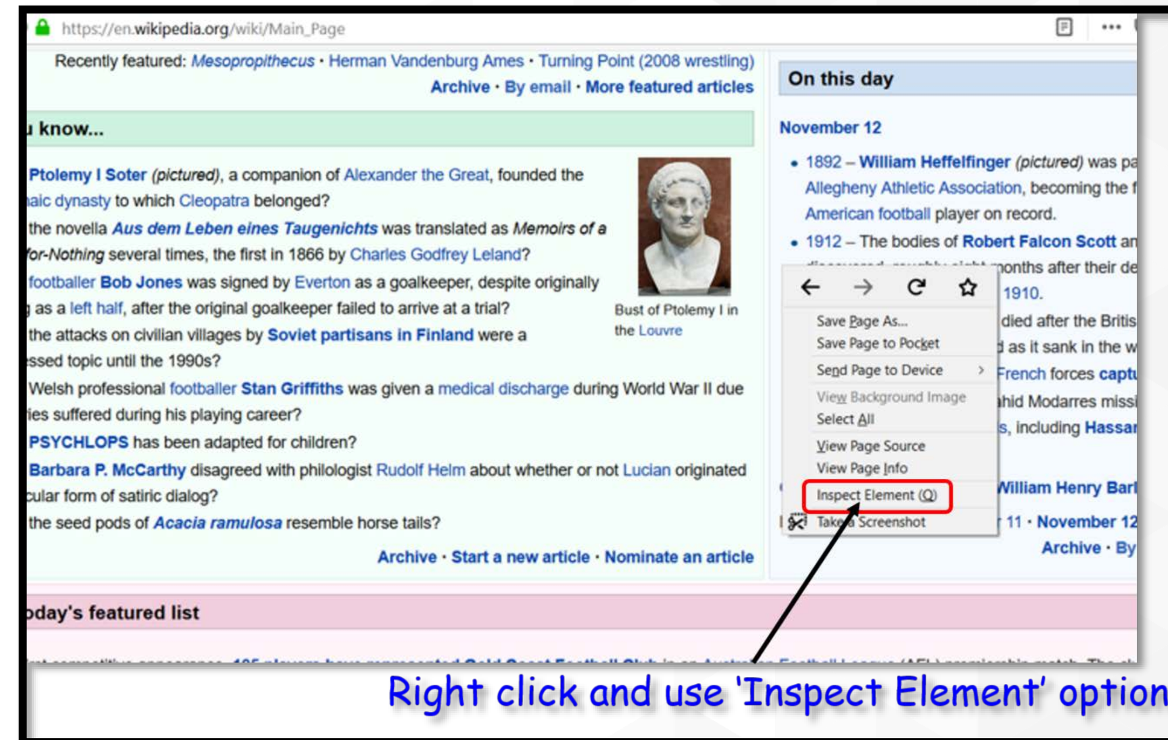
Extracting Important Historical Events That Happened on Today's Date

- We cannot apply the same technique as we did for "***From today's featured article***", because there is text just below where we want our extraction to end, which is not fixed, unlike in the previous case.
- For that, we find out the start of the "***On this day***" string and print the next 1000 characters:

```
idx3=txt_dump.find("On this day")
print(txt_dump[idx3 + len("On this day"):idx3 + len("On this day") + 1000])
```

Exercise 85: Using Advanced bs4 Techniques to Extract Relevant Text without Guessing or Hardcoding

- In Mozilla Firefox, we can easily do this by right-clicking and selecting the '***Inspect Element***' option and using the HTML tags.
- As you hover around with the mouse, you will see different portions of the page being highlighted. By doing this, it is easy to discover the precise block of markup text that is responsible for the textual information we are interested in. Here, we see that a certain <ul> block contains the text.



Exercise 85: Using Advanced BS4 Techniques to Extract Relevant Text without Guessing or Hardcoding (continued…)

14

- Find the <div> tag that contains this <ul> block within it. We find the <div> and also its ID.
- Use the `find_all` method from BeautifulSoup to find and extract the text associated with this particular <div> element.
- Note how we are utilizing the `mp-otd` ID of <div> to identify it among tens of other <div> elements.
- The `find_all` method returns a `NavigableString` class, which has a useful text method associated with it for extraction. Create an empty list and append the text to this list as we traverse the page.

This is the <div> which contains the <ul>. It has the id "mp-otd"

```
<div id="mp-otd" style="padding:0.1em 0.6em 0.5em;">
  <p>…</p>
  <div id="mp-otd-img" style="float:right;margin-left:0.5em;">…</div>
  <ul>…</ul>
  <p>…</p>
  <div style="margin-top: 0.5em;">…</div>
  <div class="otd-footer hlist noprint" style="text-align: right;">
    <ul>…</ul>
  </div>
</div>
```

This is the <ul> block which contains the text

Exercise 85: Using Advanced BS4 Techniques to Extract Relevant Text without Guessing or Hardcoding (continued…)

15

- Here is the code:

```
text_list=[] #Empty list
for d in soup.find_all('div'):
    if (d.get('id') == 'mp-otd'):
            for i in d.find_all('ul'):
                text_list.append(i.text)
```

- Now, if we examine the text_list list, we will see it has three elements. If we print the elements, separated by a marker, we will see that the text we are interested in appears as the first element!

This is the text, we are interested in!

```
1892 – William Heffelfinger (pictured) was paid $500 by the Allegheny Athletic Association, becoming the first profes
sional American football player on record.
1912 – The bodies of Robert Falcon Scott and his companions were discovered, roughly eight months after their deaths
during the ill-fated British Antarctic Expedition 1910.
1928 – At least 110 people died after the British ocean liner SS Vestris was abandoned as it sank in the western Atla
ntic Ocean.
1940 – World War II: Free French forces captured Gabon from Vichy France.
2011 – A blast in Iran's Shahid Modarres missile base led to the death of 17 members of the Revolutionary Guards, inc
luding Hassan Tehrani Moghaddam, a key figure in Iran's missile program.
--
November 11
November 12
November 13
------------------------------------------------------------------------------------------------------
Archive
By email
List of historical anniversaries
------------------------------------------------------------------------------------------------------
```

Exercise 86: Create a Compact Function to Extract the "On this day" Text from the Wikipedia Home Page

- As we discussed before, it is always good to try to functionalize specific tasks, particularly in a web-scraping application.
- We show the code for a function block, whose only job is to take the URL (as a string) and to return the text corresponding to the “On this day” section.
- The benefit of such a functional approach is that you can call this function from any Python script and use it anywhere in another program as a standalone module.
- Create a function that extracts the text from the "On this day" section on the Wikipedia home page.
- Accept the Wikipedia home page URL as a string. A default URL is provided.
- Note how this function utilizes the status check and prints out an error message if the request failed. When we test this function with an intentionally incorrect URL, it behaves as expected.

Reading Data from XML

35 mins

Exercise 87: Creating an XML File and Reading XML Element Objects

- Create an XML file:

```
data = '''
<person>
  <name>Dave</name>
  <surname>Piccardo</surname>
  <phone type="intl">
     +1 742 101 4456
   </phone>
   <email hide="yes">
   dave.p@gmail.com</email>
</person>'‘’
```


```
<person>
  <name>Dave</name>
  <surname>Piccardo</surname>
  <phone type="intl">
     +1 742 101 4456
   </phone>
   <email hide="yes">
   dave.p@gmail.com</email>
</person>
```


- Read it as an Element object using the Python XML parser engine:

```
tree = ET.fromstring(data)

type (tree)
>> xml.etree.ElementTree.Element
```

Exercise 88: Finding Various Elements of Data within a Tree

- We can use the **find** method to search for various pieces of useful data within an XML element object and print them (or use them in whatever processing code we want) using the **text** method. We can also use the **get** method to extract the specific attribute we want:
- For ***Name***:

```
print('Name:', tree.find('name').text)
```

- For ***Surname***:

```
print('Surname:', tree.find('surname').text)
```

- For ***Phone*** (note the use of the **strip** method to strip away any trailing spaces/blanks):

```
print('Phone:', tree.find('phone').text.strip())
```

- For ***email status*** and ***actual email*** (note the use of the **get** method to extract the status):

```
print('Email hidden:', tree.find('email').get('hide'))
print('Email:', tree.find('email').text.strip())
```

Reading from a Local XML File into an ElementTree Object

- We can also read from an XML file (saved locally on disk).
- This is a fairly common situation where a frontend web-scraping module has already downloaded a lot of XML files by reading a table of data on the web and now the data wrangler needs to parse through this XML file to extract meaningful pieces of numerical and textual data:

```
tree2=ET.parse('xml1.xml')
type(tree2)
>> xml.etree.ElementTree.ElementTree
```

- Note how we use the `parse` method to read this XML file. This is slightly different than using the `fromstring` method used in the previous exercise, where we were directly reading from a string object. This produces an `ElementTree` object instead of a simple `Element`.

Exercise 89: Traversing the Tree, Finding the Root, and Exploring all Child Nodes and Their Tags and Attributes

- Every node in the XML tree has tags and attributes. Find the root and explore all of the child nodes and their tags and attributes:

```
root=tree2.getroot()
 for child in root:
    print ("Child:", child.tag, "| Child attribute:", child.attrib)
```

- Still, it is best to examine the raw XML file structure once and understand the data format before attempting automatic extractions.

```
<?xml version="1.0"?>
<data>
    <country1 name="Norway">
        <rank>5</rank>
        <year>2016</year>
        <gdppc>70617</gdppc>
        <neighbor name="Sweden" direction="E"/>
    </country1>
    <country2 name="Austria">
        <rank>16</rank>
        <year>2016</year>
        <gdppc>44857</gdppc>
        <neighbor name="Germany" direction="N"/>
        <neighbor name="Hungary" direction="E"/>
        <neighbor name="Italy" direction="S"/>
        <neighbor name="Switzerland" direction="W"/>
    </country2>
    <country3 name="Israel">
        <rank>23</rank>
        <year>2016</year>
        <gdppc>38788</gdppc>
        <neighbor name="Lebanon" direction="N"/>
        <neighbor name="Syria" direction="NE"/>
    </country3>
</data>
```

Tags

Attribute

```
Child tag: country1 | Child attribute: {'name': 'Norway'}
Child tag: country2 | Child attribute: {'name': 'Austria'}
Child tag: country3 | Child attribute: {'name': 'Israel'}
```

Exercise 90: Using the `text` Method to Extract Meaningful Data

- Access the element root[0][2], we see following:

```
root[0][2]
```

- We can use the text method to access it:

```
root[0][2].text
root[0][2].tag
```

- We looked at root[0][2], but what about root[0] itself?

```
root[0]
 root[0].tag
 root[0].attrib
```

- So, root[0] is again an element but it has a different set of tags and attributes than root[0][2]. **This last piece of code output is interesting because it returns a dictionary object**. Therefore, we can just index it by its keys.

