# Discovering Disease Outbreaks from News Headlines

## Introduction

### **about this liveProject**

In this liveProject, you will fill the role of a data scientist at the World Health Organization (WHO) and identify disease outbreaks around the world using data science techniques in Python. The [WHO](https://www.who.int/) is a specialized agency of the United Nations responsible for monitoring and responding to international disease epidemics. A critical component of the monitoring process is analyzing news headlines published around the world for signs of epidemics. Each day, the WHO receives hundreds of news headlines concerning potential disease outbreaks in diverse geographic locations.  
  
The daily deluge of news headlines is too overwhelming to be analyzed manually due to the limited resources of the WHO and the large quantity of data. The WHO therefore needs data scientists to extract geographic information from the headlines and determine if and where there are clusters of disease outbreaks. This critical information will be used to dispatch resources where they are needed.  
  
Your work identifying and mapping epidemics is crucial to prevent or respond to deadly disease outbreaks. Your first assignment for the WHO is as follows:

* Extract the locations (city and/or country name) from each news headline.
* Find the geographic coordinates of each headline using the city/country.
* Cluster (group) the headlines based on the geographic location.
* Visualize the clusters on a map and analyze them for patterns indicating an epidemic.
* Investigate the largest clusters for signs of disease outbreaks.
* Review headlines in the largest clusters within the United States and around the world. If any disease outbreak is particularly dominant, visualize all worldwide mentions of that disease.
* Provide a summary of your findings to your superiors at the WHO so they can direct resources.

This liveProject is based on a case study from the Manning book [*Data Science Bookcamp*](https://livebook.manning.com/book/data-science-bookcamp/welcome/v-3/) by Leonard Apeltsin.

### **Techniques employed**

The following are some of the techniques you’ll employ throughout this project. Don’t worry if you haven’t mastered one of these areas; we’ll give you the resources to learn more. Data scientists must use a diverse range of techniques, many of which are picked up on the job for a specific project!  
  
Listed under the bullets are the Python libraries for the technique.

* Extracting city and country name data from text using regular expressions
  + re: text extraction with regular expressions
* Manipulating data and matching location names to geographic coordinates
  + pandas: data manipulation and analysis
* Clustering geographic coordinates with k-means and/or DBSCAN
  + scikit-learn: machine learning
* Visualizing clusters on a geographic map
  + basemap: geographic visualization
* Analyzing algorithm output and tuning model settings to improve result
* Sorting between clusters based on size and within clusters based on distance
  + matplotlib: general visualization
* Interpreting algorithm results in the problem domain
* Summarizing findings of a data science project effectively
  + nbconvert: converting Jupyter Notebooks to reports

Resources are provided for working with these techniques/libraries both as embedded in Manning liveBooks and as links to external documentation. On the job, you will often be given partial information and must learn how to seek out relevant resources on your own!

### **Project outline**

The project is broken into five parts.

**1. Extracting City and Country Information from News Headlines**

**2. Finding Geographic Locations of Headlines**

**3. Clustering Headlines Based on Location**

**4. Identifying Disease Outbreaks**

**5. Presenting the Disease Outbreak Data**

The skills covered in order are the following: text extraction, data manipulation, clustering, interpreting algorithm outputs, and producing an actionable report. The deliverable from each part is a Jupyter notebook (uploaded to GitHub) documenting your workflow and results. The final milestone will be a notebook converted to html or a PDF to share the conclusions with your superiors at the WHO. Each section builds upon the previous and will test your skills in a different area of data science. As you go through the project, keep in mind the overall objective: to identify disease outbreaks around the world. The project is representative of the problems solved by data scientists in industry or academia and utilizes the most popular tools for data science in Python.

### **Prerequisites**

The liveProject is for intermediate Python programmers who know the basics of data science. To begin this liveProject, you will need to be familiar with:

* TOOLS
  + Basics of pandas
  + Basics of scikit-learn
  + Basics of text extraction
  + Basics of k-means and DBSCAN clustering
  + Basics of Jupyter Notebook

An enthusiasm for solving problems and self-led learning will also prove quite useful (and not just in this project)!

### **Python libraries and setup**

This uses Python 3.7. It is recommended to use the [Anaconda distribution of Python](https://www.anaconda.com/distribution/) and conda for managing the libraries.

The following Python libraries will be utilized in this project. Again, you don’t need working experience with all of these, as you can pick up the basics by reading documentation and putting them to use.

* re: For text extraction with regular expressions
* geonamescache: For identifying cities and countries and their locations
* pandas: For data manipulation and analysis
* matplotlib: For general visualization
* basemap: For geographic visualization
* jupyter: For running Jupyter Notebooks
* scikit-learn: For implementing k-means and DBSCAN clustering
* nbconvert: For converting Jupyter Notebooks to HTML and/or pdf

To install these libraries, clone the base [Git repository](https://github.com/WillKoehrsen/discovering-disease-outbreaks-base) to your computer and run *conda env create -f environment.yml* from the root project directory (assuming you are using conda).

Once the virtual environment is created, you can activate it with *conda activate discovering-disease-outbreaks* and run a Jupyter Notebook with *jupyter notebook*.

Any additional libraries you need can be installed with *conda install “library.”* Managing Python package versions with virtual libraries is a crucial data science skill to avoid situations where code that works on one machine causes errors on another!

### **Dataset**

[headlines.txt](https://livevideo-resources.s3-us-west-2.amazonaws.com/course-data/93/headlines.txt) (Available in the GitHub repository)

* This file contains a curated list of actual worldwide news headlines referring to diseases.
* Each headline is on a separate line with a total of 650 headlines.

**Example headlines:**

Zika spreads to Winter Park

Durango is infested with Hepatitis B

Mpika authorities confirmed the spread of Chikungunya

Gladstone Encounters Severe Symptoms of Dengue

Mad Cow Disease Hits London

Zika symptoms spotted in Quito

The headlines in the dataset were synthetically created, but the dataset’s distribution of disease and location-mentions is modeled on real health mentions in news articles over the first few months of 2016. These articles were detected by matching health keywords within international sections of popular online news sources. Such sections are commonly demarcated by continent or country within the news source’s URL. For instance, <https://cnn.com/africa> contains English-language news coverage from Africa, while <https://reuters.com/places/thailand> contains English-language news coverage from Thailand. Article content ranges from local politics to newsworthy health incidents.

We matched health terms in the body and headline of the articles to find the health-related articles. Inclusion of the body text would have added surplus noise to our analysis and all mentions of location-specific diseases have been transformed into simple, single-line article titles. About half of the headlines originated in the United States, which might necessitate splitting the data into two categories: The United States and the rest of the world.

### **Recommended resources**

* [Data Science Bookcamp](https://livebook.manning.com/book/data-science-bookcamp/welcome/v-3/) by Leonard Apeltsin
* [The Quick Python Book, Third Edition](https://livebook.manning.com/book/the-quick-python-book-third-edition/about-this-book/) by Naomi Ceder
* [Think Like a Data Scientist](https://livebook.manning.com/book/think-like-a-data-scientist/about-this-book/) by Brian Godsey
* Documentation on Python libraries:
  + [re](https://docs.python.org/3/library/re.html)
  + [geonamescache](https://pypi.org/project/geonamescache/)
  + [pandas](https://pandas.pydata.org/pandas-docs/stable/)
  + [matplotlib](https://matplotlib.org/3.1.1/contents.html)
  + [basemap](https://matplotlib.org/basemap/users/intro.html)
  + [scikit-learn](http://scikit-learn.org/stable/documentation.html)
* [Stack Overflow](https://stackoverflow.com/)
* [Jupyter documentation](https://jupyter.readthedocs.io/en/latest/)
* [Python official docs](https://docs.python.org/3/)

Additional resources and tutorials are provided throughout the project. Feel free to use any resources you can find to complete the project.

If you run into problems or have questions, refer to the [Frequently Asked Questions](https://liveproject.manning.com/module/93_7_2/discovering-disease-outbreaks-from-news-headlines/project-conclusions/moduleName?) (FAQs) within the Summary section.

# 1. Extracting City and Country Information from News Headlines

# **1.1 Parsing the News Headlines**

**Objective**

* Find any city and/or country names mentioned in each of the news headlines.

**Workflow**

1. Load in the headline data and examine it for any data quality issues.
   * Use any library/data structure to read in the headlines.
   * Read through some of the headlines and identify potential problems.
2. Using regular expressions and the cities and countries within the geonamescache library, match any cities/countries within each headline.
   * Make sure to normalize headlines and city/country names by removing accent marks. This can be done with the unidecode library.
   * Watch out for multiple cities in a headline and matches on short words! We want the match to be on the entire city—for example San Marino—and not a partial match—San.
3. Put the extracted data into a pandas DataFrame with three columns: headline, city, country.
4. Make sure there were no issues with the extraction by sampling some of the headlines and examining the city and country names.
   * One method for finding problems is to look for the most common names and see if there are any issues.
5. Once you are confident you’ve found all the cities/countries in each headline, save the DataFrame for the next part.

**Importance to project**

* We can’t do much with just the headlines; although they contain the city/country names, they do not contain the geographic information—latitude and longitude—we need to find clusters of disease outbreaks. The first step in getting the geographic information is to isolate the names.
* Later, we will use the names to find the location of each headline, which requires bringing in external data (through geonamescache).
* This workflow is common in data science. First, we separate the useful information from the noise—data mining—and then we augment it with external data—data engineering.

**Resources**

* Chapter 16, [Regular Expressions](https://liveproject.manning.com/module/93_2_2/discovering-disease-outbreaks-from-news-headlines/1--extracting-city-and-country-information-from-news-headlines/1-2--regular-expressions?), from The Quick Python Book, Third Edition by Naomi Ceder in section 1.2 can help you with the text extraction phase.
* Chapter 11, [Matching Location Names in Text](https://liveproject.manning.com/module/93_2_3/discovering-disease-outbreaks-from-news-headlines/1--extracting-city-and-country-information-from-news-headlines/1-3--matching-location-names-in-text?), from Data Science Bookcamp by Leonard Apeltsin in section 1.3 can help you with finding the names of cities/countries.

# **1.2 Regular Expressions**

excerpt from The Quick Python Book, Third Edition | Naomi Ceder[go to book](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-16)

### What is a regular expression?

A*regular expression* (regex) is a way of recognizing and often extracting data from certain patterns of text. A regex that recognizes a piece of text or a string is said to *match* that text or string. A regex is defined by a string in which certain characters (the so-called *metacharacters*) can have a special meaning, which enables a single regex to match many different specific strings.

It’s easier to understand this through example than through explanation. Here’s a program with a regular expression that counts how many lines in a text file contain the word *hello*. A line that contains *hello* more than once is counted only once:

Top of Form

1

2

3

4

5

6

7

8

9

import re

regexp = re.compile("hello")

count = 0

**file** = **open**("textfile", 'r')

**for** line in **file**.readlines():

**if** regexp.search(line):

count = count + 1

**file**.**close**()

**print**(count)

Bottom of Form

copy

The program starts by importing the Python regular expression module, called re. Then it takes the text string "hello" as a *textual regular expression* and compiles it into a *compiled regular expression*, using the re.compile function. This compilation isn’t strictly necessary, but compiled regular expressions can significantly increase a program’s speed, so they’re almost always used in programs that process large amounts of text.

What can the regex compiled from "hello" be used for? You can use it to recognize other instances of the word "hello" within another string; in other words, you can use it to determine whether another string contains "hello" as a substring. This task is accomplished by the search method, which returns None if the regular expression isn’t found in the string argument; Python interprets None as false in a Boolean context. If the regular expression is found in the string, Python returns a special object that you can use to determine various things about the match (such as where in the string it occurred). I discuss this topic later.

### Regular expressions with special characters

The previous example has a small flaw: It counts how many lines contain "hello" but ignores lines that contain "Hello" because it doesn’t take capitalization into account.

One way to solve this problem would be to use two regular expressions—one for "hello" and one for "Hello"—and test each against every line. A better way is to use the more advanced features of regular expressions. For the second line in the program, substitute

Top of Form

1

regexp = re.compile("hello|Hello")

Bottom of Form

copy

This regular expression uses the vertical-bar special character |. A *special character* is a character in a regex that isn’t interpreted as itself; it has some special meaning. | means *or*, so the regular expression matches "hello" *or* "Hello".

Another way of solving this problem is to use

Top of Form

1

regexp = re.compile("(h|H)ello")

Bottom of Form

copy

In addition to using |, this regular expression uses the *parentheses* special characters to group things, which in this case means that the | chooses between a small or capital *H*. The resulting regex matches either an *h* or an *H*, followed by *ello*.

Another way to perform the match is

Top of Form

1

regexp = re.compile("[hH]ello")

Bottom of Form

copy

The special characters [ and ] take a string of characters between them and match any single character in that string. There’s a special shorthand to denote ranges of characters in [ and ]; [a-z] match a single character between *a* and *z*, [0-9A-Z] match any digit or any uppercase character, and so forth. Sometimes, you may want to include a real hyphen in the [], in which case you should put it as the first character to avoid defining a range; [-012] match a hyphen, a *0*, a *1*, or a *2*, and nothing else.

Quite a few special characters are available in Python regular expressions, and describing all of the subtleties of using them in regular expressions is beyond the scope of this book. A complete list of the special characters available in Python regular expressions, as well as descriptions of what they mean, is in the online documentation of the regular expression re module in the standard library. For the remainder of this chapter, I describe the special characters I use as they appear.

##### QUICK CHECK: SPECIAL CHARACTERS IN REGULAR EXPRESSIONS

What regular expression would you use to match strings that represent the numbers -5 through 5?

What regular expression would you use to match a hexadecimal digit? Assume that allowed hexadecimal digits are 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, a, B, b, C, c, D, d, E, e, F, and f.

### Regular expressions and raw strings

The functions that compile regular expressions, or search for matches to regular expressions, understand that certain character sequences in strings have special meanings in the context of regular expressions. regex functions understand that \n represents a newline character, for example. But if you use normal Python strings as regular expressions, the regex functions typically never see such special sequences, because many of these sequences also possess a special meaning in normal strings. \n, for example, also means newline in the context of a normal Python string, and Python automatically replaces the string sequence \n with a newline character before the regex function ever sees that sequence. The regex function, as a result, compiles strings with embedded newline characters—not with embedded \n sequences.

In the case of \n, this situation makes no difference because regex functions interpret a newline character as exactly that and do the expected thing: attempt to match the character with another newline character in the text being searched.

Now look at another special sequence, \\, which represents a *single* backslash to regular expressions. Assume that you want to search text for an occurrence of the string "\ten". Because you know that you have to represent a backslash as a double backslash, you might try

Top of Form

1

regexp = re.compile("\\ten")

Bottom of Form

copy

This example compiles without complaining, but it’s wrong. The problem is that \\ also means a single backslash in Python strings. Before re.compile is invoked, Python interprets the string you typed as meaning \ten, which is what is passed to re.compile. In the context of regular expressions, \t means *tab*, so your compiled regular expression searches for a tab character followed by the two characters *en*.

To fix this problem while using regular Python strings, you need four backslashes. Python interprets the first two backslashes as a special sequence representing a single backslash, and likewise for the second pair of backslashes, resulting in two *actual* backslashes in the Python string. Then that string is passed in to re.compile, which interprets the two actual backslashes as a regex special sequence representing a single backslash. Your code looks like this:

Top of Form

1

regexp = re.compile("\\\\ten")

Bottom of Form

copy

That seems confusing, and it’s why Python has a way of defining strings that doesn’t apply the normal Python rules to special characters. Strings defined this way are called *raw strings*.

#### Raw strings to the rescue

A raw string looks similar to a normal string except that it has a leading *r* character immediately preceding the initial quotation mark of the string. Here are some raw strings:

Top of Form

1

2

3

4

r"Hello"

r"""\tTo be\n\tor not to be"""

r'Goodbye'

r'''12345'''

Bottom of Form

copy

As you can see, you can use raw strings with either the single or double quotation marks and with the regular or triple-quoting convention. You can also use a leading *R* instead of *r* if you want to. No matter how you do it, raw-string notation can be taken as an instruction to Python saying “Don’t process special sequences in this string.” In the previous examples, all the raw strings are equivalent to their normal string counterparts except the second example, in which the \t and \n sequences aren’t interpreted as tabs or newlines but are left as two-string character sequences beginning with a backslash.

Raw strings aren’t different types of strings. They represent a different way of *defining* strings. It’s easy to see what’s happening by running a few examples interactively:

Top of Form

1

2

3

4

5

6

7

8

9

10

>>> r"Hello" == "Hello"

True

>>> r"\the" == "\\the"

True

>>> r"\the" == "\the"

False

>>> print(r"\the")

\the

>>> print("\the")

he

Bottom of Form

copy

Using raw strings with regular expressions means that you don’t need to worry about any funny interactions between string special sequences and regex special sequences. You use the regex special sequences. Then the previous regex example becomes

Top of Form

1

regexp = re.compile(r"\\ten")

Bottom of Form

copy

which works as expected. The compiled regex looks for a single backslash followed by the letters *ten*.

You should get into the habit of using raw strings whenever defining regular expressions, and you’ll do so for the remainder of this chapter.

### Extracting matched text from strings

One of the most common uses of regular expressions is to perform simple pattern-based parsing on text. This task is something you should know how to do, and it’s also a good way to learn more regex special characters.

Assume that you have a list of people and phone numbers in a text file. Each line of the file looks like this:

Top of Form

1

surname, firstname middlename: phonenumber

Bottom of Form

copy

You have a surname followed by a comma and space, followed by a first name, followed by a space, followed by a middle name, followed by colon and a space, followed by a phone number.

But to make things complicated, a middle name may not exist, and a phone number may not have an area code. (It might be 800-123-4567 or 123-4567.) You *could* write code to explicitly parse data out from such a line, but that job would be tedious and error-prone. Regular expressions provide a simpler answer.

Start by coming up with a regex that matches lines of the given form. The next few paragraphs throw quite a few special characters at you. Don’t worry if you don’t get them all on the first read; as long as you understand the gist of things, that’s all right.

For simplicity’s sake, assume that first names, surnames, and middle names consist of letters and possibly hyphens. You can use the [] special characters discussed in the previous section to define a pattern that defines only name characters:

Top of Form

1

[-a-zA-z]

Bottom of Form

copy

This pattern matches a single hyphen, a single lowercase letter, or a single uppercase letter.

To match a full name (such as McDonald), you need to repeat this pattern. The + metacharacter repeats whatever comes before it one or more times as necessary to match the string being processed. So the pattern

Top of Form

1

[-a-zA-Z]+

Bottom of Form

copy

matches a single name, such as Kenneth or McDonald or Perkin-Elmer. It also matches some strings that aren’t names, such as --- or -a-b-c-, but that’s all right for purposes of this example.

Now, what about the phone number? The special sequence \d matches any digit, and a hyphen outside [] is a normal hyphen. A good pattern to match the phone number is

Top of Form

1

\d\d\d-\d\d\d-\d\d\d\d

Bottom of Form

copy

That’s three digits followed by a hyphen, followed by three digits, followed by a hyphen, followed by four digits. This pattern matches only phone numbers with an area code, and your list may contain numbers that don’t have one. The best solution is to enclose the area-code part of the pattern in (); group it; and follow that group with a ? special character, which says that the thing coming immediately before the ? is optional:

Top of Form

1

(\d\d\d-)?\d\d\d-\d\d\d\d

Bottom of Form

copy

This pattern matches a phone number that may or may not contain an area code. You can use the same sort of trick to account for the fact that some of the people in your list have middle names (or initials) included and others don’t. (To do so, make the middle name optional by using grouping and the ? special character.)

You can also use {} to indicate the number of times that a pattern should repeat, so for the phone-number examples above, you could use:

Top of Form

1

(\**d**{3}-)?\**d**{3}-\**d**{4}

Bottom of Form

copy

This pattern also means an optional group of three digits plus a hyphen, three digits followed by a hyphen, and then four digits.

Commas, colons, and spaces don’t have any special meanings in regular expressions; they mean themselves.

Putting everything together, you come up with a pattern that looks like this:

Top of Form

1

[-a-zA-Z]+, [-a-zA-Z]+( [-a-zA-Z]+)?: (\**d**{3}-)?\**d**{3}-\**d**{4}

Bottom of Form

copy

A real pattern probably would be a bit more complex, because you wouldn’t assume that there’s exactly one space after the comma, exactly one space after the first and middle names, and exactly one space after the colon. But that’s easy to add later.

The problem is that, whereas the above pattern lets you check to see whether a line has the anticipated format, you can’t extract any data yet. All you can do is write a program like this:

1

2

3

4

5

6

7

8

9

10

11

**import** re

regexp = re.compile(r"[-a-zA-Z]+,"

r" [-a-zA-Z]+"

r"( [-a-zA-Z]+)?"

r": (\d{3}-)?\d{3}-\d{4}"

)

file = open("textfile", 'r')

**for** line **in** file.readlines():

**if** regexp.search(line):

print("Yeah, I found a line with a name and number. So what?")

file.close()

**1**

**2**

**3**

**4**

copy

Notice that you’ve split your regex pattern, using the fact that Python implicitly concatenates any set of strings separated by whitespace. As your pattern grows, this technique can be a great aid in keeping the pattern maintainable and understandable. It also solves the problem with the line length possibly increasing beyond the right edge of the screen.

Fortunately, you can use regular expressions to extract data from patterns, as well as to see whether the patterns exist. The first step is to group each subpattern corresponding to a piece of data you want to extract by using the () special characters. Then give each subpattern a unique name with the special sequence ?P<name>, like this:

Top of Form

1

2

(?P<**last**>[-a-zA-Z]+), (?P<**first**>[-a-zA-Z]+)( (?P<**middle**>([-a-zA-Z]+)))?:

(?P<**phone**>(\d{3}-)?\d{3}-\d{4}

Bottom of Form

copy

(Please note that you should enter these lines as a single line, with no line breaks. Due to space constraints, the code can’t be represented here in that manner.)

There’s an obvious point of confusion here: The question marks in ?P<...> and the question-mark special characters indicating that the middle name and area code are optional have nothing to do with one another. It’s an unfortunate semi-coincidence that they happen to be the same character.

Now that you’ve named the elements of the pattern, you can extract the matches for those elements by using the group method. You can do so because when the search function returns a successful match, it doesn’t return just a truth value; it also returns a data structure that records what was matched. You can write a simple program to extract names and phone numbers from your list and print them out again, as follows:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

**import** re

regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"

r" (?P<first>[-a-zA-Z]+)"

r"( (?P<middle>([-a-zA-Z]+)))?"

r": (?P<phone>(\(\d{3}-)?\d{3}-\d{4})"

)

file = open("textfile", 'r')

**for** line **in** file.readlines():

result = regexp.search(line)

**if** result == None:

print("Oops, I don't think this is a record")

**else**:

lastname = result.group('last')

firstname = result.group('first')

middlename = result.group('middle')

**if** middlename == None:

middlename = ""

phonenumber = result.group('phone')

print('Name:', firstname, middlename, lastname,' Number:', phonenumber)

file.close()

**1**

**2**

**3**

**4**

copy

There are some points of interest here:

* You can find out whether a match succeeded by checking the value returned by search. If the value is None, the match failed; otherwise, the match succeeded, and you can extract information from the object returned by search.
* group is used to extract whatever data matched your named subpatterns. You pass in the name of the subpattern you’re interested in.
* Because the middle subpattern is optional, you can’t count on it to have a value, even if the match as a whole is successful. If the match succeeds, but the match for the middle name doesn’t, using group to access the data associated with the middle subpattern returns the value None.
* Part of the phone number is optional, but part isn’t. If the match succeeds, the phone subpattern must have some associated text, so you don’t have to worry about it having a value of None.

##### TRY THIS: EXTRACTING MATCHED TEXT

Making international calls usually requires a + and the country code. Assuming that the country code is two digits, how would you modify the code above to extract the + and the country code as part of the number? (Again, not all numbers have a country code.) How would you make the code handle country codes of one to three digits?

### Substituting text with regular expressions

In addition to extracting strings from text, you can use Python’s regex module to find strings in text and substitute other strings in place of those that were found. You accomplish this task by using the regular substitution method sub. The following example replaces instances of "the the" (presumably, a typo) with single instances of "the":

Top of Form

1

2

3

4

5

6

>>> **import** re

>>> string = "If the the problem is textual, use the the re module"

>>> pattern = r"the the"

>>> regexp = re.compile(pattern)

>>> regexp.sub("the", string)

'If the problem is textual, use the re module'

Bottom of Form

copy

The sub method uses the invoking regex (regexp, in this case) to scan its second argument (string, in the example) and produces a new string by replacing all matching substrings with the value of the first argument ("the", in this example).

But what if you want to replace the matched substrings with new ones that reflect the value of those that matched? This is where the elegance of Python comes into play. The first argument to sub—the replacement substring, "the" in the example—doesn’t have to be a string at all. Instead, it can be a function. If it’s a function, Python calls it with the current match object; then it lets that function compute and return a replacement string.

To see this function in action, build an example that takes a string containing integer values (no decimal point or decimal part) and returns a string with the same numerical values but as floating numbers (with a trailing decimal point and zero):

Top of Form

1

2

3

4

5

6

7

8

9

>>> **import** re

>>> int\_string = "1 2 3 4 5"

>>> **def** **int\_match\_to\_float**(match\_obj):

... **return**(match\_obj.group('num') + ".0")

...

>>> pattern = r"(?P<num>[0-9]+)"

>>> regexp = re.compile(pattern)

>>> regexp.sub(int\_match\_to\_float, int\_string)

'1.0 2.0 3.0 4.0 5.0'

Bottom of Form

copy

In this case, the pattern looks for a number consisting of one or more digits (the [0-9]+ part). But it’s also given a name (the ?P<num>... part) so that the replacement string function can extract any matched substring by referring to that name. Then the sub method scans down the argument string "1 2 3 4 5", looking for anything that matches [0-9]+. When sub finds a substring that matches, it makes a match object defining exactly which substring matched the pattern, and it calls the int\_match\_to\_float function with that match object as the sole argument. int\_match\_to\_float uses group to extract the matching substring from the match object (by referring to the group name num) and produces a new string by concatenating the matched substring with a ".0". sub returns the new string and incorporates it as a substring into the overall result. Finally, sub starts scanning again right after the place where it found the last matching substring, and it keeps going like that until it can’t find any more matching substrings.

##### TRY THIS: REPLACING TEXT

In the checkpoint in [section 16.4](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-16/ch16lev1sec4), you extended a phone-number regular expression to also recognize a country code. How would you use a function to make any numbers that didn’t have a country code now have +1 (the country code for the United States and Canada)?

##### LAB 16: PHONE-NUMBER NORMALIZER

In the United States and Canada, phone numbers consist of ten digits, usually separated into a three-digit area code, a three-digit exchange code, and a four-digit station code. As mentioned in [section 16.4](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-16/ch16lev1sec4), they may or may not be preceded by +1, the country code. In practice, however, you have many ways to format a phone number, such as (NNN) NNN-NNNN, NNN-NNN-NNNN, NNN NNN-NNNN, NNN.NNN.NNNN, and NNN NNN NNNN, to name a few. Also, the country code may not be present, may not have a +, and usually (not always) is separated from the number by a space or dash. Whew!

In this lab, your task is to create a phone-number normalizer that takes any of the formats and returns a normalized phone number 1-NNN-NNN-NNNN.

The following are all possible phone numbers:

|  |  |  |
| --- | --- | --- |
| +1 223-456-7890 | 1-223-456-7890 | +1 223 456-7890 |
| (223) 456-7890 | 1 223 456 7890 | 223.456.7890 |

*Bonus:* The first digit of the area code and the exchange code can only be 2-9, and the second digit of an area code can’t be 9. Use this information to validate the input and return a ValueError exception of invalid phone number if the number is invalid.

### Summary

* For a complete list and explanation of the regex special characters, refer to the Python documentation.
* In addition to the search and sub methods, many other methods can be used to split strings, extract more information from match objects, look for the positions of substrings in the main argument string, and precisely control the iteration of a regex search over an argument string.
* Besides the \d special sequence, which can be used to indicate a digit character, many other special sequences are listed in the documentation.
* There are also regex flags, which you can use to control some of the more esoteric aspects of how extremely sophisticated matches are carried out.

# **1.3 Matching Location Names in Text**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-11)

Data scientists frequently analyze text for patterns. Python offers a simple syntax for carrying out such string matching analyses. In Python, we can easily determine if one string is a substring of another, or if the start of a string contains some predefined text.

##### This listing Basic string matching

1

2

3

**assert** 'Boston' **in** 'Boston Marathon'

**assert** 'Boston Marathon'.startswith('Boston')

**assert** 'Boston Marathon'.endswith('Boston') == **False**

copy

Nevertheless, in more complex analyses, Python’s basic string syntax can be quite limiting. For example, there is no direct string method for executing case-insensitive substring comparison. Furthermore, Python’s string methods can’t directly distinguish between sub-characters in a string and sub-phrases in a sentence. So if we wish to determine if the phrase 'in a' is present in a sentence, then we cannot safely rely on basic matching. Otherwise, we the run the risk of incorrectly matching character sequences such as 'sin apple' or 'win attached'.

##### This listing Basic sub-string matching errors

1

2

assert 'in a' **in** 'sin apple'

assert 'in a' **in** 'win attached'

copy

To overcame these limitations, we must rely on Python’s built-in regular expression processing library, re. A **regular expression** (or **regex** for short) is a string-encoded pattern that can be compared against some text. Coded regex patterns range from simple string copies to incredibly complex formulations that very few people can decipher. In this section, we will only focus on simple regex composition and matching.

Most regex-matching in Python can be executed with the re.search function. The function takes 2 inputs; a regex pattern, and also the text against which the pattern will be matched. It returns a Match object if a match is found, and a None otherwise. The Match object contains start method and an end method. These methods will return the start-index and the end-index of the matched string in the text.

##### This listing String matching using regexes

1

2

3

4

5

6

7

8

9

10

11

12

**import** re

regex = 'Boston'

random\_text = 'Clown Patty'

match = re.**search**(regex, random\_text)

**assert** match **is** **None**

matchable\_text = 'Boston Marathon'

match = re.**search**(regex, matchable\_text)

**assert** match **is** **not** **None**

**start**, end = match.**start**(), match.**end**()

matched\_string = matchable\_text[**start**: **end**]

**assert** matched\_string == 'Boston'

copy

Additionally, case-insensitive string matching is a breeze with re.search. We simply pass re.IGNORECASE as an added flags parameter.

##### This listing Case-insensitive matching using regexes

1

2

**for** text **in** ['BOSTON', 'boston', 'BoSTOn']:

**assert** re.**search**(regex, text, flags=re.IGNORECASE) **is** **not** **None**

**1**

copy

Also, regexes permit us to match exact words, and not just substrings, using word boundary detection. The addition of the \b pattern to a regex string will capture the start and end points of words (as defined by whitespaces and punctuation). However, because the backslash is a special character in the standard Python lexicon, we must take measures to ensure the backslash is interpreted like a regular raw character. We do this by either adding another backslash to the backslash, (a rather cumbersome approach) or by preceding the string with an r literal. The later solution guarantees that the input regex get treated as a raw string during analysis.

