# Data Science Bookcamp

Ten Python Projects

Leonard Apeltsin

MEAP

MANNING

**MEAP Edition**
**Manning Early Access Program**
# Data Science Bookcamp
**Ten case studies**
**Version 2**



For more information on this and other Manning titles go to
manning.com

# *welcome*

Thank you for purchasing the MEAP of *Data Science Bookcamp: Ten case studies*.

This book is intended for Python coders who are interested in a data science career. The only prerequisite is a basic knowledge of Python. Previous experience with complex algorithms is not required. A mathematical background is not necessary. An amateur who finishes this book will gain the necessary skills to get their first high-paying data science job. These skills include:

- The fundamentals of probability and statistics.
- Supervised and unsupervised machine learning techniques.
- Key data science libraries such as NumPy, SciPy, Pandas, Matplotlib and Scikit-Learn.
- Open-ended problem-solving abilities.

Open-ended problem-solving abilities are essential for a data science career. Unfortunately, these abilities cannot be acquired simply by reading. In order to become a problem solver, you must persistently solve difficult problems. With this mind, I've structured my book around case studies: open-ended problems that are modeled on real-world situations. The case studies range from online advertisement analysis to tracking disease outbreaks using news data.

Each case study begins with a detailed problem statement. Afterwards, the reader is taught the skills required to solve the problem. These skill-sections cover fundamental libraries, as well as mathematical and algorithmic techniques. All mathematical concepts are illustrated using common-sense coding examples. I promise my readers, there will be no Greek symbols in my book! This is a Python book, and therefore all the math will be explained using Python code.

Upon completing the skills sections, the readers are strongly encouraged to solve the case study on their own. Afterwards, they can compare their personal solution to the solution included in the book.

Reader, I'm excited to join you on your data science journey. My end-goal is grow your data science skillset, so that you get a data science job. Your feedback can help achieve this goal. Please share your thoughts and comments in the liveBook discussion forum. Your posts will aid me in improving my book.

Thanks again for your interest, and for purchasing the MEAP!

—Dr Leonard Apeltsin

# brief contents

CS1

# Case Study 1: Finding the Winning Strategy in a Card Game

## CS1.1 Problem Statement

Would you like to win a bit of money? Lets wager on a card game for minor stakes. In front of you is a shuffled deck of cards. All 52 cards lie face-down. Half the cards are red and half are black. I will proceed to flip the cards over, one by one. If the last card I flip over is red, you'll win a dollar. Otherwise, you will lose a dollar.

Here's the twist; you can ask me to halt the game at any time. Once you say "halt", I will flip over the next card and end the game. That next card will serve as the final card. You will win a dollar if it's red.

We can play the game as many times as you like. The deck will be reshuffled every time. After each round, we'll exchange money. What is your best approach to winning this game?

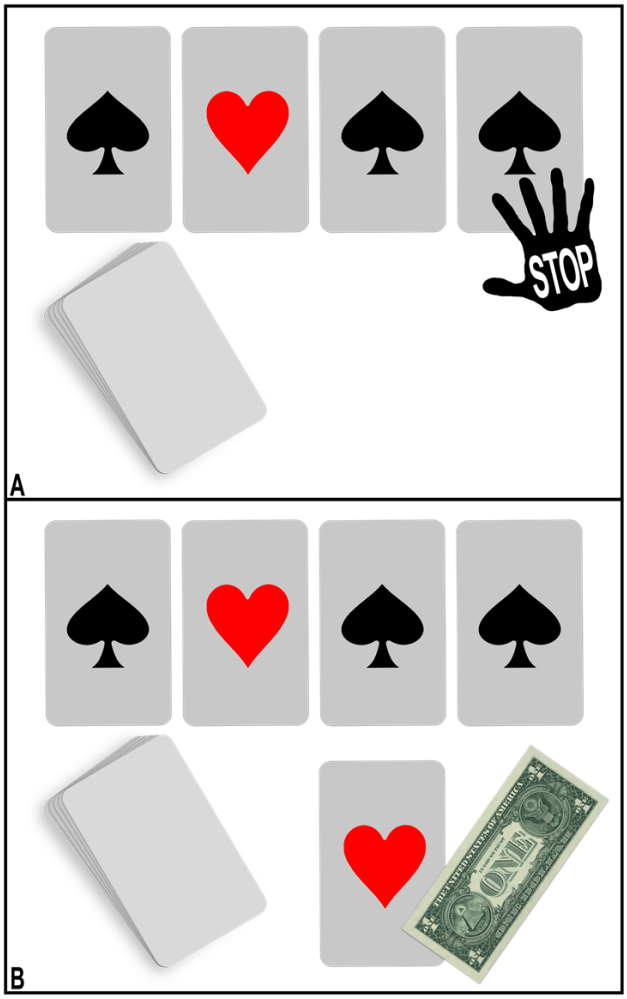


**Figure CS1.1 The card flipping game. We start with a shuffled deck. I repeatedly flip over the top card from the deck. A) I have just flipped the fourth card. You instruct me to stop. B) I flip over the fifth and final card. The final card is red. You win a dollar.**

## *CS1.2 Overview*

In order to address the problem at hand we will need to know how to:

**A.** Compute the probabilities of observable events using sample space analysis.

**B.** Plot the probabilities of events across a range of interval values.

**C.** Simulate random processes, such as coin-flips and card shuffling, using Python.

**D.** Evaluate our confidence in decisions drawn from simulations using confidence interval analysis.

# 1 Computing Probabilities Using Python

***This section covers:***

- What are the basics of probability theory?
- Computing probabilities of a single observation.
- Computing probabilities across a range of observations.