Extracting and Printing the GDP/Per Capita Information Using a Loop

- Now that we know how to read the GDP/per capita data and how to get a dictionary back from the tree, we can easily construct a simple dataset by running a loop over the tree:

```
for c in root:
    for c in root:
    country_name=c.attrib['name']
    gdppc = int(c[2].text)
    print("{}: {}".format(country_name,gdppc))
   Norway: 70617
   Austria: 44857
   Israel: 38788
```

- We can put these in a DataFrame or CSV file for saving back to a local disk or further processing, such as a simple plot!

Exercise 91: Finding All the Neighboring Countries for Each Country and Print Them

- There are efficient search algorithms for tree structures, and one such method for XML trees is **findall**. We can use this to find all the neighbors a country has and print them.
- Why do we need to use **findall** over **find**? Because not all the countries have an equal number of neighbors and **findall** searches for all the data with that tag that is associated with a particular node, and we want to traverse all of them.

```
for c in root:
    ne = c.findall('neighbor') # Find all the neighbors
    print("Neighbors\n" + "-" *25)
    for i in ne:
        print(i.attrib['name'])
        print('\n')
```

```
Neighbors
-------------------------
Sweden


Neighbors
-------------------------
Germany
Hungary
Italy
Switzerland


Neighbors
-------------------------
Lebanon
Syria
```

Exercise 92: A Simple Demo of Using XML Data Obtained by Web Scraping

- Read a website called **receipepuppy.com**, which aggregates links to other sites with the recipe.
- Execute the following code. It will ask the user for input. For example, I entered ‘chicken tikka’:

```
import urllib.request, urllib.parse, urllib.error
serviceurl = 'http://www.recipepuppy.com/api/?'
item = str(input('Enter the name of a food item (enter \'quit\' to quit): '))
url = serviceurl + urllib.parse.urlencode({'q':item})+'&p=1&format=xml'
uh = urllib.request.urlopen(url)
data = uh.read().decode()
print('Retrieved', len(data), 'characters')
tree3 = ET.fromstring(data)
```

```
item = str(input('Enter the name of a food item (enter \'quit\' to quit): '))
url = serviceurl + urllib.parse.urlencode({'q':item})+'&p=1&format=xml'
uh = urllib.request.urlopen(url)

Enter the name of a food item (enter 'quit' to quit): chicken tikka
```

Exercise 92: A Simple Demo of Using XML Data Obtained by Web Scraping (contd)

- We get back data in XML format and decode it before creating an XML tree out of it:

```
data = uh.read().decode()
tree3 = ET.fromstring(data)
```

- Now, we can use another useful method, called **iter**, which basically iterates over the nodes under an element. If we traverse the tree and extract the text, we get the following:

```
for elem in tree3.iter():
    print(elem.text)
```

```
Chicken Tikka Masala
http://allrecipes.com/Recipe/Chicken-Tikka-Masala/Detail.aspx
black pepper, chicken, butter, cayenne, cinnamon, cumin, cumin, garlic, heavy cream, jalapeno,
t, yogurt


Chicken Tikka With Chickpea Flour
http://www.recipezaar.com/Chicken-Tikka-With-Chickpea-Flour-224938
chicken, chickpea flour, chili powder, cumin, garlic, ginger, lemon juice, nutmeg, salt, turme


Chicken Tikka Masala
http://www.recipezaar.com/Chicken-Tikka-Masala-289402
black pepper, chicken, tomato, cayenne, chicken broth, garam masala, garlic, ginger, cardamom,
ns, paprika, yogurt, salt, tomato paste, turmeric, vegetable oil


Chicken Tikka Masala Recipe
http://www.grouprecipes.com/37802/chicken-tikka-masala.html
cumin, garam masala
```

Exercise 92: A simple demo of using XML data obtained by web scraping (contd)

- This is great! But let's say **we want to just extract the top links from this data for scraping in more detail**. How can we do that?
- Now we know what tags to search for. So, we can run the following code to nicely accumulate and print all the hyperlinks in the XML data:

```
for e in tree3.iter():
    h = e.find('href')
    t  =e.find('title')
    if h != None and t != None:
        print("Recipe Link for:",t.text)
        print(h.text)
        print("-" * 100)
```

This is <title> tag

This is <href> tag which stores the hyperlink

```
print(data)

<?xml version="1.0"?>
<recipes>
<recipe>
<title>Chicken Tikka Masala</title>
<href>http://allrecipes.com/Recipe/Chicken-Tikka-Masala/Detail.aspx</href>
<ingredients>black pepper, chicken, butter, cayenne, cinnamon, cumin, cumin
a, salt, salt, yogurt</ingredients>
</recipe>
<recipe>
<title>Chicken Tikka With Chickpea Flour</title>
<href>http://www.recipezaar.com/Chicken-Tikka-With-Chickpea-Flour-224938</h
<ingredients>chicken, chickpea flour, chili powder, cumin, garlic, ginger,
</recipe>
<recipe>
<title>Chicken Tikka Masala</title>
<href>http://www.recipezaar.com/Chicken-Tikka-Masala-289402</href>
```

```
Receipe Link for: Chicken Tikka Masala
http://allrecipes.com/Recipe/Chicken-Tikka-Masala/Detail.aspx
----------------------------------------------------------------------------------------------------
Receipe Link for: Chicken Tikka With Chickpea Flour
http://www.recipezaar.com/Chicken-Tikka-With-Chickpea-Flour-224938
----------------------------------------------------------------------------------------------------
Receipe Link for: Chicken Tikka Masala
http://www.recipezaar.com/Chicken-Tikka-Masala-289402
----------------------------------------------------------------------------------------------------
Receipe Link for: Chicken Tikka Masala Recipe
http://www.grouprecipes.com/37802/chicken-tikka-masala.html
----------------------------------------------------------------------------------------------------
Receipe Link for: Chicken Tikka Masala
http://www.recipezaar.com/Chicken-Tikka-Masala-166811
----------------------------------------------------------------------------------------------------
```

Reading Data from an API

40 mins

What is a web API and can data be read from it?

- Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.
- There are two major types of web services:
  - SOAP (Simple Object Access Protocol), which is an XML-based protocol.
  - REST (Representational State Transfer), which is a particular style of software architecture. Data exchange is generally done using JSON format.
- API stands for **A**pplication **P**rogramming **I**nterface, which is an interfacing protocol used by various software components (within a large system or over a network) to communicate with each other.
- In the context of client-server web architecture and services, naturally, an API serves as the interfacing protocol to read data from the web on the basis of client requests. In general, an API is open source and it can be used by any client that understands JSON or XML formats.

Defining the Base URL (or API Endpoint)

- First we need to set the base URL. When we are dealing with API microservices, this is often called the '**API endpoint**'. Therefore, look for such a phrase in the web service portal you are interested in and use the endpoint URL they give you.
- API-based microservices are extremely dynamic in nature in terms of what and how they offer their service and data. At the time of this lesson planning, we found this particular API a nice choice for extracting data easily and **without using authorization keys** (login or special API keys).
- For most APIs, however, **you need to have your own API key**. You get that by registering with their service. **A basic usage (up to a fixed number of requests or a data flow limit) is often free, but after that you will be charged**. To register for an API key, you often need to enter credit card information.
- We wanted to avoid all that hassle to teach you the basics and that's why chose this example, which does not require such authorization.

Exercise 93: Defining and Testing a Function to Pull Country Data from an API

- Define a function to pull out data when we pass the name of a country as an argument:

```
url = serviceurl + country_name
        uh = urllib.request.urlopen(url)
```

- The first line of code appends the country name as a string to the base URL and the second line sends a get request to the API endpoint. If all goes well, we get back the data, decode it, and read it as a JSON file.
- Test the function by passing some arguments.

Using the Built-in JSON Library to Read and Examine Data

- In this exercise, we will use Python's **json** module to read raw data in that format:

```
x=json.loads(data)
y=x[0]
type(y)

>> dict
```

- So, we get a list back when we use the **loads()** method from the json module. It reads a string datatype into a list of dictionaries. In this case, we get only one element in the list, so we extract that and check its type to make sure it is a dictionary:

```
dict_keys(['name', 'topLevelDomain', 'alpha2Code', 'alpha3Code', 'callingCodes',
n', 'population', 'latlng', 'demonym', 'area', 'gini', 'timezones', 'borders', '
uages', 'translations', 'flag', 'regionalBlocs', 'cioc'])
```

Printing All the Data Elements

- This task is extremely simple given that we have a dictionary at our disposal! All we have to do is to iterate over the dictionary and print the keys/items pair one by one:

```
for k,v in y.items():
    print("{}: {}".format(k,v))
```

```
name: Switzerland
topLevelDomain: ['.ch']
alpha2Code: CH
alpha3Code: CHE
callingCodes: ['41']
capital: Bern
altSpellings: ['CH', 'Swiss Confederation', 'Schweiz', 'Suisse', 'Svizzera', 'Svizra']
region: Europe
subregion: Western Europe
population: 8341600
latlng: [47.0, 8.0]
demonym: Swiss
area: 41284.0
gini: 33.7
timezones: ['UTC+01:00']
borders: ['AUT', 'FRA', 'ITA', 'LIE', 'DEU']
nativeName: Schweiz
numericCode: 756
currencies: [{'code': 'CHF', 'name': 'Swiss franc', 'symbol': 'Fr'}]
languages: [{'iso639_1': 'de', 'iso639_2': 'deu', 'name': 'German', 'nativeName': 'Deutsch'},
ra', 'name': 'French', 'nativeName': 'français'}, {'iso639_1': 'it', 'iso639_2': 'ita', 'name
ano'}]
translations: {'de': 'Schweiz', 'es': 'Suiza', 'fr': 'Suisse', 'ja': 'スイス', 'it': 'Svizzera
'nl': 'Zwitserland', 'hr': 'Švicarska', 'fa': 'سوئیس'}
flag: https://restcountries.eu/data/che.svg
```

Dictionary Values

- Note, **the items in the dictionary are not of the same type.**
- This is fairly common with JSON data. The internal data structure of JSON can be arbitrarily complex and multilevel, that is, you can have a **dictionary of lists of dictionaries of dictionaries of lists of lists…. and so on**.
- We will write a small loop to extract the languages spoken in Switzerland. First, let's examine the dictionary closely and see where the language data is:

This is the key: 'languages'

This is the key of the dictionaries which are in the list: 'name'

```
nativeName: Schweiz
numericCode: 756
currencies: [{'code': 'CHF', 'name': 'Swiss franc', 'symbol': 'Fr'}]
languages: [{'iso639_1': 'de', 'iso639_2': 'deu', 'name': 'German', 'nativeName': 'Deutsch'}, {'iso639_1': 'fr', 'iso639_2': 'f
ra', 'name': 'French', 'nativeName': 'français'}, {'iso639_1': 'it', 'iso639_2': 'ita', 'name': 'Italian', 'nativeName': 'Itali
ano'}]
translations: {'de': 'Schweiz', 'es': 'Suiza', 'fr': 'Suisse', 'ja': 'スイス', 'it': 'Svizzera', 'br': 'Suíça', 'pt': 'Suíça',
'nl': 'Zwitserland', 'hr': 'Švicarska', 'fa': 'سوئیس'}
```

Dictionary Values

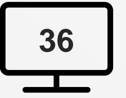
- So, the data is embedded inside a list of dictionaries, which is accessed by a particular key of the main dictionary.
- We can write simple two-line code to extract this data:

```
for lang in y['languages']:
    print(lang['name'])
```

```
for lang in y['languages']:
    print(lang['name'])
```

```
German
French
Italian
```

Using a Function That Extracts a DataFrame Containing Key Info



- Here, we are interested in writing a function that can take a list of countries and return a pandas DataFrame with some key info:
  - Capital
  - Region
  - Sub-region
  - Population
  - Latitude/longitude

This is the kind of wrapper function you are generally expected to write in real-life data wrangling tasks, that is, a utility function that can take a user argument and output a useful data structure (or a mini database type object) with key information extracted over the internet about the item the user is interested in.

Using a Function That Extracts a DataFrame Containing Key Info

- It starts by building an empty dictionary of lists. This is the chosen format for finally passing to the pandas **DataFrame()** method, which can accept such a format and returns a nice DataFrame with column names set to dictionary keys' names.
- We use the previously defined **get_country_data** function to extract data for each country in the user-defined list. For this, we just simply iterate over the list and call this function.
- We check the output of the **get_country_data** function. If, for some reason, it returns a **None** object, we will know that the API reading was not successful and it print out a suitable message. Again, this is an example of an **error-handling mechanism** and you must have them in your code. Without such error-checking code, your application won't be robust enough for the occasional incorrect input or API malfunction!

Exercise 94: Testing the Function by Building a Small Database of Countries' Information

- To test the robustness, we pass in an erroneous name – something funny like '**Turmeric**'!

A wrong entry!

```
df1=build_country_database(['Nigeria','Switzerland','France',
                            'Turmeric','Russia','Kenya','Singapore'])

Retrieved data on Nigeria. Total 1004 characters read.
Retrieved data on Switzerland. Total 1090 characters read.
Retrieved data on France. Total 1047 characters read.
Sorry! Could not retrive anything on Turmeric
Retrieved data on Russia. Total 1120 characters read.
Retrieved data on Kenya. Total 1052 characters read.
Retrieved data on Singapore. Total 1223 characters read.
```

Function catches the error and handles it gracefully!

- Finally, the output is a pandas DataFrame and it is shown here:

Single or multiple pieces of data do not matter. They are extracted correctly

	Area	Capital	Country	Currencies	Gini	Languages	Lattitude	Longitude	Population	Region	Sub-region
0	923768.0	Abuja	Nigeria	Nigerian naira	48.8	English	10.000000	8.0	186988000	Africa	Western Africa
1	41284.0	Bern	Switzerland	Swiss franc	33.7	German,French,Italian	47.000000	8.0	8341600	Europe	Western Europe
2	640679.0	Paris	France	Euro	32.7	French	46.000000	2.0	66710000	Europe	Western Europe
3	17124442.0	Moscow	Russian Federation	Russian ruble	40.1	Russian	60.000000	100.0	146599183	Europe	Eastern Europe
4	580367.0	Nairobi	Kenya	Kenyan shilling	47.7	English,Swahili	1.000000	38.0	47251000	Africa	Eastern Africa
5	710.0	Singapore	Singapore	Brunei dollar,Singapore dollar	48.1	English,Malay,Tamil,Chinese	1.366667	103.8	5535000	Asia	South-Eastern Asia

Fundamentals of Regular Expressions (***RegEx***)

30 mins

What are regular expressions and why do you need to know about them?

- **Reg**ular **ex**pressions or RegEx are used to identify whether a pattern exists in a given sequence of characters (a string) or not.
- They help in manipulating textual data, which is often a prerequisite for data science projects that involve text mining.
- They can **greatly simplify your code** while searching for a complicated pattern match. In theory, you can always avoid using regex using native string methods and complicated logic, but they make the pattern search much simpler and concise.
- It is, therefore, very useful for a data wrangling professional to know about regex and use them for appropriate text manipulation tasks.

Exercise 95: Using the match Method to Check Whether a Pattern Matches a String/Sequence.

- Let's define a string and a pattern:

```
string1 = 'Python'
pattern = r"Python"
```

- Let's write a conditional expression to check for a match:

```
if re.match(pattern,string1):
    print("Matches!")
else:
    print("Doesn't match.")
```

- Now, let's test that with a string that only differs in the first letter by making it lowercase:

```
string2 = 'python'
if re.match(pattern,string2):
    print("Matches!")
else:
    print("Doesn't match.")
>> Doesn't match.
```

Using the compile and use Methods to Create a Regex Program

- In a program or module, if we are making heavy use of a particular pattern, then it is better to use the compile method and create a regex program and then call methods on this program. Here is how you compile a regex program:

```
prog = re.compile(pattern)
prog.match(string1)
>> <_sre.SRE_Match object; span=(0, 6), match='Python'>
```

- This code produced an SRE.Match object that has a span of (0,6) and the matched string of ‘Python’.
- The span here simply denotes the start and end indices of the pattern that was matched. These indices may come handy in a text-mining program where the subsequent code uses the indices for further search or decision-making purposes.

Exercise 96: Compiling Programs to Match Objects


```
#string1 = 'Python'

#string2 = 'python'

#pattern = r"Python"
```

- Here, we match strings by writing a simple conditional:

```
if prog.match(string2)!=None:
    print("Matches!")
else:
    print("Doesn't match.")
>> Doesn't match.
```

Exercise 97: Using Additional Parameters in match to Check for Positional Matching

- By default, **match** looks for pattern matching at the beginning of the given string
- The following example matches ***y*** for the second position (index/pos 1):

```
prog = re.compile(r'y')
prog.match('Python', pos=1)
>> <_sre.SRE_Match object; span=(1, 2), match='y'>
```

- The following example checks for a pattern called 'thon' starting from pos=2:

```
prog = re.compile(r'thon')
prog.match('Python', pos=2)
>> <_sre.SRE_Match object; span=(2, 6), match='thon’>
```

- The following example looks for match in a different string:

```
prog.match('Marathon', pos=4)
<_sre.SRE_Match object; span=(4, 8), match='thon'>
```

Finding the Number of Words in a List That End with 'ing'

- Suppose we want to find out whether a given string has the last three letters ‘***ing***.’
- **What is a possible use for this**? This kind of query may come up in a text analytics/text-mining program where somebody is interested in finding instances of present continuous tense words, which are highly likely to end with ‘***ing***.’ However, other nouns may also have ending with ‘***ing***’ (as we will see in the example):

```
prog = re.compile(r'ing')
words = ['Spring', 'Cycling', 'Ringtone']
for w in words:
    if prog.match(w, pos=len(w)-3) != None:
        print("{} has last three letters 'ing'".format(w))
    else:
        print("{} does not have last three letter as 'ing'".format(w))
```

Exercise 98: The Search Method in Regex

- **Search** and **match** are related concepts and they both return the same **Match** object. The real difference between them is that **match works for only the first match** (either at the beginning of the string or at a specified position, as we saw in the previous exercises), whereas **search looks for the pattern anywhere in the string** and returns the appropriate position if it finds a match. See the following example:

```
prog = re.compile('ing')
if prog.match('Spring') == None:
    print("None")
>> None
 prog.search('Spring')
<_sre.SRE_Match object; span=(3, 6), match='ing'>
prog.search('Ringtone')
<_sre.SRE_Match object; span=(1, 4), match='ing'>
```

Exercise 99: Using the span Method of the Match Object to Locate the Position of the Matched Pattern

- As you will understand by now, the span contained in the Match object, is useful for locating the exact position of the pattern as it appears in the string:

```
prog = re.compile(r'ing')
words = ['Spring', 'Cycling', 'Ringtone']
 for w in words:
    mt = prog.search(w)
    start_pos = mt.span()[0] # Starting position of the match
    end_pos = mt.span()[1] # Ending position of the match
    print("The word '{}' contains 'ing' in the position {}-{}".format(w, start_pos,
    end_pos))
>> The word 'Spring' contains 'ing' in the position 3-6
>> The word 'Cycling' contains 'ing' in the position 4-7
>> The word 'Ringtone' contains 'ing' in the position 1-4
```

Exercise 100: Examples of Single Character Pattern Matching with Search

Now, we will start getting into the real usage of regex with examples of various useful pattern matching.