##### This listing Word boundary matching using regexes

1

2

3

4

5

6

**for** regex **in** ['\\bin a\\b', r'\bin a\b']:

**for** text **in** ['sin apple', 'win attached']:

**assert** re.search(regex, text) **is** None

text = 'Match in a string'

**assert** re.search(regex, text) **is** **not** None

copy

Now, let us carry out a more complicated match. We’ll match against the sentence f’I visited {city} yesterday, where {city} represents one of 3 possible locations; 'Boston', 'Philadelphia', or 'San Francisco'. The correct regex syntax for executing the match is r’I visited \b(Boston|Philadelphia|San Francisco)\b yesterday'.

##### NOTE

The pipe | is an Or condition. It requires the regex to match from one of the 3 cities in our list. Furthermore, the parentheses limit the scope of the matched cities. Without them, the matched text-range would stretch beyond 'San Francisco', all the way to 'San Francisco yesterday'.

##### This listing Multi-city matching using regexes

1

2

3

4

5

6

regex = r'I visited \b(Boston|Philadelphia|San Francisco)\b yesterday.'

**assert** re.search(regex, 'I visited Chicago yesterday.') **is** None

cities = ['Boston', 'Philadelphia', 'San Francisco']

**for** city **in** cities:

**assert** re.search(regex, f'I visited {city} yesterday.') **is** **not** None

copy

On a final note, lets discuss how to run a regex search efficiently. Suppose we want to match a regex against 100 strings. For every match, re.search will transform the regex into Python PatternObject. Each such transformation is computationally costly. We’re better off executing the transformation only once using re.compile. Afterwards, we can leverage the compiled object’s built-in search function while avoiding any additional compilation.

##### NOTE

If we intend to use the compiled pattern for case-independent matching, then we must pass flags=re.IGNORECASE into re.compile.

##### This listing String matching using compiled regexes

1

2

3

4

compiled\_re = re.compile(regex)

text = 'I visited Boston yesterday.'

**for** i **in** range(1000):

**assert** compiled\_re.**search**(text) **is** **not** **None**

copy

##### COMMON REGEX MATCHING TECHNIQUES.

match = re.search(regex, text): Returns a Match object if regex is present in text, and None otherwise.

match = re.search(regex, text, flags=re.IGNORECASE): Returns a Match object if regex is present in text, and None otherwise. Matching is carried out independent of case.

match.start(): Returns the start index of a regex matched to an input text.

match.end(): Returns an end index of a regex matched to an input text.

compiled\_regex = re.compile(regex): Transforms the regex string a compiled pattern-matching matching object.

match = compiled\_regex.search(text): We leverage the compiled object’s built-in search method to match a regex against text.

re.compile('Boston'): Compiles a regex to match the string 'Boston' against the text.

re.compile('Boston', flags=re.IGNORECASE): Compiles a regex to match the string Boston against the text. The matching will be independent of text-case.

re.compile('\\bBoston\\b'): Compiles a regex to match the word Boston against the text. Word boundaries will be used to execute an exact word match.

re.compile(r'\bBoston\b'): Compiles a regex to match the word Boston against the text. The inputted regex is treated as raw string, because of the r literal. Thus, we don’t need to add additional backslashes to our \b word boundary delimiters.

re.compile(r'\b(Boston|Chicago)\b'): Compiles a regex to match either the word Boston or the word Chicago to the text.

Regex-matching allows us to find location names text. Thus, the re module will prove invaluable to solving Case Study Three.

# **1.4 Submit Your Work**

The deliverable is a Jupyter Notebook documenting your workflow as you take the *headlines.txt* file, extract the city/country names, and put the results into a pandas DataFrame. This DataFrame will allow us to quickly perform analysis on the headlines and geographic data that we will find in the next part. An example of the structure and content of the expected DataFrame is below:

Upload a link to your Jupyter Notebook (preferably hosted on GitHub) in the blank below and hit submit. After submitting, you can view an example solution in the next section.

# **Solution**

Below is an example of the structure and content of the pandas DataFrame you should create in this section. The full DataFrame has just over 600 rows—yours may have slightly more or less depending on how many cities/countries you matched in the headlines.

For an in-depth solution to this section, refer to the notebook below. Your end outcome pandas DataFrame should be similar, but your workflow may differ from the notebook. In data science, there are often multiple methods—equally effective—to achieve the same result.

* [1. Extracting City and Country Information from News Headlines.ipynb](https://liveproject-resources.s3.amazonaws.com/93/22360/2020-03-03-09-34-20/1.%20Extracting%20City%20and%20Country%20Information%20from%20News%20Headlines.ipynb)

# 2. Finding Geographic Locations of Headlines

# **2.1 Adding Latitude and Longitude Coordinates**

**Objective**

* Find the geographic location of each headline in latitude and longitude coordinates from the city/country names.

**Workflow**

1. Load in the pandas DataFrame with headline, countries, and cities.
   * If a headline contains multiple cities/countries, decide which single one to keep.
2. For each city/country, match the name to the latitude and longitude in geonamescache.
   * You can use the function *gc.get\_cities\_by\_names\_ \_(“city\_name”)*.
   * Some cities will return multiple matches with the previous function in different countries. You’ll have to decide which city to keep based on a heuristic (rule of thumb).
   * If you have trouble, work with a single problematic city until you figure it out, then write a function to apply on all headlines.
3. Add longitude and latitude coordinates to your DataFrame for each headline.
   * It will be helpful to get the *countrycode* of each headline at this point.
   * If you were not able to find many countries, think about dropping the column. You also need to decide what to do with headlines that have no coordinates.
   * You should end up with over 600 headlines that have geographic coordinates.

**Importance to project**

* In this section, we are augmenting the original data with external information. Our hypothesis is that there may be groups of headlines in close proximity referring to similar diseases. We will use the locations from this part to test our hypothesis by clustering headlines in the next part.
* If we just searched the headlines for diseases, we might find repeated but geographically unrelated (not close to each other) mentions. These could lead to false positives that are only isolated cases.

**Notes**

* Make sure you are inspecting your data statistically and visually at each step. Compute summary statistics and make basic distribution plots (histograms) to search for anomalies.
* When you find issues (we all make mistakes), go back and correct them. You may find yourself repeatedly carrying out data cleaning operations; this is normal for a data science project!
* Keep in mind that the more accurate the geographic locations of the headlines, the better will be the clustering and the overall conclusions. Checking your data one more time is worth it when lives are at stake.

**Resources**

* [Python and Pandas](https://liveproject.manning.com/module/93_3_2/discovering-disease-outbreaks-from-news-headlines/2--finding-geographic-locations-of-headlines/2-2--python-and-pandas?), Chapter 24 in The Quick Python Book, Third Edition, by Naomi Ceder in section 2.2 can help you with manipulating pandas dataframes.
* Chapter 11, [Location Tracking Using GeoNames Cache](https://liveproject.manning.com/module/93_3_3/discovering-disease-outbreaks-from-news-headlines/2--finding-geographic-locations-of-headlines/2-3--location-tracking-using-geonamescache?), from Data Science Bookcamp by Leonard Apeltsin in section 2.3 can help you with matching cities/countries to geographic coordinates.

# **Python and Pandas**

excerpt from The Quick Python Book, Third Edition | Naomi Ceder[go to book](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-24)

#### Python’s advantages for exploring data

Python has become one of the leading languages for data science and continues to grow in that area. As I’ve mentioned, however, Python isn’t always the fastest language in terms of raw performance. Conversely, some data-crunching libraries, such as NumPy, are largely written in C and heavily optimized to the point that speed isn’t an issue. In addition, considerations such as readability and accessibility often outweigh pure speed; minimizing the amount of developer time needed is often more important. Python is readable and accessible, and both on its own and in combination with tools developed in the Python community, it’s an enormously powerful tool for manipulating and exploring data.

#### Python can be better than a spreadsheet

Spreadsheets have been the tools of choice for ad-hoc data manipulation for decades. People who are skilled with spreadsheets can make them do truly impressive tricks: spreadsheets can combine different but related data sets, pivot tables, use lookup tables to link data sets, and much more. But although people everywhere get a vast amount of work done with them every day, spreadsheets do have limitations, and Python can help you go beyond those limitations.

One limitation that I’ve already alluded to is the fact that most spreadsheet software has a row limit—currently, about 1 million rows, which isn’t enough for many data sets. Another limitation is the central metaphor of the spreadsheet itself. Spreadsheets are two-dimensional grids, rows and columns, or at best stacks of grids, which limits the ways you can manipulate and think about complex data.

With Python, you can code your way around the limitations of spreadsheets and manipulate data the way you want. You can combine Python data structures such as lists, tuples, sets, and dictionaries in endlessly flexible ways, or you can create your own classes to package both data and behavior exactly the way you need.

### Jupyter notebook

Probably one of the most compelling tools for exploring data with Python doesn’t augment what the language itself does, but changes the way you use the language to interact with your data. Jupyter notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations, and explanatory text. Although several other languages are now supported, it originated in connection with IPython, an alternative shell for Python developed by the scientific community.

What makes Jupyter such a convenient and powerful tool is the fact that you interact with it in a web browser. It lets you combine text and code, as well as modify and execute your code interactively. You can not only run and modify code in chunks, but also save and share the notebooks with others.

The best way to get a feel for what Jupyter notebook can do is start playing with it. It’s fairly easy to run a Jupyter process locally on your machine, or you can access online versions. For some options, see the sidebar on ways to run Jupyter.

##### WAYS TO RUN JUPYTER

*Jupyter online:* Accessing online instances of Jupyter is one of the easiest ways to get started. Currently, Project Jupyter, the community behind Jupyter, hosts free notebooks at <https://jupyter.org/try>. You can also find demo notebooks and kernels for other languages. At this writing, you can also access free notebooks on Microsoft’s Azure platform at [https://notebooks.azure.com](https://notebooks.azure.com/), and many other ways are available.

*Jupyter locally:* Although using an online instance is quite convenient, it’s not very much work to set up your own instance of Jupyter on your computer. Usually for local versions, you point your browser to localhost:8888.

If you use Docker, you have several containers to choose among. To run the data science notebook container, use something like this:

Top of Form

1

docker run -**it** --rm -p 8888:8888 jupyter/datascience-notebook

Bottom of Form

copy

If you’d rather run directly on your system, it’s easy to install and run Jupyter in a virtualenv.

*macOS and Linux systems:* First, open a command window, and enter the following commands:

Top of Form

1

2

3

4

5

> python3 -m venv jupyter

> cd jupyter

> source bin/activate

> pip install jupyter

> jupyter-notebook

Bottom of Form

copy

*Windows systems:*

Top of Form

1

2

3

4

5

> python3 -m venv jupyter

> cd jupyter

> Scripts/bin/activate

> pip install jupyter

> Scripts/jupyter-notebook

Bottom of Form

copy

The last command should run the Jupyter notebook web app and open a browser window pointing at it.

#### Starting a kernel

When you have Jupyter installed, running, and open in your browser, you need to start a Python kernel. One nice thing about Jupyter is that it lets you run multiple kernels at the same time. You can run kernels for different versions of Python and for other languages such as R, Julia, and even Ruby.

Starting a kernel is easy. Just click the new button and select Python 3 ([this figure](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-24/ch24fig01)).

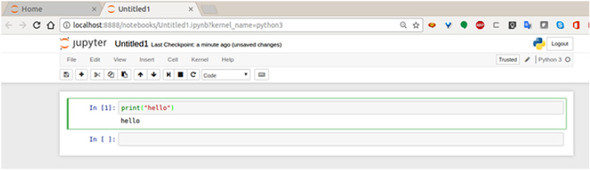
##### Starting a Python kernel



#### Executing code in a cell

When you have a kernel running, you can start entering and running Python code. Right away, you’ll notice a few differences from the ordinary Python command shell. You won’t get the >>> prompt that you see in the standard Python shell, and pressing Enter just adds new lines in the cell. To execute the code in a cell, illustrated in [this figure](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-24/ch24fig02), choose Cell > Run Cells, click the Run button immediately to the left of the down arrow on the button bar, or use the key combination Alt-Enter. After you use Jupyter notebook a little bit, it’s quite likely that the Alt-Enter key combination will become quite natural to you.

##### Executing code in a notebook cell



You can test how it works by entering some code or an expression into the first cell of your new notebook and then pressing Alt-Enter.

As you can see, any output is shown immediately below the cell, and a new cell is created and ready for your next input. Also note that each cell that’s executed is numbered in the order in which it’s executed.

##### TRY THIS: USING JUPYTER NOTEBOOK

Enter some code in the notebook and experiment with running it. Check out the Edit, Cell, and Kernel menus to see what options are there. When you have a little code running, use the Kernel menu to restart the kernel, repeat your steps, and then use the Cell menu to rerun the code in all of the cells.

### Python and pandas

In the course of exploring and manipulating data, you perform quite a few common operations, such as loading data into a list or dictionary, cleaning data, and filtering data. Most of these operations are repeated often, have to be done in standard patterns, and are simple and often tedious. If you think that this combination is a strong reason to automate those tasks you’re not alone. One of the now-standard tools for handling data in Python—pandas—was created to automate the boring heavy lifting of handling data sets.

#### Why you might want to use pandas

pandas was created to make manipulating and analyzing tablular or relational data easy by providing a standard framework for holding the data, with convenient tools for frequent operations. As a result, it’s almost more of an extension to Python than a library, and it changes the way you can interact with data. The plus side is that after you grok how pandas work, you can do some impressive things and save a lot of time. It does take time to learn how to get the most from pandas, however. As with many tools, if you use pandas for what it was designed for, it excels. The simple examples I show you in the following sections should give you a rough idea whether pandas is a tool that’s suited for your use cases.

#### Installing pandas

pandas is easy to install with pip. It’s often used along with matplotlib for plotting, so you can install both tools from the command line of your Jupyter virtual environment with this code:

Top of Form

1

> pip **install** pandas matplotlib

Bottom of Form

copy

From a cell in a Jupyter notebook, you can use

Top of Form

1

In [ ]: !pip **install** pandas matplotlib

Bottom of Form

copy

If you use pandas, life will be easier if you use the following three lines:

Top of Form

1

2

3

%matplotlib inline

**import** pandas **as** pd

**import** numpy **as** np

Bottom of Form

copy

The first line is a Jupyter “magic” function that enables matplotlib to plot data in the cell where your code is (which is very useful). The second line imports pandas with the alias of pd, which is both easier to type and common among pandas users; the last line also imports numpy. Although pandas depends quite a bit on numpy, you won’t use it explicitly in the following examples, but it’s reasonable to get into the habit of importing it anyway.

#### Data frames

One basic structure that you get with pandas is a data frame. A *data frame* is a two-dimensional grid, rather similar to a relational database table except in memory. Creating a data frame is easy; you give it some data. To keep things absolutely simple, give it a 3 × 3 grid of numbers as the first example. In Python, such a grid is a list of lists:

Top of Form

1

2

3

4

grid = [[1,2,3], [4,5,6], [7,8,9]]

print(grid)

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Bottom of Form

copy

Sadly, in Python the grid won’t look like a grid unless you make some additional effort. So see what you can do with the same grid as a pandas data frame:

Top of Form

1

2

3

4

5

6

7

8

**import** pandas as pd

df = pd.DataFrame(grid)

print(df)

0 1 2

0 1 2 3

1 4 5 6

2 7 8 9

Bottom of Form

copy

That code is fairly straightforward; all you needed to do was turn your grid into a data frame. You’ve gained a more gridlike display, and now you have both row and column numbers. It’s often rather bothersome to keep track of what column number is what, of course, so give your columns names:

Top of Form

1

2

3

4

5

6

7

df = pd.DataFrame(grid, columns=["one", "two", "three"] )

print(df)

one two three

0 1 2 3

1 4 5 6

2 7 8 9

Bottom of Form

copy

You may wonder whether naming the columns has any benefit, but the column names can be put to use with another pandas trick: the ability to select columns by name. If you want the contents only of column "two", for example, you can get it very simply:

Top of Form

1

2

3

4

5

print(df["two"])

0 2

1 5

2 8

Name: two, dtype: int64

Bottom of Form

copy

Here, you’ve already saved time in comparison to Python. To get only column two of your grid, you’d need to use a list comprehension while also remembering to use a zero-based index (and you still wouldn’t get the nice output):

Top of Form

1

2

print([x[1] **for** x **in** grid])

[2, 5, 8]

Bottom of Form

copy

You can loop over data frame column values just as easily as the list you got by using a comprehension:

Top of Form

1

2

3

4

5

for x in df["two"]:

print(x)

2

5

8

Bottom of Form

copy

That’s not bad for a start, but by using a list of columns in double brackets, you can do better, getting a subset of the data frame that’s another data frame. Instead of getting the middle column, get the first and last columns of your data frame as another data frame:

Top of Form

1

2

3

4

5

6

edges = df[["one", "three"]]

print(edges)

one three

0 1 3

1 4 6

2 7 9

Bottom of Form

copy

A data frame also has several methods that apply the same operation and argument to every item in the frame. If you want to add two to every item in the data frame’s edges, you could use the add() method:

Top of Form

1

2

3

4

5

print(edges.add(2))

one three

0 3 5

1 6 8

2 9 11

Bottom of Form

copy

Here again, it’s possible to get the same result by using list comprehensions and/or nested loops, but those techniques aren’t as convenient. It’s pretty easy to see how such functionality can make life easier, particularly for someone who’s more interested in the information that the data contains than in the process of manipulating it.

### Data cleaning

In earlier chapters, I discussed a few ways to use Python to clean data. Now that I’ve added pandas to the mix, I’ll show you examples of how to use its functionality to clean data. As I present the following operations, I also refer to ways that the same operation might be done in plain Python, both to illustrate how using pandas is different and to show why pandas isn’t right for every use case (or user, for that matter).

#### Loading and saving data with pandas

pandas has an impressive collection of methods to load data from different sources. It supports several file formats (including fixed-width and delimited text files, spreadsheets, JSON, XML, and HTML), but it’s also possible to read from SQL databases, Google BiqQuery, HDF, and even clipboard data. You should be aware that many of these operations aren’t actually part of pandas itself; pandas relies on having other libraries installed to handle those operations, such as SQLAlchemy for reading from SQL databases. This distinction matters mostly if something goes wrong; quite often, the problem that needs to be fixed is outside pandas, and you’re left to deal with the underlying library.

Reading a JSON file with the read\_json() method is simple:

Top of Form

1

mars = pd.read\_json("mars\_data\_01.json")

Bottom of Form

copy

This code gives you a data frame like this:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

report

abs\_humidity None

atmo\_opacity Sunny

ls 296

max\_temp -1

max\_temp\_fahrenheit 30.2

min\_temp -72

min\_temp\_fahrenheit -97.6

pressure 869

pressure\_string Higher

season Month 10

sol 1576

sunrise 2017-01-11T12:31:00Z

sunset 2017-01-12T00:46:00Z

terrestrial\_date 2017-01-11

wind\_direction --

wind\_speed None

Bottom of Form

copy

For another example of how simple reading data into pandas is, load some data from the CSV file of temperature data from [chapter 21](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-21/ch21) and from the JSON file of Mars weather data used in [this chapter](https://livebook.manning.com/book/the-quick-python-book-third-edition/chapter-22/ch22). In the first case, use the read\_csv() method:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

temp = pd.read\_csv("temp\_data\_01.csv")

4 5 6 7 8 9 10 11 12 13 14 \

0 1979/01/01 17.48 994 6.0 30.5 2.89 994 -13.6 15.8 NaN 0

1 1979/01/02 4.64 994 -6.4 15.8 -9.03 994 -23.6 6.6 NaN 0

2 1979/01/03 11.05 994 -0.7 24.7 -2.17 994 -18.3 12.9 NaN 0

3 1979/01/04 9.51 994 0.2 27.6 -0.43 994 -16.3 16.3 NaN 0

4 1979/05/15 68.42 994 61.0 75.1 51.30 994 43.3 57.0 NaN 0

5 1979/05/16 70.29 994 63.4 73.5 48.09 994 41.1 53.0 NaN 0

6 1979/05/17 75.34 994 64.0 80.5 50.84 994 44.3 55.7 82.60 2

7 1979/05/18 79.13 994 75.5 82.1 55.68 994 50.0 61.1 81.42 349

8 1979/05/19 74.94 994 66.9 83.1 58.59 994 50.9 63.2 82.87 78

15 16 17

0 NaN NaN 0.0000

1 NaN NaN 0.0000

2 NaN NaN 0.0000

3 NaN NaN 0.0000

4 NaN NaN 0.0000

5 NaN NaN 0.0000

6 82.4 82.8 0.0020

7 80.2 83.4 0.3511

8 81.6 85.2 0.0785

**1**

copy

Clearly, loading the file in a single step is appealing, and you can see that pandas had no issues loading the file. You can also see that the empty first column has been translated into NaN (not a number). You do still have the same issue with 'Missing' for some values, and in fact it might make sense to have those 'Missing' values converted to NaN:

Top of Form

1

temp = pd.read\_csv("temp\_data\_01.csv", na\_values=['Missing'])

Bottom of Form

copy

The addition of the na\_values parameter controls what values will be translated to NaN on load. In this case, you added the string 'Missing' so that the row of the data frame was translated from

Top of Form

1

2

NaN Illinois 17 Jan 01, 1979 1979/01/01 17.48 994 6.0 30.5 2.89994

-13.6 15.8 Missing 0 Missing Missing 0.00%

Bottom of Form

copy

to

Top of Form

1

2

NaN Illinois 17 Jan 01, 1979 1979/01/01 17.48 994 6.0 30.5 2.89994

-13.6 15.8 NaN0 NaN NaN 0.00%

Bottom of Form

copy

This technique can be particularly useful if you have one of those data files in which, for whatever reason, “no data” is indicated in a variety of ways: NA, N/A, ?, -, and so on. To handle a case like that, you can inspect the data to find out what’s used and then reload it, using the na\_values parameter to standardize all those variations as NaN.

##### Saving data

If you want to save the contents of a data frame, a pandas data frame has a similarly broad collection of methods. If you take your simple grid data frame, you can write it in several ways. This line

1

df.**to**\_csv("df\_out.csv", index=False)

**1**

copy

writes a file that looks like this:

Top of Form

1

2

3

4

one,two,three

1,2,3

4,5,6

7,8,9

Bottom of Form

copy

Similarly, you can transform a data grid to a JSON object or write it to a file:

1

2

3

df.to\_json()

'{"one":{"0":1,"1":4,"2":7},"two":{"0":2,"1":5,"2":8},"three":{"0":3,"1":6,"2

":9}}'

**1**

copy

#### Data cleaning with a data frame

Converting a particular set of values to NaN on load is a very simple bit of data cleaning that pandas makes trivial. Going beyond that, data frames support several operations that can make data cleaning less of a chore. To see how this works, reopen the temperature CSV file, but this time, instead of using the headers to name the columns, use the range() function with the names parameter to give them numbers, which will make referring to them easier. You also may recall from an earlier example that the first field of every line—the "Notes" field—is empty and loaded with NaN values. Although you could ignore this column, it would be even easier if you didn’t have it. You can use the range() function again, this time starting from 1, to tell pandas to load all columns except the first one. But if you know that all of your values are from Illinois and you don’t care about the long-form date field, you could start from 4 to make things much more manageable:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

temp = pd.read\_csv("temp\_data\_01.csv", na\_values=['Missing'], header=0,

names=range(18), usecols=range(4,18))

print(temp)

4 5 6 7 8 9 10 11 12 13 14 \

0 1979/01/01 17.48 994 6.0 30.5 2.89 994 -13.6 15.8 NaN 0

1 1979/01/02 4.64 994 -6.4 15.8 -9.03 994 -23.6 6.6 NaN 0

2 1979/01/03 11.05 994 -0.7 24.7 -2.17 994 -18.3 12.9 NaN 0

3 1979/01/04 9.51 994 0.2 27.6 -0.43 994 -16.3 16.3 NaN 0

4 1979/05/15 68.42 994 61.0 75.1 51.30 994 43.3 57.0 NaN 0

5 1979/05/16 70.29 994 63.4 73.5 48.09 994 41.1 53.0 NaN 0

6 1979/05/17 75.34 994 64.0 80.5 50.84 994 44.3 55.7 82.60 2

7 1979/05/18 79.13 994 75.5 82.1 55.68 994 50.0 61.1 81.42 349

8 1979/05/19 74.94 994 66.9 83.1 58.59 994 50.9 63.2 82.87 78

15 16 17

0 NaN NaN 0.00%

1 NaN NaN 0.00%

2 NaN NaN 0.00%

3 NaN NaN 0.00%

4 NaN NaN 0.00%

5 NaN NaN 0.00%

6 82.4 82.8 0.20%

7 80.2 83.4 35.11%

8 81.6 85.2 7.85%

**1**

copy

Now you have a data frame that has only the columns you might want to work with. But you still have an issue: the last column, which lists the percentage of coverage for the heat index, is still a string ending with a percentage sign rather than an actual percentage. This problem is apparent if you look at the first row’s value for column 17:

Top of Form

1

2

temp[17][0]

'0.00%'

Bottom of Form

copy

To fix this problem, you need to do two things: Remove the % from the end of the value and then cast the value from string to a number. Optionally, if you want to represent the resulting percentage as a fraction, you need to divide it by 100. The first bit is simple because pandas lets you use a single command to repeat an operation on a column:

Top of Form

1

2

3

temp[17] = temp[17].str.strip("%")

temp[17][0]

'0.00'

Bottom of Form

copy

This code takes the column and calls a string strip() operation on it to remove the trailing %. Now when you look at the first value in the column (or any of the other values), you see that the offending percentage sign is gone. It’s also worth noting that you could have used other operations, such as replace("%", ""), to achieve the same result.

The second operation is to convert the string to a numeric value. Again, pandas lets you perform this operation with one command:

Top of Form

1

2

3

temp[17] = pd.to\_numeric(temp[17])

temp[17][0]

0.0

Bottom of Form

copy

Now the values in column 17 are numeric, and if you want to, you can use the div() method to finish the job of turning those values into fractions:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

temp[17] = temp[17].div(100)

temp[17]

0 0.0000

1 0.0000

2 0.0000

3 0.0000

4 0.0000

5 0.0000

6 0.0020

7 0.3511

8 0.0785

Name: 17, dtype: float64

Bottom of Form

copy

In fact, it would be possible to achieve the same result in a single line by chaining the three operations together:

Top of Form

1

temp[17] = pd.**to**\_numeric(temp[17].str.strip("%")).div(100)

Bottom of Form

copy

This example is very simple, but it gives you an idea of the convenience that pandas can bring to cleaning your data. pandas has a wide variety of operations for transforming data, as well as the ability to use custom functions, so it would be hard to think of a scenario in which you couldn’t streamline data cleaning with pandas.

Although the number of options is almost overwhelming, a wide variety of tutorials and videos is available, and the documentation at [http://pandas.pydata.org](http://pandas.pydata.org/) is excellent.

##### TRY THIS: CLEANING DATA WITH AND WITHOUT PANDAS

Experiment with the operations. When the final column has been converted to a fraction, can you think of a way to convert it back to a string with the trailing percentage sign?

By contrast, load the same data into a plain Python list by using the csv module, and apply the same changes by using plain Python.

### Data aggregation and manipulation

The preceding examples probably gave you some idea of the many options pandas gives you for performing fairly complex operations on your data with only a few commands. As you might expect, this level of functionality is also available for aggregating data. In this section, I walk through a few simple examples of aggregating data to illustrate some of the many possibilities. Although many options are available, I focus on merging data frames, performing simple data aggregation, and grouping and filtering.

#### Merging data frames

Quite often in the course of handling data, you need to relate two data sets. Suppose that you have one file containing the number of sales calls made per month by members of a sales team, and in another file, you have the dollar amounts of the sales in each of their territories:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

calls = pd.read\_csv("sales\_calls.csv")

print(calls)

Team member Territory Month Calls

0 Jorge 3 1 107

1 Jorge 3 2 88

2 Jorge 3 3 84

3 Jorge 3 4 113

4 Ana 1 1 91

5 Ana 1 2 129

6 Ana 1 3 96

7 Ana 1 4 128

8 Ali 2 1 120

9 Ali 2 2 85

10 Ali 2 3 87

11 Ali 2 4 87

revenue = pd.read\_csv("sales\_revenue.csv")

print(revenue)

Territory Month Amount

0 1 1 54228

1 1 2 61640

2 1 3 43491

3 1 4 52173

4 2 1 36061

5 2 2 44957

6 2 3 35058

7 2 4 33855

8 3 1 50876

9 3 2 57682

10 3 3 53689

11 3 4 49173

Bottom of Form

copy

Clearly, it would be very useful to link revenue and team-member activity. These two files are very simple, yet merging them with plain Python isn’t entirely trivial. pandas has a function to merge two data frames:

Top of Form

1

calls\_revenue = pd.merge(calls, revenue, on=['Territory', 'Month'])

Bottom of Form

copy

The merge function creates a new data frame by joining the two frames on the columns specified in the column field. The merge function works similarly to a relational-database join, giving you a table that combines the columns from the two files:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

14

print(calls\_revenue)

Team member Territory Month Calls Amount

0 Jorge 3 1 107 50876

1 Jorge 3 2 88 57682

2 Jorge 3 3 84 53689

3 Jorge 3 4 113 49173

4 Ana 1 1 91 54228

5 Ana 1 2 129 61640

6 Ana 1 3 96 43491

7 Ana 1 4 128 52173

8 Ali 2 1 120 36061

9 Ali 2 2 85 44957

10 Ali 2 3 87 35058

11 Ali 2 4 87 33855

Bottom of Form

copy

In this case, you have a one-to-one correspondence between the rows in the two fields, but the merge function can also do one-to-many and many-to-many joins, as well as right and left joins.

##### QUICK CHECK: MERGING DATA SETS

How would you go about merging to data sets like the ones in the Python example?

#### Selecting data

It can also be useful to select or filter the rows in a data frame based on some condition. In the example sales data, you may want to look only at territory 3, which is also easy:

Top of Form

1

2

3

4

5

6

7

print(calls\_revenue[calls\_revenue.Territory==3])

Team member Territory Month Calls Amount

0 Jorge 3 1 107 50876

1 Jorge 3 2 88 57682

2 Jorge 3 3 84 53689

3 Jorge 3 4 113 49173

Bottom of Form

copy

In this example, you select only rows in which the territory is equal to 3 but using exactly that expression, revenue.Territory==3, as the index for the data frame. From the point of view of plain Python, such use is nonsense and illegal, but for a pandas data frame, it works and makes for a much more concise expression.

More complex expressions are also allowed, of course. If you want to select only rows in which the amount per call is greater than 500, you could use this expression instead:

Top of Form

1

2

3

4

5

6

7

print(calls\_revenue[calls\_revenue.Amount/calls\_revenue.Calls>500])

Team member Territory Month Calls Amount

1 Jorge 3 2 88 57682

2 Jorge 3 3 84 53689

4 Ana 1 1 91 54228

9 Ali 2 2 85 44957

Bottom of Form

copy

Even better, you could calculate and add that column to your data frame by using a similar operation:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

calls\_revenue['Call\_Amount'] = calls\_revenue.Amount/calls\_revenue.Calls

print(calls\_revenue)

Team member Territory Month Calls Amount Call\_Amount

0 Jorge 3 1 107 50876 475.476636

1 Jorge 3 2 88 57682 655.477273

2 Jorge 3 3 84 53689 639.154762

3 Jorge 3 4 113 49173 435.159292

4 Ana 1 1 91 54228 595.912088

5 Ana 1 2 129 61640 477.829457

6 Ana 1 3 96 43491 453.031250

7 Ana 1 4 128 52173 407.601562

8 Ali 2 1 120 36061 300.508333

9 Ali 2 2 85 44957 528.905882

10 Ali 2 3 87 35058 402.965517

11 Ali 2 4 87 33855 389.137931

Bottom of Form

copy

Again, note that pandas’s built-in logic replaces a more cumbersome structure in plain Python.