Few things in life are certain; most things are driven by chance. Whenever we cheer for our favorite sports-team, or purchase a lottery ticket, or make an investment in the stock-market, we hope for some particular outcome, but that outcome cannot ever be guaranteed. Randomness permeates our day to day experiences. Fortunately, that randomness can still be mitigated and controlled. We know that some unpredictable events occur more rarely than others, and that certain decisions carry less uncertainty than other much more risky choices. Driving to work in a car is safer than riding a motor-cycle. Investing a part of one's savings in a retirement account" is a safer than betting it all on a single hand of Blackjack. We can intrinsically sense these trade-offs in certainty because even the most unpredictable systems still show some predictable behaviors. These behaviors have been rigorously studied using **probability theory**. Probability theory is an inherently complex branch of math. However, aspects of the theory can be understood without knowing the mathematical underpinnings. In fact, difficult probability problems can be solved in Python without needing to know a single math equation. Such an equation-free approach to probability requires a baseline understanding of what mathematicians call a **sample space**.

## 1.1 Sample Space Analysis: An Equation-Free Approach for Measuring Uncertainty in Outcomes

Certain actions have measurable outcomes. A sample space is the set of all the possible outcomes that an action could produce. Lets take the simple action of flipping a coin. The coin will land on either heads or tails. Thus, the coin-flip will produce one of 2 measurable outcomes: 'Heads' or 'Tails'. By storing these outcomes in a Python set, we can create a sample space of coin-flips.

**Listing 1.1 Creating a Sample Space of Coin-Flips**

```
sample_space = {'Heads', 'Tails'}  ❶
```

❶ Storing elements in curly brackets will create a Python set. A Python set is a collection of unique, unordered elements.

Suppose we choose an element of `sample_space` at random. What fraction of the time will the chosen element equal `'Heads'`? Well, our sample space holds 2 possible elements. Each element occupies an equal fraction of space within the set. Therefore, we expect `'Heads'` to be selected with a frequency of 1/2. That frequency is formally defined as the **probability** of an outcome. All outcomes within `sample_space` share an identical probability, which is equal to `1 / len(sample_space)`.

**Listing 1.2 Computing the probability of heads**

```
probability_heads = 1 / len(sample_space)
print(f'Probability of choosing heads is {probability_heads}')
```

```
Probability of choosing heads is 0.5
```

The probability of choosing `'Heads'` equals 0.5. This relates directly to the action of flipping a coin. We'll assume the coin is unbiased, which means the coin is equally likely to fall on either heads or tails. Thus, a coin-flip is conceptually equivalent to choosing a random element from `sample_space`. The probability of the coin landing on heads is therefore 0.5. The probability of it landing on tails is also equal to 0.5.

We've assigned probabilities to our 2 measurable outcomes. However, there are additional questions we could ask. What is the probability that the coin lands on either heads or on tails? Or, more exotically, what is the probability that the coin will spin forever in the air, landing on neither heads nor tails? To find rigorous answers, we'll need to define the concept of an **event**. An event is the subset of those elements within `sample_space` that satisfy some **event condition** . An event condition is a simple Boolean function whose input is a single `sample_space` element. The function returns `True` only if the element satisfies our condition constraints.

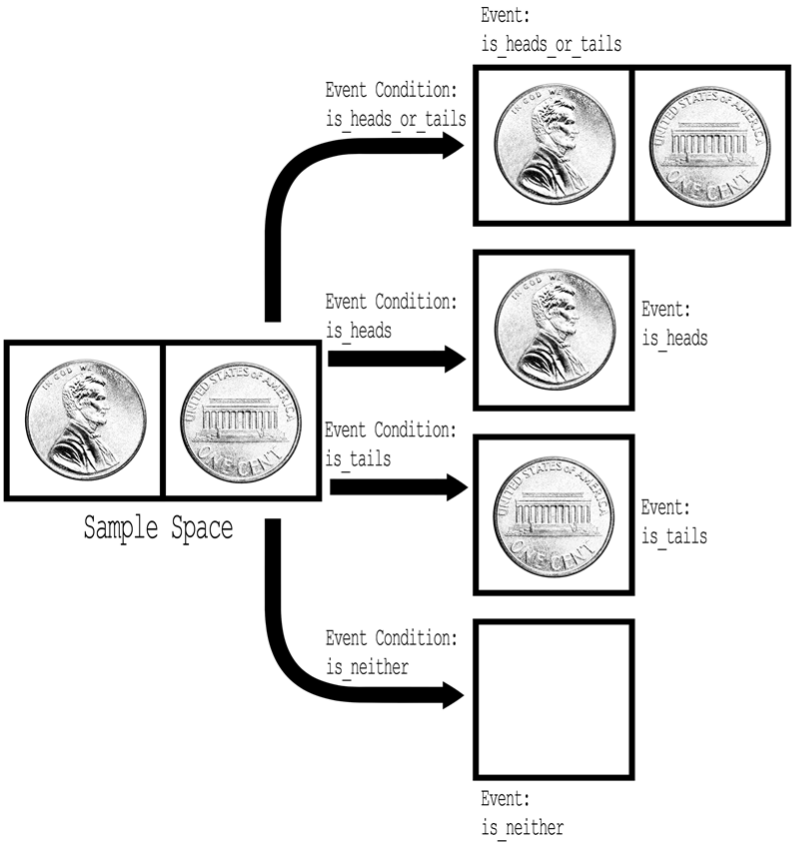


**Figure 1.1 4 event conditions applied to a sample space. The sample space contains 2 outcomes; heads and tails. Arrows represent the event conditions. Every event condition is a yes-or-no function. Each function filters out those outcomes that do not satisfy its terms. The remaining outcomes form an event. Each event contains a subset of the outcomes that are found in the sample space. 4 events are possible; heads, tails, heads or tails, neither heads nor tails.**

Lets define 2 event conditions: one where the coin lands on either heads or tails, and another where the coin lands on neither heads nor tails.