- **Dot (.) matches any single character except a newline character:**

```
prog = re.compile(r'py.')
print(prog.search('pygmy').group())
print(prog.search('Jupyter').group())
>> pyg
>> pyt
```

- **\w (lowercase w) matches any single letter, digit, or underscore:**

```
prog = re.compile(r'c\wm')
print(prog.search('comedy').group())
print(prog.search('camera').group())
print(prog.search('pac_man').group())
print(prog.search('pac2man').group())
>> com
>> cam
>> c_m
>> c2m
```

Exercise 100: Examples of Single-Character Pattern Matching with **search**

- **\W (uppercase W) matches anything not covered with \w:**

```
prog = re.compile(r'9\W11')
print(prog.search('9/11 was a terrible day!').group())
print(prog.search('9-11 was a terrible day!').group())
>> 9/11
>> 9-11
```

- **\s (lowercase s) matches a single whitespace character, such as space, newline, tab, or return:**

```
prog = re.compile(r'Data\swrangling')
print(prog.search("Data wrangling is cool").group())
print("Data\twrangling is the full string")
print(prog.search("Data\twrangling is the full string").group())
>> Data wrangling
Data    wrangling is the full string
Data    wrangling
```

Exercise 101: Examples of Pattern Matching at the Start or End of the String

- First, let's write a small function to handle cases where a match is not found:

```
def print_match(s):
    if prog.search(s)==None:
        print("No match")
    else:
 print(prog.search(s).group())
```

^ (Caret) matches a pattern at the start of the string:

```
prog = re.compile(r'^India')
print_match("Russia implemented this law")
print_match("India implemented that law")
print_match("This law was implemented by India")
>> No match
>> India
>> No match
```

Exercise 101: Examples of Pattern Matching at the Start or End of the String (contd)

- $ (dollar sign) matches a pattern at the end of the string:

```
prog = re.compile(r'Apple$')
print_match("Patent no 123456 belongs to Apple")
print_match("Patent no 345672 belongs to Samsung")
>> Apple
>> No match
```

- * matches 0 or more repetitions of the preceding RE:

```
prog = re.compile(r'ab*')
print_match("a")
print_match("ab")
print_match("abbb")
print_match("b")
>> a
>> ab
>> abbb
>> No match
```

Exercise 102: Examples of Pattern Matching with Multiple Characters

- **+ causes the resulting RE to match 1 or more repetitions of the preceding RE:**

```
prog = re.compile(r'ab+')
print_match("a")
print_match("ab")
print_match("abbb")
>> No match
>> ab
>> abbb
```

- **? causes the resulting RE to match precisely 0 or 1 repetitions of the preceding RE:**

```
prog = re.compile(r'ab?')
print_match("a")
print_match("ab")
print_match("abbb")
>> a
>> ab
>> ab
```

Exercise 103: Greedy versus Non-Greedy Matching

- The standard (default) mode of pattern matching in regex is greedy, that is, the program tries to match as much as it can. Sometimes, this behavior is natural, but in some cases, you may want to match minimally. We'll show examples of such cases in the following code to illustrate the point:

```
prog = re.compile(r'<.*>')
print_match('<a> b <c>')
>> <a> b <c>
```

- So, the preceding regex found both tags with <> pattern, but what if we wanted to match the first tag only and stop there? We can use '?' inserted after any regex expression to make it non-greedy:

```
prog = re.compile(r'<.*?>')
print_match('<a> b <c>')
>> <a>
```

Exercise 104: Controlling how many Repetitions to Match

- {m} specifies exactly *m* copies of RE to match. Fewer matches cause a non-match and returns None:

```
prog = re.compile(r'A{3}')

print_match("ccAAAdd")
print_match("ccAAAAdd")
print_match("ccAAdd")

>> AAA
>> AAA
>> No match
```

Exercise 104: Controlling how many Repetitions to Match(contd)

- {m,n} specifies exactly *m* to *n* copies of RE to match:

```
prog = re.compile(r'A{2,4}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")

>> AAAB
>> No match
>> AAB
>> AAAAB
```

Exercise 104: Controlling how many Repetitions to Match(contd)

- **Omitting *m* specifies a lower bound of zero:**

```
prog = re.compile(r'A{,3}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")

>> AAAB
>> AB
>> AAB
>> AAAB
```

Exercise 104: Controlling how many Repetitions to Match (contd)

- **Omitting *n* specifies an infinite upper bound:**

```
prog = re.compile(r'A{3,}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")

>> AAAB
>> No match
>> No match
>> AAAAAAAB
```

Exercise 104: Controlling how many Repetitions to Match (contd)

- {m,n}? specifies *m* to *n* copies of RE to match in a non-greedy fashion:

```
prog = re.compile(r'A{2,4}')
print_match("AAAAAAA")

prog = re.compile(r'A{2,4}?')
print_match("AAAAAAA")

>> AAAA
>> AA
```

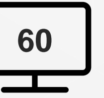
Exercise 105: Sets of Matching Characters

- To match an arbitrarily complex pattern, we need to be able to include a logical combination of characters together as a bunch. Regex gives us that kind of capability:

- **[xyz] matches x, y, or z:**

```
prog = re.compile(r'[AB]')
print_match("ccAd")
print_match("ccABd")
print_match("ccXdB")
print_match("ccXdZ")

>> A
>> A
>> B
>> No match
```

Exercise 105: Sets of Matching Characters(contd)



- **A range of characters can be matched inside the set using -. This is one of the most widely used regex techniques!**
- Suppose, we want to pick out an email address from a text. Email address are generally of the form **<some name>@<some domain name>.<some domain identifier>:**

```
prog = re.compile(r'[a-zA-Z]+@+[a-zA-Z]+\.com')

print_match("My email is coolguy@xyz.com")
print_match("My email is coolguy12@xyz.com")

>> coolguy@xyz.com
>> No match
```

Exercise 105: Sets of Matching Characters(contd)

- What happened with the second email ID?
- The regex could not capture it because it had the number ‘12’ in the name! That pattern is not captured by the expression [a-zA-Z].
- Let’s change that and add the digits as well:

```
prog = re.compile(r'[a-zA-Z0-9]+@+[a-zA-Z]+\.com')

print_match("My email is coolguy12@xyz.com")
print_match("My email is coolguy12@xyz.org")

>> coolguy12@xyz.com
>> No match
```

Exercise 105: Sets of Matching Characters(contd)

- Now we catch the first email ID perfectly. But what's going on with the second one? We again got a mismatch. The reason is that we changed the **.com** to **.org** in that email, and in our regex expression, that portion was hardcoded as .com, so it did not find a match.
- Let's try to address this in the following regex:

```
prog = re.compile(r'[a-zA-Z0-9]+@+[a-zA-Z]+\.+[a-zA-Z]{2,3}')
print_match("My email is coolguy12@xyz.org")
print_match("My email is coolguy12[AT]xyz[DOT]org")

>> coolguy12@xyz.org
>> No match
```

- In this regex, we used the fact that most domain identifiers have 2 or 3 characters, so we used **[a-zA-Z]{2,3}** to capture that.

Exercise 106: The use of OR in regex using |

- Because regex patterns are like complex and compact logical constructors themselves, it makes perfect sense that we want to combine them to construct even more complex programs when needed.
- We can do that using the | operator. The following example demonstrates the point:

```
prog = re.compile(r'[0-9]{10}')

print_match("3124567897")
print_match("312-456-7897")

>> 3124567897
>> No match
```

Exercise 106: The use of OR in regex using | (contd)

- Here, we are trying to extract patterns of 10-digit number that could be phone numbers. Note the use of **{10}** to denote exactly 10-digit numbers in the pattern. But the second number could not be matched for obvious reasons – it had '**-**' symbols inserted in between groups of numbers. We can tackle this by using multiple smaller regex and logically combining them:

```
prog = re.compile(r'[0-9]{10}|[0-9]{3}-[0-9]{3}-[0-9]{4}')

print_match("3124567897")
print_match("312-456-7897")

>> 3124567897
>> 312-456-7897
```

Exercise 106: The use of OR in regex using | (contd)

- Phone numbers are written in a myriad of ways and if you search on the web, you will see examples of very complex regex for capturing phone numbers.

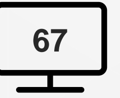
```
p1= r'[0-9]{10}'
p2=r'[0-9]{3}-[0-9]{3}-[0-9]{4}'
p3 = r'\([0-9]{3}\)[0-9]{3}-[0-9]{4}'
p4 = r'[0-9]{3}\.[0-9]{3}\.[0-9]{4}'
pattern= p1 + '|' + p2 + '|' + p3 + '|' +p4
prog = re.compile(pattern)

print_match("3124567897")
print_match("312-456-7897")
print_match("(312)456-7897")
print_match("312.456.7897")

>> 3124567897
>> 312-456-7897
>> (312)456-7897
>> 312.456.7897
```

The **findall** method

- The last regex method that we will learn for lesson is **findall**.
- Essentially, it is a **search-and aggregate** method, that is, it puts all the instances that match with the regex pattern in a given text and returns them in a list. This is extremely useful, as we can just count the length of the returned list to count the number of occurrences or pick and use the returned pattern-matched words one by one as we see fit:

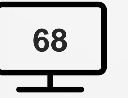
```
ph_numbers = """Here are some phone numbers.
Pick out the numbers with 312 area code: 312-423-3456, 456-334-6721,
312-5478-9999, 312-Not-a-Number,777.345.2317, 312.331.6789"""

re.findall('312+[-\.][0-9-\.]+',ph_numbers)

>> ['312-423-3456', '312-5478-9999', '312.331.6789']
```

Activity 9: Extracting the Top 10 eBooks from Gutenberg.org



- This activity aims to scrape the URL of Project Gutenberg's Top 100 eBooks (yesterday's ranking) to identify the eBooks' links.
- Import the necessary libraries, including regex and BeautifulSoup, and check the SSL certificate.
- Read the HTML from the URL and write a small function to check the status of the web request.
- Decode the response and pass this on to BeautifulSoup for HTML parsing, and find all the href tags and store them in the list of links. Check what the list looks like – print the first 30 elements.
- Use a regular expression to find the numeric digits in these links. Use the findall method.
- Use the .text method and print only the first 2,000 characters. Search in the extracted text from the soup object to find the names of the top 100 eBooks.
- Create a starting index. It should point at the text *Top 100 Ebooks yesterday*. Use the splitlines method of soup.text. It splits the lines of text of the soup object.
- Loop 1-100 to add the strings of the next 100 lines to this temporary list. Use regular expression to extract only the text from the name strings and append them to an empty list. Use match and span to find indices and use them.

Activity 10: Building Your Own Movie Database by Reading from an API



The aims of this activity are as follows:

- To retrieve and print basic data about a movie (the title is entered by the user) from the web (OMDb database)
- If a poster of the movie can be found, it downloads the file and saves it a user-specified location

Summary

- We started by reading data from a web page using two of the most popular Python libraries – requests and BeautifulSoup.
- We extracted meaningful data from the Wikipedia home page during this process.
- We learned how to read data from XML and JSON files, two of the most widely used data streaming/exchange formats on the web.
- For the XML part, we showed how to traverse a tree-structure data string efficiently to extract key information. For the JSON part, we read data from the web using an API (Application Program Interface).
- We went through a detailed exercise of using regex techniques in tricky string-matching problems to scrape useful information from a large and messy text corpus, parsed from HTML.

Practice Questions

Practice Questions

1. True or false? Python does not have a built-in library for executing HTTP requests and that's why you have to always use a third-party library such as Requests.
2. For a web-scraping task, why do you need to check the status code of the response first?
3. If we just extract the raw contents of a web page using the decode method of the requests library, we do not get human-readable text but all the raw HTML formatted content. How did we solve this issue in this lesson?