##### QUICK CHECK: SELECTING IN PYTHON

What Python code structure would you use to select only rows meeting certain conditions?

#### Grouping and aggregation

As you might expect, pandas has plenty of tools to summarize and aggregate data as well. In particular, getting the sum, mean, median, minimum, and maximum values from a column uses clearly named column methods:

Top of Form

1

2

3

4

5

6

7

8

9

10

print(**calls\_revenue**.Calls.sum())

print(**calls\_revenue**.Calls.mean())

print(**calls\_revenue**.Calls.median())

print(**calls\_revenue**.Calls.max())

print(**calls\_revenue**.Calls.min())

1215

101.25

93.5

129

84

Bottom of Form

copy

If, for example, you want to get all of the rows in which the amount per call is above the median, you can combine this trick with the selection operation:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

print(calls\_revenue.Call\_Amount.median())

print(calls\_revenue[calls\_revenue.Call\_Amount >=

calls\_revenue.Call\_Amount.median()])

464.2539427570093

Team member Territory Month Calls Amount Call\_Amount

0 Jorge 3 1 107 50876 475.476636

1 Jorge 3 2 88 57682 655.477273

2 Jorge 3 3 84 53689 639.154762

4 Ana 1 1 91 54228 595.912088

5 Ana 1 2 129 61640 477.829457

9 Ali 2 2 85 44957 528.905882

Bottom of Form

copy

In addition to being able to pick out summary values, it’s often useful to group the data based on other columns. In this simple example, you can use the groupby method to group your data. You may want to know the total calls and amounts by month or by territory, for example. In those cases, use those fields with the data frame’s groupby method:

Top of Form

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

print(calls\_revenue[['Month', 'Calls', 'Amount']].groupby(['Month']).sum())

Calls Amount

Month

1 318 141165

2 302 164279

3 267 132238

4 328 135201

print(calls\_revenue[['Territory', 'Calls',

'Amount']].groupby(['Territory']).sum())

Calls Amount

Territory

1 444 211532

2 379 149931

3 392 211420

Bottom of Form

copy

In each case, you select the columns that you want to aggregate, group them by the values in one of those columns, and (in this case) sum the values for each group. You could also use any of the other methods mentioned earlier in this chapter.

Again, all these examples are simple, but they illustrate a few of the options you have for manipulating and selecting data with pandas. If these ideas resonate with your needs, you can learn more by studying the pandas documentation at [http://pandas.pydata.org](http://pandas.pydata.org/).

##### TRY THIS: GROUPING AND AGGREGATING

Experiment with pandas and the data in previous examples. Can you get the calls and amounts by both team member and month?

# **2.2 Location Tracking Using GeoNamesCache**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-11)

The GeoNames database ([geonames.org](http://geonames.org/)) is an excellent resource for obtaining geographic data.. GeoNames contains over eleven million place-names spanning all the countries in the World. Beyond just place-names, GeoNames also stores valuable information such as latitude and longitude. Thus, we can leverage the database to determine the precise geographic locations of cities and countries discovered in text.

How do we access the GeoNames data? Well, we could manually download the GeoNames data dump, ([download.geonames.org/export/dump/](http://download.geonames.org/export/dump/)), parse it and then store the output data structure. That would take a lot of work. Fortunately, someone has already done the hard work for us by creating the GeoNamesCache library.

GeoNamesCache is designed to efficiently retrieve data pertaining to continents, countries, and cities, as well US counties and US states. The library provides eight easy-to-use methods to support the access to location data. These method are get\_continents, get\_countries, get\_cities, get\_countries\_by\_name, get\_cities\_by\_name, get\_us\_counties. Lets install the library and explore its usage in more detail. We’ll begin by initializing a GeonamesCache location-tracking object.

##### NOTE

Call "pip install geonamescache" from the command-line terminal in order to install the GeoNamesCache library.

##### This listing Initializing a GeonamesCache object

1

2

**from** geonamescache **import** GeonamesCache

gc = GeonamesCache()

copy

Lets use our gc object to explore the 7 continents. We’ll run gc.get\_continents() in order to retrieve a dictionary of continent-related information. Afterwards, we’ll investigate the dictionary’s structure by printing out its keys.

##### This listing Fetching all 7 continents from GeoNamesCache

1

2

continent\_dictionary = gc.get\_continents()

**print**(continent\_dictionary.keys())

copy

1

**dict\_keys**(['AF', 'AS', 'EU', 'NA', 'OC', 'SA', 'AN'])

copy

The dictionary keys represent shorthand encoding of continent names, in which Africa is transformed into 'AF', and North America is transformed into 'NA'. Lets check the values mapped to every key by passing in the code for North America.

##### NOTE

continents is a nested dictionary. Thus, the 7 top-level keys map to content-specific dictionary structures. In the code below, we’ll output the content-specific keys contained within the continents['NA'] dictionary.

##### This listing Fetching North America from GeoNamesCache

1

2

north\_america = continents['NA']

**print**(north\_america.keys())

copy

1

**dict\_keys**(['lng', 'geonameId', 'timezone', 'bbox', 'toponymName', 'asciiName', 'astergdem', 'fcl', 'population', 'wikipediaURL', 'adminName5', 'srtm3', 'adminName4', 'adminName3', 'alternateNames', 'cc2', 'adminName2', 'name', 'fclName', 'fcodeName', 'adminName1', 'lat', 'fcode', 'continentCode'])

copy

Many of the north\_america data elements represent the various naming schemes for the North American continent. Such information is not very useful.

##### This listing Printing North America’s naming schemas

1

2

**for** **name\_key** **in** ['name', 'asciiName', 'toponymName']:

**print**(**north\_america**[name\_key])

copy

1

2

3

**North** America

**North** America

**North** America

copy

However, other elements do hold more value. For example, the 'lat' and 'lng' keys map to the latitude and longitude of the central-most location in the continent. We can utilize these coordinates to plot a map projection centered at the heart of North America.

##### This listing Mapping North America’s central coordinates

1

2

3

4

5

6

7

8

9

latitude = float(north\_america['lat'])

longitude = float(north\_america['lng'])

fig = plt.figure(figsize=(12, 8))

map\_plotter = Basemap(**projection**='ortho',lat\_0=40, **lon\_0**=-95)

map\_plotter.scatter([longitude], [latitude], **latlon**=True, **s**=200)

map\_plotter.drawcoastlines()

map\_plotter.drawcountries()

plt.show()

**1**

copy

##### The central North American latitude and longitude plotted on a map of North America.



### Accessing Country Information

The ability to access continental data is useful, though our primary concern remains the analysis of cities and countries. We can analyze countries using the get\_countries method. It returns a dictionary whose 2-character keys encode the names of 252 different countries. As with the continents, the country-codes capture the abbreviated country names. For example, the code for Canada is 'CA' and the code for United States is 'US'. Accessing gc.get\_countries()['US'] will return a dictionary containing useful US data. Lets examine that data output.

##### This listing Fetching US data from GeoNamesCache

1

2

3

4

5

6

7

countries = gc.get\_countries()

num\_countries = len(countries)

**print**(f"GeonamesCache holds data for {num\_countries} countries.")

us\_data = countries['US']

**print**("The following data pertains to the United States:")

**print**(us\_data)

copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

GeonamesCache holds data for 252 countries.

The following data pertains **to** the United States:

{'geonameid': 6252001,

'name': 'United States',

'iso': 'US',

'iso3': 'USA',

'isonumeric': 840,

'fips': 'US',

'continentcode': 'NA',

'capital': 'Washington',

'areakm2': 9629091,

'population': 310232863,

'tld': '.us',

'currencycode': 'USD',

'currencyname': 'Dollar',

'phone': '1',

'postalcoderegex': '^\\d{5}(-\\d{4})?$',

'languages': 'en-US,es-US,haw,fr',

'neighbours': 'CA,MX,CU'}

**1**

**2**

**3**

**4**

**5**

**6**

**7**

copy

The outputted country data holds many useful elements, such as the country’s capital, currency, square area, spoken languages, and population size. Regrettably, GeoNamesCache does fail to provide the central latitude and longitude associated with the country’s area. However, as we will shortly discover, a country’s centrality can be estimated using city coordinates.

Additionally, there is valuable information within each country’s 'neighbours' element (the spelling is written in British English). The 'neighbours' key maps to a comma-delimited string of country codes that signify all neighboring territories. We can obtain more details about each neighbor by splitting the string and passing the codes into the 'countries' dictionary.

##### This listing Fetching neighboring countries

1

2

3

us\_neighbors = us\_data['neighbours']

for neighbor\_code in us\_neighbors.split(','):

print(countries[neighbor\_code]['name'])

copy

1

2

3

**Canada**

**Mexico**

**Cuba**

copy

According to GeoNamesCache, the immediate neighbors of the United States are Canada, Mexico, and Cuba. We can all agree on the first 2 locations, though whether Cuba is an actual neighbor remains questionable. Cuba does not directly border the United States. Also, if Caribbean island-nation the indeed a neighbor, then why is Haiti not included in that list? More importantly, how did Cuba get labeled as a neighbor in the first place? Well, GeoNames is a collaborative project run by a community of editors (like a location-focused Wikipedia). At some point, an editor had decided that Cuba is a neighbor the US. Some might disagree with this decision, which is why it is important to remember that GeoNames is not a golden-standard repository of location information. Instead, it is a tool for quickly accessing large quantities of location data. Some of that data might be imprecise. Thus, please be cautious in your use of GeoNamesCache

The get\_countries method requires a country’s 2-character code. However, for most countries, we will not know the code. Fortunately, we can query all countries by name using the get\_countries\_by\_names method. This method returns a dictionary whose elements are country names rather than codes.

##### This listing Fetching countries by name

1

2

result = gc.get\_countries\_by\_names()['United States']

**assert** result == countries['US']

copy

### Accessing City Information

Now, lets turn our attention to analyzing cities. The get\_cities method returns a dictionary whose keys are unique ids mapping back to city data. Below, we’ll output that data for a single city.

##### This listing Fetching cities from GeoNamesCache

1

2

3

4

5

cities = gc.get\_cities()

num\_cities = len(cities)

**print**(f"GeoNamesCache holds data for {num\_cities} total cities")

city\_id = list(cities.keys())[0]

**print**(cities[city\_id])

copy

cities is a dictionary mapping a unique city\_id to geographic information.

1

2

3

4

5

6

7

{'geonameid': 3041563,

'name': 'Andorra la Vella',

'latitude': 42.50779,

'longitude': 1.52109,

'countrycode': 'AD',

'population': 20430,

'timezone': 'Europe/Andorra'}

**1**

**2**

**3**

**4**

**5**

**6**

**7**

copy

The data for each city contains the city name, its latitude and longitude, its population, and also the reference code for the country where that city is located. By utilizing the country code, we can create a new mapping between a country and all its territorial cities. Lets isolate and count all US cities stored in GeoNamesCache.

##### NOTE

As we’ve discussed, GeoNames is not perfect. Certain US cities might be missing from the database.

##### This listing Fetching US cities from GeoNamesCache

1

2

3

4

us\_cities = [city for city **in** cities.values()

**if** city['countrycode'] == 'US']

num\_us\_cities = **len**(us\_cities)

**print**(f"GeoNamesCache holds data for {num\_us\_cities} US cities.")

copy

1

GeoNamesCache holds **data** for 3248 US cities

copy

GeoNamesCache contains information on over 3,000 US cities. Each city’s data-dictionary contains a latitude and a longitude. Lets find the average US latitude and longitude. This average will approximate the central coordinates of the United States.

Please note that the approximation will not be perfect. The calculated average will not take into account the curvature of the Earth. Additionally, the approximation will be inappropriately weighted by city-location. For instance, a disproportionate number of US cities are located near the American East coast. Thus, a final approximation of the US center will be overly skewed towards the East. In the code below, we will approximate and plot the US center, while remaining fully aware that our approximation is not ideal.

##### This listing Approximating US central coordinates

1

2

3

4

5

6

7

8

9

10

11

12

13

center\_lat = np.mean([city['latitude']

**for** city **in** us\_cities])

center\_lon = np.mean([city['longitude']

**for** city **in** us\_cities])

fig = plt.figure(figsize=(12, 8))

map\_lcc = Basemap(**projection**='lcc', **lon\_0**=-95, **llcrnrlon**=-119,

**llcrnrlat**=22, **urcrnrlon**=-64, **urcrnrlat**=49, **lat\_1**=33,

**lat\_2**=45)

map\_lcc.scatter([center\_lon], [center\_lat], **latlon**=True, **s**=200)

map\_lcc.drawcoastlines()

map\_lcc.drawstates()

plt.show()

copy

##### The central location of the United States is approximated by averaging the coordinates of every US city in GeonamesCache. The approximation is slightly skewed towards the East.



The get\_cities method is suitable for iterating over city information, but not for querying cities by name. To run a name-based city search, we must rely on get\_cities\_by\_name. This method takes as an input a city-name. It then returns a list of data-outputs for all the cities with that name.

##### This listing Fetching cities by name

1

2

matched\_cities\_by\_name = gc.get\_cities\_by\_name('Philadelphia')

print(matched\_cities\_by\_name)

copy

1

[{'4560349': {'geonameid': 4560349, 'name': 'Philadelphia', 'latitude': 39.95233, 'longitude': -75.16379, 'countrycode': 'US', 'population': 1567442, 'timezone': 'America/New\_York'}}]

copy

The get\_cities\_by\_name method may return more than one city, because city-names are not always unique. For example, GeoNamesCache contains 6 different instances of the city San Francisco, spanning across 5 different countries. Calling gc.get\_cities\_by\_name('San Francisco') will return data for each of these San Francisco instances. Lets iterate over that data, and print the country where each San Francisco is found.

##### This listing Fetching multiple cities with a shared name

1

2

3

4

5

6

7

matched\_cities\_list = gc.get\_cities\_by\_name('San Francisco')

for i, san\_francisco in enumerate(matched\_cities\_list):

city\_info = list(san\_francisco.values())[0]

country\_code = city\_info['countrycode']

country = countries[country\_code]['name']

print(f"The San Francisco at index {i} is located in {country}")

copy

1

2

3

4

5

6

The San Francisco **at** index 0 **is** located **in** Argentina

The San Francisco **at** index 1 **is** located **in** Costa Rica

The San Francisco **at** index 2 **is** located **in** Philippines

The San Francisco **at** index 3 **is** located **in** Philippines

The San Francisco **at** index 4 **is** located **in** El Salvador

The San Francisco **at** index 5 **is** located **in** United States

copy

Its common for multiple cities to share the same name. Choosing among such cities is quite difficult. Suppose, for instance, that someone queries a search engine for the "weather in Athens". The search engine must then choose between Athens, Ohio and Athens, Greece. Additional context is required to correctly disambiguate between the locations. Is the user from Ohio? Are they planning a trip to Greece? Without that context, the search engine must guess. Usually, the safest guess is the city with the largest population. From a statistical standpoint, the more populous cities are more likely to be referenced in everyday conversation. Choosing the most populated city isn’t guaranteed to work all the time, but it still better than making a completely random choice. Lets see what happens when we plot the most populated San Francisco location.

##### This listing Mapping the most populous San Francisco

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

best\_sf = max(gc.get\_cities\_by\_name('San Francisco'),

**key**=lambda x: list(x.values())[0]['population'])

sf\_data = list(best\_sf.values())[0]

sf\_lat = sf\_data['latitude']

sf\_lon = sf\_data['longitude']

fig = plt.figure(figsize=(12, 8))

map\_lcc = Basemap(**projection**='lcc', **lon\_0**=-95, **llcrnrlon**=-119,

**llcrnrlat**=22, **urcrnrlon**=-64, **urcrnrlat**=49, **lat\_1**=33,

**lat\_2**=45)

map\_lcc.scatter([sf\_lon], [sf\_lat], **latlon**=True, **s**=200)

map\_lcc.drawcoastlines()

map\_lcc.drawstates()

x, y = map\_lcc(sf\_lon, sf\_lat)

plt.text(x, y, ' San Francisco', **fontsize**=16)

plt.show()

**1**

copy

##### Among the 6 San Franciscos stored GeoNamesCache, the city with the highest population appears in California, as expected.



Selecting the San Francisco with the largest population returns the well-known Californian city, rather than any of the lesser-known locations outside of the US.

Common GeoNamesCache Methods

gc = GeonamesCache(): Initializes a GeonamesCache object

gc.get\_continents(): Returns a dictionary mapping continent ids to continent data.

gc.get\_countries(): Returns a dictionary mapping country ids to country data.

gc.get\_countries\_by\_names(): Returns a dictionary mapping country names to country data.

gc.get\_cities(): Returns a dictionary mapping city ids to city data.

gc.get\_cities\_by\_name(city\_name): Returns a list of cities that share a name of city\_name.

### Limitations of the GeoNamesCache Library

GeoNamesCache is a useful tool, but it does carry certain significant flaws. First of all, the library’s record of cities is far from complete. Certain sparsely populated locations in rural areas (whether the rural United States or rural China) are missing from the stored database records. Furthermore, the get\_cities\_by\_name method maps only one version of a city’s name to its geographic data. This poses a problem for cities like New York, which carry more than one commonly referenced name.

##### This listing Fetching New York City from GeoNamesCache

1

2

3

4

5

**for** ny\_name **in** ['New York', 'New York City']:

**if** **not** gc.get\_cities\_by\_name(ny\_name):

print(f"'{ny\_name}' is not present in GeoNamesCache database.")

**else**:

print(f"'{ny\_name}' is present in GeoNamesCache database.")

copy

1

2

'New York' **is** **not** present **in** GeoNamesCache **database**.

'New York City' **is** present **in** GeoNamesCache **database**.

copy

The limits of single references become particularly obvious when we examine diacritics in city names. Diacritics are accent marks that designate the proper pronunciation of foreign-sounding words. They are regularly present in the names of certain cities. Examples include Cañon City, Colorado, and Hagåtña, Guam.

##### This listing Fetching accented cities from GeoNamesCache

1

2

print(gc.get\_cities\_by\_name(u'Cañon City'))

print(gc.get\_cities\_by\_name(u'Hagåtña'))

copy

1

2

[{'5416005': {'geonameid': 5416005, 'name': 'Cañon City', 'latitude': 38.44098, 'longitude': -105.24245, 'countrycode': 'US', 'population': 16400, 'timezone': 'America/Denver'}}]

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU', 'population': 1051, 'timezone': 'Pacific/Guam'}}]

copy

How many of the cities stored in GeoNamesCache contain diacritics in their name? We can find out using the unidecode function from the external Unidecode library. The function strips out all accent marks within an input text. By checking for differences between the input text and the output text, we should be able to detect all city-names where accent-marks are present.

##### NOTE

Call "pip install Unidecode" from the command-line terminal in order to install the Unidecode library.

##### This listing Counting all accented cities in GeoNamesCache

1

2

3

4

5

6

7

**from** unidecode **import** unidecode

accented\_names = [city['name'] **for** city **in** gc.get\_cities().values()

**if** city['name'] != unidecode(city['name'])]

num\_accented\_cities = len(accented\_names)

print(f"An example accented city name is '{accented\_names[0]}'")

print(f"{num\_accented\_cities} cities have accented names")

copy

1

2

An example accented city **name** **is** 'Khawr Fakkān'

4896 cities have accented names

copy

Approximately 5000 stored cities contain diacritics in their names. These cities are commonly referenced without an accent in published text-data. One way to ensure we match all such cities is to create a dictionary of alternative city names. Within that dictionary, the accent-free unidecode output will map back to the original accented names.

##### This listing Stripping accents in alternative city names

1

2

3

alternative\_names = {unidecode(**name**): **name**

for **name** **in** accented\_names}

print(gc.get\_cities\_by\_name(alternative\_names['Hagatna']))

copy

1

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU', 'population': 1051, 'timezone': 'Pacific/Guam'}}]

copy

We can now match the stripped dictionary keys against all inputted text by passing the accented dictionary values into GeoNamesCache, whenever a key-match is found.

##### This listing Finding accent-free city-names in text

1

2

3

4

5

text = u'This sentence matches Hagatna'

**for** key, value **in** alternative\_names.items():

**if** key **in** text:

print(gc.get\_cities\_by\_name(value))

**break**

copy

1

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU', 'population': 1051, 'timezone': 'Pacific/Guam'}}]

copy

GeoNamesCache provides us with a way to track locations, along with their geographical coordinates. Using the library, we can also search for mentioned location-names within any inputted text. However, finding names within text is not a trivial process. If we wish to match location-names appropriately, then we must learn proper Python text-matching techniques while also avoiding common pitfalls.

# **2.3 Submit Your Work**

The deliverable is a Jupyter Notebook documenting your work as you add three additional columns to the DataFrame: longitude, latitude, and countrycode. We will use these coordinates to cluster the headlines in the next section. An example of the structure and content of the expected DataFrame is below:

Upload a link to your Jupyter Notebook (preferably hosted on GitHub) in the blank below and hit submit. After submitting, you can view an example solution in the next section.

# **Solution**

Below is an example of the structure and content of the pandas DataFrame you should create in this section. The last 3 columns were added to capture the location of each headline.

For an in-depth solution to this section, refer to the notebook below. Your end outcome pandas DataFrame should be similar, but your workflow may differ from the notebook.

* [2. Finding Geographic Locations of Headlines.ipynb](https://liveproject-resources.s3.amazonaws.com/93/22365/2020-03-03-09-34-52/2.%20Finding%20Geographic%20Locations%20of%20Headlines.ipynb)

# 3. Clustering Headlines Based on Location

# **3.1 Applying Clustering Algorithms to Find Geographically Similar Headlines**

**Objective**

* Cluster (find groups of) headlines based on the geographic coordinates using both k-means clustering and DBSCAN. Visualize the clusters on a world map to check the results. Try different parameters and distance measures in the algorithms to produce better clusters.

**Workflow**

1. Apply k-means clustering and the DBSCAN algorithm to the latitude and longitude of each headline.
   * Use the default initial parameters for the algorithm or, if you have prior experience, choose parameters you think will work well.
   * Assign the cluster labels as another column on the DataFrame.
2. Visualize the clusters on a world map using the Basemap library. Color the headlines by the cluster assignment.
   * Determine if the clusters are reasonable: Are headlines geographically close to one another in the same cluster?
   * Write a visualization function to quickly check clustering results.
3. In the likely case that the first clustering is not ideal, adjust the parameters of the algorithm you choose or use a different algorithm.
   * You can use an [elbow plot to select](https://bl.ocks.org/rpgove/0060ff3b656618e9136b) the number of clusters in k-means.
   * The two most important parameters for DBSCAN are eps and min\_samples
4. Try using DBSCAN with the great circle distance, which finds the distance between two geographic points on a spherical globe.
   * Write a function to return the Great Circle distance between two coordinate points.
   * Use this function as the metric for DBSCAN.
5. Repeat the above steps—cluster, visualize, analyze, tune—as many times as is required until the algorithm correctly assigns close points to the same cluster without too many outliers.

**Importance to project**

* Clustering is a form of unsupervised learning that tries to match similar entities with features (for example, geographic coordinates) and a measure (for example, Euclidean distance between the coordinates). We don’t know ahead of time which headlines go in which clusters, so we need a machine learning algorithm to find the clusters for us based on the data we provide.
* We are using machine learning to extract additional information from our data set, but this time we are compressing the data instead of adding more. We use the algorithm’s output to focus on only the most important clusters and headlines instead of having to examine them all.
* **In the next section, we will interpret the algorithm’s output to identity disease outbreaks.**

**Notes**

* You may be surprised that we are using our intuition—by visualizing the clusters—to determine if they are acceptable. This is common in data science: often the fastest way to find out if an algorithm is working properly is to assess the outputs with your own knowledge. We can be quite accurate in immediately spotting if something is off with the model.
* Again, notice the iterative nature of this section. It will require several tries to find the best model parameters. Don’t get frustrated if you can’t figure them out at first. Also, don’t be afraid to go back and redo the previous sections if data quality is causing problems.

**Resources**

* Chapter 10, [Clustering with K-Means and DBSCAN](https://liveproject.manning.com/module/93_4_2/discovering-disease-outbreaks-from-news-headlines/3--clustering-headlines-based-on-location/3-2--clustering-with-kmeans-and-dbscan?), from Data Science Bookcamp by Leonard Apeltsin in section 3.2 can help you work with the clustering algorithms and tune the parameters.
* Chapter 11, [Plotting Maps with Basemap](https://liveproject.manning.com/module/93_4_3/discovering-disease-outbreaks-from-news-headlines/3--clustering-headlines-based-on-location/3-3--plotting-maps-using-basemap?), from Data Science Bookcamp by Leonard Apeltsin in section 3.3 can help you plot geographic maps of clustered headlines.
* Referring to the Scikit-Learn documentation on these algorithms is crucial.
  + DBSCAN:
    - [Method documentation](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html)
    - [User Guide](https://scikit-learn.org/stable/modules/clustering.html#dbscan)
  + K-Means:
    - [Method Documentation](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html)
    - [User Guide](https://scikit-learn.org/stable/modules/clustering.html#k-means)
* [The Basemap documentation](https://basemaptutorial.readthedocs.io/en/latest/index.html#basemap-tutorial) is quite useful for picking up what you need to make a world map with the clustered headlines.

# **3.2 Clustering with K-Means and DBSCAN**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-10)

Clustering is the process of organizing data-points into conceptually meaningful groups. What makes a given group "conceptually meaningful"? There is no easy answer to that question. The usefulness of any clustered output is dependent on the task we’ve been assigned.

Image that we’re asked to cluster a collection of pet photos. Do we cluster fish and lizards in one group and the fluffy pets (such as hamsters, cats, and dogs) into another? Or should hamsters, cats, and dogs be assigned 3 separate clusters of their own? If so, perhaps we should consider clustering pets by breed. Thus, Chihuahuas and Great Danes fall into diverging clusters. Differentiating between dog breeds will not be easy. However, we can easily distinguish between Chihuahuas and Great Danes based on breed size. Maybe we should compromise? We’ll cluster on both fluffiness and size, thus bypassing the distinction between the Cairn Terrier and similar-looking Norwich Terrier.

Is the compromise worth it? It depends on our data science task. Suppose we work for a pet-food company, and our aim is to estimate demand for dog food, cat food, and lizard food. Under these conditions, we must distinguish between fluffy dogs, fluffy cats, and scaly lizards. However, we won’t need resolve differences between separate dog-breeds. On the other hand, image an analyst at a vet’s office, who’s trying to group pet-patients by their breed. This second task requires a much more granular level of group resolution.

Different situations depend on different clustering techniques. Its up to us as data scientists to choose the correct clustering solution. Over course of our careers, we will cluster thousands (if not tens of thousands) of datasets, using a variety of clustering algorithms. The most commonly used algorithms will rely on some notion of "centrality" to distinguish between clusters.

## Using Centrality to Discover Clusters

In Section Five, we learned how the centrality of data can be represented using the mean. Later, in Section Seven, we computed the mean-length within a single group of fish. Eventually, we compared 2 separate sets of fish by analyzing the difference between their means. We utilized that difference to determine if all the fish belonged to the same group. Intuitively, all data-points within a single group should cluster around one central value. Meanwhile, the measurements in 2 divergent groups should cluster around 2 different means. Thus, we can utilize centrality to distinguish between 2 divergent groups. Lets explore this notion in concrete detail.

Suppose we take a field trip to a lively local pub and see 2 dartboards hanging side-by-side. Each of the dartboards is completely covered in darts, and there are also darts protruding from the walls. The tipsy players in the pub aim for the bull’s-eye of one board or the other. Frequently, they miss. This leads to the observed scattering of darts centered around the 2 bull’s-eyes.

Lets simulate the scattering numerically. We’ll treat each bull’s-eye location as a 2D coordinate. Darts are randomly flung at that coordinate. Consequently, the 2D position of each dart is randomly distributed. The most appropriate distribution for modeling dart position is the Normal distribution. This is true for the following 2 reasons:

* A typical dart thrower will aim at the bull’s-eye, not at the edge of the dartboard. Thus, each dart is more likely to strike close to the center of the board. This behavior is consistent with random Normal samples. Sampled Normal values closer to the mean will occur more frequently than values that are further from the mean.
* We expect the darts to strike the board symmetrically, relative to the center. Darts will strike 3 inches left of center and 3 inches right of center with equal frequency. This symmetry is captured by the bell-shaped Normal curve.

Suppose the first bull’s-eye is located at a coordinate of [0, 0]. A dart is thrown at that coordinate. We’ll model the x and y positions of the dart using 2 Normal distributions. These distributions share a mean of 0. We’ll also assume that they share a variance of 2. The code below will generate the random coordinates of the dart.

##### This listing Modeling dart coordinates using 2 Normal distributions

1

2

3

4

5

6

7

8

import numpy as np

np.random.seed(0)

mean = 0

variance = 2

x = np.random.normal(mean, variance \*\* 0.5)

y = np.random.normal(mean, variance \*\* 0.5)

print(f"The x coordinate of a randomly thrown dart is {x:.2f}")

print(f"The y coordinate of a randomly thrown dart is {y:.2f}")

copy

1

2

The x coordinate of a randomly thrown dart **is** 2.49

The y coordinate of a randomly thrown dart **is** 0.57

copy

##### NOTE

We can more efficiently model dart position using the np.random.multivariate\_normal method. That method will select a single random point from a **Multivariate Normal distribution.** The Multivariate Normal curve is simply a Normal curve that is extended to more than one dimension. Our 2D Multivariate Normal distribution will resemble a round hill whose summit is positioned at [0, 0].

Lets simulate 5,000 random darts tossed at the bulls'-eye positioned at [0, 0]. We’ll also simulate 5,000 random darts tossed at a second bull’s-eye, positioned at [0, 6]. Afterwards, we’ll generate a scatter plot of all the random dart coordinates.