### Listing 1.3 Defining event conditions

```
def is_heads_or_tails(outcome):  return outcome in {'Heads', 'Tails'}
def is_neither(outcome): return not is_heads_or_tails(outcome)
```

Also, for completeness-sake, lets define event conditions for the 2 basic events in which the coin satisfies exactly one of our 2 potential outcomes.

### Listing 1.4 Defining additioning event conditions

```
def is_heads(outcome): return outcome == 'Heads'
def is_tails(outcome): return outcome == 'Tails'
```

We can pass event conditions into a generalized `get_events` function. That function is defined below. Its inputs are an event condition, and also a generic sample space. The function iterates through `generic_sample_space` and returns the set of outcomes where

`event_condition(outcome)` is `True`.

**Listing 1.5 Defining an event detection function**

```
def get_event(event_condition, sample_space):
    return set([outcome for outcome in sample_space
                if event_condition(outcome)])
```

Lets execute `get_event` on our 4 event conditions. Afterwards, we'll output the 4 extracted events.

**Listing 1.6 Detecting events using event conditions**

```
event_conditions = [is_heads_or_tails, is_heads, is_tails, is_neither]

for event_condition in event_conditions:
    print(f"Event Condition: {event_condition.__name__}")  ❶
    event = get_event(event_condition, sample_space)
    print(f'Event: {event}\n')
```

❶ Prints the name of an `event_condition` function.

```
Event Condition: is_heads_or_tails
Event: {'Tails', 'Heads'}

Event Condition: is_heads
Event: {'Heads'}

Event Condition: is_tails
Event: {'Tails'}

Event Condition: is_neither
Event: set()
```

We've successfully extracted 4 events from `sample_space`. What is the probability of each event occurring? Earlier, we showed that the probability of a single-element outcome for a fair coin is `1 / len(sample_space)`. This property can be generalized to include multi-element events. The probability of an event is equal to `len(event) / len(sample_space)`, but only if all outcomes are known to occur with equal likelihood. In other words, the probability of a multi-element event for a fair coin is equal to event size divided by the sample space size. We'll now leverage event size to compute the 4 event probabilities.

**Listing 1.7 Computing event probabilities**

```
def compute_probability(event_condition, generic_sample_space):
    event = get_event(event_condition, generic_sample_space)  ❶
    return len(event) / len(generic_sample_space)  ❷

for event_condition in event_conditions:
    prob = compute_probability(event_condition, sample_space)
    name = event_condition.__name__
    print(f"Probability of event arising from '{name}' is {prob}")
```

❶ The `compute_probability` function will extract the event associated with an inputted event condition in order to compute its probability.

❷ Probability is equal to event size divided by sample space size.

```
Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_heads' is 0.5
Probability of event arising from 'is_tails' is 0.5
Probability of event arising from 'is_neither' is 0.0
```

The executed code outputs a diverse range of event probabilities, the lowest of which is 0.0 and the largest of which is 1.0. These values represent the lower and upper bounds of probability; no probability can ever fall below 0.0 or rise above 1.0.

### 1.1.1 Analyzing a Biased Coin

We computed probabilities for an unbiased coin. What would happen if that coin was biased? Suppose, for instance, that a coin is 4 times more likely to land on heads relative to tails. How do we compute the likelihoods of outcomes that are not weighted in an equal manner? Simple! We'll construct a weighed sample space, represented by a Python dictionary. Each outcome will be treated as a key whose value maps to the associated weight. In our example, heads is weighted 4 times as heavily as tails. Therefore, we'll map `'Tails'` to 1 and `'Heads'` to 4.

**Listing 1.8 Representing a weighted sample space**

```
weighted_sample_space = {'Heads': 4, 'Tails': 1}
```

Our new sample space is stored within a dictionary. This allows us to redefine sample-space size as the sum of all dictionary weights. Within `weighted_sample_space`, that sum will equal 5.

**Listing 1.9 Checking the weighted sample space size**

```
sample_space_size = sum(weighted_sample_space.values())
assert sample_space_size == 5
```

We can redefine event size in similar manner. Each event is a set of outcomes. These outcomes map to weights. Summing over these weights will yield the event size. Thus, the size of the event satisfying the `is_heads_or_tails` event condition is also 5.

**Listing 1.10 Checking the weighted event size**

```
event = get_event(is_heads_or_tails, weighted_sample_space)
event_size = sum(weighted_sample_space[outcome] for outcome in event)
assert event_size == 5
```

Our generalized definitions of sample-space size and event size permit us to create a `compute_event_probability` function. The function takes as input a `generic_sample_space` variable that can be either a weighted dictionary or an unweighted set.

**Listing 1.11 Defining a generalized event probability function**

```
def compute_event_probability(event_condition, generic_sample_space):
    event = get_event(event_condition, generic_sample_space)
    if type(generic_sample_space) == type(set()):   ❶
        return len(event) / len(generic_sample_space)

    event_size = sum(generic_sample_space[outcome]
                     for outcome in event)
    return event_size / sum(generic_sample_space.values())
```

❶ Checks if `generic_event_space` is a set.

We can now output all the event probabilities for the biased coin without needing to redefine our 4 event condition functions.

**Listing 1.12 Computing weighted event probabilities**

```
for event_condition in event_conditions:
    prob = compute_weighted_probability(event_condition, weighted_sample_space)
    name = event_condition.__name__
    print(f"Probability of event arising from '{name}' is {prob}")
```

```
Probability of event arising from 'is_heads' is 0.8
Probability of event arising from 'is_tails' is 0.2
Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_neither' is 0.0
```

With just few lines of code, we have constructed a tool for solving many problems in probability. Lets apply this tool to a series of problems more complex than a simple coin-flip.

## 1.2 Computing Non-Trivial Probabilities

We will now proceed to solve several example problems using `compute_event_probability`.