8
RDBMS And SQL

35 mins

Lesson Objectives & Introduction

By the end of this lesson, you will be able to:

- Apply the basics of RDBMS to query databases in Python
- Convert data from SQL into a Python DataFrame
- RDBMS stands for **R**elational **D**ata**b**ase **M**anagement **S**ystem.

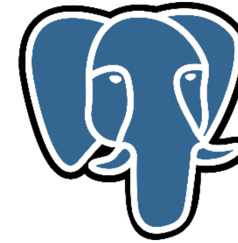
- DBMS is the most popular way to store and retrieve structured data.
- They store data as tables. Columns of those tables represent different data headings. Each row represents one data tuple.
- The main power comes from establishing relationships among tables.
- SQL (pronounced "sequel") stands for **S**tructured **Q**uery **L**anguage.SQL is a declarative and easy-to-use language.

Refresher of RDBMS and SQL

10 mins

Examples of RDBMS

We can divide RDBMSes into two categories:



Open source:

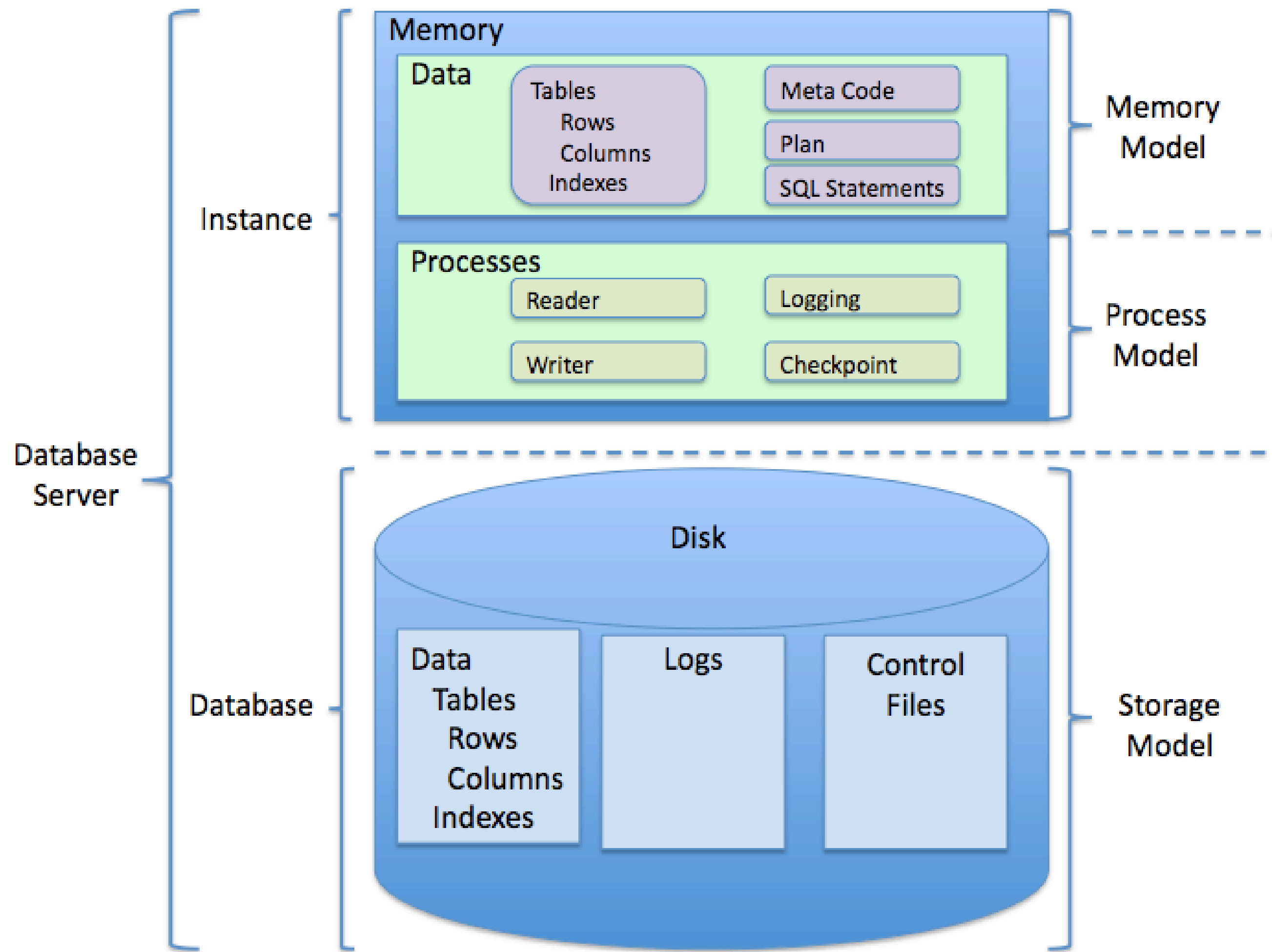
- MySQL
- PostgreSQL
- SQLite

Closed source (Enterprise)

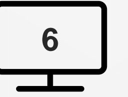
- Oracle
- IBM DB2
- Microsoft SQL Server

We are going to use SQLite.

Structure of an RDBMS



Storage Engine and Log Management

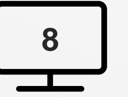


- Responsible for efficiently storing and retrieving data on request.
- Each database has one (or more) storage engines built into them, and some are suitable for some special purposes.
- An application developer never interacts with it directly.

Log management:

- Creates and maintains logs of every action and their results
- Important as it can be used as a “source of truth” for any kind of recovery process
- Also important (with the augmentation of **W**rite **A**head **L**og or **WAL**) for replication and partitions (PostgreSQL, as an example, uses this method)

Query Engine



- Middle layer between the storage engine and the end user.
- Accepts an SQL query and then efficiently parses it.
- Creates an “Execution Plan”.
- Executes the query by talking with storage engine and returns the results in a human readable form.
- As an application developer, we interact with it either by writing a direct SQL query or via the database driver and library provided by the language of choice.

Discuss: What Are the Different Types of Databases?

NoSQL databases

- Mongo
- CouchDB
- RocksDB
- Cassandra
- And many more…

Graph Database

- Neo4J
- OrientDB

SQL Components – 1

DDL:

- Stands for Data Definition Language.
- We define what the structure of our data looks like.
- This is where we define the tables.
- We define the names of the columns

and also their data types.

- Example –

```
CREATE TABLE tab1 (id int, name varchar(255))
```

- It can drop tables or alter tables.

```
mysql> SELECT * FROM student
    -> LIMIT 5;
+--------+---------+---------+--------+
| stu_id | name    | city    | pin    |
+--------+---------+---------+--------+
|      1 | Raj     | Ranchi  | 123456 |
|      2 | Sona    | Delhi   | 652345 |
|      3 | Anu     | Kolkata | 879845 |
|      4 | Sonu    | NULL    | 764839 |
|      5 | Shubham | HYD     | 542367 |
+--------+---------+---------+--------+
5 rows in set (0.00 sec)

mysql> SELECT * FROM student
```

SQL Components – 2

DML:

- Stands for Data Manipulation Language.
- We actually store data using DML commands.
- Works on one (or more) rows of an already defined table.
- Examples are INSERT INTO, UPDATE, and DELETE FROM.

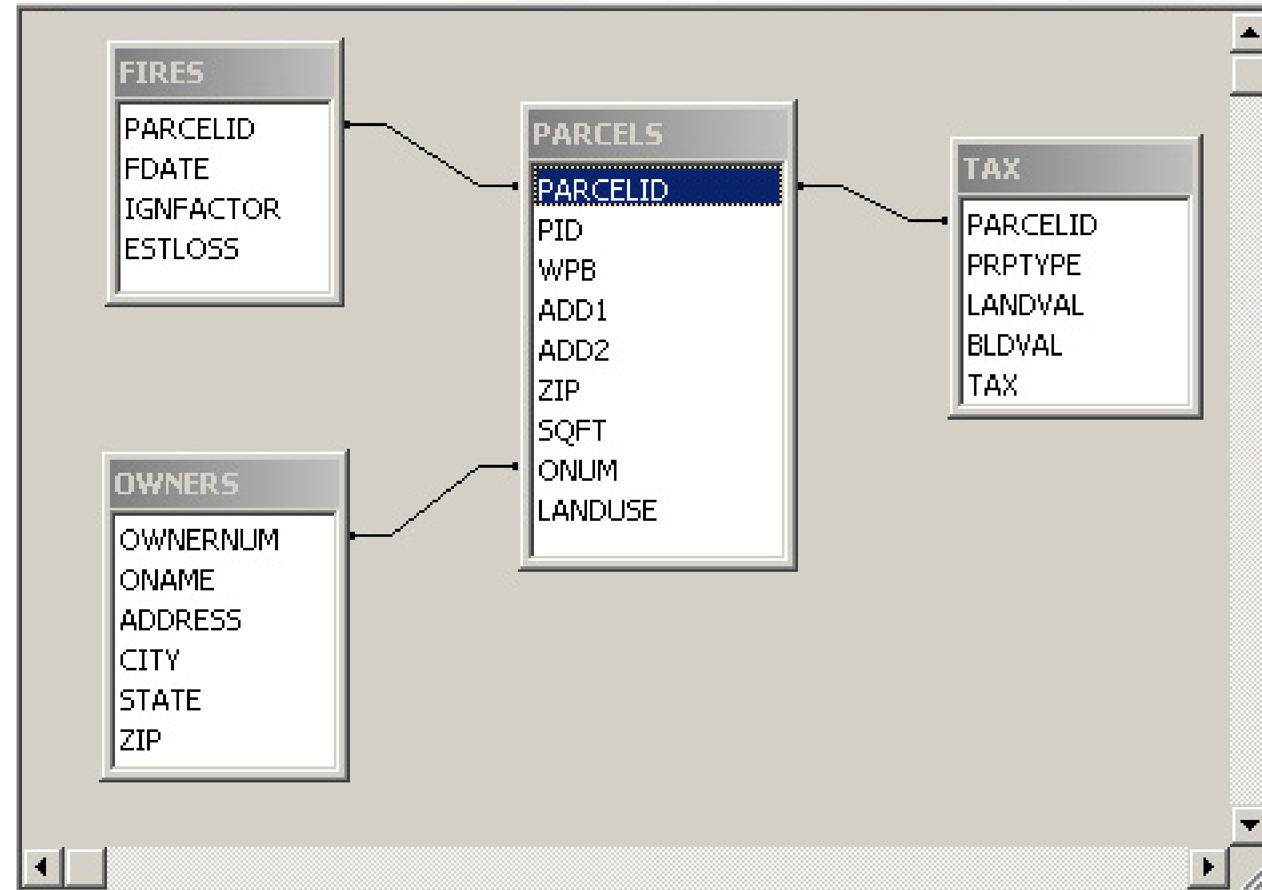
SQL Components – 3

DQL:

- Stands for Data Query Language
- Lets us view the already present data
- It can get data from a single table or from multiple tables using relationships between them.
- We just tell it what data to get, but not how to get it. The query engine figures it out.
- An frequently used command is SELECT.
- SELECT is often accompanied with WHERE, GROUP BY, and more.