##### This listing Simulating randomly thrown darts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

**import** matplotlib.pyplot as plt

np.random.seed(1)

bulls\_eye1 = [0, 0]

bulls\_eye2 = [6, 0]

bulls\_eyes = [bulls\_eye1, bulls\_eye2]

x\_coordinates, y\_coordinates = [], []

**for** bulls\_eye **in** bulls\_eyes:

**for** \_ **in** range(5000):

x = np.random.normal(bulls\_eye[0], variance \*\* 0.5)

y = np.random.normal(bulls\_eye[1], variance \*\* 0.5)

x\_coordinates.append(x)

y\_coordinates.append(y)

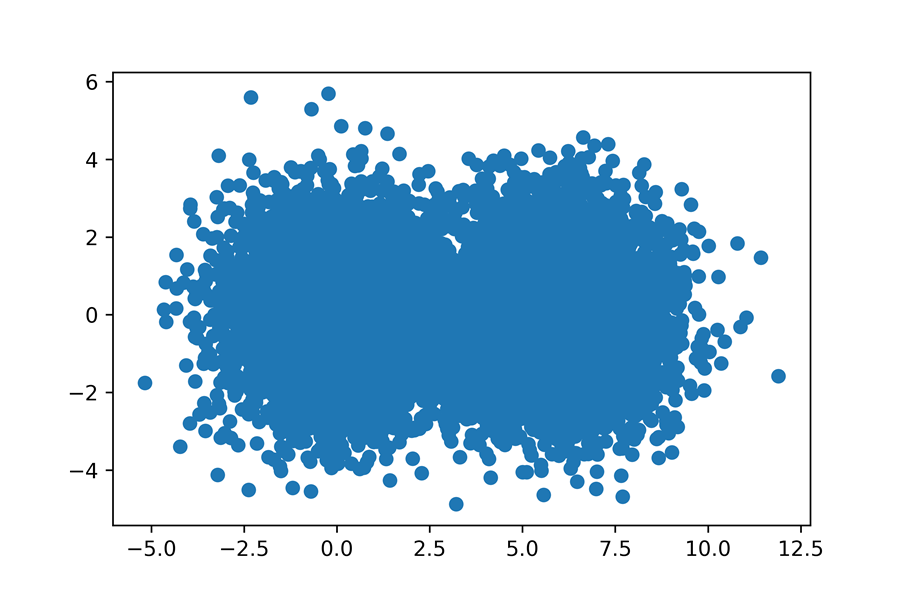
plt.scatter(x\_coordinates, y\_coordinates)

plt.show()

**1**

copy

##### Figure-10.1. A simulation of darts randomly scattered around 2 bull’s-eye targets.



Two overlapping dart-groups appear within the plot. The 2 groups contains 10,000 darts. Half the darts were aimed at the bull’s-eye on the left. The other darts were aimed at the bull’s-eye on the right. Each dart has an intended target. We can estimate that target just by looking at the plot. Darts closer to [0, 0] were probably aimed at the bull’s-eye on the left. We’ll incorporate this assumption into our dart-plot.

Lets assign each dart to its nearest bull’s-eye. We’ll start by defining a nearest\_bulls\_eye function. The function will take as input a dart list. That list will hold the x and y positions of some dart. The function will return will return the index of the bull’s-eye that is most proximate to dart. We’ll measure dart-proximity using **Euclidean distance**, which is the standard straight-line distance between 2 points.

##### NOTE

Euclidean distance arises from the Pythagorean theorem. Suppose we examine a dart at position [x\_dart, y\_dart] relative to a bull’s-eye at position [x\_bull, y\_bull]. According to the Pythagorean theorem, distance2 = (x\_dart - x\_bull)2 + (y\_dart - y\_bull)2. We can solve for distance using a custom Euclidean function. Alternatively, we can use the scipy.spatial.distance.euclidean function provided by SciPy.

Below, we’ll define nearest\_bulls\_eye, and apply it to darts [0, 1] and [6, 1].

##### This listing Assigning darts to the nearest bull’s-eye

1

2

3

4

5

6

7

8

9

**from** scipy.spatial.distance **import** euclidean

**def** **nearest\_bulls\_eye**(dart):

distances = [euclidean(dart, bulls\_e) **for** bulls\_e **in** bulls\_eyes]

**return** np.argmin(distances)

darts = [[0,1], [6, 1]]

**for** dart **in** darts:

index = nearest\_bulls\_eye(dart)

print(f"The dart at position {dart} is closest to bulls-eye {index}")

**1**

**2**

copy

Now, we will apply the nearest\_bulls\_eye function to all our computed dart coordinates. Afterwards, each dart-point will be plotted using one of 2 colors, in order to distinguish between the 2 bull’s-eye assignments.

##### This listing Coloring darts based on nearest bull’s-eye

1

2

3

4

5

6

7

8

9

10

11

12

13

def color\_by\_cluster(darts):

nearest\_bulls\_eyes = [nearest\_bulls\_eye(dart) for dart in darts]

for bs\_index in range(len(bulls\_eyes)):

selected\_darts = [darts[i] for i in range(len(darts))

if bs\_index == nearest\_bulls\_eyes[i]]

x\_coordinates, y\_coordinates = np.array(selected\_darts).T

plt.scatter(x\_coordinates, y\_coordinates,

color=['g', 'k'][bs\_index])

plt.show()

darts = [[x\_coordinates[i], y\_coordinates[i]]

for i in range(len(x\_coordinates))]

color\_by\_cluster(darts)

**1**

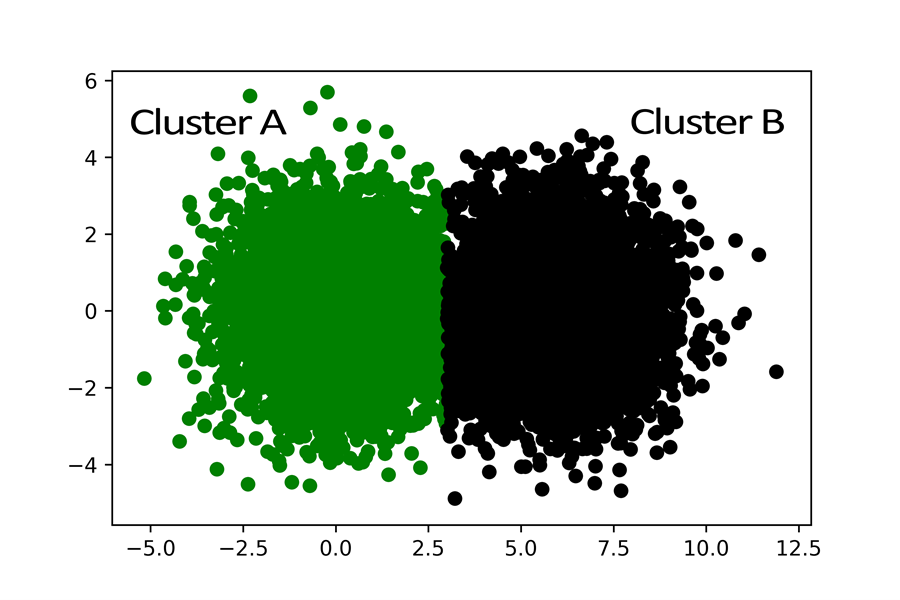
**2**

**3**

**4**

copy

##### Figure-10.2. Darts colored based on proximity to the nearest bull’s-eye. Cluster A represents all points closest to the left bull`s-eye, and Cluster B represents all points closet to the right bull`s-eye.



The colored darts sensibly split into 2 even clusters. Our discovery of these clusters depended our knowledge of the bulls'-eye coordinates. How would we identify such clusters if no central coordinates were provided? Well, one primitive strategy is to simply guess the location of the bull’s-eyes. We can pick 2 random darts. We’ll hope these darts are somehow relatively close to each of the bull’s-eyes, though the likelihood of that happening is incredibly low. Coloring darts based on 2 randomly chosen centers will in most cases not yield good results.

##### This listing Assigning darts to randomly chosen centers

1

2

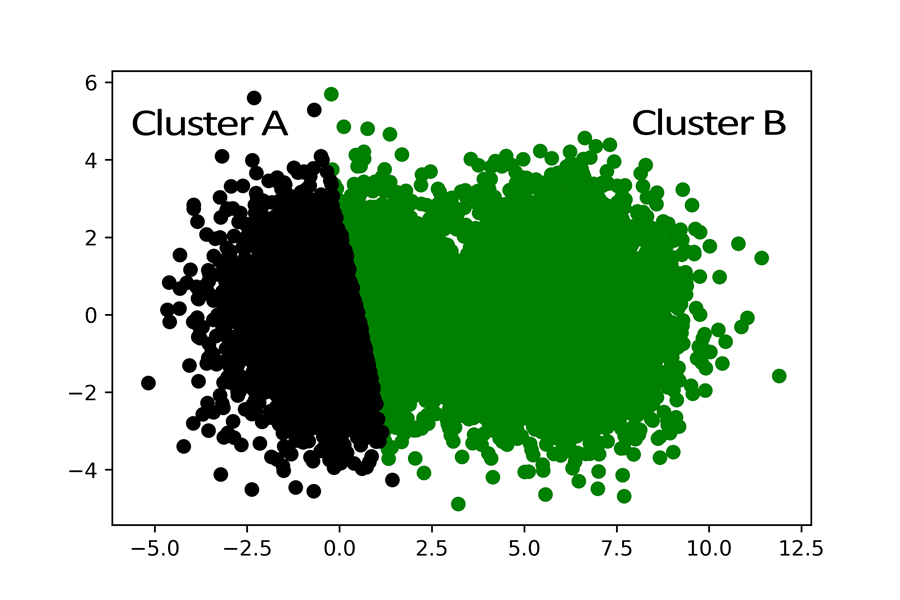
bulls\_eyes = np.array(darts[:2])

color\_by\_cluster(darts)

**1**

copy

##### Figure-10.3. Darts colored based on proximity to randomly selected centers. Cluster B is stretched too far out to the left.



From a qualitative standpoint, our indiscriminately selected centers simply feel wrong. For instance, Cluster B on the right seems to be stretching way too far to the left. The arbitrary center we’ve assigned it doesn’t appear to match its actual bulls'-eye point. Yet there’s a way to remedy our error. We can compute the mean coordinates of all the points within the stretched right clustered group, and afterwards utilize these coordinates to adjust our estimation of the group’s center. After assigning the cluster’s mean coordinates to the bulls'-eye, we can re-apply our distance-based grouping technique in order to adjust the right-most cluster’s boundaries. In fact, for maximum effectiveness, we will also reset the left-most cluster’s center to its mean prior to re-running our centrality-based clustering.

##### NOTE

When we compute the mean of 1D array, we return a single value. We are now extending that definition to encompass multiple dimensions. When we compute the mean of 2D array, we return the mean of all x-coordinates, and also the mean of all y-coordinates. The final output is a 2D array containing means across the x-axis, and the y-axis.

##### This listing Assigning darts to centers based on mean

1

2

3

4

5

6

7

8

9

10

11

12

13

14

def update\_bulls\_eyes(darts):

updated\_bulls\_eyes = []

nearest\_bulls\_eyes = [nearest\_bulls\_eye(dart) **for** dart **in** darts]

**for** bs\_index **in** range(len(bulls\_eyes)):

selected\_darts = [darts[i] **for** i **in** range(len(darts))

**if** bs\_index == nearest\_bulls\_eyes[i]]

x\_coordinates, y\_coordinates = np.array(selected\_darts).T

mean\_center = [np.mean(x\_coordinates), np.mean(y\_coordinates)]

updated\_bulls\_eyes.append(mean\_center)

return updated\_bulls\_eyes

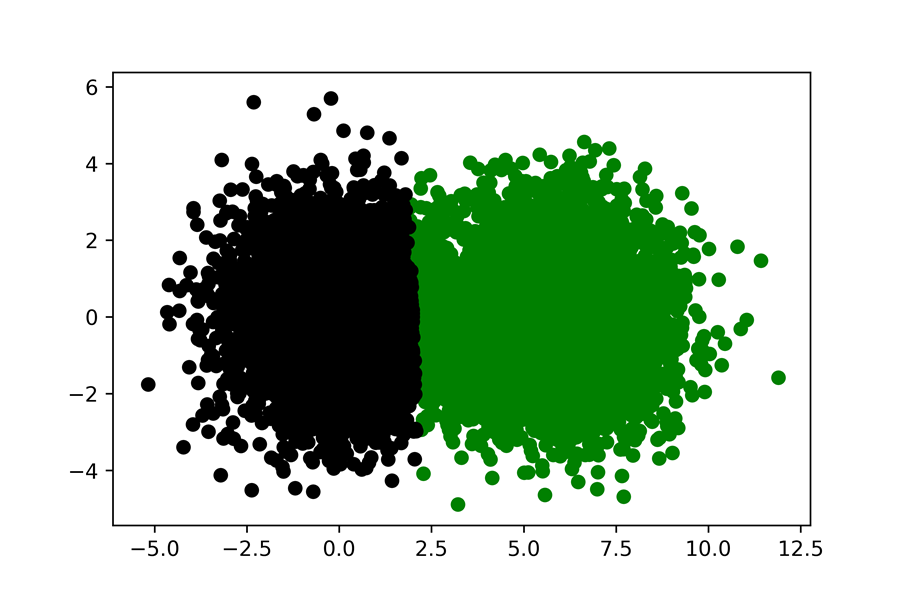
bulls\_eyes = update\_bulls\_eyes(darts)

color\_by\_cluster(darts)

**1**

copy

##### Figure-10.4. Darts colored based on proximity to recomputed centers. The 2 clusters now appear to be more even.



Already the results are looking better, though they’re not quite as effective as they could be. The cluster’s centers still appear a little off. Lets remedy the results by repeating the mean-based centrality adjustment over 10 additional iterations.

##### This listing Adjusting bull’s-eye positions over 10 iterations

1

2

3

4

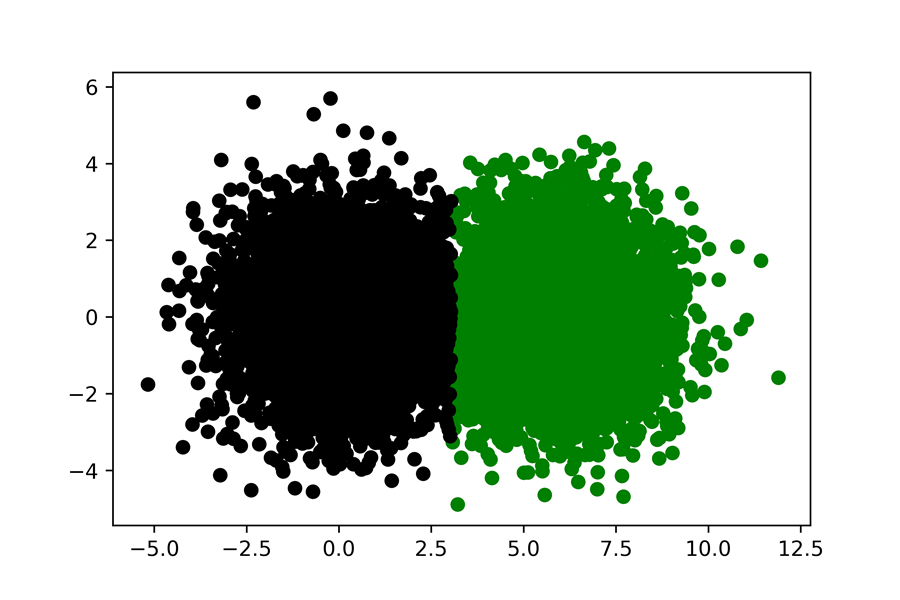
for i **in** **range**(10):

bulls\_eyes = **update\_bulls\_eyes**(darts)

**color\_by\_cluster**(darts)

copy

##### Figure-10.5. Darts colored based on proximity to iteratively recomputed centers



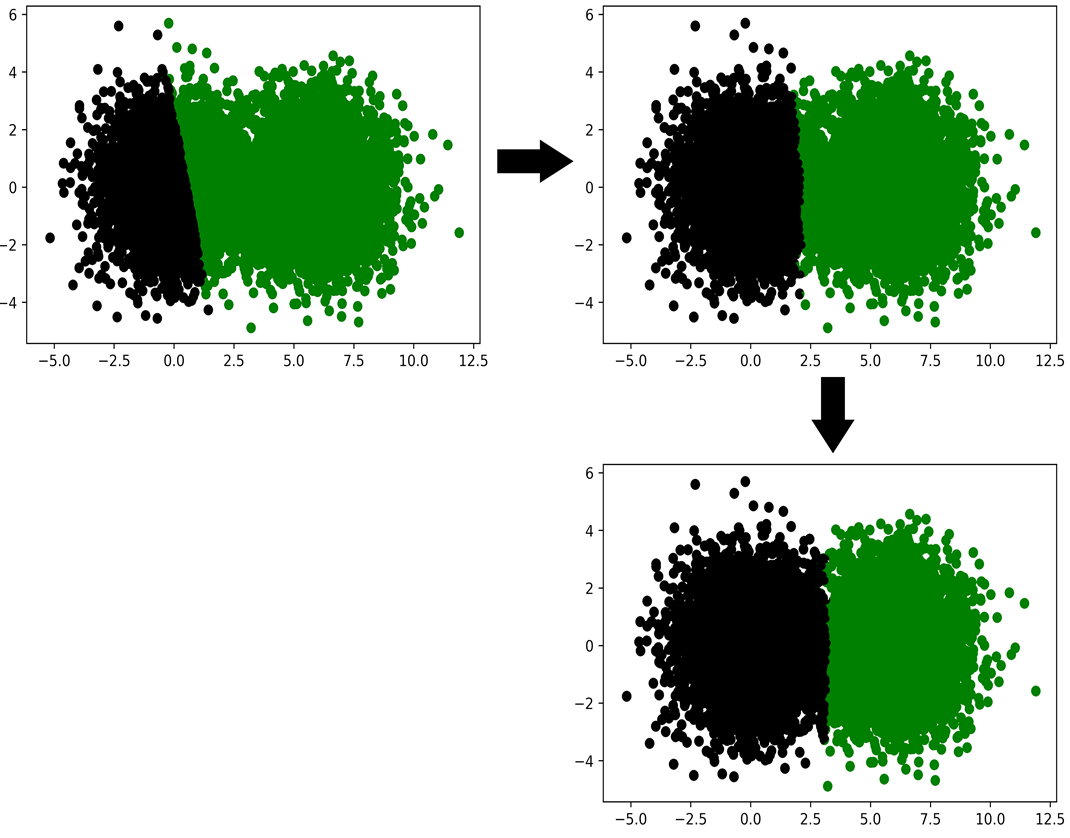
Viola! Now the 2 sets of darts have been perfectly clustered. We have essentially replicated the **K-means** clustering algorithm, which organizes data using centrality.

## K-Means: A Clustering Algorithm for Grouping Data into K Central Groups

The K-means algorithm assumes that inputted data-points swirl around K different centers. Each central coordinate is like a hidden bulls'-eye surrounded by scattered data-points. The purpose of the algorithm is to uncover these hidden central coordinates.

We initialize K-means by first selecting K, which is the number of central coordinates we will search for. In our dartboard analysis, K was set to 2, though generally K can equal any whole number. The algorithm proceeds to choose K data-points at random. These data-points are treated as though they were true centers. Afterwards the algorithm iterates by updating the chosen central locations, which data scientists call **centroids**. During a single iteration, every data-point is assigned to its closest center. This leads to the formation of K groups. Next, the center of each group is updated. The new center equals the the mean the group’s coordinates. If we repeat the process long enough, the group-means will converge to K representative centers. The convergence is mathematically guaranteed. However, we cannot know in advance the number of iterations required for the convergence to take place. Thus, a common trick is to halt the iterations when the all of the newly computed centers do not deviate significantly from their predecessors.

##### Figure-10.6. The K-means algorithm iteratively converging from 2 randomly selected centroids to the actual bulls'-eye centroids.



K-means is not without its limitations. The algorithm is predicated on our knowledge of K; the number of clusters to look for. Frequently, such knowledge is not available. Also, while K-means commonly finds reasonable centers, its not mathematically guaranteed to find the best possible centers in the data. Occasionally, K-means will return non-intuitive or sub-optimal groups due to poorly selected random centroids at the initialization step of the algorithm. Finally, K-means pre-supposes that the clusters in the data actually swirl around K central locations. However, as we’ll learn later in the Section, this supposition does not always hold.

### K-means Clustering Using Scikit-learn

The K-means algorithm will run in reasonable time, if it has been implemented efficiently. A speedy implementation of the algorithm is available through the external Scikit-Learn library. Scikit-learn is an extremely popular machine learning toolkit built on-top of NumPy and Scipy. It features a variety of core classification, regression, and clustering algorithms, including of course, K-means. Lets install the library. Afterwards, we’ll import Scikit-learn’s KMeans clustering class.

##### NOTE

Call "pip install scikit-learn" from the command-line terminal in order to install the Scikit-learn library.

##### This listing Importing KMeans from Scikit-learn

1

**from** sklearn.**cluster** **import** KMeans

copy

Applying KMeans to our darts data is easy. First, we need to run KMeans(n\_clusters=2). This will create a cluster\_model object capable of finding 2 bull’s-eye centers. Afterwards, we can execute K-means by running cluster\_model.fit\_predict(darts). That method-call will return an assigned\_bulls\_eyes array, which will store the bull’s-eye index of each dart.

##### This listing K-means clustering using Scikit-learn

1

2

3

4

5

cluster\_model = **KMeans**(n\_clusters=2)

assigned\_bulls\_eyes = cluster\_model.fit\_predict(darts)

**print**("Bull's-eye assignments:")

**print**(assigned\_bulls\_eyes)

**1**

**2**

copy

1

2

Bull's-eye assignments:

[0 0 0 ... 1 1 1]

copy

Lets quickly color our darts based on their clustering assignments, in order to confirm that the assignments makes sense.

##### This listing Plotting K-means cluster assignments

1

2

3

4

5

6

7

for bs\_index in range(len(bulls\_eyes)):

selected\_darts = [darts[i] for i in range(len(darts))

if bs\_index == assigned\_bulls\_eyes[i]]

x\_coordinates, y\_coordinates = np.array(selected\_darts).T

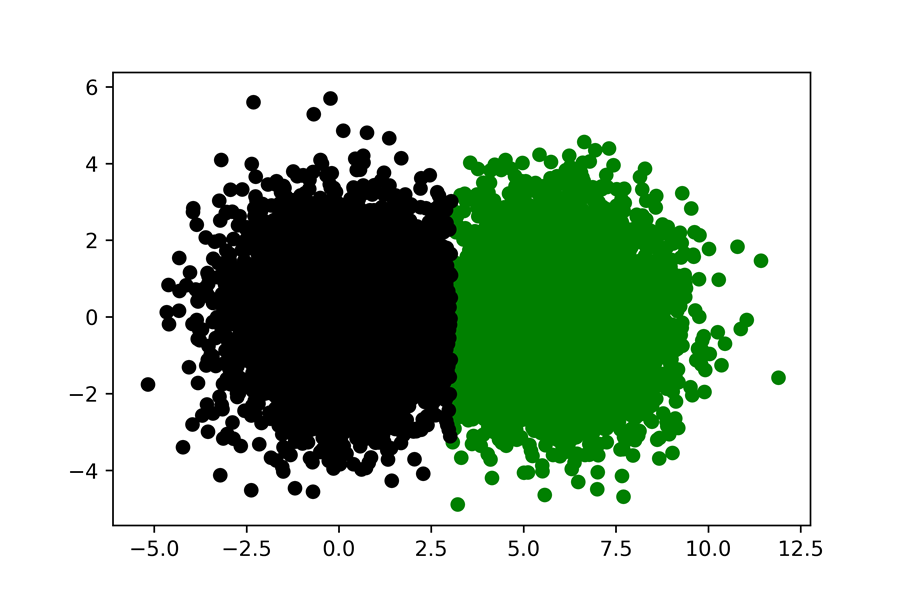
plt.scatter(x\_coordinates, y\_coordinates,

color=['g', 'k'][bs\_index])

plt.show()

copy

##### Figure-10.7. The K-means clustering results returned by Scikit-learn are consistent with our expectations.



Our clustering model has located the centroids in the data. Now, we can reuse these centroids to analyze new data-points that the model has not seen before. Executing cluster\_model.predict([x, y]) will assign a centroid to a data-point defined by x and y. We’ll use the predict method to cluster 2 new data-points below.

##### This listing Using cluster\_model to cluster new data

1

2

3

4

5

**new**\_darts = [[500, 500], [-500, -500]]

**new**\_bulls\_eye\_assignments = cluster\_model.predict(**new**\_darts)

**for** i, dart **in** enumerate(**new**\_darts):

bulls\_eye\_index = **new**\_bulls\_eye\_assignments[i]

print(f"Dart at {dart} is closest to bull's-eye {bulls\_eye\_index}")

copy

1

2

Dart **at** [500, 500] **is** closest to bull's-eye 0

Dart **at** [-500, -500] **is** closest to bull's-eye 1

copy

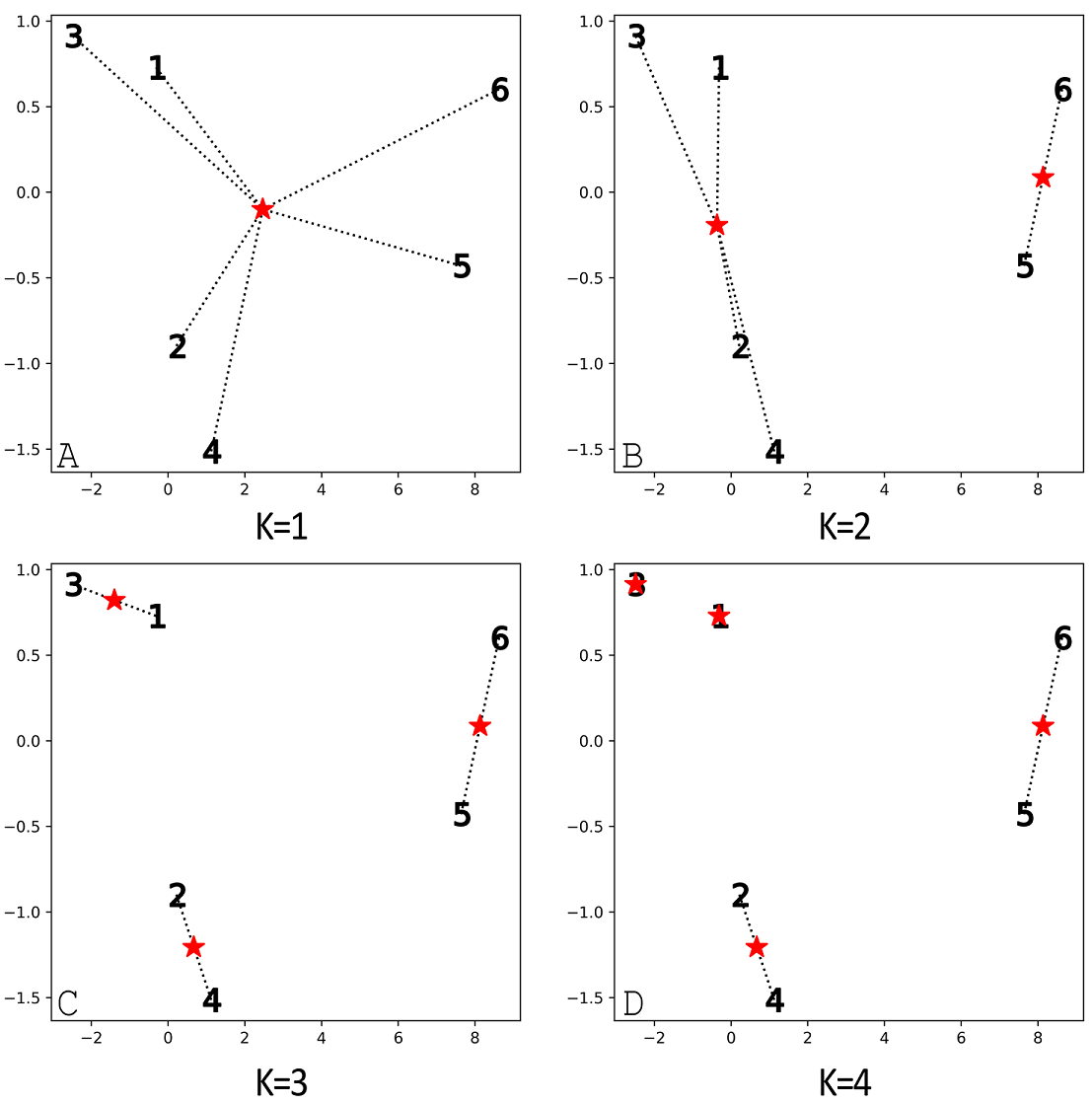
### Selecting the Optimal K Using the Elbow Method

K-means relies on an inputted K. This can be a serious hindrance when the number of authentic clusters in the data isn’t known in advance. We can however, estimate an appropriate value for K using a technique known as the **Elbow method**.

The Elbow method depends on a calculated value called **inertia**, which is the sum of the squared distances between each point and its closest K-means center. If K is 1, then the inertia will equal the sum of all squared distances to the dataset’s mean. This value, as discussed in Section Five, is directly proportional to the variance. Variance, in turn, is a measure of dispersion. Thus, if K is 1, then the inertia is an estimate of dispersion. This property holds true even if K is greater than 1. Basically, inertia estimates total dispersion around our K computed means.

Inertia’s estimate of dispersion allows us to determine if our K-value is too high or too low. For example, imagine if we set K to 1. Potentially, many of our data-points will be positioned too far from one center. Our dispersion will be large, and our inertia will be large. As we increase K towards a more sensible number, the additional centers will cause the inertia to decrease. Eventually, if we go overboard and set K to equal to the total number of points, then each data point will fall into its very own private cluster. Dispersion will be eliminated and inertia will drop to zero.

##### Figure-10.8. Six points, numbered 1 through 6, are plotted in 2D space. Centers, marked by stars, are computed across various values of K. A line is drawn from every point to its nearest center. Inertia is computed by summing the squared lengths of the six lines. A) K=1. All six lines stretch out from a single center. The inertia is quite large. B) K=2. Points 5 and 6 are now very close to a second center. The inertia is reduced. C) K=3. Points 1 and 3 are substantially closer to a newly formed center. Points 2 and 4 are also substantially closer to a newly formed center. The inertia has radically decreased. D) K=4. Points 1 and 3 now overlap with their centers. Their contribution to the inertia has shifted from a very low value to zero. The distances between the remaining four points and their associated centers remains unchanged. Thus, increasing K from 3 to 4 caused a very small decrease in inertia.



Some inertia values are too large. Others are too low. Somewhere in-between might lie value that’s just right. How do we find it?

Lets work out a solution. We’ll begin by plotting the inertia of our dartboard dataset over a large range of K values. Inertia is automatically computed for each Scikit-learn KMeans object. We can access this stored value through the model’s \_inertia variable.

##### This listing Plotting the K-means inertia

1

2

3

4

5

6

7

8

k\_values = range(1, 10)

inertia\_values = [KMeans(k).fit(darts).inertia\_

**for** k **in** k\_values]

plt.plot(k\_values, inertia\_values)

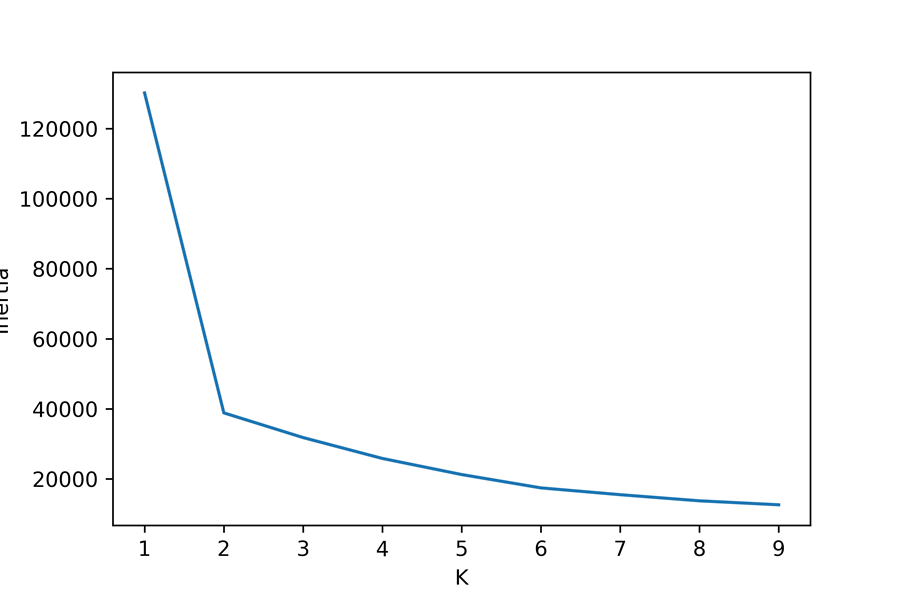
plt.xlabel('K')

plt.ylabel('Inertia')

plt.show()

copy

##### Figure-10.9. An inertia plot for a dartboard simulation containing 2 bull’s-eyes targets. The plot resembles an arm bent at the elbow. The elbow points directly to a K of 2.