### 1.2.1 Problem 1: Analyzing a Family with 4 Children

Suppose a family has 4 children. What is the probability that exactly 2 of the children are boys? We'll assume that each child is equally likely to be either a boy or a girl. This allows us to construct an unweighted sample space where each outcome is a 4-element tuple representing one possible sequence of 4 children.

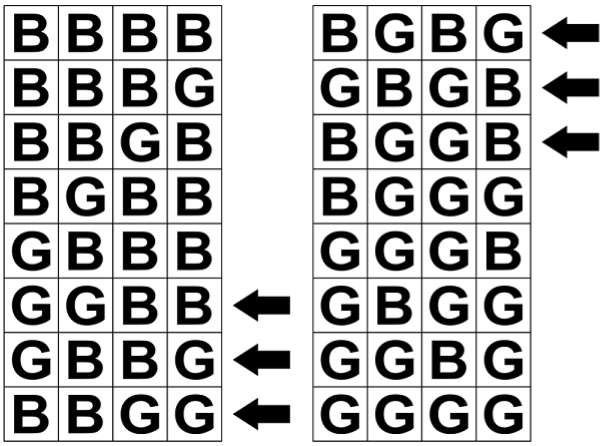


**Figure 1.2 The sample space for 4 successive children. Each row in the sample space contains one of 16 possible outcomes. Every outcome represents a unique combination of 4 children. The sex of each child is signaled by a letter; B for Boy and G for Girl. Outcomes with 2 boys are marked by an arrow. There are 6 such arrows present. Thus, the probability of 2 boys equals 6 / 16.**

### Listing 1.13 Computing the sample space of children

```
possible_children = ['Boy', 'Girl']
sample_space = set()
for child1 in possible_children:
    for child2 in possible_children:
        for child3 in possible_children:
            for child4 in possible_children:
                outcome = (child1, child2, child3, child4)
                sample_space.add(outcome)
```

We ran 4 nested for-loops to explore the sequence of 4 births. This is not an efficient use of code. We can more easily generate our sample space using Python's build-in `itertools.product` function. The function returns all pairwise combinations of all elements across all input lists. Below, we input 4 instances of the `possible_children` list into `intertools.product`. The product function then proceeds to iterate over all 4 instances of the list, computing all the combinations of list elements. The final output equals our sample space.

### Listing 1.14 Computing the sample space using `product`

```
from itertools import product
all_combinations = product(*(4 * [possible_children]))   ❶
assert set(all_combinations) == sample_space
```

❶ The `*` operator unpacks multiple arguments stored within a list. These arguments are then passed into a specified function. Thus, calling ` product(*(4 * [possible_children]))` is equivalent to calling `product(possible_children, possible_children, possible_children, possible_children)`.

We can make our code even more efficient by executing `set(product(possible_children, repeat=4))`. In general, running `product(possible_children, repeat=n)` will return an iterable over all possible combinations of n children.

**Listing 1.15 Passing `repeat` into `product`**

```
sample_space_efficient = set(product(possible_children, repeat=4))
assert sample_space == sample_space_efficient
```

Lets calculate the fraction of `sample_space` that is composed of families with 2 boys. We'll first define a `has_two_boys` event condition. Afterwards, will pass that condition into `compute_event_probability`.

**Listing 1.16 Computing the probability of 2 boys**

```
def has_two_boys(outcome): return len([child for child in outcome
                                       if child == 'Boy']) == 2

prob = compute_event_probability(has_two_boys, sample_space)
print(f"Probability of 2 boys is {prob}")
```

```
Probability of 2 boys is 0.375
```

The probability of exactly 2 boys being born in family of 4 children is 0.375. By implication, we expect 37.5% of families with 4 children to contain an equal number of boys and girls. Of course, the actual observed percentage of families with 2 boys will vary due to random chance.

### *1.2.2 Problem 2: Analyzing Multiple Dice Rolls*

Suppose we're shown a fair six-sided die whose faces are numbered from 1 to 6. The die is rolled 6 times. What is the probability that these 6 dice-rolls add up to 21?

We'll begin by defining the possible values of any single roll. These are integers that range from 1 to 6.

**Listing 1.17 Defining all possible rolls of a six-sided die**

```
possible_rolls = list(range(1, 7))
print(possible_rolls)
```

```
[1, 2, 3, 4, 5, 6]
```

Next, we'll create the sample space for 6 consecutive rolls using the `product` function.

**Listing 1.18 The sample space for 6 consecutive dice rolls**

```
sample_space = set(product(possible_rolls, repeat=6))
```

Finally, we'll define a `has_sum_of_21` event condition that we'll subsequently pass into `compute_event_probability`.

**Listing 1.19 Computing the probability of a dice-roll sum**

```
def has_sum_of_21(outcome): return sum(outcome) == 21
prob = compute_event_probability(lambda x: x == 21,
                                 weighted_sample_space)
assert prob == compute_event_probability(has_sum_of_21, sample_space)
print(f"Probability of dice summing to 21 is {prob}")
```

```
Probability of dice summing to 21 is 0.09284979423868313
```

The 6 dice-rolls will sum to 21 more than 9% of the time. We should note that the above analysis can be executed in a single line of code. Lets simplify our code with a lambda expression. Lambda expressions are one-line anonymous functions that do not require a name. In this book, we'll use lambda expressions to pass short functions into other functions.

**Listing 1.20 Computing the probability using a lambda expression**

```
prob = compute_event_probability(lambda x: sum(x) == 21, sample_space)  ❶
assert prob == compute_event_probability(has_sum_of_21, sample_space)
```

❶ Lambda expressions allow us to define short functions in a single line of code. Coding `lambda x:` is functionally equivalent to coding `func(x):`. Thus, `lambda x: sum(x) == 21` is functionally equivalent to `has_sum_of_21`.