Relationship

- A table usually has one unique identifier.
- The ID can be an integer, or any other type.
- It is often declared as the primary key.
- Another table can have several rows that each have a reference of a primary key from first table.
- These keys are called foreign keys, and they relate the second table to the first table.



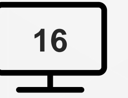
Discuss

- Is there any other way we can interact with an RDBMS other than using SQL?

Discuss

If you create a view in RDBMS from a table, what happens to it when you delete the table?

Exercise 107: Connecting to a Database in SQLite



- Use the **connect** function to connect to a database
- The main database engine of SQLite is embedded:

```
import sqlite3

with sqlite3.connect("lesson.db") as conn:
    pass
```

Discuss

- What will happen if you do not close a database connection properly?

Exercise 108: DDL and DML Commands in SQLite

- Creating a table:

```
with sqlite3.connect("lesson.db") as conn:
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS user (email text, first_name 
text, last_name text, address text, age integer, PRIMARY KEY (email))")
```

- Inserting data:

```
cursor.execute("INSERT INTO user VALUES ('bob@example.com', 'Bob', 'Codd', 
'123 Fantasy lane, Fantasy City', 31)")
```

Discuss

What will happen if you do not close a database connection properly?

Reading Data from a Database in SQLite

- The **SELECT** clause is immensely powerful, and it is really important for a data practitioner to master **SELECT** and everything related to it (such as conditions, joins, group-by, and so on).
- The * after **SELECT** tells the engine to select all of the columns from the table. It is a useful shorthand.

Discuss

- Imagine a table that has 100 rows. How will you select the last 50 rows?

Exercise 109: Sorting Values That Are Present in the Database

- We can sort a collection of rows based on one (or more columns) when we query.
- The results will be returned in the sorted order:

```
rows = cursor.execute('SELECT * FROM user ORDER BY age DESC')
```

- The default order is ASCENDING.
- To sort in descending order, we add DESC at the end of the ORDER BY clause. Sorting is performed one after another from left to right.
- We can sort by as many columns as we want. Each ORDER BY must be separated by a space.
- We can sort on columns with numerical or alphanumeric data types.

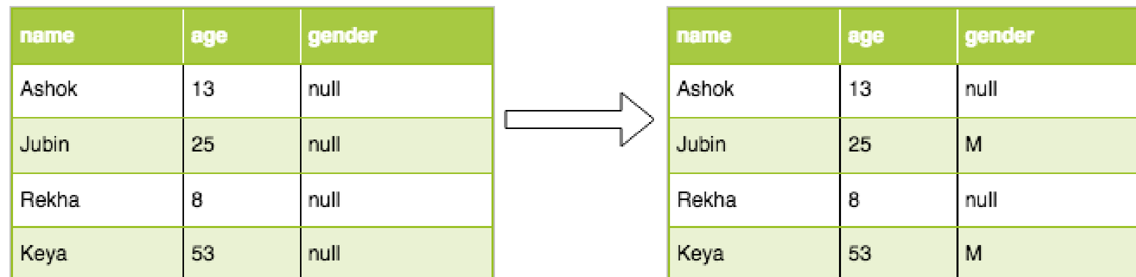
Exercise 110: Altering the Structure of Table and Updating New Fields

- We update (edit) the original table definition using ALTER TABLE:

```
cursor.execute("ALTER TABLE user ADD COLUMN gender text")
```

- It adds the “gender” column and fills it up with “null” values.
- Use this to update one (or multiple) rows of a table.
- Usually it is never used without a WHERE clause, as it will update all the rows of a table, which is seldom the desired behavior:

```
cursor.execute("UPDATE user SET gender='M' WHERE age>20")
```



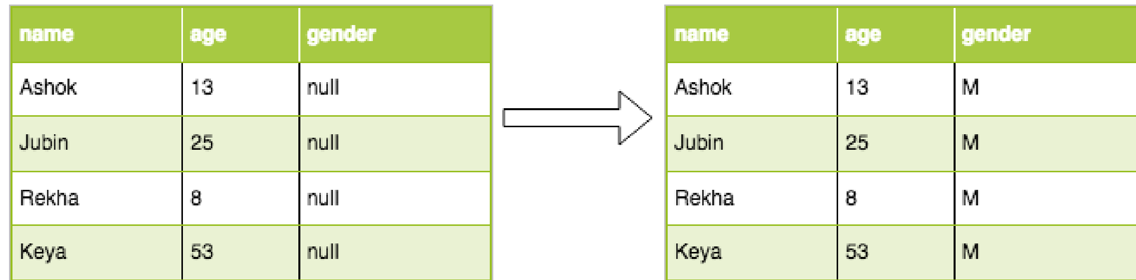
name	age	gender
Ashok	13	null
Jubin	25	null
Rekha	8	null
Keya	53	null

name	age	gender
Ashok	13	null
Jubin	25	M
Rekha	8	null
Keya	53	M

Exercise 110: Altering the Structure of Table and Updating New Fields


```
cursor.execute("UPDATE user SET gender='M'")
```

- The second statement updates the whole table.



name	age	gender
Ashok	13	null
Jubin	25	null
Rekha	8	null
Keya	53	null

name	age	gender
Ashok	13	M
Jubin	25	M
Rekha	8	M
Keya	53	M

Exercise 111: Grouping Values in Tables

- Creating buckets or groups based on unique values in a column.
- It’s conceptually same as grouping in pandas.

```
rows = cursor.execute("SELECT COUNT(*), gender FROM user GROUP BY gender")
```

Table - 1

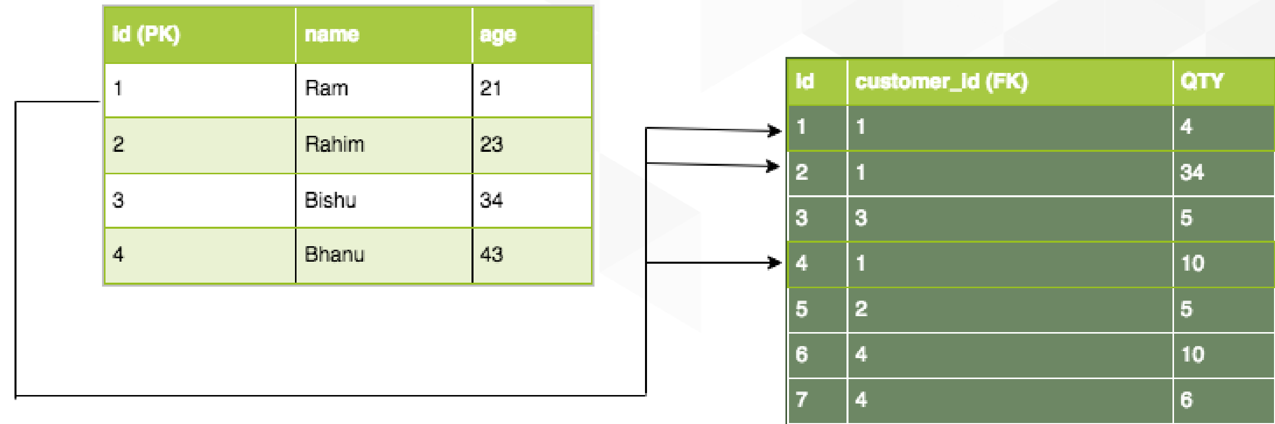
Col1	Col2	Col3	
		A	Group A
		A	
		A	
		B	Group B
		B	
		A	Group A
		A	
		A	
		B	Group B
		B	
		B	
		B	

COUNT(*), col3 -> 6, a AND 6, B

Relations

- Primary key in one table Referred as Foreign Key in another.
- Relations can be 1:1, 1:N, N:M, N:1
- Mentioned at the time of table creation

One to many relationship



Id (PK)	name	age
1	Ram	21
2	Rahim	23
3	Bishu	34
4	Bhanu	43

Id	customer_Id (FK)	QTY
1	1	4
2	1	34
3	3	5
4	1	10
5	2	5
6	4	10
7	4	6

```
sql = """
        CREATE TABLE comments (
            user_id text,
            comments text,
            FOREIGN KEY (user_id) REFERENCES user (email)
            ON DELETE CASCADE ON UPDATE NO ACTION
        )
    """
cursor.execute(sql)
conn.commit()
```

Discuss

- Can you give a real-life example of all the different relations (1:1, 1:N, N:1, N:M)?

Adding Rows in the Comment Table

- Prepare a template SQL:

```
sql = "INSERT INTO comments VALUES ('{}', '{}')"

rows = cursor.execute('SELECT * FROM user ORDER BY age')
for row in rows:
        email = row[0]
        name = row[1] + " " + row[2]
```

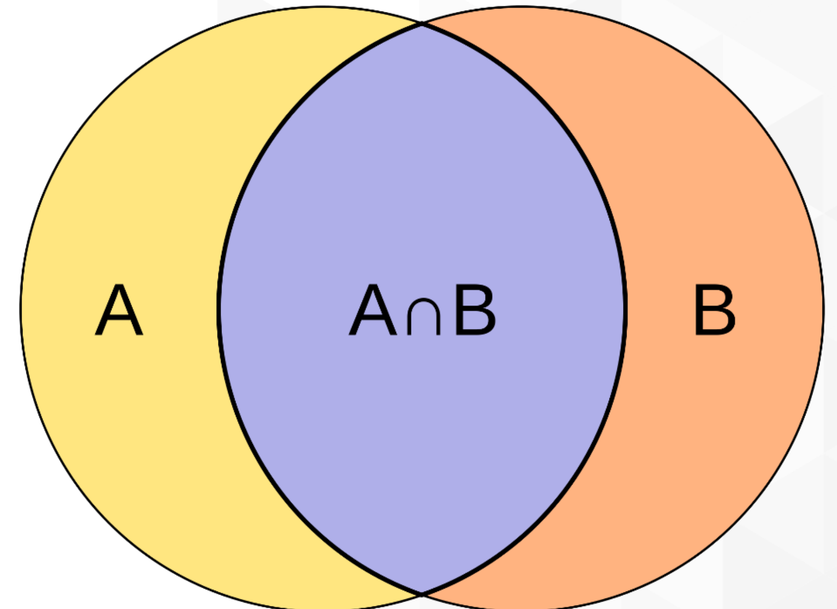
- Insert 10 dynamically generated comments with this code:

```
for i in range(10):
        comment = "This is comment {} by {}".format(i, name)
        conn.cursor().execute(sql.format(email, comment))
```

Joins

- This is where we tap into the power of relationships to fetch data from multiple tables.
- There are five types of join – INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, and CROSS
- Can be very time-consuming.

```
sql = """
        SELECT * FROM comments
        JOIN user ON comments.user_id = user.email
        WHERE user.email='bob@example.com'
    """
    rows = cursor.execute(sql)
    for row in rows:
        print(row)
```



Discuss

Do you think an inner join works with more than two tables? Is it possible, for example, to fetch rows from two tables, both of which are related to a third one (the parent table)?