The generated plot resembles an arm bent at the elbow. The elbow points directly to a K of 2. As we already know, this K accurately captures the two centers we have pre-programmed into the dataset.

Will the approach still hold if the number of present centers is increased? We can find out by adding an additional bull’s-eye to our dart-throwing simulation. After we raise the cluster count to 3, we’ll regenerate our inertia plot.

##### This listing Plotting inertia for a 3-dartboard simulation

1

2

3

4

5

6

7

8

9

10

11

12

13

new\_bulls\_eye = [12, 0]

**for** \_ **in** range(5000):

x = np.random.normal(new\_bulls\_eye[0], variance \*\* 0.5)

y = np.random.normal(new\_bulls\_eye[1], variance \*\* 0.5)

darts.append([x, y])

inertia\_values = [KMeans(k).fit(darts).inertia\_

**for** k **in** k\_values]

plt.plot(k\_values, inertia\_values)

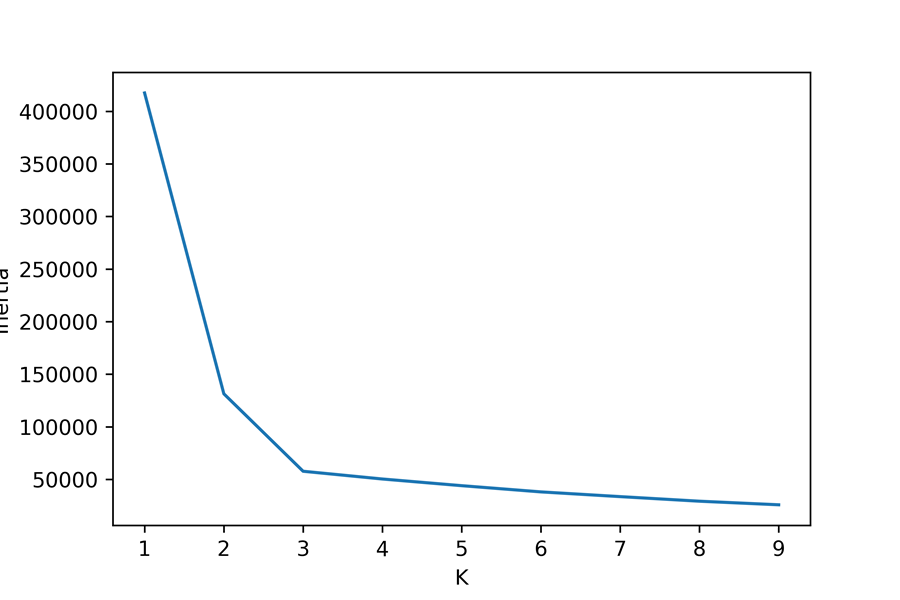
plt.xlabel('K')

plt.ylabel('Inertia')

plt.show()

copy

##### Figure-3.10. An inertia plot for a dartboard simulation containing 3 bull’s-eyes targets. The plot resembles an arm bent at the elbow. The lower-most portion of the elbow points to a K of 3.



Adding a third center leads to a new elbow whose lower-most inclination points to a K of 3. Essentially, our elbow plot traces the dispersion captured by each incremental K. A rapid decrease in inertia between consecutive K-values implies that scattered data-points have been assigned to a tighter cluster. The reduction in inertia incrementally loses its impact as the inertia curve flattens out. This transition from a vertical drop to more level angle leads to the presence of an elbow shape in our plot. We can use the position of the elbow to select a proper K in the K-means algorithm.

The Elbow method selection criterion is a useful heuristic, but it is not guaranteed to work in every case. Under certain conditions, the elbow will level off slowly over multiple K values, making it difficult select a single valid cluster count.

##### NOTE

There exist more powerful K-selection methodologies, such as the **Silhouette score**, which captures the distance of each point to neighboring clusters. A thorough discussion of the Silhouette score is beyond the scope of this book. However, you’re encouraged explore the score on your own, using the sklearn.metrics.silhouette\_score method.

K-means Clustering Methods

k\_means\_model = KMeans(n\_clusters=K): Creates a K-means model that’s intended to search for K different centroids. We’ll need to fit these centroids to inputted data.

clusters = k\_means\_model.fit\_predict(data): Executes K-means on inputted data, using an initialized KMeans object. The returned clusters array contains cluster ids ranging from 0 to K. The cluster id of data[i] is equal to clusters[i].

clusters = KMeans(n\_clusters=K).fit\_predict(data): Executes K-means in a single line of code, and returns the resulting clusters.

new\_clusters = k\_means\_model.predict(new\_data): Finds the nearest centroids to previously unseen data, using the existing centroids within a data-optimized KMeans object.

inertia = k\_means\_model.inertia\_: Returns the inertia associated with a data-optimized KMeans object.

inertia = KMeans(n\_clusters=K).predict(data).inerta\_: Executes K-means in a single line of code, and returns the resulting inertia.

The Elbow method isn’t perfect, but it will perform reasonably well if the data is clearly centered on K distinct means. This of course, assumes that our data-clusters differ due to centrality. However, in many instances, data-clusters differ to due density of data-points in space. Lets explore the concept of density-driven clusters, which are not dependent on centrality.

## Using Density to Discover Clusters

Suppose that an astronomer discovers a new planet at the far-flung edges of the solar system. The plant, much like our Saturn, has multiple rings spinning in constant orbit around its center. Each ring is formed from thousands of rocks. We’ll model these rocks as individual points, defined by x and y coordinates. Lets generate 3 rock rings composed of many rocks, using Sckit-Learn’s makes\_circles function.

##### This listing Simulating rings around a planet

1

2

3

4

5

6

7

8

9

10

11

12

13

**from** sklearn.datasets **import** make\_circles

x\_coordinates = []

y\_coordinates = []

**for** factor **in** [.3, .6, 0.99]:

rock\_ring, \_ = make\_circles(n\_samples=800, factor=factor,

noise=.03, random\_state=1)

**for** rock **in** rock\_ring:

x\_coordinates.append(rock[0])

y\_coordinates.append(rock[1])

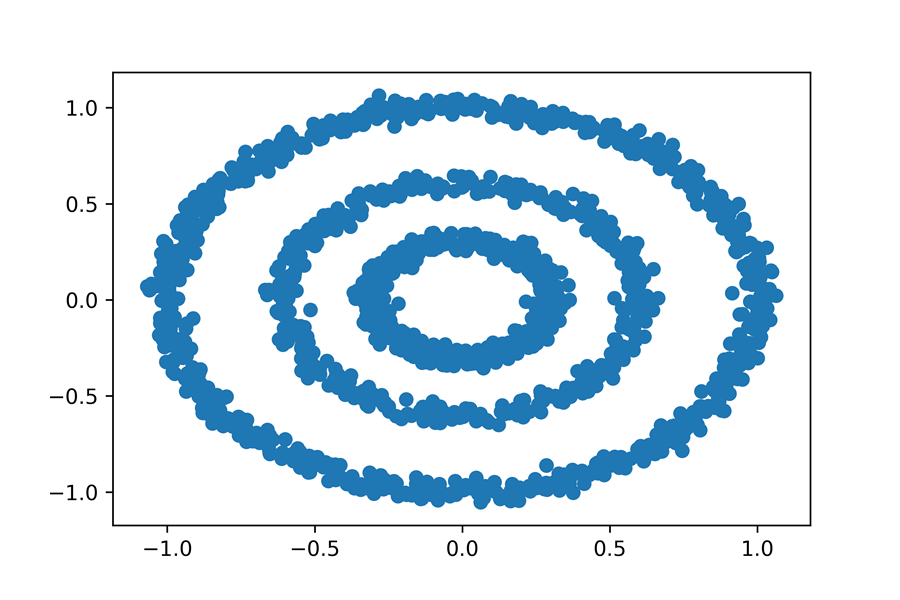
plt.scatter(x\_coordinates, y\_coordinates)

plt.show()

**1**

copy

##### Figure-10.11. A simulation of 3 rock rings positioned around a central point.



Three ring-groups are clearly present in the plot. Lets search for these 3 clusters using K-means. Obviously, we’ll set our K to 3.

##### This listing Using K-means to cluster rings

1

2

3

4

5

6

7

rocks = [[x*\_coordinates[i], y\_*coordinates[i]]

for i in range(len(x\_coordinates))]

rock*\_clusters = KMeans(3).fit\_*predict(rocks)

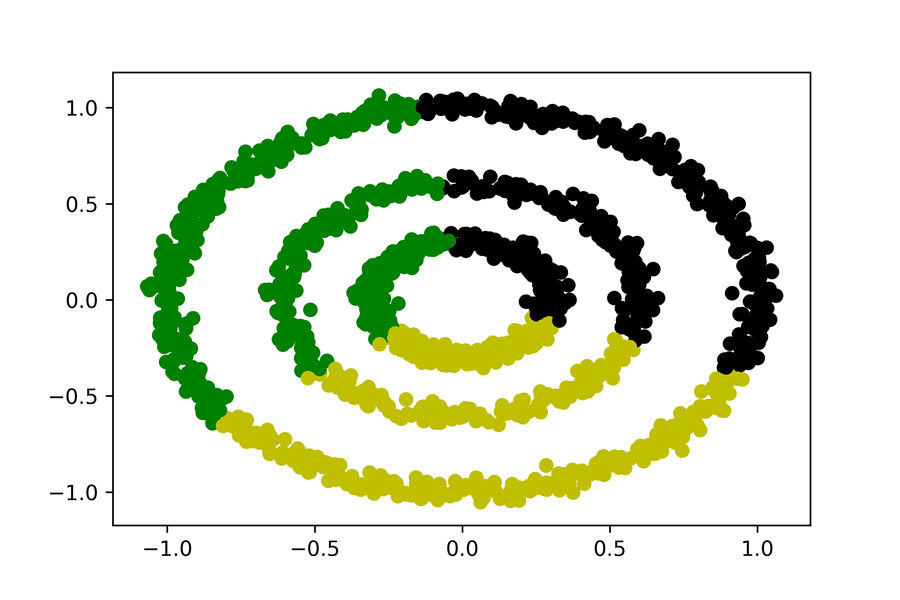
colors = [['g', 'y', 'k'][cluster] for cluster in rock\_clusters]

plt.scatter(x*\_coordinates, y\_*coordinates, color=colors)

plt.show()

copy

##### Figure-10.12. K-means clustering fails to properly identify the 3 distinct rock rings



The output is an utter failure! K-means dissects the data into 3 symmetric segments, and each segment spans across multiple rings. The solution doesn’t align with our intuitive expectation that each ring should fall into its own distinct group. What went wrong? Well, K-means assumed that the 3 clusters are defined by three unique centers, but the actual rings spin around a single central point. The difference between clusters is driven not by centrality, but by density. Each ring is constructed from a dense collection of points, with empty areas of sparsely populated space serving as the boundaries between rings.

We need to design an algorithm that will cluster data within dense regions of space. Doing so requires that we define whether a given region is dense or sparse. One simple definition of density is as follows; a point is in a dense region only if it’s located within a distance X of Y other points. We’ll refer to X and Y as epsilon and min\_points, respectively. Below, we’ll set epsilon to 0.1 and min\_points to 10. Thus, our rocks are present in a dense region of space if they’re within a 0.1 radius of at-least 10 other rocks.

##### This listing Specifying density parameters

1

2

epsilon=0.1

min\_points = 10

copy

Lets analyze the density of the first rock in our rocks list. We’ll begin by searching for all the other rocks that are within epsilon units of rocks[0]. We’ll store the indices of these neighboring rocks in a neighbor\_indices list.

##### This listing Finding the neighbors of rocks[0]

1

2

neighbor\_indices = [i for i, rock in enumerate(rocks[1:])

if euclidean(rocks[0], rock) <= epsilon]

copy

Now, we’ll compare the number of neighbors to min\_points, in order to determine if rocks[0] lies in a dense region of space.

##### This listing Checking the density of rocks[0]

1

2

3

4

5

6

7

num\_neighbors = **len**(neighbor\_indices)

**print**(f"The rock at index 0 has {num\_neighbors} neighbors.")

**if** num\_neighbors >= min\_points:

**print**("It lies in a dense region.")

**else**:

**print**("It does not lie in a dense region.")

copy

1

2

The rock at index 0 has 40 neighbors.

It lies **in** a dense region.

copy

The rock at index 0 lies in a dense region of space. Do the neighbors of rocks[0] also share that dense region of space? This is a tricky question to answer. After all, its possible that every neighbor holds less than min\_points neighbors of its own. Under our rigorous density definition, we wouldn’t consider these neighbors to be dense points. However, this would lead to a ludicrous situation in which the dense region is composed of just a single point; rocks[0]. We need to avoid such absurd outcomes. Thus, we’ll need to update our density definition. Let’s formally define density as follows:

1. If a point is located within an epsilon distance of min\_point neighbors, then that point is in a dense region of space.
2. Every neighbor of a point in a dense region of space will also cluster in that space.

Based our updated definition, we can combine rocks[0] and its neighbors into a single dense cluster.

##### This listing Creating a dense cluster

1

2

3

4

dense\_region\_indices = [0] + neighbor\_indices

dense\_region\_cluster = [rocks[i] for i **in** dense\_region\_indices]

dense\_cluster\_size = **len**(dense\_region\_cluster)

**print**(f"We found a dense cluster containing {dense\_cluster\_size} rocks")

copy

1

We found a dense **cluster** containing 41 rocks

copy

The rock and index 0 and its neighbors form a single 41-element dense cluster. What about the neighbors of the neighbors? Do any neighbors-of-neighbors belong to a dense region of space? If so, then by our updated definition, these rocks also belong to the dense cluster. Thus, by analyzing additional neighboring points, we expand the size of dense\_region\_cluster.

##### This listing Expanding a dense cluster

1

2

3

4

5

6

7

8

9

10

11

dense\_region\_indices = **set**(dense\_region\_indices)

**for** index in neighbor\_indices:

point = rocks[index]

neighbors\_of\_neighbors = [i **for** i, rock in enumerate(rocks)

**if** euclidean(point, rock) <= epsilon]

**if** len(neighbors\_of\_neighbors) >= min\_points:

dense\_region\_indices.**update**(neighbors\_of\_neighbors)

dense\_region\_cluster = [rocks[i] **for** i in dense\_region\_indices]

dense\_cluster\_size = len(dense\_region\_cluster)

**print**(**f**"We expanded our cluster to include {dense\_cluster\_size} rocks")

**1**

copy

1

We expanded our **cluster** **to** **include** 781 rocks

copy

We’ve iterated over neighbors of neighbors, and expanded our dense cluster by nearly 20-fold. Why stop there? We can expand our cluster even further by analyzing the density of newly encountered neighbors. Iteratively repeating our analysis will increase the breadth of our cluster boundary. Eventually, the boundary will spread to completely encompass one of our rock rings. Afterwards, with no new neighbors to absorb, we repeated the iterative analysis on a rocks element that has not been analyzed thus far. The repetition will lead to the clustering of additional dense rings.

The procedure described in the previous paragraph is known as **DBSCAN**. The DBSCAN algorithm organizes data based on its spatial distribution.

## DBSCAN: A Clustering Algorithm for Grouping Data Based on Spatial Density

DBSCAN is an acronym, which stands for **Density-based Spatial Clustering of Applications with Noise**. This is a ridiculously long name for what essentially is a very simple technique. The technique is executed thusly:

1. We select a random point coordinate from a data list.
2. We obtain all neighbors within an epsilon distance of that point.
3. If less than min\_points neighbors are discovered, we repeat step a using a different random point. Otherwise, we group our point and its neighbors into a single cluster.
4. We iteratively repeat steps b and c across all newly discovered neighbors. All neighboring dense points will get merged into the cluster. Our will iterations terminate after the cluster stops expanding.
5. Once we have extracted the entire cluster, we repeat steps a through e on all data-points whose density hasn’t yet been analyzed.

The DBSCAN procedure we have outlined can be programmed in under 20 lines of code. However, any basic implementation will probably run quite slowly on our rocks list. The difficulty in programming DBSCAN is that a speedy version requires some very nuanced optimizations. These optimizations improve neighbor traversal speed, and are beyond the scope of this book. Fortunately, there’s no need for us to rebuild the algorithm from scratch. Scikit-Learn makes DBSCAN available for use. We simply need to import the DBSCAN class from sklearn.cluster. Afterwards, we can initialize the class by assigning epsilon and min\_points using the eps and min\_samples parameters. Lets utilize DBSCAN to cluster our 3 rings.

##### This listing Using DBSCAN to cluster rings

1

2

3

4

5

6

from sklearn.cluster import DBSCAN

cluster*\_model = DBSCAN(eps=epsilon, min\_*samples=min\_points)

rock*\_clusters = cluster\_*model.fit\_predict(rocks)

colors = [['g', 'y', 'k'][cluster] for cluster in rock\_clusters]

plt.scatter(x*\_coordinates, y\_*coordinates, color=colors)

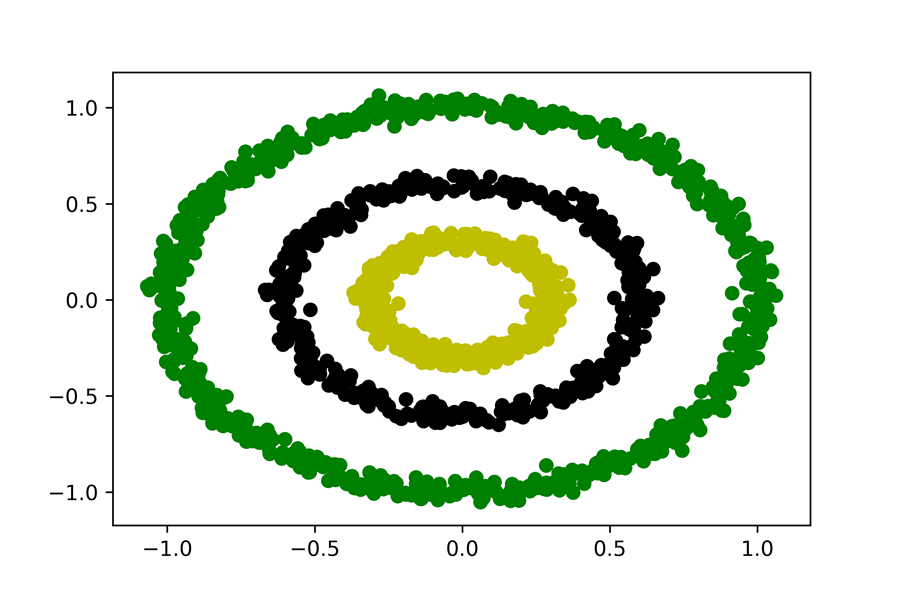
plt.show()

**1**

**2**

copy

##### Figure-10.13. DBSCAN clustering accurately identifies the 3 distinct rock rings.



DBSCAN has successfully identified the 3 rock rings. The algorithm succeeded where K-means had failed.

### Comparing DBSCAN and K-means

DBSCAN is an advantageous algorithm for clustering data composed of curving and dense shapes. Also, unlike K-means, the algorithm doesn’t require an approximation of the cluster count prior to execution. Additionally, DBSCAN can filter random outliers located in sparse regions of space. For example, if we add an outlier located beyond the boundary of the rings, then DBSCAN will assign it a cluster id of -1. The negative value indicates that the outlier cannot be clustered with the rest of the dataset.

##### NOTE

Unlike K-means, a fitted DBSCAN model cannot be re-applied to brand-new data. Instead, we’ll need to combine new and old data, and execute the clustering from scratch. The reason for this is obvious; computed K-means centers can easily be compared to additional data-points. However, the additional data-points could influence the density distribution of previously seen data, which forces DBSCAN to recompute all clusters.

##### This listing Finding outliers using DBSCAN

1

2

3

4

noisy\_data = rocks + [[1000, -1000]]

clusters = DBSCAN(eps=epsilon,

min\_samples=min\_points).fit\_predict(noisy\_data)

assert clusters[-1] == -1

copy

There is one other advantage to the DBSCAN technique that is missing from K-means. DBSCAN does not depend on the mean. Meanwhile, the K-means algorithm requires us to compute the mean coordinates of grouped points. As we discussed in Section Five, these mean coordinates will minimize the sum of squared distances to the center. The minimization property only holds if the squared distances are Euclidean. Thus, if our coordinates are not Euclidean, then the mean is not very useful, and the K-means algorithm should not be applied. However, the Euclidean distance is not the only metric for gaging separation between points. In fact, there exist infinite metrics for defining distance. We’ll explore a few of them in the subsequent sub-section. In the process, we will learn how to integrate these metrics into our DBSCAN clustering output.

### Clustering Based on Non-Euclidean Distance

Suppose we are visiting Manhattan. We wish to know the walking distance from the Empire State Building to Columbus Circle. The Empire State Building is located at the intersection of 34th street and 5th avenue. Meanwhile, Columbus Circle is located is located at the intersection of 57th street and 8th avenue. The streets and avenues in Manhattan are always perpendicular to each other. This lets us represent Manhattan as a 2D coordinate system, where streets are positioned on the x-axis and avenues are positioned on the y-axis. Under this representation, the Empire State Building is located at coordinate (34, 5) and Columbus Circle is located at coordinate (57, 8). We can easily calculate a straight-line Euclidean distance between the 2 coordinate points. However, that final length would be impassable because towering steel buildings occupy the area outlined by every city block. A more correct solution would be limited to a path across the perpendicular sidewalks that form the City’s grid. Such a route requires us to walk three blocks between 5th avenue and 3rd avenue, and then 23 blocks between 34th street and 57th street, for a distance of 26 blocks total. Manhattan’s average block-length is .17 miles, so we can estimate the walking distance as 4.42 miles. Lets compute that walking distance directly using a generalized manhattan\_distance function.

##### This listing Computing the Manhattan distance

1

2

3

4

5

6

7

8

9

def manhattan\_distance(point\_a, point\_b):

num\_blocks = np.sum(np.absolute(point\_a - point\_b))

return .17 \* num\_blocks

x = np.array([34, 5])

y = np.array([57, 8])

distance = manhattan\_distance(x, y)

print(f"Manhattan distance is {distance} miles")

**1**

copy

1

Manhattan distance **is** 4.42 miles

copy

Now, suppose we wish to cluster more than 2 Manhattan locations. We’ll assume each cluster holds a point that is within a 1-mile walk of 3 other clustered points. This assumption lets us apply DBSCAN clustering, using Scikit-Learn’s DBSCAN class. We’ll set eps to 1 and min\_samples to 3, during DBSCAN’s initialization. Furthermore, we will pass metric= manhattan\_distance into the initialization method. The metric parameter will swap Euclidean distance for our custom distance metric. Consequently, the clustering distance will correctly reflect the grid-based constraints within the City.

The code below will cluster Manhattan coordinates. Subsequently, these coordinates will be plotted on a grid, along with their cluster designations.

##### This listing Clustering using Manhattan distance

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

points = [[35, 5], [33, 6], [37, 4], [40, 7], [45, 5]]

clusters = DBSCAN(**eps**=1, **min\_samples**=3,

**metric**=manhattan\_distance).fit\_predict(points)

**for** i, cluster **in** enumerate(clusters):

point = points[i]

**if** cluster == -1:

print(f"Point at index {i} is an outlier")

plt.scatter(point[0], point[1], **marker**='x', **color**='k')

**else**:

print(f"Point at index {i} is in cluster {cluster}")

plt.scatter(point[0], point[1], **color**='g')

plt.grid(True, **which**='both', **alpha**=0.5)

plt.minorticks\_on()

plt.show()

**1**

**2**

**3**

copy

1

2

3

4

5

Point at **index** 0 **is** **in** **cluster** 0

Point at **index** 1 **is** **in** **cluster** 0

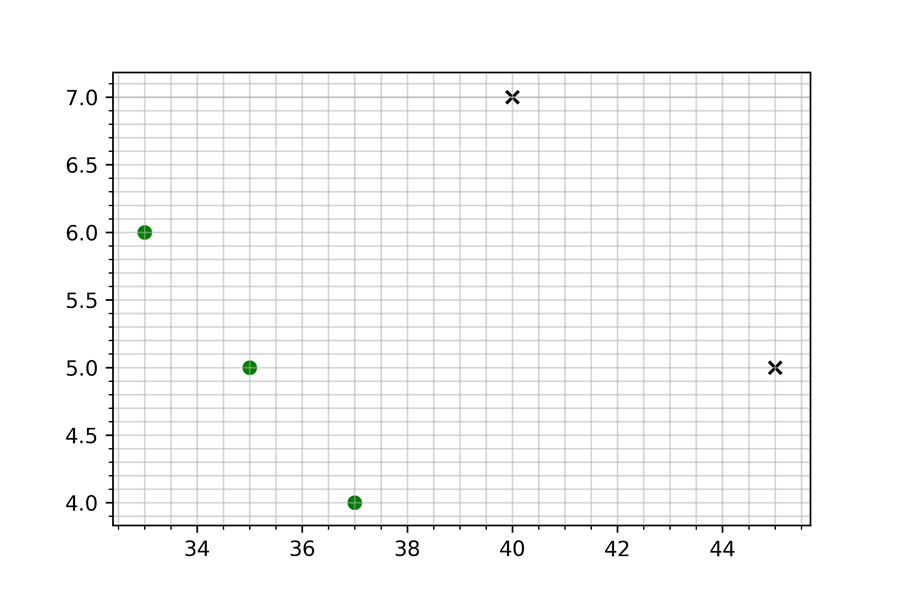
Point at **index** 2 **is** **in** **cluster** 0

Point at **index** 3 **is** an outlier

Point at **index** 4 **is** an outlier

copy

##### Figure-10.14. 5 points in a rectangular grid have been clustered using the Manhattan distance. The 3 points in the lower-left corner of the grid all fall within a single cluster. The remaining two points are outliers, marked by an x.



The first 3 locations fall within a single cluster, and the remaining points are outliers. Could we have detected that cluster using the K-means algorithm? Perhaps. After all, our Manhattan block coordinates can be averaged out, making them compatible with a K-means implementation. What if we swap Manhattan distance for a different metric where average coordinates are not so easily obtained? Lets define a non-linear distance metric with the following properties: two points are zero units apart if all their elements are negative, 2 units apart if all their elements are non-negative, and 10 units apart otherwise. Given this ridiculous measure of distance, can we compute the mean of any 2 arbitrary points? We can’t, and K-means cannot be applied. A weakness of the algorithm is that it depends on the existence of an average distance. Unlike K-means, the DBSCAN algorithm does not require our distance function to be linearly divisible. Thus, we can easily run DBSCAN clustering using our ridiculous distance metric.

##### This listing Clustering using a ridiculous measure of distance

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

def ridiculous\_measure(point\_a, point\_b):

is\_negative\_a = np.array(point\_a) < 0

is\_negative\_b = np.array(point\_b) < 0

**if** is\_negative\_a.all() **and** is\_negative\_b.all():

**return** 0

elif is\_negative\_a.any() **and** is\_negative\_b.any():

**return** 10

**else**:

**return** 2

points = [[-1, -1], [-10, -10], [-1000, -13435], [3,5], [5,-7]]

clusters = DBSCAN(eps=.1, min\_samples=2,

metric=ridiculous\_measure).fit\_predict(points)

**for** i, cluster **in** enumerate(clusters):

point = points[i]

**if** cluster == -1:

print(f"{point} is an outlier")

**else**:

print(f"{point} falls in cluster {cluster}")

**1**

**2**

**3**

**4**

copy

1

2

3

4

5

[-1, -1] falls **in** cluster 0

[-10, -10] falls **in** cluster 0

[-1000, -13435] falls **in** cluster 0

[3, 5] **is** an outlier

[5, -7] **is** an outlier

copy

Running DBSCAN with our ridiculous\_measure metric leads to the clustering of negative coordinates into a single group. All other coordinates are treated as outliers. These results are not conceptually practical. Still, the flexibility with regards to custom metric usage is much appreciated. We are not constrained in our metric choice! We could for instance, set the metric to compute traversal distance based on the curvature of the Earth. Such a metric would be particularly useful for clustering geographic locations.

DBSCAN Clustering Methods

dbscan\_model = DBSCAN(eps=epsilon, min\_samples=min\_points): Creates a DBSCAN model that is intended to cluster by density. A dense point is defined as having at-least min\_points neighbors within a distance of epsilon. The neighbors are considered to be part of the same cluster as the point.

clusters = `dbscan\_model.fit\_predict(data): Executes DBSCAN on inputted data, using an initialized DBSCAN object. The clusters array contains cluster ids. The cluster id of data[i] is equal to clusters[i]. Unclustered outlier points are assigned an id of -1.

clusters = DBSCAN(eps=epsilon, min\_samples=min\_points).fit\_predict(data): Executes DBSCAN in a single line of code, and returns the resulting clusters.

dbscan\_model = DBSCAN(eps=epsilon, min\_samples=min\_points, metric=metric\_function): Creates a DBSCAN model where the distance metric is defined by a custom metric function. The metric\_function distance metric does not need to be Euclidean.

### Limitations of the DBSCAN Algorithm

DBSCAN does carry certain drawbacks. The algorithm is intended to detect clusters with similar point-density distributions. However, real-world data varies in density. For instance, pizza shops in Manhattan are distributed more densely than the pizza shops in Orange County. Thus, we might have trouble choosing density parameters that will let us cluster shops in both locations. This brings us a to a second limitation of the algorithm. DBSCAN requires meaningful values for the eps and min\_samples parameters. In particular, varying eps inputs will greatly impact the quality of clustering. Unfortunately, there is no one reliable procedure for estimating the appropriate eps. While certain heuristics are occasionally mentioned in the literature, their benefit is minimal. Most of the time, we must rely on our gut-level understanding of the problem in order to assign practical inputs to the 2 DBSCAN parameters. For example, if we were to cluster a set of geographic locations, then our eps and min\_samples values would depend on whether the locations are spread out across the entire globe or whether they are constrained to a single geographic region. In each instance, our understanding of density and distance would vary. Generally speaking, if we are clustering random cities spread out across the Earth, then we can set the min\_samples and eps parameters to equal 3 cities and 250 miles, respectively. This will assume each cluster holds a city that is within 250 miles of at-least 3 other clustered cities. For a more regional location distribution, a lower eps value will be required.