### *1.2.3 Problem 3: Computing Dice-Roll Probabilities using Weighted Sample Spaces*

We've just computed the likelihood of 6 rolled dice summing to 21. Now, lets recompute that probability using a weighted sample space. How do we convert our unweighted sample-space set into a weighted sample-space dictionary? The solution is simple. We must first identify all possible sums of 6 dice-rolls. Then, we must count the number of times each sum appears across all possible 6-dice-roll-combinations. These combinations are already stored in our computed `sample_space` set. By mapping the dice-roll sums to their occurrence counts, we will produce a `weighted_sample_space` result.

**Listing 1.21 Mapping dice-roll sums to ocurrence counts**

```
from collections import defaultdict   ❶
weighted_sample_space = defaultdict(int)    ❷
for outcome in sample_space:   ❸
    total = sum(outcome)    ❹
    weighted_sample_space[total] += 1   ❺
```

❶ This module returns dictionaries whose keys are all assigned a default value. For instance, `defaultdict(int)` returns a dictionary where the default value for each key is set to zero.

❷ The `weighted_sample` dictionary maps each summed 6-dice-roll combination to its occurrence count.

❸ Each `outcome` contains a unique combination of six rolled dice.

❹ We compute the summed value of 6 unique dice rolls.

❺ We update the occurrence count for a summed dice value.

Before we recompute our probability, lets briefly explore the properties of `weighted_sample_space`. Not all weights in the sample space are equal. Some of the weights are much smaller than others. For instance, there is only one way for the dice to sum to 6. We must roll precisely 6 ones to achieve that dice-sum combination. Hence, we expect `weighted_sample_space[6]` to equal 1. Furthermore, we expect `weighted_sample_space[36]` to also equal 1, since we must roll 6 sixes to achieve a sum of 36.

**Listing 1.22 Checking very rare dice-roll combinations**

```
assert weighted_sample_space[6] == 1
assert weighted_sample_space[36] == 1
```

Meanwhile, the value of `weighted_sample_space[21]` is noticeably higher.

**Listing 1.23 Checking a more common dice-roll combination**

```
num_combinations = weighted_sample_space[21]
print(f"There are {num_combinations } ways for six rolled dice to sum to 21")
```

```
There are 4332 ways for six rolled dice to sum to 21
```

There are 4332 ways for 6 rolled dice to sum to 21. For example, we could roll 4 fours, followed by a three and then a two. Or we could roll 3 fours followed by a five, a three, and a one. Thousands of other combinations are possible. This is why a sum of 21 is much more probable than a sum of 6.

**Listing 1.24 Exploring different ways of summing to 21**

```
assert sum([4, 4, 4, 4, 3, 2]) == 21
assert sum([4, 4, 4, 5, 3, 1]) == 21
```

Please note that the observed count of 4332 is equal to the length of an unweighted event whose dice-rolls add up to 21. Also, the sum of values in `weighted_sample` is equal to the length of `sample_space`. Hence, there exists a direct link between unweighted and weighted event probability computation.

**Listing 1.25 Comparing weighted events and regular events**

```
event = get_event(lambda x: sum(x) == 21, sample_space)
assert weighted_sample_space[21] == len(event)
assert sum(weighted_sample_space.values()) == len(sample_space)
```

Lets now recompute the probability using the `weighted_sample_space` dictionary. The final probability of rolling a 21 should remain unchanged.

**Listing 1.26 Computing the weighted event probability of dice rolls**

```
prob = compute_event_probability(lambda x: x == 21,
                                 weighted_sample_space)
assert prob == compute_event_probability(has_sum_of_21, sample_space)
print(f"Probability of dice summing to 21 is {prob}")
```

```
Probability of dice summing to 21 is 0.09284979423868313
```

What is the benefit of using a weighted sample space over an unweighted one? Less memory usage! As we see below, the unweighted `sample_space` set has on the order of 1500x more elements than the weighted sample space dictionary.

**Listing 1.27 Comparing weighted to unweighted event space size**

```
print('Number of Elements in Unweighted Sample Space:')
print(len(sample_space))
print('Number of Elements in Weighted Sample Space:')
print(len(weighted_sample_space))
```

```
Number of Elements in Unweighted Sample Space:
46656
Number of Elements in Weighted Sample Space:
31
```

## 1.3 Computing Probabilities Over Interval Ranges

So far, we've only analyzed event conditions that satisfy some single value. Now, we'll analyze event conditions that span across intervals of values. An **interval** is the set of all the numbers that are sandwiched between 2 boundary cutoffs. Lets define an `is_in_interval` function that checks whether a number falls within a specified interval. We'll control the interval boundaries by passing a `minimum` and a `maximum` parameter.

**Listing 1.28 Defining an interval function**

```
def is_in_interval(number, minimum, maximum):
    return minimum <= number <= maximum
```

Given the `is_in_interval` function, we can compute the probability that an event's associated value falls within some numeric range. For instance, let's compute the likelihood that our 6 consecutive dice-rolls sum up to a value between 10 and 21.

**Listing 1.29 Computing the probability over an interval**

```
prob = compute_event_probability(lambda x: is_in_interval(x, 10, 21),   ❶
                                 weighted_sample_space)
print(f"Probability of interval is {prob}")
```

❶ We define a lambda function that takes an input some `x`, and returns `True` if `x` falls in an interval between 10 and 21. This one-line lambda function serves as our event condition.

```
Probability of interval is 0.5446244855967078
```

The 6 dice rolls will fall into that interval range more than 54% of the time. Thus, if a roll-sum of 13 or 20 comes up, we should not be surprised.

### 1.3.1 Evaluating Extremes Using Interval Analysis

Interval analysis is critical to solving a whole class of very important problems in probability and statistics. One such problem involves the evaluation of extremes. The problem boils down to whether observed data is too extreme to be believable.