Retrieving Specific Columns from a JOIN Query

- The last example fetched all columns from all the tables.
- This is not often a desired behavior.
- Modify it like as follows:

```
sql = """
        SELECT comments.* FROM comments
        JOIN user ON comments.user_id = user.email
        WHERE user.email='bob@example.com'
    """
rows = cursor.execute(sql)
for row in rows:
    print(row)
cursor.execute("UPDATE user set first_name='Chris' where email='tom@web.com'")
```

- This only updates the rows where the condition is true.

Exercise 112: Deleting Rows


```
cursor.execute("DELETE FROM user WHERE email='bob@example.com'")
conn.commit()
```

- This deletes all the rows from the user table where the condition is true.
- It also (automatically) deletes all the rows from the comments table linked with this user because we mentioned (while creating the comments table) ON DELETE CASCADE.

Discuss

Compare the delete, truncate, and drop commands in SQL.

Updating Specific Values in a table

- Combine UPDATE with WHERE to selectively update the first name of the user with the email address **tom@web.com:**

```
with sqlite3.connect("lesson.db") as conn:
    cursor = conn.cursor()
    cursor.execute("PRAGMA foreign_keys = 1")
    cursor.execute("UPDATE user set first_name='Chris' where email='tom@web.com'")
    conn.commit()
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

Discuss

- Can you find out what it means to do a LEFT OUTER or RIGHT OUTER join?
- Can you draw a diagram, as we did for INNER JOIN, to explain them?

Exercise 113: RDBMS and DataFrames

- Gather data:

```
sql = """
        SELECT user.email, user.first_name, user.last_name, user.age, user.gender,
        comments.comments FROM comments
        JOIN user ON comments.user_id = user.email
        WHERE user.email = 'tom@web.com'
    """
rows = cursor.execute(sql)
for row in rows:
    data.append(row)
```

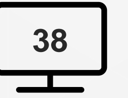
- Export DataFrame