## Analyzing Clusters Using Pandas

So far, we have kept separate our data-inputs and our clustering outputs. For instance, in our rock-ring analysis, the input data is held in rocks list while the clustering output is held in a rock\_clusters array. Tracking both the coordinates and the clusters requires us to map indices between the input list and the output array. Thus, if we wish to extract all the rocks in cluster zero, then we must obtain all instances of rocks[i] where rock\_clusters[i] == 0. This index analysis is slightly convoluted. We can more intuitively analyze clustered rocks by combining the coordinates and the clusters together in a single Pandas table.

The following code will create a Pandas table. The table will hold 3 columns; X, Y, and Cluster. Each i-th row in the table with hold the x-coordinate, the y-coordinate, and the cluster of the rock located at rocks[i].

##### This listing Storing clustered coordinates in a table

1

2

3

4

**import** pandas **as** pd

x\_coordinates, y\_coordinates = np.**array**(rocks).T

df = pd.DataFrame({'X': x\_coordinates, 'Y': y\_coordinates,

'Cluster': rock\_clusters})

copy

Our Pandas table lets us easily access the rocks in any cluster. Lets plot those rocks that fall into cluster zero, using techniques described in Section Eight.

##### This listing Plotting a single cluster using Pandas

1

2

3

df\_cluster = df[df.Cluster == 0]

plt.scatter(df\_cluster.X, df\_cluster.Y)

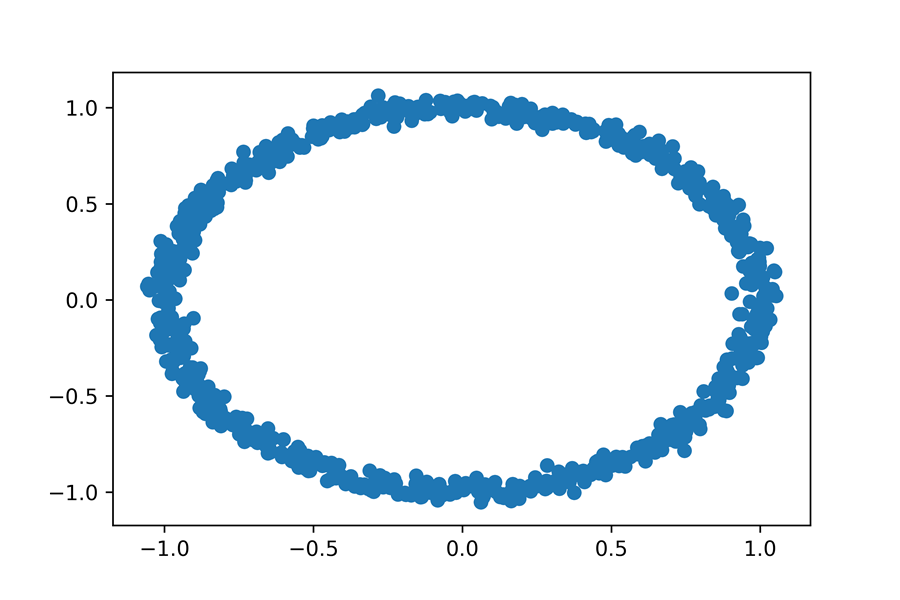
plt.show()

**1**

**2**

copy

##### Figure-3.15. Rocks that fall into cluster zero.



Pandas allows us to obtain a table containing elements from any single cluster. Alternatively, we might want to obtain multiple tables, where each table maps to a cluster id. In Pandas, this can easily be done by calling df.groupby('Cluster'). The groupby method will create 3 tables; one for each cluster. It will return an iterable over the mappings between cluster ids and tables. Lets use the groupby method to iterate over our 3 clusters. We’ll subsequently plot the rocks in cluster 2 and cluster 3, but not the rocks in cluster zero.

##### NOTE

Calling df.groupby('Cluster') returns more than just an iterable. It returns a DataFrameGroupBy object, which provides additional methods for cluster filtering and analysis.

##### This listing Iterating over clusters using Pandas

1

2

3

4

5

6

7

8

9

10

**for** cluster\_id, df\_cluster **in** df.groupby('Cluster'):

**if** cluster\_id == 0:

print(f"Skipping over cluster {cluster\_id}")

**continue**

print(f"Plotting cluster {cluster\_id}")

plt.scatter(df\_cluster.X, df\_cluster.Y)

plt.show()

**1**

copy

1

2

3

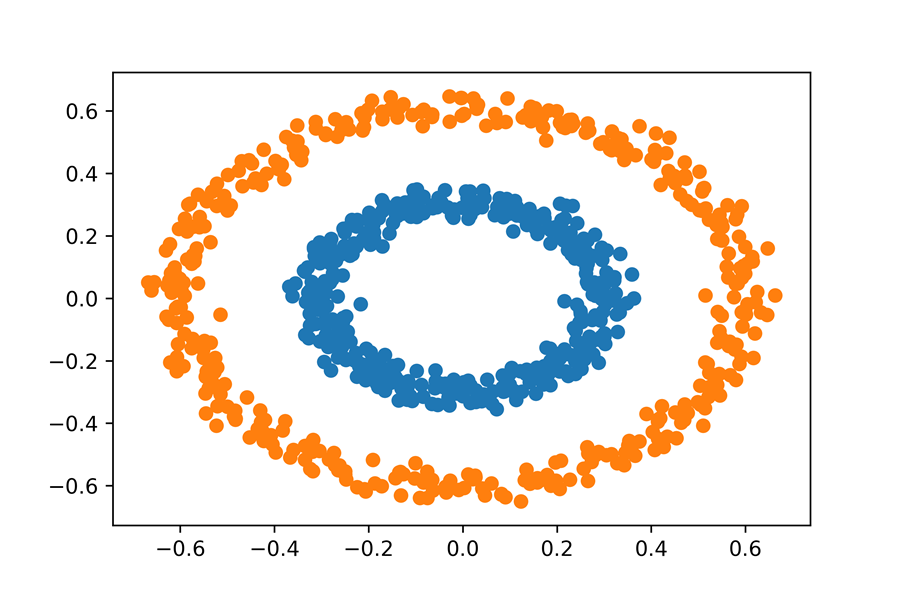
Skipping **over** **cluster** 0

Plotting **cluster** 1

Plotting **cluster** 2

copy

##### Figure-10.16. Rocks that fall into clusters 1 and 2.



The Pandas groupby method lets us iteratively examine different clusters. This could prove useful in our Case Study Three analysis.

## Summary

* The **K-means** algorithm clusters inputted data by searching for K **centroids**. These centroids represent the mean coordinates of the discovered data groups. K-means is initialized by selecting K random centroids. Each data-point is then clustered based on its nearest centroid. Afterwards, the centroids are iteratively recomputed until they converge on stable locations.
* K-means is guaranteed to converge to a solution. However, that solution might not be the optimal solution.
* K-means requires Euclidean distance to distinguish between points. The algorithm is not intended to cluster non-Euclidean coordinates.
* After executing K-means clustering, we can compute the **inertia** of the result. Inertia equals the sum of the squared distances between each data-point and its closest center.
* Plotting the inertia across a range of K-values will generate an **Elbow plot**. The elbow component within the elbow-shaped plot should point downwards to a reasonable K-value. Using the Elbow plot, we can heuristically select a meaningful K input into K-means.
* The **DBSCAN** algorithm clusters data based on density. Density is defined using the epsilon and min\_points parameters. If a point is located within an epsilon distance of min\_point neighbors, then that point is in a dense region of space. Every neighbor of a point in a dense region of space will also cluster in that space. DBSCAN iteratively expands the boundaries of a dense region of space until a complete cluster is detected.
* Points in non-dense region are not clustered by the DBSCAN algorithm. They are treated as outliers.
* DBSCAN is an advantageous algorithm for clustering data composed of curving and dense shapes.
* DBSCAN can cluster using arbitrary, non-Euclidean distances.
* There is no reliable heuristic for choosing appropriate epsilon and min\_points parameters. However, if we wish to cluster global cities, then we can set the 2 parameters to 250 miles and 3 cities, respectively.
* Storing clustered data in a Pandas table allows us to intuitively iterate over clusters with the groupby method.

# **3.3 Plotting Maps Using Basemap**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-11)

Visualizing geographic data is a common data science task. One external library used to map such data is Basemap; a Matplotlib extension for generating maps in Python. Lets install the Basemap library. Once installation is complete, we will proceed to import the Basemap mapping class. Afterwards, we’ll initialize the class as map\_plotter = Basemap().

##### NOTE

Call "conda install Basemap" from the command-line terminal in order to install the Basemap library.

##### This listing Initializing the Basemap mapping class

1

2

from mpl\_toolkits.**basemap** import **Basemap**

map\_plotter = **Basemap()**

copy

We are ready to visualize the Earth, by plotting the coastline boundaries of all 7 continents. We’ll generate the coastline plot by executing map\_plotter.drawcoastlines(). Afterwards, we’ll visualize the plot in Matplotlib, by calling plt.show().

##### This listing Visualizing the Earth using Basemap

1

2

3

fig = plt.figure(figsize=(12, 8))

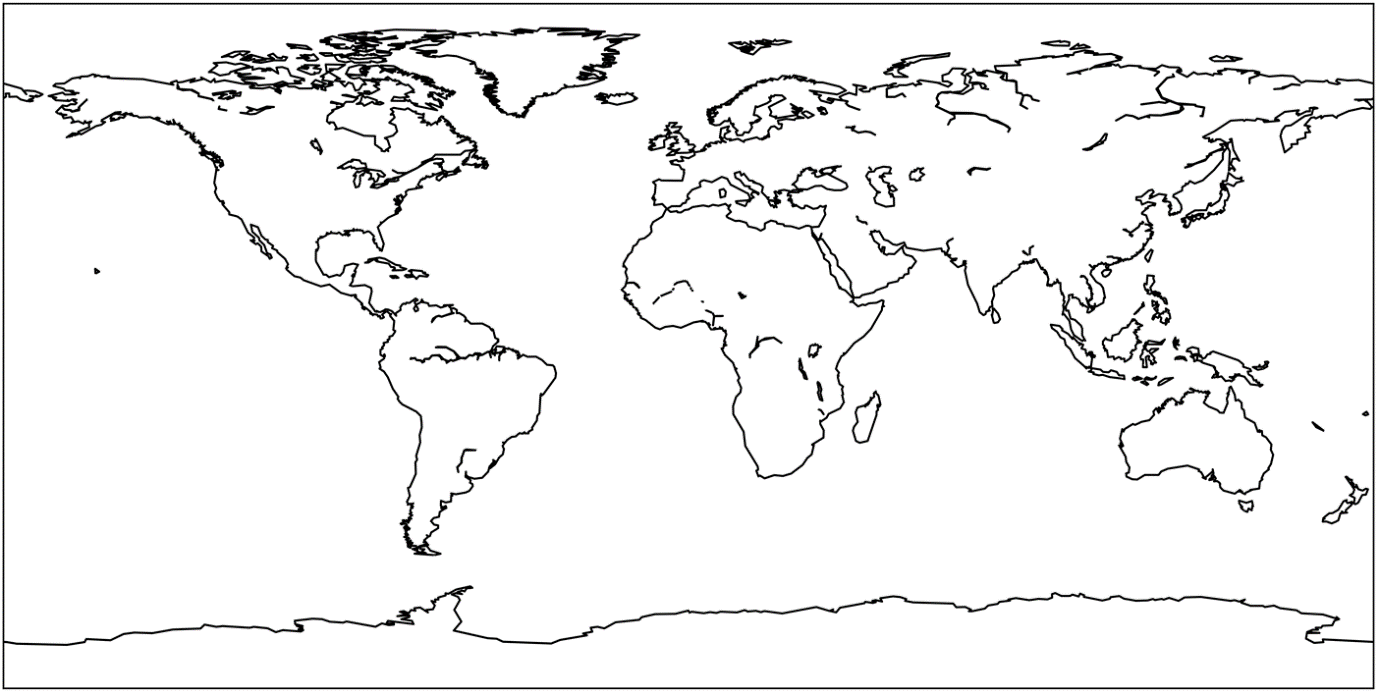
map\_plotter.drawcoastlines()

plt.show()

**1**

copy

##### Figure-11.3. A standard map of the Earth, in which the coastlines of the continents have been plotted.



The map uses a standard **Equidistant cylindrical projection**, in which the spherical globe is superimposed on an unrolled cylinder. This is the most popular 2D map representation. Consequently, the Basemap class is preset to display geographic data in this manner.

##### NOTE

We can also manually specify the Equidistant cylindrical crojection while initializing the Basemap class. To do this, we must execute ` map\_plotter = Basemap(projection='cyl')`.

Our visualized map is composed of coastal boundaries, which outline all 7 continents. National boundaries are currently missing from the plot. We can incorporate country boundaries by calling the map\_plotter.drawcountries() method.

##### This listing Mapping coastlines and countries

1

2

3

4

fig = plt.figure(figsize=(12, 8))

map\_plotter.drawcoastlines()

map\_plotter.drawcountries()

plt.show()

**1**

copy

##### Figure-11.4. A standard map of the Earth where coastlines and national boundary lines are present.



So far our map looks sparse and uninviting. We can improve the quality by calling map\_plotter.shadedrelief(). The method-call will color the map using topographic information. Oceans will be colored blue, and forested regions will be colored green.

##### This listing Coloring a map of the Earth

1

2

3

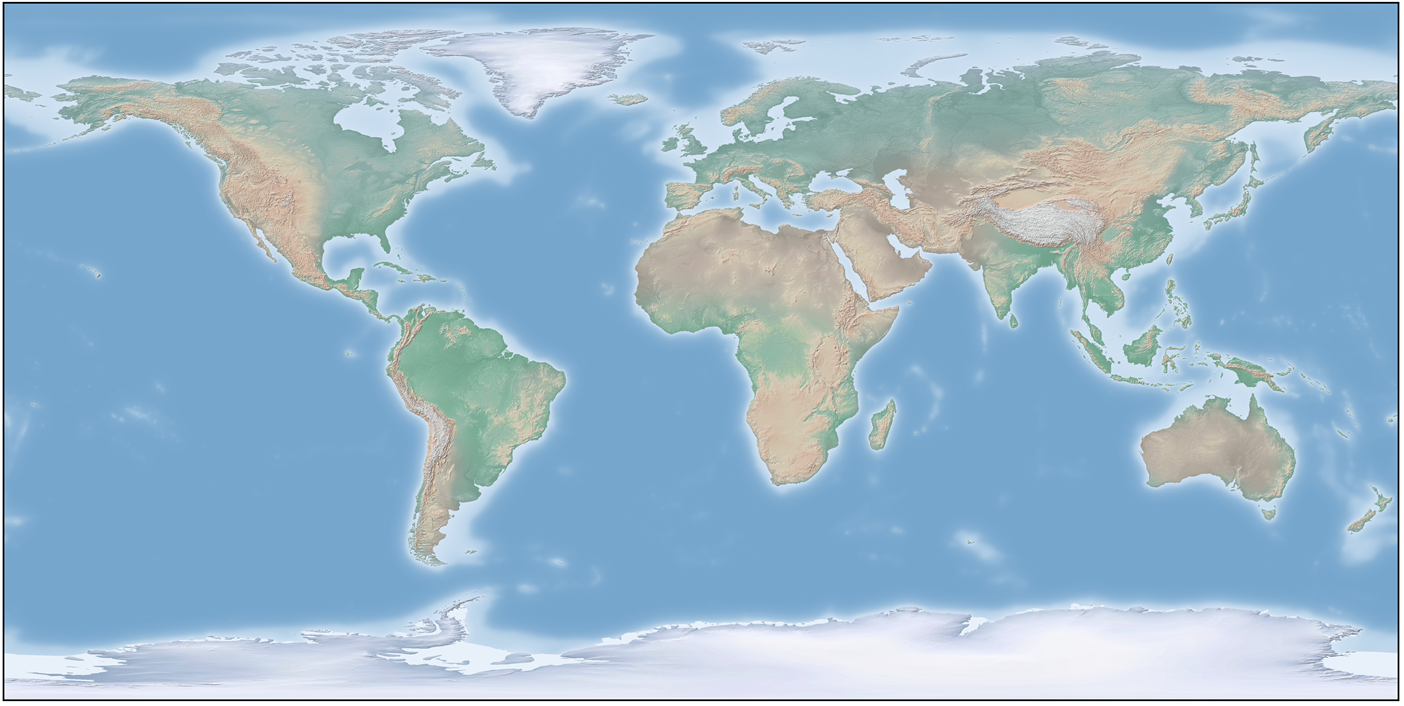
fig = plt.figure(figsize=(12, 8))

map\_plotter.shadedrelief()

plt.show()

copy

##### Figure-11.5. A standard map of the Earth that has been colored to contain oceanographic and topographic details.



Suppose we are given a list of locations defined by pairs of latitudes and longitudes. We can plot these locations on our map by separating the latitudes from the longitudes and then passing the results into map\_plotter.scatter.

##### NOTE

We’ll also need to pass lattlon=True into the map\_plotter.scatter method, so that the plotted points are treated as spherical coordinates.

##### This listing Plotting coordinates on a map

1

2

3

4

5

6

7

8

9

fig = plt.figure(figsize=(12, 8))

coordinates = [(39.9526, -75.1652), (37.7749, -122.4194),

(40.4406, -79.9959), (38.6807, -108.9769),

(37.8716, -112.2727), (40.7831, -73.9712)]

latitudes, longitudes = np.array(coordinates).T

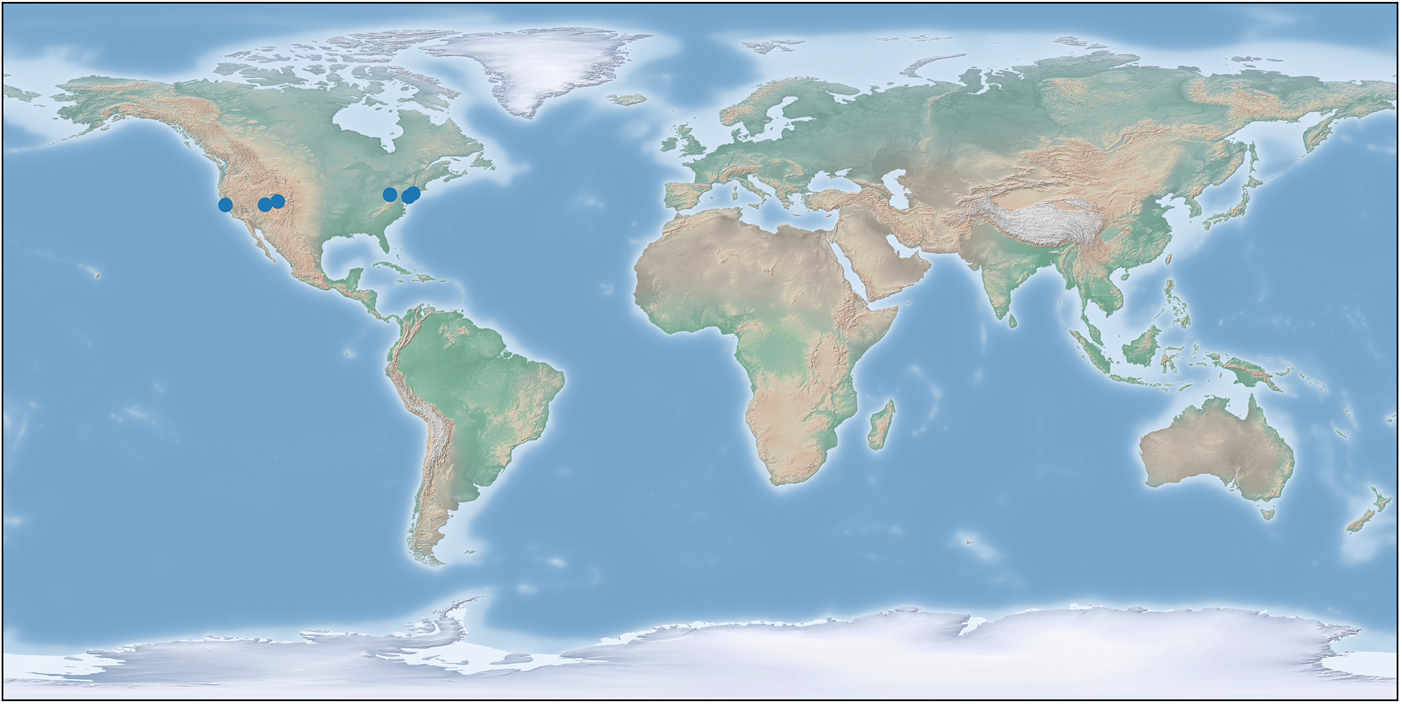
map\_plotter.scatter(longitudes, latitudes, latlon=True)

map\_plotter.shadedrelief()

plt.show()

copy

##### Figure-11.6. A standard map of the Earth with plotted latitude and longitude coordinates.



The plotted points all appear within the boundaries of North America. We thusly can simplify the map by zooming in on North America. In order to adjust the map, we must alter our projection. Lets implement an **Orthographic projection**. This projection plots the Earth from the perspective of a viewer in the outer reaches of the galaxy. The distant observer cannot see the entire Earth, but only a part of it. We’ll center the perspective of the viewer on North America. Specifically, we’ll center their perspective on a latitude and longitude of (40, -95).

We’ll generate the Orthographic projection by initializing the Basecamp class as ` Basemap(projection='ortho', lat\_0=40, lon\_0=-95). The `projection parameter specifies the projection type, while lat\_0 and lon\_0 denote the center of the projected perspective. We’ll use our new mapping object to plot coordinates within North America.

##### This listing Plotting North American Coordinates

1

2

3

4

5

6

fig = plt.figure(figsize=(12, 8))

map\_ortho = Basemap(**projection**='ortho', **lat\_0**=40, **lon\_0**=-95)

map\_ortho.scatter(longitudes, latitudes, **latlon**=True,

**s**=70)

map\_ortho.drawcoastlines()

plt.show()

**1**

copy

##### An Orthographic projection of North America with plotted latitude and longitude coordinates.



We successfully zoomed in on North America. Now, we’ll zoom in further, onto the United States. Unfortunately, the Orthographic projection will prove insufficient for this purpose. Instead, we will on rely on the **Lambart conformal conic projection**, (more commonly called the LCC). By setting our projection parameter to equal 'lcc', we can create a 2D map of the United States. Additional parameters will also be required to execute the projection properly. These supplementary parameters are present in the code below, which plots coordinates on the zoomed United States.

##### NOTE

We need all these parameters because the LCC is a complex, multi-step procedure. Initially, the projection places a cone on-top of the spherical Earth. The cone’s circular base covers the region we intend to map. Afterwards, coordinates in the region are projected onto the surface of the cone. Finally, the cone is unrolled to create a 2D map. This process requires many fine-tuned parameters to execute appropriately. The functionality of the parameters is beyond the scope of this section.

##### This listing Plotting USA Coordinates

1

2

3

4

5

6

7

8

fig = plt.figure(figsize=(12, 8))

map\_lcc = Basemap(**projection**='lcc', **lon\_0**=-95, **llcrnrlon**=-119,

**llcrnrlat**=22, **urcrnrlon**=-64, **urcrnrlat**=49, **lat\_1**=33,

**lat\_2**=45)

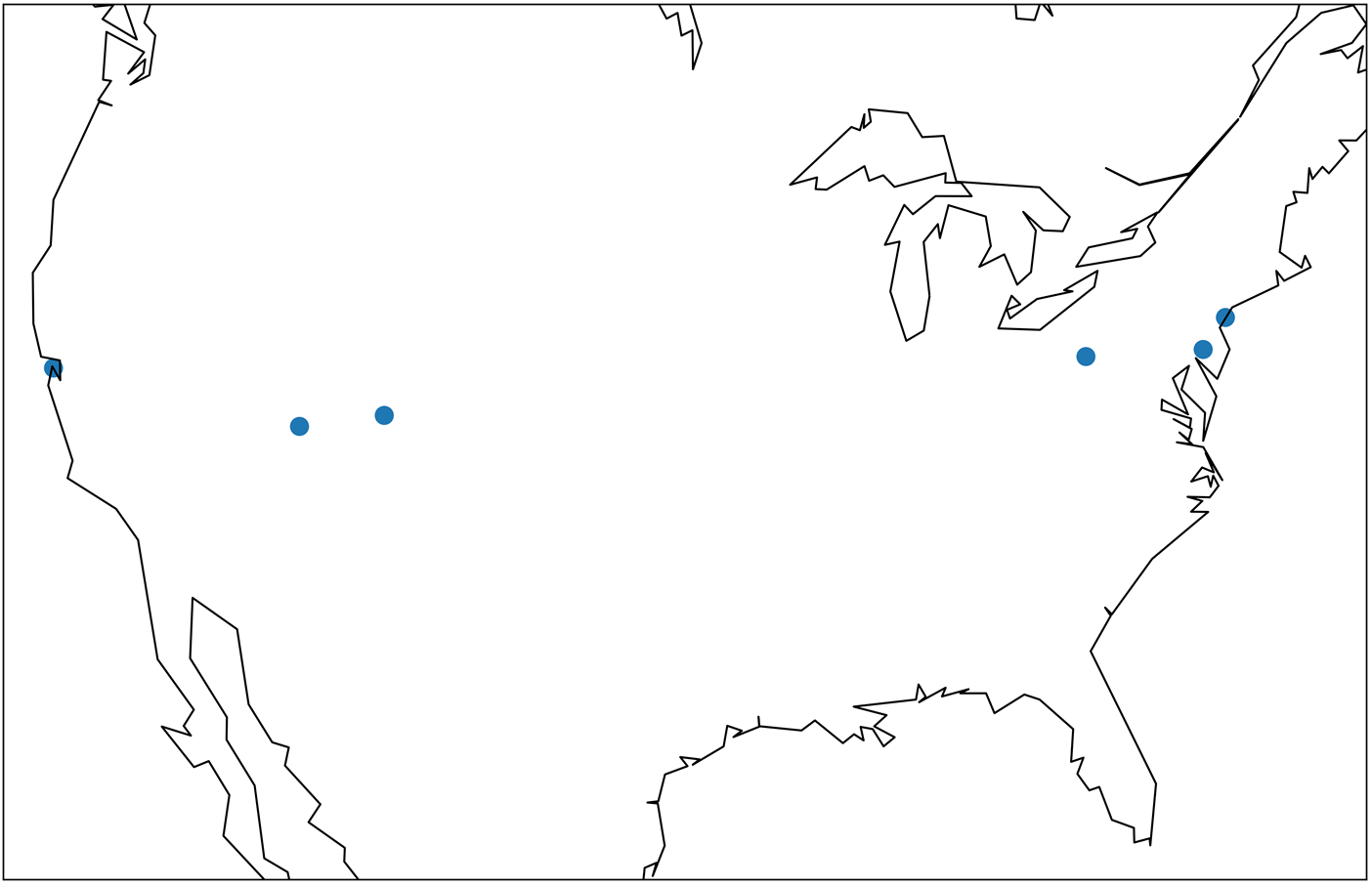
map\_lcc.scatter(longitudes, latitudes, **latlon**=True, **s**=70)

map\_lcc.drawcoastlines()

plt.show()

copy

##### An LCC projection of the United States with plotted latitude and longitude coordinates.



Our map of the United States is looking a little sparse. Lets add state boundaries to the map by calling map\_lcc.drawstates().

##### This listing Mapping state boundaries in the USA

1

2

3

4

5

fig = plt.figure(figsize=(12, 8))

map\_lcc.scatter(longitudes, latitudes, **latlon**=True, **s**=70)

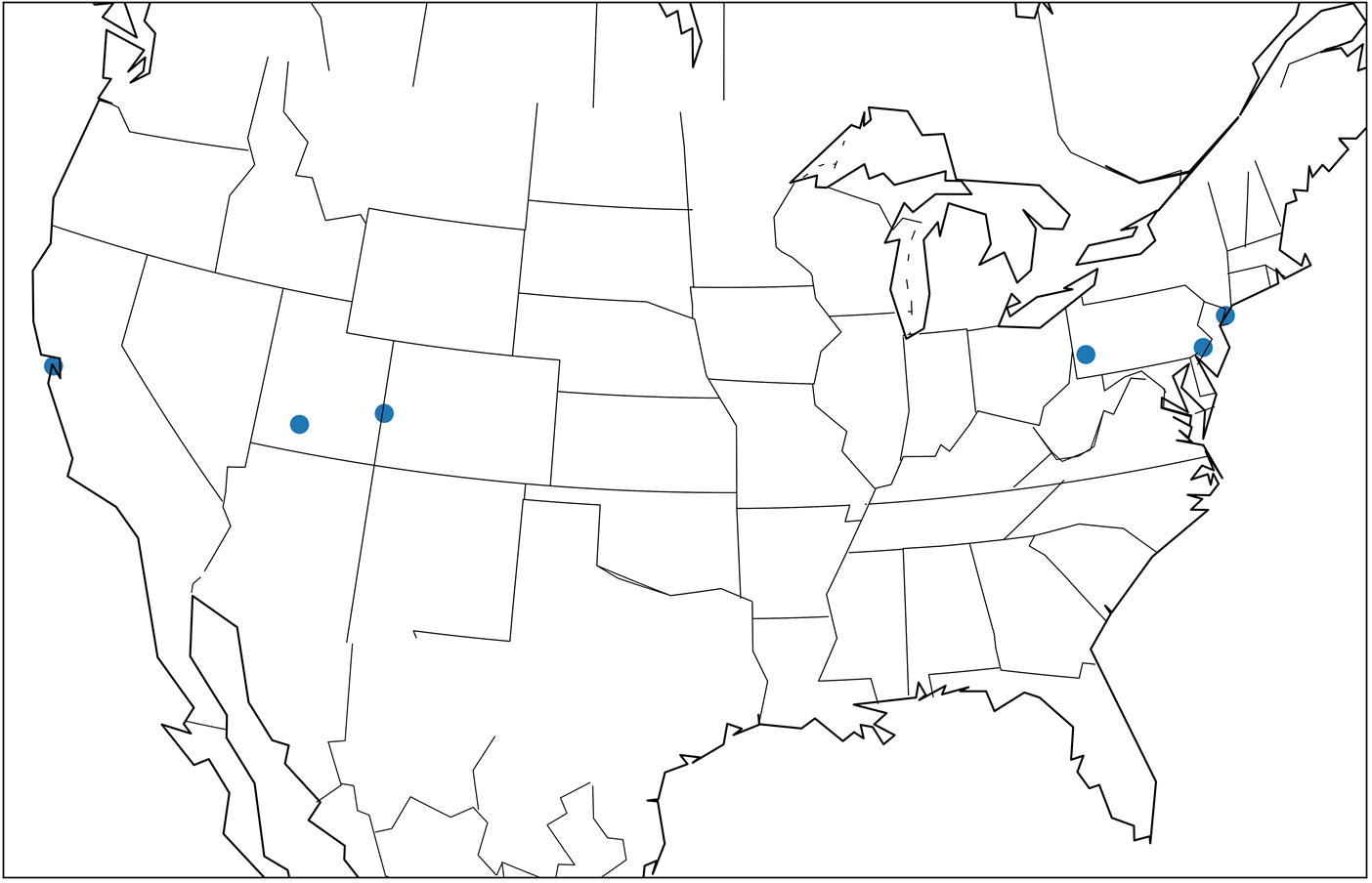
map\_lcc.drawcoastlines()

map\_lcc.drawstates()

plt.show()

copy

##### An LCC projection of the United States where state boundaries are present..



Common Basemap methods

map\_plotter = Basemap(): Creates a Basemap object for generating maps using an Equidistant cylindrical projection.

map\_plotter = Basemap(projection='cyl'): Creates a Basemap where the Equidistant cylindrical projection is set explicitly.

map\_plotter.drawcoastlines(): Plots continental coastlines on a map.

map\_plotter.drawcountries(): Plots national boundaries on a map.

map\_plotter.shadedrelief(): Colors a plotted map, using topographic information.

map\_plotter.scatter(longitudes, latitudes, latlon=True): Plots latitude and longitude coordinates on a map.

map\_ortho = Basemap(projection='ortho', lat\_0=lat, lon\_0=lon): Creates a Basemap object for generating maps using an Orthographic projection. The projection’s perspective is centered on lat and lon.

map\_ortho = Basemap(projection='ortho', lat\_0=40, lon\_0=-95): Creates a Basemap object for generating a map of North America using an Orthographic projection.

map\_lcc = Basemap(projection='lcc', lon\_0=-95, llcrnrlon=-119, llcrnrlat=22, urcrnrlon=-64, urcrnrlat=49, lat\_1=33, lat\_2=45): Creates a Basemap object for generating a map of the United States using a Lambart conformal conic projection.

map\_lcc.drawstates(): Plots US state boundaries on a map.