Data seems extreme when it is too unusual to have occurred by random chance. For instance, suppose we observe 10 flips of an allegedly fair coin, and that coin lands on heads 8 out of 10 times. Now, we had expected the coin to hit tails half the time, not 20% of the time, so the observations seem a little strange. Is the coin actually fair? Or has it been secretly replaced with a trick coin that falls on heads a majority of the time? We'll try to find out by asking the following question: what is the probability that 10 fair coin-flips lead to an extreme number of heads? We'll define an extreme head-count as observing of 8 heads or more. Thus, we can describe the problem as follows: what is the probability that 10 fair coin-flips produce between 8 and 10 heads?

We'll find our answer by computing an interval probability. However, first we need the sample space for every possible sequence of 10 flipped coins. Lets generate a weighted sample space. As previously discussed, this is more efficient then using a non-weighted representation.

The following code creates a `weighted_sample_space` dictionary. Its keys equal the total number of observable heads; ranging from 0 to 10. These head-counts map to values. Each value holds the number of coin-flip combinations that contain the associated head-count. We thus expect `weighted_sample_space[10]` to equal 1, since there is just one possible way to flip a coin 10 times and get 10 heads. Meanwhile, we expect `weighted_sample_space[9]` to equal 10, since a single tail among 9 heads can occur across 10 different positions.

### Listing 1.30 Computing the sample space for 10 coin-flips

```
def generate_coin_sample_space(num_flips=10):  ❶
    weighted_sample_space = defaultdict(int)
    for coin_flips in product(['Heads', 'Tails'], repeat=num_flips):
        heads_count = len([outcome for outcome in coin_flips  ❷
                           if outcome == 'Heads'])
        weighted_sample_space[heads_count] += 1

    return weighted_sample_space

weighted_sample_space = generate_coin_sample_space()
assert weighted_sample_space[10] == 1
assert weighted_sample_space[9] == 10
```

❶ For reusability, we'll define a general function that returns a weighted sample space for `num_flips` coin-flips. The `num_flips` parameter is preset to 10 coin-flips.

❷ The number of heads in a unique sequences of `num_flips` coin-flips.

Our weighted sample space is ready. We'll now compute the probability of observing an interval between 8 and 10 heads.

### Listing 1.31 Computing an extreme head-count probability

```
prob = compute_event_probability(lambda x: is_in_interval(x, 8, 10),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads is {prob}")
```

```
Probability of observing more than 7 heads is 0.0546875
```

10 fair coin-flips produce more than 7 heads approximately 5% of the time. Our observed head-count does not commonly occur. Does this mean the coin is biased? Not necessarily. Observing 8 out of 10 tails is as extreme as observing 8 out of 10 heads. Had we observed 8 tails and not 8 heads, we would still be suspicious of the coin. Our computed interval did not take this tails-driven extreme into account. Instead, we treated 8 or more tails as just another normal possibility. If we truly wish to measure the fairness of our coin, we'll need to update our interval computations. We'll need to include the likelihood of observing 8 or more tails. This is equivalent to observing 2 heads or less.

Lets formulate the problem as follows; what is the probability that 10 fair coin-flips produce either 0 to 2 heads or 8 to 10 heads? Or, stated more concisely, what is the probability that the coin-flips do NOT produce between 3 and 7 heads? That probability is computed below.

### Listing 1.32 Computing an extreme interval probability

```
prob = compute_event_probability(lambda x: not is_in_interval(x, 3, 7),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads or 7 tails is {prob}")
```

```
Probability of observing more than 7 heads or 7 tails is 0.109375
```

10 fair coin flips produce more than 7 heads or 7 tails approximately 10% of the time. That probability is not particularly high, but neither is it particularly low. Therefore our observations are both anomalous but also within the realm of plausibility. We're stuck in an indecisive limbo. The observed head-count is plausible enough to prevent us from fully proving bias. Without additional evidence, we simply cannot show that the coin is biased beyond a reasonable doubt. So lets collect that evidence. Suppose we flip the coin 10 additional times, and 8 more heads come up. This brings us to 16 heads out of 20 coin-flips total. Our confidence in the fairness of the coin has been reduced; but by how much? We don't know. We need to measure the change in probability. Lets find the probability of 20 fair coin-flips not producing between 5 and 15 heads.

**Listing 1.33 Analyzing extreme head-counts for 20 fair coin-flips**

```
weighted_sample_space_20_flips = generate_coin_sample_space(num_flips=20)
prob = compute_event_probability(lambda x: not is_in_interval(x, 5, 15),
                                 weighted_sample_space_20_flips)
print(f"Probability of observing more than 15 heads or 15 tails is {prob}")
```

```
Probability of observing more than 15 heads or 15 tails is 0.01181793212890625
```

The updated probability has dropped from approximately .1 to approximately .01. Thus, the added evidence has caused a 10-fold decrease in our confidence of fairness. Despite this probability drop, the ratio of heads to tails has remained constant at 4-to-1. Both our original and updated experiments produced 80% heads and 20% tails. This leads to an interesting an question: why does the probability of observing 80% or more heads decrease as the supposedly fair coin gets flipped more times? We can find out through detailed mathematical analysis. However, a much more intuitive solution is to just visualize the distribution of head-counts across our 2 sample space dictionaries. The visualization would effectively be a plot of keys (head-counts) vs values (combination counts) present in each dictionary. We can carry out this plot using Matplotlib; Python's most popular visualization library. In the subsequent section, we will discuss Matplotlib usage, and its application to probability theory.

## 1.4 Summary

- A sample space is the set of all the possible outcomes that an action can produce.
- An event is a subset of the sample space containing just those outcomes that satisfy some event condition. An event condition is a Boolean function that takes as input an outcome and returns either `True` or `False`.
- The probability of an event equals the fraction of event outcomes that cover the entire sample space.
- Probabilities can be computed over numeric intervals. An interval is defined as the set of all the numbers that are sandwiched between 2 boundary values.
- Interval probabilities are useful for determining whether an observation appears extreme.