```
df = pd.DataFrame(data, columns=columns)
```

Discuss

What are the other methods in pandas that can transform a set of data and retrieve it using a query? Can you figure out what the other functions we can use are? What benefits do they offer over the one shown in exercise 113?

Activity 11: Retrieving Data Correctly from Databases



Query the datasets to answer the following questions:

- Find the different age groups in the persons database.
- Find the age group that has the maximum number of people.
- Find the people who do not have a last name.
- Find how many people has more than one pet.
- Find how many pets have received treatment.
- Find how many pets are there from the city called *east port*.
- Find how many pets are there from the city called *east port* and who received treatment.

Summary

We took a deep dive into RDBMS and SQL and we learned many things, including the following:

- The fundamentals of RDBMS and SQL
- Relations and how they work in the worlds of RDBMS.
- How to efficiently open and close database connections using Python
- How to create, insert, and read data from a single table.
- The concept of a primary key.
- The concept of a foreign key and how to relate a second table with the first one.
- The concept of grouping.
- The concept of INNER JOIN and how to use JOIN to read data from two tables at the same time.
- How to export data to a pandas DataFrame.

Practice Questions

Practice Questions

1. Compare truncate, delete, and drop.

2. True or False. Closing a database is optional.

3. True or False. All databases are structured.

4. True or False. Delete and DELETE ON CASADE can perform the same operation in certain conditions.

5. True or False. It is easy to drop databases in SQL.

6. True or False. SQLite is lighter than SQL.

Packt

Lesson 9
Application of Data Wrangling in Real Life

1 Hour

Lesson Objectives

By the end of this lesson, you will be able to:

- Implement data wrangling on multiple full-fledged datasets from renowned sources
- Create a unified dataset that can be passed on to a data science team for analysis
- Relate data wrangling to version control, containerization, cloud services for data analytics, and big data technologies such as Apache Spark and Hadoop

Additional Skills to Pick up and Practice Regularly

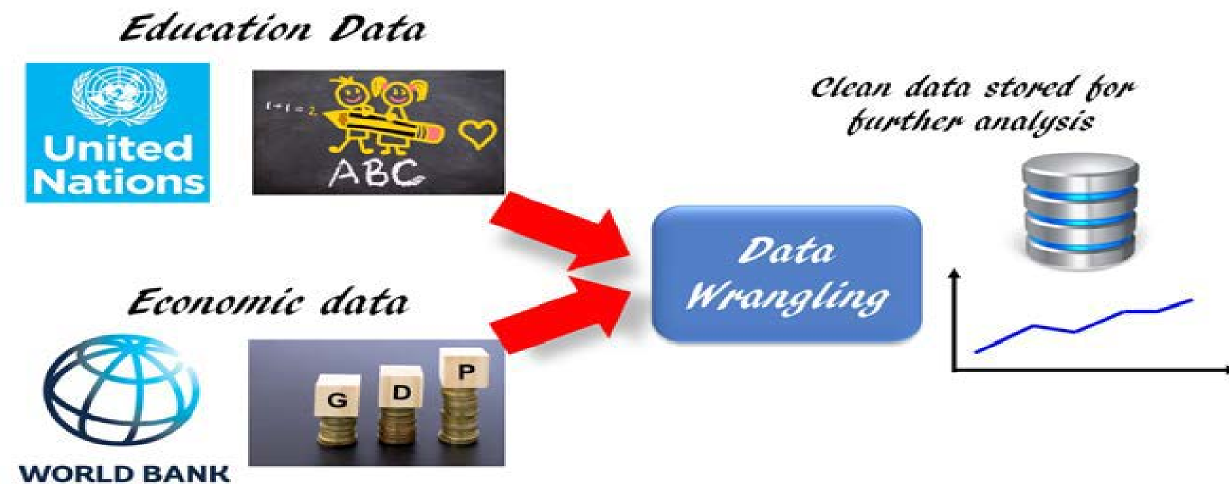
- To practice as a fully qualified data scientist/analyst, you should have some basic skills in your repertoire, irrespective of the particular programming language you chose to focus on.
- These skills and know-how are language-agnostic and can be utilized with any framework you need to use, depending on your organization’s and business needs.
- We describe them in brief here.

Introduction

- The primary job of a data wrangling expert is to pull data from multiple sources, format and clean it (impute the data if it is missing), and finally combine it in a coherent manner to prepare a dataset for further analysis by data scientists or machine learning engineers.
- In this topic, we will try to mimic such a typical task flow by downloading and using two different datasets from reputable web portals. Each of the datasets contains partial data pertaining to the key question that is being asked. Let's examine it more closely.

Applying Your Knowledge to Real-life Data Wrangling Task

- Suppose you are asked this question: **In India, did the enrollment in primary/ secondary/tertiary education increase with the improvement of per capita GDP in the past 15 years?**



Activity 12: Data Wrangling Task – Fixing UN Data

1. Download the dataset from the UN data from GitHub.
2. Clean the data to prepare a simple final dataset and save it as a SQL database file.
3. Use the **pd.read_csv** method of pandas.
4. The first row does not contain useful information, skip it using the **skiprows** parameter.
5. Drop the column region/country/area and source.
6. Assign Region/County/Area, Year, Data, Value, and Footnotes as columns of the DataFrame and check how many unique values in the **Footnotes** column.
7. Create a function to convert the value column into a floating point.
8. Use this function to convert value to floating point and print the unique values.

Activity 13: Data Wrangling Task – Cleaning GDP Data

1. Create the **df_primary, df_secondary,** and **df_tertiary DataFrames** for students enrolled in primary education, secondary education, and tertiary education in thousands, respectively.
2. Plot bar charts of the enrollment of primary students
3. Use pandas imputation methods to impute these data points by simple linear interpolation between data points. To do that, create a DataFrame with missing values inserted and append a new DataFrame with missing values to the current DataFrame.
4. Append the rows corresponding to the missing years - **2004 - 2009**, **2011 – 2013**.
5. Create a dictionary of values with **np.nan**. Create a DataFrame of missing values that we can append.

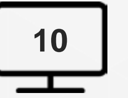
Activity 13: Data Wrangling Task – Cleaning GDP Data (contd)

6. Append the DataFrames together.
7. Sort by Year and reset the indices using **reset_index**. Use **inplace=True** to execute the changes on the DataFrame itself.
8. Use the interpolate method for linear interpolation. It fills all the NaNs by linearly interpolated values.
9. Repeat the same steps for USA (or other countries).
10. If there are values that are unfilled, use the **limit** and **limit_direction** parameters with the interpolate method to fill them in.
11. Plot the final graph using the new data.

Activity 13: Data Wrangling Task – Cleaning GDP Data (contd)

12. Read the GDP data using the pandas **read_csv** method. It will generally throw an error.
13. To avoid errors, try the **error_bad_lines = False** option.
14. Since there is no delimiter in the file, add the **\t** delimiter. Use the **skiprows** function to remove rows that are not useful.
15. Examine the dataset. Filter the dataset with information that it is similar to the previous education dataset. Reset the index for this new dataset. Drop the rows that are not useful and re-index the dataset.
16. Rename the columns properly. We will concentrate only on the data from 2003 to 2016. Eliminate the remaining data.
17. Create a new DataFrame called **df_gdp** with rows 43 to 56.

Activity 14: Data Wrangling Task – Merging UN Data and GDP Data



1. Reset the indexes for merging.
2. Merge the two DataFrames, **primary_enrollment_india** and **df_gdp**, on the Year column.
3. Drop the data, footnotes, and region/county/area.
4. Rearrange the columns for proper viewing and presentation.

Activity 15: Data Wrangling Task – Connecting the New Data to the Database

1. Import the **sqlite3** module of Python and use the **connect** function to connect to the database. The main database engine is embedded. But for a different database like **Postgresql** or **MySQL**, we will need to connect to them using those credentials. We designate **Year** as the **PRIMARY KEY** of this table.
2. Then, run a loop with the dataset rows one by one to insert them into the table.
3. If we look at the current folder, we should see a file called **Education_GDP.db**, and if we examine that using a database viewer program, we can see the data transferred there.

An Extension to Data Wrangling

15 mins

Git, Version Control, and GitHub

- Git is to version control what an RDBMS is to data storage and query.
- It simply means that there is a huge gap between the pre and post Git era of version controlling your code.
- As you may have noticed, all the notebooks for this course/book are hosted on GitHub, and this was done to take advantage of the powerful Git VCS.
- It gives you, out of the box, version control, history, branching facilities for different code, merging different code branches, and advanced operations such as cherry picking, diff, and so on.
- It is an essential tool to master, as you can be almost sure that you will face it at one point of time in your journey.

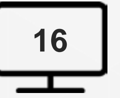
Linux Command Line

- People coming from a Windows background (or even Mac, if you have not done any development before) are not very familiar, usually, with the command line.
- The superior UI of those OSes hides the low-level details of interaction with the OS using command line.
- However, as a data professional, it is important that you know the command line well. There are so many operations that you can do simply using the command line that it is astonishing.
- Try learning about it in as much detail as possible. You can look into the KDNuggets tutorial if needed.

SQL and Basic Relational Database Concepts

- We dedicated an entire lesson to SQL and RDBMS.
- However, as we already mentioned there, it was really not enough. This is a vast subject and needs years of study to master it.
- Try to read more about it (including theory and practical) from books and online sources.
- Do not forget that, despite all the other sources of data being used nowadays, we still have hundreds of millions of structured data stored in legacy database systems. You can be sure to come across one sooner or later in your professional career.

Docker and Containerization



- Since its first release in 2013, Docker has changed the way we distribute and deploy software in server-based applications.
- It gives you a clean and lightweight abstraction over the underlying OS and lets you iterate fast on development without the headache of creating and maintaining a proper environment.
- It is very useful in both the development and the production phase. With virtually no competitor, they are becoming the default in the industry very fast.
- We strongly advise you to explore it in great detail.

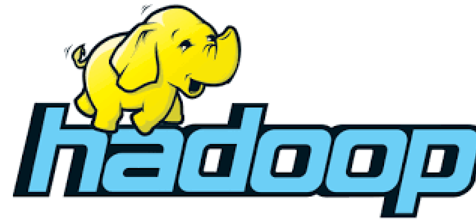
Basic Familiarity with Big Data and Cloud Technologies

- Big data and cloud platforms are everywhere these days, and it is very important that you have a clear idea and grasp on them.
- We introduce them here with one or two short sentences, and we encourage you to go ahead and learn them as much as you can.
- If you are planning to grow as a data professional, then you can be sure that without these necessary skills it will be hard for you to transition to the next level.

Fundamental Characteristics of Big Data

- Big data is simply data which is very big in size.
- The term *size* is a bit ambiguous here. It can mean one static chunk of data (such as the detailed census data of a big country such as India or the US) or data that is dynamically generated as time passes, and each time it is huge.
- To give an example for the second category, we can think of how much data is generated via Facebook per day. Statistics shows that *500+ terabytes* of new data gets ingested into the databases of Facebook every day. That is a very large amount. And you can easily imagine that we will need specialized tools to deal with that amount of data.
- There are three different categories of big data: Structured, Unstructured, and Semi-Structured. The main characteristics that define big data are Volume, Variety, Velocity, and Variability.

Hadoop Ecosystem



- Apache Hadoop (and the related ecosystem) is a software framework that aims to simplify the storage and processing of big data using the Map-Reduce programming model. It was originally introduced in 2011 and it has since become one of the backbones of the big data industry.
- All the modules in Hadoop are designed with the fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.
- The four base modules of Hadoop are common, HDFS, YARN, and Map-Reduce. The Hadoop ecosystem consists of Apache Pig, Apache Hive, Apache Impala, Apache Zookeeper, Apache Hbase, and more.
- They are very important bricks in many high demand and cutting edge data pipelines. We encourage you to study more about them. They are essential in any industry that aims to leverage data.

Apache Spark

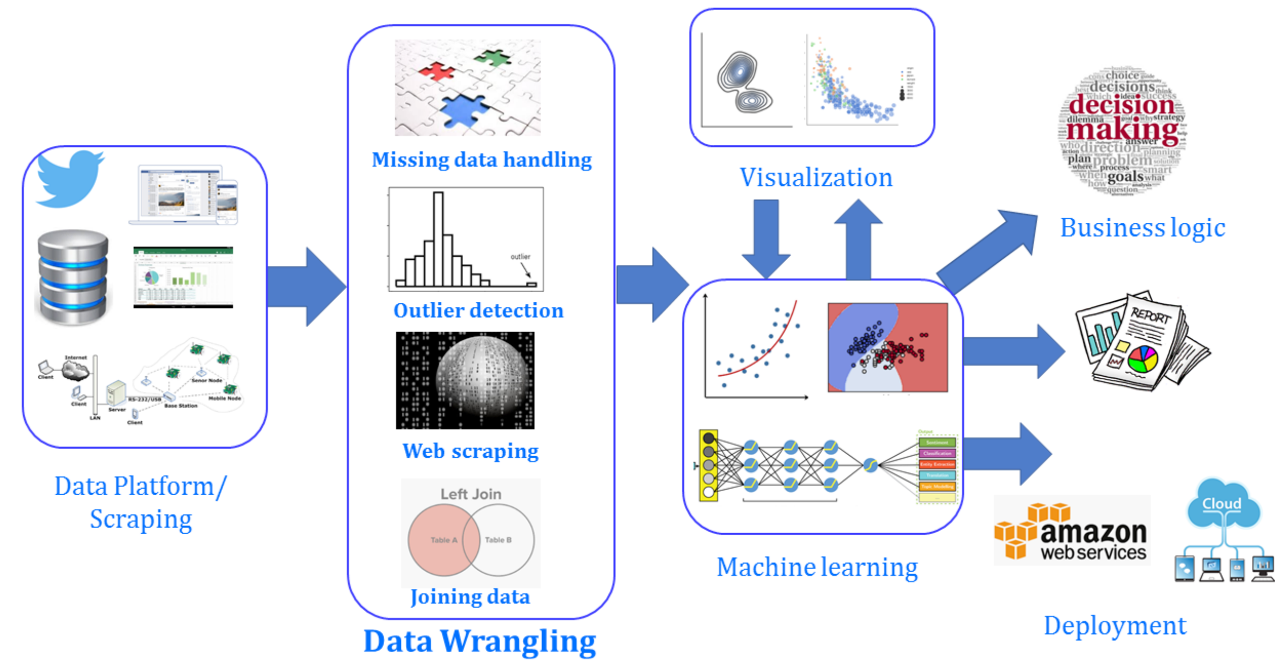
- Apache Spark is a general-purpose cluster computing framework initially developed at University of California, Berkley, and released in 2014. It gives you an interface to program an entire cluster of computers with built-in data parallelism and fault tolerance.
- It contains Spark Core, Spark SQL, Spark Streaming, MLib (for machine learning), and GraphX.
- It is now one of the main frameworks used in the industry to process a huge amount of data in real time based on streaming data. We encourage you to read and master it if you want to go toward real-time data engineering.

Amazon Web Service (AWS)

- Amazon Web Services (often abbreviated as AWS) is a bunch of managed services offered by Amazon ranging from Infrastructure-as-a-Service, Database-as-a-Service, Machine Learning-as-a-Service, Cache, Load Balancer, and NoSQL database, to Message Queues, and several other types.
- They are very useful for all sorts of applications. It can be a simple web app or a multi-cluster data pipeline. Many famous companies run their entire infrastructure on AWS (Netflix, for example).
- They give us on-demand provision, easy scaling, a managed environment, a slick UI to control everything, and also a very powerful command-line client. They also expose a rich set of APIs and we can find AWS API client in virtually any programming language.

What Goes with Data Wrangling?

- As we showed in the flow in Lesson 1, data wrangling sits in between data gathering and advanced analytics, including visualization and machine learning.
- However, the boundaries between the disciplines are not very strict, particularly depending on the organizational culture and team composition. Therefore, it is essential that you have a fair bit of knowledge about all the other components of a data science platform and can contribute as and when necessary.



Business Logic and Domain Expertise

- Business logic and domain expertise is the most varied topic, and it can only be learned on the job, however it will come naturally with experience.
- If you have an academic background and/or work experience in any particular domain, such as finance, medicine and healthcare, and engineering (for example, electronics or telecommunication), that knowledge will come in handy in your data science career.
- You can selectively choose to apply for and work in jobs that can benefit from the deep domain expertise you provide, along with your data wrangling and analysis skills.

Machine Learning

- The fruit of the hard work of data wrangling is realized fully in the domain of **machine learning.** It is the science and engineering of making machines learn patterns and insights from data for predictive analytics and intelligent, automated decision-making with a deluge of data, which cannot be analyzed efficiently by humans.
- Machine learning has become one of the most sought-after skills in the modern technology landscape. It has truly become one of the most exciting and promising intellectual fields, with applications ranging from e-commerce to healthcare and virtually everything in between.
- **Data wrangling is intrinsically linked with machine learning as it prepares the data suitable for intelligent algorithms to process**. Even if you start your career in data wrangling, it could be a natural progression to move to machine learning.

Some Tips and Tricks for Mastering Machine Learning

- You can start with some videos on YouTube. Also, you can read a couple of good books or articles. For example, you can read *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*.
- You must think clearly and learn to differentiate between the buzzwords—*machine learning, artificial intelligence, deep learning, data science, computer vision, robotics*. Read or listen to talks given by experts on each of them.
- **Cultivate the habit of following some good, influential blogs**: KDnuggets, Mark Meloon's blog about data science career, Brandon Rohrer's blog, Open AI's blog about their research.

Summary

- Data is everywhere and it is all around us. In these nine chapters, we have learned how data from different types and sources can be cleaned, corrected, and combined.
- Using the power of Python and the knowledge of data wrangling and applying the tricks and tips that you have studied in this course, you are ready to be a data wrangler.

Hearty congratulations for working hard on this course and making it to the end.

Best of luck for your continued success in the exciting world of data science and engineering!

Lesson 10

Course Summary

Packt
LEARNING TREE INTERNATIONAL

Course Objectives

- Extract and parse data from various sources
- Transform and clean data using Numpy and Pandas
- Summarize and visualize data with Matplotlib
- Read HTML, XML, and JSON data from internet resources
- Search and filter data sets
- Apply Python tools and techniques to process data sets efficiently

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.