Basemap allows us to plot any location on a map. All we need is the location’s latitude and longitude. Of course, we must first know these geographic coordinates prior to plotting them on a map. Thus, we need a mapping between location names and their geographic properties. That mapping is provided by the GeoNamesCache location-tracking library.

# **3.4 Submit Your Work**

The deliverable from this section is another Jupyter Notebook documenting your work as you cluster the headlines with the coordinates and produce a world map with clustered headlines. The map should show the locations of headlines colored by the cluster assignment. We will use this DataFrame and map to find disease outbreaks in the next section. A sample of the DataFrame showing the structure and content is below along with a partial map view:

Upload a link to your Jupyter Notebook (preferably hosted on GitHub) in the blank below and hit submit. After submitting, you can view an example solution in the next section.

# **Solution**

Below is an example of the structure and content of the pandas DataFrame you should create in this section. The last column shows the cluster assignment of each headline.

Below is the complete world map produced in this section. Yours may appear slightly different based on the algorithm and parameters you used (and the colors for the map).

For an in-depth solution to this section, refer to the notebook below. Your end outcome pandas DataFrame should be similar, but your workflow may differ from the notebook.

* [3. Clustering Headlines Based on Location.ipynb](https://liveproject-resources.s3.amazonaws.com/93/22368/2020-03-03-09-35-24/3.%20Clustering%20Headlines%20Based%20on%20Location.ipynb)

# 4. Identifying Disease Outbreaks

# **4.1 Discovering Pandemics within the United States and Worldwide**

**Objective**

* Identify pandemics by analyzing the centrally located headlines in the largest clusters of headlines. Look at the prevalence of disease in each cluster. Any disease outbreaks in the United States or around the world will be reported back to superiors at the WHO to direct help where it’s needed.

**Workflow**

1. Separate the data into world and US datasets.
   * This can be done using the countrycode for each city from geonamescache.
2. Re-cluster the world and US data separately.
   * This may require different clustering parameters for each dataset.
   * Put your code into functions to quickly cluster and visualize results of the model.
3. Sort the clusters by the number of headlines from largest to smallest.
4. Sort headlines within each cluster by finding headlines closest to the center of the cluster.
   * You can average the location coordinates in each cluster to get an approximation of the center. If you want to be more accurate, then you’ll need to apply a geographic formula to find the true geographic center of each cluster.
5. Examine the headlines closest to the center of each cluster. Record any repeated diseases in the headlines. Do this for both the US and the world data.
   * If no diseases appear to be repeating, try the clustering again. Also make sure to look at enough headlines within each cluster to find repeated diseases.
6. Pinpoint disease outbreaks based on repeated diseases within headlines in clusters. Prepare final maps of the United States and the world.
   * If you find one or two major diseases, you can find all mentions of it in headlines using regular expressions.
   * Remove any irrelevant headlines from the map for presentation.

**Importance to project**

* The clustering has separated out the relevant data for us to examine, and now we have to find information that could help the WHO. With limited resources, we need to prioritize our responses, therefore, we focus on the largest clusters. Once we have found a disease with numerous mentions, then we can search for clusters where the disease has a high prevalence.
* If a cluster contains a large percentage of a single disease, then it’s likely an indication of an epidemic. If there are multiple clusters of the disease around the world, then we have found a pandemic requiring immediate response.

**Notes**

* We are making several assumptions in this section, such as using the average of geographic coordinates as the center of a cluster. Doing effective data science usually requires making some assumptions. These are acceptable as long as they are reasonable. If in doubt, ask a domain expert what to do in a situation. Sometimes we make an approximation to get an initial answer, and then refine our methods through iteration to increase accuracy.
* We examined the headlines closest to the center of each cluster, which assumes that proximity to the center of the cluster is relevant. Try a different approach and check the results.
* People often use pandemic and epidemic as if they mean the same thing, but they are in fact, quite different. Epidemics affect a greater amount of people than what is usual for the location or spread to areas that don’t usually have any interaction with said disease. A pandemic, on the other hand, is an epidemic that happens world-wide.

**Resources**

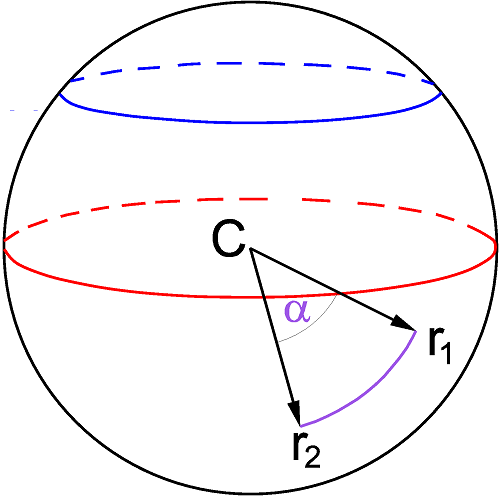
* Chapter 11, [Computing Distances Between 2 Global Points](https://liveproject.manning.com/module/93_5_2/discovering-disease-outbreaks-from-news-headlines/4--identifying-disease-outbreaks/4-2--computing-distances-between-2-global-points?), from Data Science Bookcamp by Leonard Apeltsin in section 4.2 can help you with finding distances between headlines.
* You may also want to refer back to [Python and Pandas](https://liveproject.manning.com/module/93_3_2/discovering-disease-outbreaks-from-news-headlines/2--finding-geographic-locations-of-headlines/2-2--python-and-pandas?), Chapter 24 in The Quick Python Book, Third Edition, by Naomi Ceder in section 2.2 if you need help manipulating dataframes.

# **4.2 Computing Distances Between 2 Global Points**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-11)

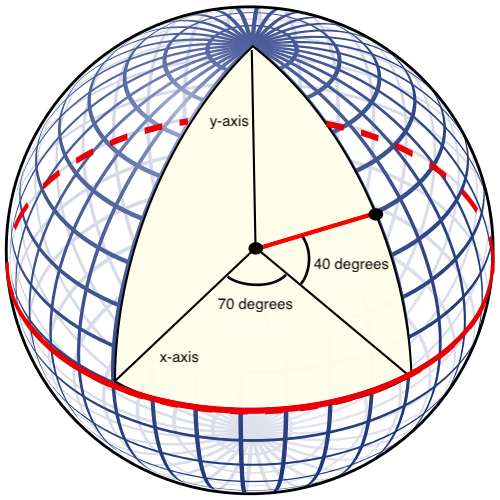
What is the shortest travel distance between any pair of points on Earth? The distance cannot be a straight line, since direct linear travel would require us to burrow deep through the Earth’s crust. A much more realistic path entails travelling along our spherical planet’s curved surface. This direct path between 2 points along the surface of a sphere is called the **great-circle distance**.

##### Figure-11.1. Visualizing the great-circle distance between 2 points on the surface of a sphere. These points are labeled as r1 and r2. A curved arc designates the traveling distance between them. The arc-length is equal to the radius of the sphere multiplied by ⍺, where ⍺ is the angle between points relative to the sphere’s center at C.



Computing the great-circle distance is easy. Our analysis simply requires a sphere and 2 points on that sphere. Any point on the sphere’s surface can be represented using **spherical coordinates** x and y, where x and y measure the angles of the point relative to the x-axis and y-axis.

##### Figure-11.2. Representing a point on the surface of a sphere using spherical coordinates. The point arises as we rotate 70 degrees away from the x-axis, and 40 degrees towards the y-axis. Hence, its spherical coordinates are (70, 40).



Lets define a basic great\_circle\_distance function, which will take as input 2 pairs of spherical coordinates. For simplicity’s sake, we will assume that the coordinates are present on a unit-sphere, with a radius of 1. This simplification allows us to define great\_circle\_distance in just 4 lines of code. The function will depend on a series of well-known trigonometric operations. A detailed derivation of these operations is beyond the scope of this book.

##### This listing Defining a great-circle distance function

1

2

3

4

5

6

7

from math **import** cos, sin, asin

def great\_circle\_distance(x1, y1, x2, y2):

delta\_x, delta\_y = x2 - x1, y2 - y2

haversin = sin(delta\_x / 2) \*\* 2 + np.product([cos(x1), cos(x2),

sin(delta\_y / 2) \*\* 2])

**return** 2 \* asin(haversin \*\* 0.5)

**1**

**2**

**3**

copy

Python’s trigonometric functions assume that the input angle is in radians, where 0 degrees equal 0 radians and 180 degrees equal π radians. Lets calculate the great-circle distance between 2 points that lie 180 degrees apart, relative to both the x-axis and the y-axis.

##### NOTE

Radians measure the length of a unit-circle arc relative to an angle. The maximum arc-length equals the unit-circle circumference of 2π. Traversing the circumference of a circle requires a 360 degree angle. Thus, 2π radians equal 360 degrees, and a single degree equals π / 180 radians.

##### This listing Computing the great-circle distance

1

2

3

from math import pi

distance = great\_circle\_distance(0, 0, pi, pi)

print(f"The distance equals {distance} units")

copy

1

The distance equals 3.141592653589793 units

copy

The points are exactly π units apart, half the distance required to circumnavigate a unit-circle. That value is the longest possible distance one can travel between 2 spherical points. This is akin to traveling between the North and South Poles of any planet. We’ll confirm by analyzing the latitudes and longitudes of Earth’s North Pole and South Pole. Terrestrial latitudes and longitudes are spherical coordinates that are measured degrees. Lets begin by recording the known coordinates of each pole.

##### This listing Defining the coordinates of the Earth’s poles

1

2

latitude\_north, longitude\_north = (90.0, 0)

latitude\_south, longitude\_south = (-90.0, 0)

**1**

copy

Latitudes and longitudes measure spherical coordinates in degrees, not radians. We’ll thus convert to radians from degrees using the np.radians function. The function takes as input an list of degrees, and returns a radian array. This result can subsequently be inputted into great\_circle\_distance.

##### This listing Computing the great-circle distance between poles

1

2

3

4

to\_radians = np.radians([latitude\_north, longitude\_north,

latitude\_south, longitude\_south])

distance = great\_circle\_distance(\*to\_radians.tolist())

print(f"The unit-circle distance between poles equals {distance} units")

**1**

copy

1

The unit-circle distance **between** poles **equals** 3.141592653589793 units

copy

As expected, the distance between poles on a unit-sphere is π . Now, let’s measure the distance between 2 poles here on Earth. The radius of the Earth is not 1 hypothetical unit, but rather 3956 actual miles. Therefore, we must multiply distance by 3956 to obtain a terrestrial measurement.

##### This listing Computing the travel distance between Earth’s poles

1

2

earth\_distance = 3956 \* distance

**print**(f"The distance between poles equals {earth\_distance} miles")

copy

1

The distance **between** poles **equals** 12428.14053760122 miles

copy

The distance between the 2 is approximately 12,400 miles. We were able to compute it by converting the latitudes and longitudes to radians, calculating their unit-sphere distance, and then multiplying that value by the radius of Earth. We can now create general travel\_distance function to calculate the travel mileage between any 2 terrestrial points.

##### This listing Defining a travel distance function

1

2

3

4

to\_radians = np.radians([lat1, lon1, lat2, lon2])

**return** 3956 \* great\_circle\_distance(\*to\_radians.tolist())

assert travel\_distance(90, 0, -90, 0) == earth\_distance

copy

Our travel\_distance function is non-Euclidean metric for measuring distances between locations. As the discussed in the previous section, we can pass such metrics into the DBSCAN clustering algorithm. Consequently, we can leverage travel\_distance to cluster locations based on their spatial distributions. Afterwards, we can visuallly validate the clusters by plotting the locations on a map. This map-plot can executed using the external Basemap visualization library.

# **4.3 Submit Your Work**

The deliverable from this section is a Jupyter Notebook documenting your work as you find disease outbreaks in the United States and worldwide. Show the information associated with the major outbreak. Produce a world map showing any disease outbreaks identified. Below is a sample of one cluster with the headlines in that cluster as well as a partial map showing the major disease outbreak:

Upload a link to your Jupyter Notebook (preferably hosted on GitHub) in the blank below and hit submit. After submitting, you can view an example solution in the next section.

# **4.4 Solution**

Below is an example of the structure and content of the pandas DataFrame you should create in this section. We’ve cut down the DataFrame to only the headlines located in the major disease outbreak. The information from this DataFrame can be used to address the outbreak.

Below is a map of the major worldwide disease outbreak—Zika—which is present in four main locations. We’ve kept only the largest clusters of headlines mentioning the disease. A map is useful for getting an overview of the situation and highlighting patterns.

A Jupyter Notebook with an example solution is below. Keep in mind that your workflow may differ from this notebook, but the overall outcome should be similar. The major disease outbreak is quite evident and should be identified.

* [4. Identifying Disease Outbreaks.ipynb](https://liveproject-resources.s3.amazonaws.com/93/22372/2020-03-03-09-35-57/4.%20Identifying%20Disease%20Outbreaks.ipynb)

# 5. Presenting the Disease Outbreak Data

# **5.1 Conclusions Drawn from Worldwide Pandemics Presentation**

**Objective**

* In this step you will do something different to show the final result to your colleagues. After all the hard (but rewarding) data science work, now comes the most important part: present your findings to your superiors at the World Health Organization in a short, easily understood document. The document should summarize your findings (perhaps in a single sentence and one image) and be accessible to a general audience.

**Workflow**

1. Write an executive summary notebook with the actionable takeaway and a visual.
2. Use the nbconvert library to convert the Jupyter Notebook into HTML or pdf.
   * Converting to pdf requires installing latex

**Importance to project**

* The purpose of data science is to enable better decisions and direct resources more efficiently by analyzing large datasets. We can’t affect any decisions without communicating our results clearly. In many respects, presentation of outputs is the most valuable part of a project.

**Notes**

* Effective communication to a nontechnical audience focuses on the what rather than the how. Tell people why they should care about your results—in this case, because lives are at stake!
* Avoid mentioning how you came to the conclusions; readers can always check the technical details if interested.

**Resources**

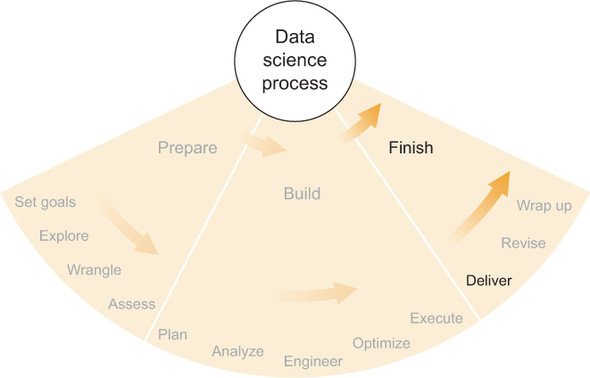
* Chapter 11, [Delivering a Product Tutorial](https://liveproject.manning.com/module/93_6_2/discovering-disease-outbreaks-from-news-headlines/5--executive-summary/5-2--delivering-a-product?), in Think Like a Data Scientist by Brian Godsey in 5.2 can help you with finding and presenting the critical parts of a data science project.
* To convert to a pdf, you’ll need to install [latex](https://www.latex-project.org/get/).

# **5.2 Delivering a Product**

excerpt from Think Like a Data Scientist: Tackle the data science process step-by-step | Brian Godsey[go to book](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-11)

[This figure](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-11/ch11fig01) shows where we are in the data science process: product delivery. Previous chapters of this book discuss setting project goals, asking good questions, and answering those questions through rigorous analysis of data. After all this is done, if you’re the lead data scientist you probably know more about every aspect of the project than anyone else, and you’re in a position to answer all sorts of questions about the project, ranging from the methods and tools used to the significance and impact of the results. But it’s not usually a good idea to stay in this position in perpetuity, making yourself the only possible source of information about the project and its results. Not only would you become the single point of failure (if you’re not available for some reason, what happens?), but you also would have created perpetual work for yourself, whenever questions come up. Because of these possibilities, it’s usually good to create something that summarizes or catalogs your results so that customers and other people can have their questions—at least the most common ones—answered without involving you.

##### The first step of the finishing phase of the data science process: product delivery



In order to create an effective product that you can deliver to the customer, first you must understand the customer perspective. Second, you need to choose the best media for the project and for the customer. And finally, you must choose what information and results to include in the product and what to leave out. Making good choices throughout product creation and delivery can greatly improve the project’s chances for success.

### Understanding your customer

In [this chapter](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-2/ch02), I discussed listening to customers and asking them questions that can help you understand their problems, as well as providing information relevant to the questions they have. Hopefully, some of the strategies I presented led to good outcomes in data gathering, exploration, design and implementation of statistical methods, and overall results. I’ll revisit that idea of understanding the customer again here, with a focus on creating a product that will most efficiently make those good results available to the customer.

#### Who is the entire audience for the results?

You probably know the customer very well by now, but there may be people other than customers who might also be interested in results. If the customer is a leader of a group or organization, other members of that group might also be part of the audience for your results. If the customer is an organization, certain departments or individuals within that organization may be part of the audience, but others may not. If the customer is an individual or a department, results may be passed up the hierarchy, to bosses or executives, so decisions can be made. In any case, it’s best not to assume that the customer you’ve dealt with regularly is the only audience for the results you’ve generated. Consider the network of people surrounding the customer and whether they’re part of the audience. If you’re not sure, ask the customer, “Who do you foresee wanting to see these results, and why?” Hopefully, you can assemble a good idea of who the audience is.

#### What will be done with the results?

Once you know the audience for your results, you’ll want to figure out what they’re going to do with them. This is often more difficult than you would think.

In [this chapter](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-2/ch02), I wrote briefly about how you might discuss deliverables with the customer, so you may already have a good idea about what types of things the customer wants to see in the results and what they might do with them. In bioinformatics, for example, a customer might intend to take the top-10 candidate genes from your results and run extensive experiments on them. If you’ve built a beer-recommendation algorithm, the customer may intend to have their friends use the algorithm and then drink the recommended beers. There are many possibilities.

In a project involving organizational behavior for which you used some techniques in social network analysis, the customer may be interested in exploring each individual’s contacts and seeing how those contacts are similar or different from the individual. This example is less of an action than an interest. If a customer begins a sentence beginning with

* “We would be interested in...”
* “We want to see...”
* “We would like to know...”

or similar, be sure to pursue the issue further and find out how exactly they intend to take action on this new knowledge. The actions they intend to take are far more important than what they’re interested in.

If, for example, the customer intends to make business decisions based on what they find out, then you should probably figure out what their tolerance is for error and incorporate that into the tailored results you present to them. Misunderstanding the intended action and its consequences can cause bigger problems later. The example at the end of this chapter gives one instance when miscommunication while delivering a product may have caused a problem.

There are an uncountable number of ways that a customer might act on the results that you deliver, so it’s best so spend some time trying to pin those down before you finalize them and their format. You might try having the customer run through a hypothetical scenario involving various types of results that are appropriate for the project, or you might even want to visit them in their workplace, observe their workflows, and witness personally how they make decisions. Talking to multiple people is also a good idea, particularly if your audience is composed of individuals with varying experience, knowledge, and interests in results. Overall, you’d like to understand as thoroughly as possible the perspective of the customer and audience and what they expect and intend with respect to the results that you’ll deliver. This understanding can help you create and deliver a product that helps the customer accomplish their goals.

### Delivery media

The thing that you create and deliver to customers—the product—can take many forms. In data science, one of the most important aspects of a product is whether the customer passively consumes information from it, or whether the customer actively engages the product and is able to use the product to answer any of a multitude of possible questions. The most common example of a *passive* product is a report or white paper; the customer can find in this only the answers that are in the text, tables, and figures present in the document. The most common example of an *active* product is an application that allows customers to interact with data and analysis in order to answer some questions on their own. Various types of products can fall anywhere along the spectrum between passive and active. Each of these types has strengths and weaknesses, which I discuss in the following sections.

#### Report or white paper

Probably the simplest option for delivering results to a customer, a report or white paper includes text, tables, figures, and other information that address some or all of the questions that your project was intended to answer. Reports and white papers might be printed on paper or delivered as PDFs or other electronic format. Because a report is a passive product, customers can read it when delivered and can consult the report as needed in the future, but the report will never be able to provide any new answers that weren’t included when it was written—this is an important distinction between reports and more active product types. On the other hand, reports and white papers are some of the simplest and most easily digestible product types.

##### Strengths

Some strengths of a report or white paper are these:

* On paper or in electronic form, reports and white papers are portable and don’t require any special technology or knowledge in order to use them, except some general domain knowledge of the report’s topic.
* Reports and white papers can provide the simplest and quickest way for the customer to find answers if the desired answers are present and if the report is concise and well organized. For most people, finding and reading an answer on a page is easier than, for example, opening an application or interpreting data in a spreadsheet.
* Reports also offer the ability to construct a narrative that can be useful for effective delivery of answers, information, caveats, and impact. Some product types provide data and answers out of context, but a narrative can establish contexts that help the readers of the report make better use of the results therein. For example, classifications generated by a machine learning algorithm can be far more useful to customers if they understand the accuracies and limits of applicability of that algorithm. A narrative can provide context prior to stating results in order to prevent misinterpretation and misuse of results.

##### Limitations

Some limitations of a report or white paper are these:

* The biggest limitation of reports and white papers is that they’re fully passive. You need to know before you write the report which questions the customer wants to have answered, and you need to answer these questions in a way that’s easily comprehensible. If you’re not successful in writing a good report, the customer will return to you with questions or, even worse, dismiss the project as a failure and lose confidence in you and/or your team, even if the results themselves are quite good.
* It can be tough to include the right amount of detail so that all the major points are covered and the most important questions are answered, while avoiding details that distract from the important points.
* Reports and papers can answer questions only at the current time and may not apply to future times or other data sets outside the current set of data. If it’s likely that the customer will want to revisit the project’s questions in the future or use another data set, a report might not be the best choice.
* Some people don’t like reading reports. People of various learning and leadership styles may prefer to see results in a different format, and if these people are stubborn and in a position of authority, writing a report would be a waste of time.

##### When to use it

A report or white paper can be a good product to deliver when

* Your project involves a few key questions that can be answered completely and succinctly in a written report that may include tables, graphics, or other figures.
* The main goals of your project involve answering a few questions one time, and these answers are useful by themselves, without an ongoing need to update or expand the answers.
* The customer would like a written report, and you don’t feel that that’s an inappropriate request.

#### Analytical tool

In some data science projects, the analyses and results from the data set can also be used on data outside the original scope of the project, which might include data generated after the original data (in the future), similar data from a different source, or other data that hasn’t been analyzed yet for one reason or another. In these cases, it can be helpful to the customer if you can create a tool for them that can perform these analyses and generate results on new data sets. If the customer can use this analytical tool effectively, it might allow them to generate any number of results and continue to answer their primary questions well into the future and on various (but similar) data sets.

A simple example of such an analytical tool is a spreadsheet that makes projections based on the current financial situation and expectations of a company and its industry. Theoretically, a customer could enter a range of values into such a spreadsheet and see how the projections change if the company’s financial situations change. Customers might not be able to create the spreadsheet themselves if it consists of complicated formulas and statistical methods, but they can understand the intent and the meaning of the results if they, for example, conform to generally accepted financial-modeling principles.

An analytical tool that you might deliver as a product of your data science project might also be a software script that accepts a data set and analyzes it, generating results that can be used by the customer in a specific, useful way. It might also be a highly specialized database query that addresses some of the project’s questions. An analytical tool delivered as a product can take many forms, but it needs to fulfill some criteria:

* The analytical tool needs to generate reliable results within the boundaries of the types of data sets for which it was intended.
* The set of applicable data sets must be well specified.
* The customer must be able to use the analytical tool correctly.

If all three of these criteria are met, then you might have a good product to deliver. The usefulness of the tool also depends on how many of the project’s questions it can answer and how important those questions are to the project’s goals and to the customer.

##### Strengths

Some strengths of analytical tools are these:

* Analytical tools allow the customer to answer some of their own questions quickly and without involving you. This saves time and effort for everyone involved.
* Within the intended scope of answerable questions, an analytical tool is more versatile than a report. Even within a narrow scope, analytical tools can usually give an unlimited number of results as inputs and data sets vary. It would be impossible to provide such unlimited results in a report.

##### Limitations

Some limitations of analytical tools are these:

* It’s often difficult to build an analytical tool that’s good at answering important questions reliably and concisely for a customer. If at some point the tool runs into an edge case and gives incorrect or misleading results, the customer may not realize it.
* Customers need to be able to understand the basics of how the tool works in order to know its limitations and interpret the results correctly.
* Customers need to be able to use the tool properly, or they’ll risk getting incorrect results. If you’re not available to assist them, you need to have reasonable guarantees that they won’t mess something up.
* If there are bugs or other problems with the tool, the customer may need support from you. Even if the analysis is good, things like data formatting, computer compatibility, and third parties whom the customer invited to share the tool can all cause unexpected problems that require your attention and slow the customer down.
* Because it’s so hard to create a foolproof analytical tool, such a tool can typically replicate only the absolute clearest of the project’s analyses. Accuracy, significance, and impact must all generally be high, and so the scope of an analytical tool must be reduced to only those analyses and results that meet these stringent criteria.

##### When to use it

An analytical tool can be a good product to deliver when

* The analysis completed within your project is conducive to being converted into such a tool, specifically that it can be made relatively easy to use and its results can be expected to be reliable.
* The customers can be expected to understand the tool to a point that they can use it correctly and interpret results correctly.
* A passive product such as a report isn’t sufficient for the customer’s needs, such as the case where the customer intends to replicate the project’s analysis for new data sets.

#### Interactive graphical application

If you want to deliver a product that’s a step more toward active than an analytical tool, you’ll likely need to build a full-fledged application of some sort. Although it can be argued that analytical tools like scripts and spreadsheets are also applications, I’ll draw a fuzzy distinction here between command-line-style, numbers-in-numbers-out analytical tools and graphical user interface (GUI) point-and-click-style applications. These aren’t well-defined categories, but I think the loose conceptual descriptions suffice here, because you can combine the two types in any number of ways and consider both sets of strengths and limitations listed here as appropriate. The former type (command-line style) I consider to fall into the analytical tool category of the previous section. In this section, I consider mainly GUI-based applications.

GUI-based applications, these days, are typically built on web frameworks, which I discussed earlier in this book. They don’t have to be web applications, but that type is most common right now. Such an interactive graphical application that you might deliver to your customer might include the following:

* Graphs, charts, and tables
* Drop-down menus that enable different analyses
* Interactive graphics, such as a timeline with movable endpoints
* The ability to import or select different data sets
* A search bar
* Results that can be filtered and/or sorted

None of these is required, but each of them enables the user (the customer) to answer more project-related questions at their leisure.

The most important thing to remember about interactive graphical applications, if you’re considering delivering one, is that you have to design, build, and deploy it. Often, none of these is a small task. If you want the application to have many capabilities and be flexible, designing it and building it become even more difficult. Software design, user experience, and software engineering are each full-time jobs at software companies, and so if you have little experience with delivering applications, it’s probably best to consult someone who does and to consider carefully the time, effort, and knowledge required before you start.

The strengths and limitations of interactive graphical applications include those of analytical tools discussed in the previous section, but I’ll add more specific ones here.

##### Strengths

Some strengths of interactive graphical applications include the following:

* If it’s well designed, an interactive graphical application can be the most powerful tool that you can deliver to a customer in terms of the information and answers it can convey.
* A well-designed and well-deployed interactive graphical application is easy to access and easy to use. It can be made clear within the application itself how to use the application properly and effectively.
* An interactive graphical application can be made portable and scalable if it’s built and deployed using common frameworks. This can be useful if you expect the number of users to grow or if you think another customer will want to use it as well.

##### Limitations

Some limitations of interactive graphical applications are these:

* Interactive graphical applications are hard to design, build, and deploy. Not only are the tasks difficult, but they also can take a lot of time.
* Interactive graphical applications often require ongoing support. The potential for bugs and problems increases with the complexity of the software and the deployment platform, and supporting the application and fixing bugs may take a considerable amount of time and resources.
* Customers might not use the application properly. If proper use isn’t clear, or if a user isn’t careful, misleading conclusions might be drawn.

##### When to use it

An interactive graphical application can be a good product to deliver when

* The guidelines from the previous section for when to use an analytical tool apply.
* A point-and-click GUI is strongly preferred over other types of analytical tools, either for ease of use or for the effectiveness of results delivery.
* You have the time and resources to design, build, deploy, and support such an application.

#### Instructions for how to redo the analysis

Whether or not you’ve already elected to create and deliver one of the products I’ve discussed, it can be a good idea to record the steps that you took to perform the project’s final analysis and to package it into an instruction book for the customer’s use or even for your own.

If you’re dealing with a smart and capable customer, possibly even a data scientist of some sort, they may be able to replicate your analysis if given instructions. This can be helpful for them if they want to analyze new data or other similar data in the future. As with building an analytical tool, the goal is to enable the customer to ask and answer some of their own questions without requiring much of your time. Giving them detailed instructions can accomplish this without requiring you to create and deliver a high-quality, relatively bug-free software application. If you’re giving them any code, there’s still the possibility of encountering bugs, but in this case there’s a reasonable expectation that the customer can read, edit, and fix the code themselves if they need to. You may still need to provide support sometimes, but this arrangement shifts a large part of the support burden away from you, assuming the customer is capable.

On the other hand, delivering an instruction set to a customer can create all sorts of problems if they aren’t familiar with some of the steps or if they don’t have much experience with the tools you’re using.

##### Strengths

Some strengths of delivering a set of instructions are these:

* It’s usually pretty easy to write down what you did, bundle it with your code or other tools, and deliver it to a customer.
* A set of instructions can be extremely useful to you in the future, if you ever return to this project and need to analyze data in a similar way again.