# 2 Plotting Probabilities Using Matplotlib

***This section covers:***

- Creating simple plots using Matplotlib.
- Labeling plotted data.
- What is a probability distribution?
- Plotting and comparing multiple probability distributions.

Data plots are among the most valuable tools in any data scientist's arsenal. Without good visualizations, we are effectively crippled in our ability to gleam insights from our data. Fortunately we have at our disposal the external Python Matplotlib library, which is fully optimized for outputting high-caliber plots and data visualizations.In this section, we will leverage Matplotlib to better comprehend the coin-flip probabilities that we computed in Section One.

## 2.1 Basic Matplotlib Plots

Lets begin by installing the Matplotlib library.

**NOTE** Call "pip install matplotlib" from the command-line terminal in order to install the Matplotlib library.

Once installation is complete, we'll proceed to import `matplotlib.pyplot`, which is the library's main plot generation module. According to convention, the module is commonly imported using the shortened alias `plt`.

**Listing 2.1 Importing Matplotlib**

```
import matplotlib.pyplot as plt
```

We will now plot some data using `plt.plot`. That method takes as input 2 iterables; `x` and `y`. Calling `plt.plot(x, y)` will prepare a 2D plot of `x` vs `y`. Displaying the plot requires a subsequent call to `plt.show()`. Let's assign our `x` to equal integers 0 through 10 We'll assign our y-values to equal double the values of `x`. The code below will visualize that linear relationship.

**Listing 2.2 Plotting a linear relationship**

```
x = range(0, 10)
y = [2 * value for value in x]
plt.plot(x, y)
plt.show()
```

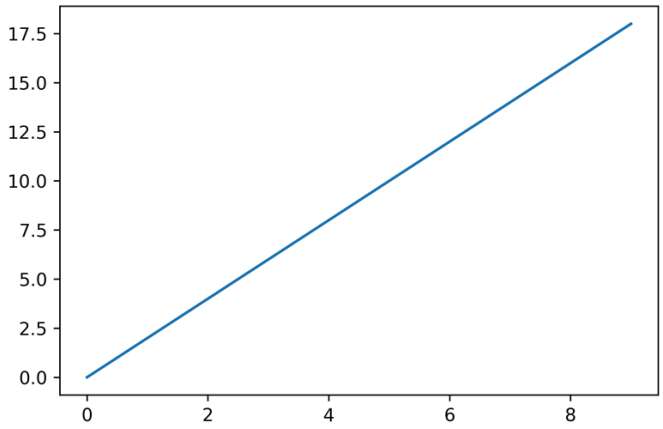


**Figure 2.1 A Matplotlib plot of x vs 2x. The x variable represents integers 1 through 10.**

**WARNING** The axes in the linear plot are not evenly spaced. The slope of the plotted line appears less steep than it actually is. We can equalize both axes by calling `plt.axis('equal)`. However, this will lead to awkward visualization containing too much empty space. Throughout this book we will rely on Matplotlib's automated axes adjustments, while also carefully observing the adjusted lengths.

The visualization is complete. Within it, our 10 y-axis points have been connected using smooth line segments. If we prefer to visualize the 10 points individually, we can do so through the `plt.scatter` method.

**Listing 2.3 Plotting individual data-points**

```
plt.scatter(x, y)
plt.show()
```

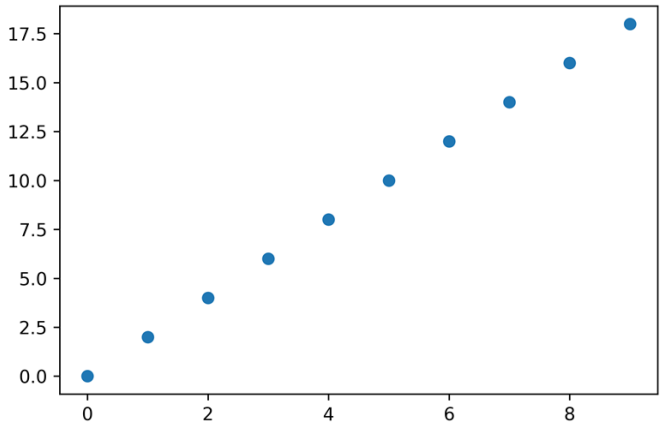


**Figure 2.2 A Matplotlib scatter plot of x vs 2 * x. The x variable represents integers 1 through 10. The individual integers are visible as scattered points in the plot.**

Suppose we want to emphasize the interval where `x` begins at 2 and ends at 6. We do this by shading the area under the plotted curve over the specified interval, using the `plt.fill_between` method. The method takes as input both `x`, and `y`, and also a `where` parameter, which defines the interval coverage. The input of the `where` parameter is a list of Boolean values, in which an element is `True` if the x-value at the corresponding index falls within the interval we specified. In the plot below, we'll set the `where` parameter to equal `[is_in_interval(value, 2, 6) for value in x]`. We'll also execute `plt.plot(x,y)`, in order to juxtapose the shaded interval with the smoothly connected line.