##### Limitations

Some limitations of delivering a set of instructions are these:

* The customer needs to understand everything you deliver, and they need to be able to replicate it, possibly with changes in data sets or other aspects, without encountering many problems that they can’t solve.
* Delivering instructions requires a lot of time and effort on the part of the customer, because they will have to read and understand some complex analyses and tools.
* Unclear instructions or messy code can make it nearly impossible to replicate the analysis reliably. You need to take care to avoid this possibility.

##### When to use it

A set of instructions for performing the analysis can be a good product to deliver when

* The exploratory work was the challenging part, and applying the statistical methods is relatively easy for the customer to do.
* The customer is smart and capable and shouldn’t have many problems working with the instructions, tools, and/or code that you deliver.
* You think there’s even a remote possibility that you’ll return to this project in the future—hold onto the instructions for yourself.

#### Other types of products

There are many other products that might also fit your project and your customer well. Here are some:

* ***A web-based API that, when queried in a certain way, returns answers and information pertaining to the query—***This can be useful when a customer wants to be able to integrate your analysis into an existing piece of software.
* ***A software component that’s built directly into the customer’s software—***This takes more coordination than a web-based API because you have to understand the architecture of the existing software, but it can still be a good idea depending on how the software will be used and deployed.
* ***An extended development project in which you work with the customer’s own software engineers to build a software component that they will then maintain themselves—***If you have the time now, letting others build and maintain software based on your analysis can save you a lot of time later.
* ***A database populated with the most useful data and/or the results of some analyses—***For customers who regularly work with databases, giving them one can sometimes be more convenient for them to work with than an API or other component, if you can figure out an efficient and relatively foolproof way to structure the database so that it’s useful and is used properly.

Each of these has its own set of strengths and limitations, many of which correspond to those I listed for other products earlier. If you’re considering a product that doesn’t fit in any of the categories I’ve described in detail, perhaps thinking through it in much the same way that I have will lead you to your own conclusions. In particular, it’s important to consider whether a particular product is more passive or active and what the time requirements will likely be for you, both in the near future and in the long term if you’ll need to provide support. Beyond that, every project and every specific potential product will have its own nuanced situations, and if you find yourself even a little unsure of what the ramifications are, it can be very helpful to consult someone with experience and ask their opinions. The internet can also be a good source of guidance if you can distinguish the good from the bad.

### Content

In addition to deciding the medium in which to deliver your results, you must also decide which results it will contain. Once you choose a product, you have to figure out the content you’ll use to fill it.

Some results and content may be obvious choices for inclusion, but the decision may not be so obvious for other bits of information. Typically, you want to include as much helpful information and as many results as possible, but you want to avoid any possibility that the customer might misinterpret or misuse any results you choose to include. This can be a delicate balance in many situations, and it depends greatly on the specific project as well as the knowledge and experience of the customer and the rest of the audience for the results. In this section, I provide some guidance on how to make decisions about inclusion and exclusion, and I also discuss how user experience can make products more and less effective.

#### Make important, conclusive results prominent

If there are critical questions that your project was intended to answer, and you now have conclusive answers to these questions, these answers should be prominent in your product. If you’re delivering a report, a summary of these important, conclusive results should appear in the first section or on the first page, and a more through discussion of methods and impact should be given later in the paper. If you’re delivering an interactive graphical application, these results should appear either on the main page of analytical results, or they should be very easily accessible through a few clicks, searches, or queries.

In general, it’s best to put the most important, conclusive, unmistakable, straightforward, and useful results front and center in whatever product you’re delivering so that the customer and the rest of the audience can find them without having to look for them and immediately understand the results and their impact.

#### Don’t include results that are virtually inconclusive

It can be tempting to tell the customer about all of the planned analyses that didn’t work alongside all of those that did. In a research setting, this might be a good idea, because failed experiments can sometimes give insight into how a system works and why other positive results turned out the way they did. But in non-academic data science, including in a report, the stories of the things that you tried that didn’t work can distract from the important, actionable information in the report.

Most people don’t need distractions from their work; they can find them on their own. If a piece of information doesn’t support valuable intelligence that can be used directly to make business decisions or otherwise help achieve any of the stated goals of a project, it’s usually best to leave it out. Feel free, however, to make note of any interesting tidbits for your own records or a supplementary report that isn’t the primary product for the customer; sometimes these can come in handy later if a project’s direction changes or if a new, related project is begun.

#### Include obvious disclaimers for less significant results

You probably have some results that fall between the labels *conclusive* and *inconclusive*. Deciding what to do with these can be tough, and there can be many reasons both for including them and for excluding them. It’s certain, though, that you wouldn’t want the customer to confuse results that are absolutely conclusive with those that are only partially so. Therefore, both as insurance for your own reputation and as a step toward making sure that the customer understands how best to use the information you’re providing, I highly recommended including disclaimers and caveats next to every result that’s less than 99.9% statistically significant or otherwise is not quite conclusive.

For example, let’s say you’re trying to detect fraudulent credit card transactions, and the customer is immediately going to reject the transactions that your software labels as fraudulent. If your software is 99.9% accurate, the customer probably won’t complain about the 0.1% of cases that were falsely rejected. But if your software has a false positive rate of 10%, the customer might complain very strongly. In similar situations, if you do have a 10% false positive rate, it’s definitely better to communicate that rate to the customer along with its potential implications before you deliver them the results and before they act on them. If there were any misunderstandings at all, and they acted on your results before they fully understood their implications and limitations, that could be bad for the customer, bad for the project, and possibly bad for you.

If you want to deliver some results that aren’t conclusive but that still might be helpful and informative, make sure you include a disclaimer stating exactly how significant the results are, what the limitations are, and what the positive and negative impacts are if the customer uses those results in certain ways. Overall, though, the customer needs to understand fully that you’re not 100% sure about these results and that acting on them might have unexpected consequences. If you can communicate this effectively, the customer will be in a good position to make good decisions based on your results, and you’ll be in a position to have a successful project.

#### User experience

Most people who use the phrase “user experience” refer to the ways that a person might interact with a piece of software. Within the software industry, user experience (UX) designer has become a lucrative career and rightly so. Understanding how people might interact with a piece of software is not an easy task, and it has been demonstrated in many contexts that how people interact with software has a large influence on whether that software is ultimately effective. User experience makes the difference between good analytical software that’s used effectively and good analytical software that most people can’t figure out how to use.

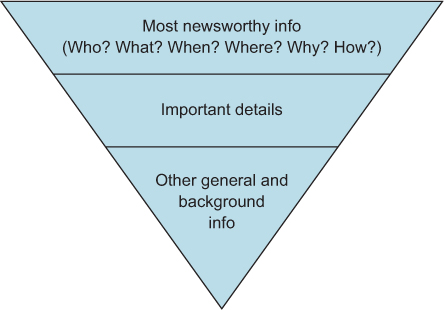
User experience can also refer to a report, an analytical tool, or any other product that you might deliver to a customer. The experience that the customer or audience has with your product is the user experience, and the principal goal is to ensure that these users use the product properly in order to draw correct conclusions from it and make good business decisions. If a customer isn’t using the product properly, you might reconsider the user experience. The goal is to enable and encourage the customer and the audience to do the right thing.

##### Inverted pyramid of journalism

The popular concept of an inverted pyramid, as illustrated in [this figure](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-11/ch11fig02), shows how a journalistic news story might be represented in order to be most effective to the reader. The implicit assumption is that a reader might not read the whole article, or at least that the reader might not read the whole article with their full attention. Based on this assumption, the most important information, the *lede* (or lead), should be at the very beginning of the article, followed by the body that supports the lede, and finally the tail that adds to the rest but isn’t absolutely necessary for the story to be complete.

##### The inverted pyramid of journalism[[1](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-11/ch11fn01)]

1From <https://en.wikipedia.org/wiki/Inverted_pyramid>, public domain image



For reports on data science projects, this means that it might be most effective to follow the same pattern to deliver results to a customer: lead with the most important, most impactful results in clear language, then include details that directly support those results, and finally include other auxiliary results that are useful but not necessary.

For analytical tools and interactive graphical applications (and other products), it can be helpful to consider the concept of the inverted pyramid when designing and building. The most important results and information should be in the user’s face as soon as they start using the application. Supporting details might take a little more effort, but not too much, and then users might need to look around a bit before they find the less important, extra information.

Although it certainly shouldn’t be a hard rule of data science projects, following the inverted pyramid from journalism can be helpful in writing a report or designing an application that delivers the most important information to the customer first and the less important but still helpful information second.

##### Plain language with no jargon

Jargon is confusing to people who don’t work in the field from which it comes. You shouldn’t use jargon in your reports or your applications or any of your products. If you do use it, you should make the definitions clear to the customers, the audience, or the users of the product you’re delivering.

The term *jargon* is hard to define. For our purposes, *jargon* is a set of terms or phrasing that’s familiar to people of a specific training, experience, or knowledge but that isn’t familiar to people working in substantially different fields. Because you can rarely guarantee that the people you’re speaking with are people with similar backgrounds to you, it’s generally best to assume that they don’t know your jargon.

I admit to being wholly anti-jargon, but I also see the value in using jargon in highly specialized conversations and writing. Jargon allows people to communicate efficiently within their fields—or at least within the subfields for which that jargon is valid. In those situations, I fully support the use of jargon, but in any situation in which people might not understand the specialized terms, it’s best to avoid them.

When it comes to presenting your work, if you must speak with or in front of people, it’s helpful to speak more plainly than you think you need to and more slowly as well. It’s rarely advantageous to use terms that a significant portion of your audience doesn’t understand.

When you’re writing a report or text for an application, the same rules apply: the text should be comprehensible by most of the audience, even if they don’t have experience in some related areas. Most important, with respect to language and understanding, using jargon isn’t proof that someone knows what they’re talking about. It’s often the contrary, in my experience. The ability to explain complex concepts in plain language is a rare talent and in my opinion a far more valuable skill than explaining anything using jargon.

##### Visualizations

Like the field of user experience design, data visualization improves your application or other product greatly, but it isn’t usually a main focus of that product. Data visualization is also very well studied, and it’s usually best to heed the warnings and follow the best practices of those who have studied and thought a lot about data visualization.

Edward Tufte’s *The Visual Display of Quantitative Information* is a must-read book for people who want to get their data visualization absolutely right. There are other great references on the topic as well, but Tufte is usually the best place to start. Not only will you learn about when it’s best to use bar charts or line graphs, but you’ll also discover some key principles that apply to any visualization—maps, timelines, scatter plots, and so on—such as “encourage the eye to compare different pieces of data” and “be closely integrated with the statistical and verbal descriptions of a data set.” Tufte’s books are packed with such tenets and plenty of examples that show exactly what he means.

Visualizations of data and results can be helpful in reports and applications, but if they’re not designed well, they can be detrimental to the product’s intent. It’s often worth taking some time to study and consider the assumptions and implications of any visualizations that you’re trying to create. Consulting some data visualization references, like Tufte’s books, or someone with experience can have large benefits later, as the visualizations continue to serve their purpose as clear, concise conveyors of useful information.

##### The science behind user experience

The study of user experience is a science, though some people don’t treat it as such. I didn’t realize that it was or could be more science than art until a few years ago, when I witnessed experience studies and evaluations in action. There are many well-studied principles regarding what makes an application easy to use, or powerful, or effective, and if you’re building a complex application, employing these principles can make a huge difference in the success of your project. I encourage you to consult an experienced UX designer if you’re building an application. Sometimes even a short conversation with a UX designer can lead to great improvements in your application’s usability.

### Example: analyzing video game play

While working with Panopticon Laboratories, an analytic software company whose goal is to characterize and detect suspicious in-game behavior in multiplayer online video game environments, we delivered a preliminary report to a customer (a video game publisher) that included a survey of the state of their in-game community as well as a list of some of the more suspicious players. To do this, as we did with all customers, we fit a proprietary statistical model to their data; this model assigned scores to each player, indicating how suspicious or fraudulent the player appeared to be in various categories. We were highly confident that the players with the highest scores were indeed fraudulent players, but the farther we progressed down the list of the most suspicious players, the less sure we were. We were contractually obligated to deliver to the customer a list of suspicious players, and we knew that we had to convey along with this list some notion of this uncertainty.

The customer didn’t employ any data scientists in the security department with which we were dealing, so we had to be careful not to mince words about statistical significance or uncertainty. The main thing we had to decide was how many suspicious players to include on the list. If we provided a relatively short list, we would likely be leaving out some very suspicious players who would continue to cost the video game publisher money. If we included too many players in the report, we would be pointing the finger at innocent players (with a not-insignificant false positive rate) and possibly misleading the video game publisher into thinking the problem was bigger than it was and possibly also causing them to take action against those innocent players, such as banning them from game play.

To resolve the decision, we worked with the customer to determine that they did indeed intend to ban players who were highly suspicious and that they were willing to accept a false positive rate of less than 5%. In order to establish a false positive rate, the customer planned to sample randomly from our initial list of suspicious players and check each one manually. We could then use the feedback to reinforce the statistical models and subsequently generate a new, more accurate report. Two or three rounds of this is usually enough to obtain high accuracy.

We had a minor setback when, after using the first-round feedback to generate another report but before we were quite ready to say that the behavior models were 100% done, the customer indicated that they were ready to begin banning the suspicious players from the game based on our lists. Luckily, before they acted, we had the chance to talk with them about how the most recent report still wasn’t necessarily actionable and that their feedback on this report would be crucial to gaining the requisite actionable intelligence from the next phase of reporting (meeting the <5% false positive rate requirement) and the subsequent software deployment. This was a classic case of not knowing what the customer was going to do with the product we were delivering. I’m not sure they would have admitted their intent beforehand (maybe they didn’t even realize it themselves), and so any line of questioning may have been fruitless anyway, but it was worth trying and worth being vigilant. The only thing worse than delivering something that isn’t entirely effective is delivering something that is effective but that is then misused, to much detriment.

After the next round of feedback, we delivered a report for which we expected a false positive rate of well under 5% (to give ourselves a cushion), and we made sure that the customer understood that the list still wasn’t perfect and that they could expect that a small percentage of the players on that list weren’t bad guys. If they took action against all of those players, they should expect some adverse effects.

After delivery of the final report, we began hooking up their data source to our real-time analytic engine that powers an interactive graphical web application that provides the same information as the reports, plus the ability to interact with and learn far more about players and their behavior. The application allows the customer to see the same curated lists of suspicious players and allows them to click players’ names and get more information about them—in particular, more information about why they’re considered suspicious. In many ways, the application is superior to the reports because of the increased amount of information available, the multiple ways and formats in which the results can be viewed, and the interactive nature of the application that lets users find the answers they need when they want them. Also, the application has informative graphics and a well-designed user experience, which makes interacting with data and results both easier and more intuitive. Supporting a customer deployment of a live application is a considerable amount of work, but the app seems to be far more useful to customers than the reports have been.

### Exercises

Continuing with the Filthy Money Forecasting personal finance app scenario first described in [this chapter](https://livebook.manning.com/book/think-like-a-data-scientist/chapter-2/ch02), and relating to previous chapters’ exercises, try these exercises:

**1.**

Suppose your boss or another management figure has asked for a report summarizing the results of your work with forecasting. What would you include in the report?

**2.**

Suppose the lead product designer for the FMF web app asks you to write a paragraph for the app users, explaining the forecasts generated by your application, specifically regarding reliability and accuracy. What would you write?

### Summary

* The product is, in a sense, the thing that you’ve been working toward for the entire duration of the project; it’s important to get the format and content right.
* The format and medium of the product should, as much as possible, meet the customer’s needs both now and in the foreseeable future.
* The content of products should focus on important, conclusive results and not distract the customer with inconclusive results or other trivia.
* It’s best to spend some time thinking formally about user experience (UX) design in order to make the product as effective as possible.
* Consider in advance whether the product will need ongoing support and plan accordingly.

# **5.3 Submit Your Work**

The deliverable from this section is a Jupyter Notebook with the overall summary of your project. The notebook should be converted to a presentation format, either HTML or pdf. There should be minimal wording in this summary, perhaps only a sentence or two along with the map of the major disease outbreak.

Upload a link to your Jupyter Notebook (preferably hosted on GitHub) in the blank below and hit submit. After submitting, you can view an example solution in the next section.

# **5.4 Solution**

The completed map showing the major disease outbreak is below. This should be within a pdf document concluding your findings to a nontechnical audience.

* [5. Executive Summary.ipynb](https://liveproject-resources.s3.amazonaws.com/93/22375/2020-03-03-09-36-24/5.%20Executive%20Summary.ipynb)

# Summary

# **Project Conclusions**

* Completing a successful data science project requires employing a variety of tools, from basic data cleaning to data manipulation to machine learning algorithms to visualization techniques.
* Data cleaning is often the most time-consuming part of a data science project. It’s also the most critical because all downstream steps depend on the quality of the data.
* Data science is an iterative process and we have to repeat steps multiple times. It’s not unusual to find mistakes in a later part of the project that we have to go backwards to fix.
* Make sure to check both the inputs to an algorithm and the outputs. Machine learning algorithms are only as good as the data that we put into them. They can also produce unreasonable results. Sometimes the quickest way to check the inputs and the outputs is to visualize them or summarize them with basic statistics. Also check for outliers at each step.
* Breaking up a data set into parts can be an effective method when the distribution has a clear differentiator (United States versus worldwide). Also, different datasets may require different parameters for a machine learning algorithm.
* Domain knowledge often plays a crucial role in helping data scientists. There are rules of thumb we can employ but, if possible, we should check these assumptions with domain experts.
* Effectively presenting a final outcome from a data science project is crucial. In an executive summary, explain what you discovered, why it’s important to a nontechnical audience, and actions that should be taken as a result.
* The objective of data science is to enable better decisions through analysis of large data sets, something we hopefully accomplished through this real-world project!

# Bonus Follow-up Project: The Coronavirus Pandemic

# **6.1 Tracking the Coronavirus Pandemic**

The coronavirus pandemic is causing massive devastation around the globe. As data scientists, we can help do our part by monitoring the spread of the disease. You can apply the skills gained in this project in order to make a difference. Consider executing the following series of steps.

**Workflow**

1. Obtain a list of popular news sites. Consider using Feedsite, which maintains a list of the top 100 US news sites, and the top 100 world news sites.
   * The individual news site URLs can be extracted from the Feedspot HTML. You simply need to leverage the Beautiful Soup library in order to parse out the necessary data.
2. Crawl each news site. Identify articles that mention coronavirus or Covid-19 within the headline. Save the article headlines and article body texts on your local machine. Avoid including articles with duplicate titles.
3. Extract the locations from each headline, as well as from the body of the text.
4. Cluster the locations, and visualize the clusters on the map.
5. Consider incorporating the article publication dates to plot each cluster’s growth rate over time. These growth rates could potentially be used to measure the virus spread in certain locations.
6. Consider running a separate analysis on small clusters, as well as singleton locations that do not cluster with any other datapoint. \*These isolated locations might represent future hot-spots in which an outbreak is just beginning.
7. Write up your analysis in a blog post, and share that analysis with others.

**Notes**

* Please note that the locations in the body might be noisier than the locations within the headline. One way to limit the level of noise is to ensure that the headline locations and the body locations align. For instance, a headline might reference the coronavirus outbreak in the country of Italy. Meanwhile, the body of the article might reference two cities: Milan and Washington, DC. Using GeonamesCache, we can identify that Milan is a city in Italy, and Washington, DC, is not. Hence, Milan is more likely to represent a valid location for the coronavirus outbreak.

**Resources**

* Chapter 16, [Parsing HTML using Beautiful Soup](https://liveproject.manning.com/editmodule/93_8_2/discovering-disease-outbreaks-from-news-headlines/tracking-the-coronavirus-pandemic/moduleName?), from Data Science Bookcamp by Leonard Apeltsin in 6.2 can give you a more thorough introduction to Beautiful Soup.
* You may want to use one of these Feedsites:
  + [USA News Feedspot](https://blog.feedspot.com/usa_news_websites/)
  + [World News Feedspot](https://blog.feedspot.com/world_news_blogs/)
* For an introduction on how to crawl and scrape the web, please see:
  + [Practical Intro to Web Scraping](https://realpython.com/python-web-scraping-practical-introduction/)
  + [Web Scraping and Crawling with Scrapy and MongoDB](https://realpython.com/web-scraping-and-crawling-with-scrapy-and-mongodb/)

# **6.2 Parsing HTML Using Beautiful Soup**

excerpt from Data Science Bookcamp: Ten case studies MEAP V03 livebook | Leonard Apeltsin[go to book](https://livebook.manning.com/book/data-science-bookcamp/chapter-16)

We’ll start by installing the Beautiful Soup library. Afterwards, we’ll import a BeautifulSoup class from bs4. Following a common convention, we’ll import BeautifulSoup as simply bs.

##### NOTE

Call "pip install bs4" from the command-line terminal in order to install the Beautiful Soup library.

##### This listing Importing the BeautifulSoup class

1

from **bs4** import **BeautifulSoup** as **bs**

copy

We’ll now initialize the BeautifulSoup class by running bs(html\_contents). In keeping with convention, we’ll assign the initialized object to a soup variable.

##### NOTE

By default, the bs class uses Python’s built-in HTML parser to extract the HTML contents. However, more efficient parsers are available through external libraries. One popular library is called lxml, which can be installed by running pip install lxml. After installation, the lxml parser can be leveraged during bs initialization. We simply need to execute bs(html\_contents, 'lxml')

##### This listing Initializing BeautifulSoup using an HTML string

1

soup = bs(html\_contents)

copy

Our soup object tracks all elements in the parsed HTML. We can output these elements in a clean, readable format by running the soup.prettify() method.

##### This listing Printing readable HTML with Beautiful Soup

1

print(**soup**.prettify())

copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

<html>

<title>

Data Science is Fun

</title>

<body>

<h1>

Data Science is Fun

</h1>

<div class="text" id="paragraphs">

<p id="paragraph 0">

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

</p>

<p id="paragraph 1">

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

</p>

<p id="paragraph 2">

Here is a link to

<a href="https://www.manning.com/books/data-science-bookcamp">

Data Science Bookcamp

</a>

</p>

</div>

<div class="text" id="list">

<h2>

Common Data Science Libraries

</h2>

<ul>

<li>

NumPy

</li>

<li>

Scipy

</li>

<li>

Pandas

</li>

<li>

Scikit-Learn

</li>

</ul>

</div>

<div class="empty" id="empty">

</div>

</body>

</html>

copy

Suppose we want to access an individual element, such as the title. The soup object provides that access through its find method. Running soup.find(title) will return all content that’s enclosed within the title’s start and end tags.

##### This listing Extracting the title with Beautiful Soup

1

2

title = soup.find('title')

print(title)

copy

1

<title>Data Science is Fun</title>

copy

The outputted title appears to be an HTML string that’s demarcated by the title tags. However, our title variable is not a string. Rather, it’s an initialized Beautiful Soup Tag class. We can verify by printing type(title)

##### This listing Outputting the title’s data type

1

**print**(type(title))

copy

1

<**class** 'bs4.element.Tag'>

copy

Each Tag object contains a text attribute, which maps to the text within the tag. Thus, printing title.text will return Data Science is Fun.

##### This listing Outputting the title’s text attribute

1

**print**(title.text)

copy

1

Data Science **is** Fun

copy

We’ve accessed our title tag by running soup.find('title'). Additionally, we can access that same tag simply by running soup.title. Therefore, running soup.title.text will return a string that’s equal to title.text.

##### This listing Accessing the title’s text attribute from soup

1

assert soup.title.**text** == title.**text**

copy

In this same manner, we can access the body of our document by running soup.body. Below, we’ll output all the text within the body of our HTML.

##### This listing Accessing the body’s text attribute from soup

1

2

**body** = soup.body

**print**(body.text)

copy

1

Data Science **is** FunParagraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

1

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

1

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

1

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

1

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

1

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

copy

1

Here is a **link** **to** Data Science BookcampCommon Data Science LibrariesNumPyScipyPandasScikit-Learn

copy

Our output is an aggregation of all the text within the body. This text blob includes all headers, bullet-points, and paragraphs. It is virtually unreadable. Rather than outputting all the text, we should instead narrow the scope of our output. Lets print the text of just the first paragraph. We can output that paragraph by printing body.p.text. Alternatively, printing soup.p.text will generate the same exact output.

##### This listing Accessing the text of the first paragraph

1

2

assert **body**.p.text == soup.p.text

**print**(soup.p.text)

copy

1

2

3

4

5

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

copy

Accessing body.p returns the first paragraph in body. How do we access the remaining two paragraphs? Well, we can utilize the find\_all method. Running body.find\_all('p') will return a list of all the paragraph tags within the body.

##### This listing Accessing all paragraphs in the body

1

2

3

4

paragraphs = body.find\_all('p')

**for** i, paragraph **in** enumerate(paragraphs):

print(f"\nPARAGRAPH {i}:")

print(paragraph.text)

copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

PARAGRAPH 0:

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

PARAGRAPH 1:

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

PARAGRAPH 2:

Here **is** a link to Data Science Bookcamp

copy

Similarly, we access our list of bullet-points by running body.find\_all('li'). Lets utilize find\_all to print all the bulleted libraries in the body.

##### This listing Accessing all bullet-points in the body

1

2

print([bullet.text for bullet

in body.find\_all('li')])

copy

1

['NumPy', 'Scipy', 'Pandas', 'Scikit-Learn']

copy

The find and find\_all methods allow us to search the elements by tag-type, and also by attribute. Suppose we wish to access an element with a unique id of x. In order to search on that attribute id, we simply need to execute find(id=x). With this in mind, lets output the text of the final paragraph, whose assigned id is paragraph 2.

##### This listing Accessing a paragraph by id

1

2

paragraph\_2 = soup.find(**id**='paragraph 2')

print(paragraph\_2.text)

copy

1

Here is a **link** **to** Data Science Bookcamp

copy

The contents of paragraph\_2 include a web link to Data Science Bookcamp. The actual url is stored within the href attribute. Beautiful Soup permits us to access any attribute using the get method. Thus, running paragraph\_2.get(id) will return paragraph 2. Subsequently, running paragraph\_2.a.get(href) will return the url. Below, we’ll print that url.

##### This listing Accessing an attribute within a tag

1

2

assert paragraph\_2.get('id') == 'paragraph 2'

print(paragraph\_2.a.get('href'))

copy

1

https://www.manning.com/books/data-science-bookcamp

copy

All attribute ids have unique values assigned to them within our HTML. However, not all our attributes are unique. For instance, two of our three division elements share the class attribute of text. Meanwhile, the third division element contains a unique class that’s set to empty. Running body.find\_all('div') will return all three division elements. How do we obtain just those two divisions where the class is set to text? Well, we just simply need to run body.find\_all('div', class\_='text'). The added class\_ parameter will limit our results to those divisions where the class is set appropriately. Below, we’ll search for these divisions. Afterwards, we’ll output their text contents.

##### NOTE

Why do we run find\_all on class\_ rather than class? Well, in Python the class keyword is a restricted identifier, which is used to define novel classes. Thus, Beautiful Soup allows for a special class\_ parameter to get around this keyword restriction.

##### This listing Accessing divisions by their shared class attribute

1

2

3

4

**for** division **in** soup.find\_all('div', **class\_**='text'):

id\_ = division.get('id')

print(f"\nDivision with id '{id\_}':")

print(division.text)

copy

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Division with id 'paragraphs':

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1

Here is a link to Data Science Bookcamp

Division with id 'list':

Common Data Science LibrariesNumPyScipyPandasScikit-Learn

copy

So far, we’ve leveraged Beautiful Soup to access elements within the HTML. However, the library also allows us to edit individual elements. For example, given a tag object, we can delete that object by running tag.decompose(). The decompose method will remove that element from all our data-structures, including soup. Thus, calling body.find(id='paragraph 0').decompose() will remove all traces of the first paragraph. Also, calling soup.find(id='paragraph 1').decompose() will delete the second paragraph from both the soup and body objects. After these deletions, only the third paragraph will remain. Lets confirm below.

##### This listing Paragraph deletion with Beautiful Soup

1

2

3

body.find(**id**='paragraph 0').decompose()

soup.find(**id**='paragraph 1').decompose()

print(body.find(**id**='paragraphs').text)

**1**

copy

1

Here is a **link** **to** Data Science Bookcamp

copy

Additionally, we’re able to insert new tags into the HTML. Suppose we wish to insert a new paragraph into our final empty division. To do so, we must first create a new paragraph element. Running soup.new\_tag('p') will return an empty paragraph Tag object.

##### This listing Initializing an empty paragraph Tag

1

2

**new**\_paragraph = soup.**new**\_tag('p')

print(**new**\_paragraph)

copy

1

**<p></p>**

copy

Next, we must update the initialized paragraph’s text, by assigning it to new\_paragraph.string. Running new\_paragraph.string = x will set the paragraph’s text to equal x.

##### This listing Updating the text of an empty paragraph

1

2

**new**\_paragraph.string = "This paragraph is new"

print(**new**\_paragraph)

copy

1

<p>This paragraph **is** **new**</**p**>

copy

Finally, we must append the updated new\_paragraph to an existing Tag object. Given two Tag objects; tag1 and tag, we can insert tag1 into tag2 by running tag2.append(tag1). Thus, running soup.find(id='empty').append(new\_paragraph) should append to paragraph to the empty division. Lets update our HTML and then confirm the changes by rendering the updated results.

##### This listing Paragraph insertion with Beautiful Soup

1

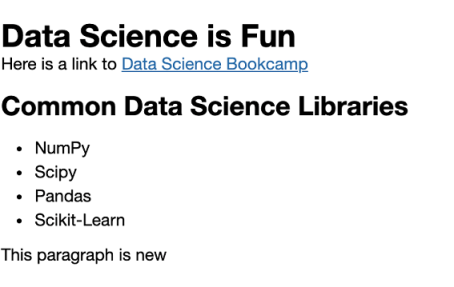
2

soup.find(**id**='empty').append(new\_paragraph)

render(soup.prettify())

copy

##### A rendered HTML document. The document has edited. Two of the three original paragraphs have been removed. Also, a new paragraph has been inserted into the HTML.



##### COMMON BEAUTIFUL SOUP METHODS

soup = bs(html\_contents): Initializes a BeautifulSoup object by from the HTML elements within the parsed html\_contents).

soup.prettify(): Returns the parsed HTML document in a clean, easily-readable format.

title = soup.title: Returns a Tag object associated with the title element of a parsed document.

title = soup.find('title'): Returns a Tag object associated with the title element of a parsed document.

tag\_object = soup.find('element\_tag'): Returns a Tag object associated with the first HTML element that is demarcated by the specified element\_tag tag.

tag\_objects = soup.find\_all('element\_tag'): Returns a list of all Tag objects that are demarcated by the specified element\_tag tag.

tag\_object = soup.find(id='unique\_id'): Returns a Tag object that contains the specified unique id attribute.

tag\_objects = soup.find\_all('element\_tag', class\_='category\_class'): Returns a list of Tag objects that are demarcated by the specified element\_tag tag, and also contain the specified class attribute.

tag\_object = soup.new\_tag('element\_tag'): Creates a new Tag object, whose HTML element type is specified by element tag.

tag\_object.decompose(): Deletes the Tag object from soup.

tab\_object.apped(tag\_object2): Given two Tag objects, tag\_object and `tag\_object2, this method inserts tag\_object2 into tag\_object.

tag\_object.text: Returns all visible text within a Tag object.

tag\_object.get('attribute'): Returns an HTML attribute that has been assigned to the Tag object.