**Listing 2.4 Shading an interval beneath a connected plot**

```
plt.plot(x, y)
where = [is_in_interval(value, 2, 6) for value in x]
plt.fill_between(x, y, where=where)
plt.show()
```

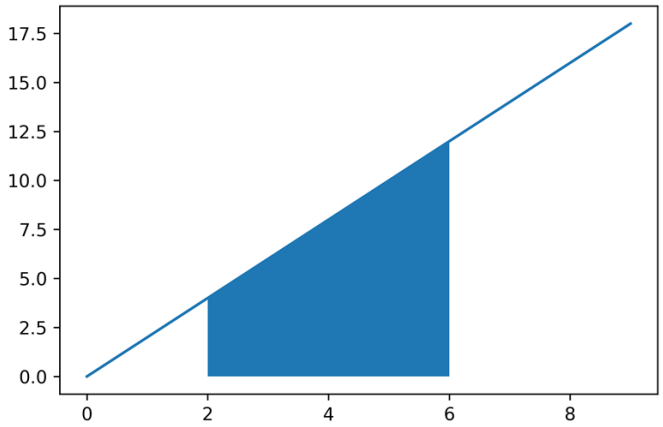


**Figure 2.3 A connected plot with a shaded interval. The interval covers all values between 2 and 6.**

So far we have reviewed 3 visualization methods; `plt.plot`, `plt.scatter`, and `plt.fill_between`. Lets execute all 3 methods in a single plot. This will highlight an interval beneath a continuous line while also exposing individual coordinates.

**Listing 2.5 Exposing individual coordinates within a continuous plot.**

```
plt.scatter(x, y)
plt.plot(x, y)
plt.fill_between(x, y, where=where)
plt.show()
```

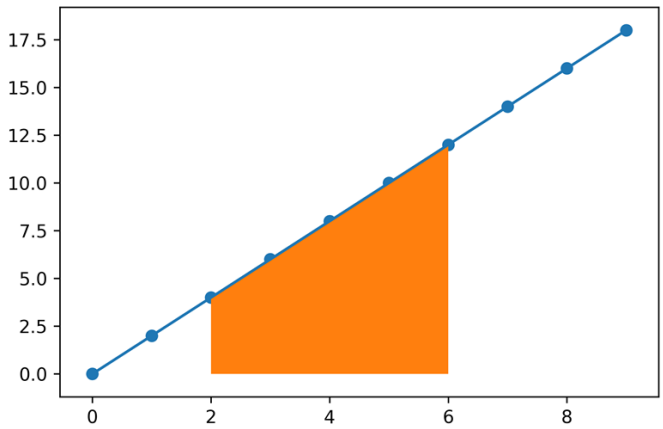


**Figure 2.4 A connected plot and a scatter plot combined together with a shaded interval. The individual integers in the plot appear as points marking a smooth, indivisible line.**

Furthermore, no data plot is ever truly complete without descriptive x-axis and y-axis labels. Such labels can be set using the `plt.xlabel` and `plt.ylabel` methods.

## Listing 2.6 Adding axes labels

```
plt.plot(x, y)
plt.xlabel('Values between zero and ten')
plt.ylabel('Twice the values of x')
plt.show()
```

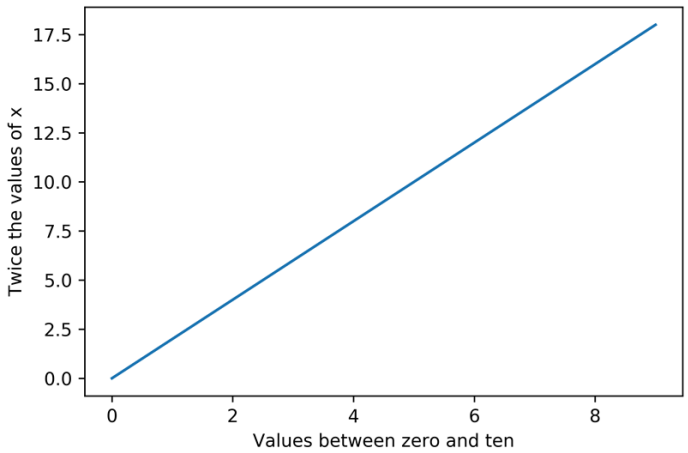


**Figure 2.5 A Matplotlib plot with x-axis and y-axis labels.**

**SIDEBAR** **Common Matplotlib method calls**

- `plt.plot(x, y)`:
  Plots the elements of `x` vs the elements of `y`. The plotted points are connected using smooth line segments.
- `plt.scatter(x, y)`:
  Plots the elements of `x` vs the elements of `y`. The plotted points are visualized individually, and are not connected by any lines.
- `plt.fill_between(x, y, where=booleans)`:
  Highlights the area beneath a plotted curve. The curve is obtained by plotting `x` vs `y`. The `where` parameter defines the highlighted interval. It takes a list of Booleans that correspond to elements of `x`. Each Boolean is `True` if its corresponding x-value is located within the highlighted interval.
- `plt.xlabel(label)`:
  Sets the x-label of the plotted curve to equal `label`.
- `plt.ylabel(label)`:
  Sets the y-label of the plotted curve to equal `label`.

## 2.2 Plotting Coin-Flip Probabilities

We now have tools to visualize the relationship between coin-flip count and the probability of heads. In Section One, we examined the probability of seeing 80% or more heads across a series of coin-flips. That probability decreased as the coin-flip count went up. We wanted to know why. We'll soon find out, by plotting head-counts vs their associated coin-flip combination counts. These values have already been computed during our Section One analysis. The keys in the `weighed_sample_set` dictionary contain all possible head-counts across 10 flipped coins. These head-counts map to combination counts. Meanwhile, the `weighted_sample_space_20_flips` dictionary contains the head-count mappings for 20 flipped-coins.

Our aim is to compare the plotted data from both these dictionaries. We will begin plotting the elements of `weighted_sample_space`. We'll plot its keys on the x-axis vs the associated values on the y-axis. The x-axis will correspond to `'Head-count'`. The y-axis will correspond to `'Number of coin-flip combinations with x heads'`. We'll use a scatter plot to visualize key-to-value relationships directly, without connecting any plotted points.

**Listing 2.7 Plotting the coin-flip weighted sample space**

```
x_10_flips = list(weighted_sample_space.keys())
y_10_flips = [weighted_sample_space[key] for key in x_10_flips]
plt.scatter(x_10_flips, y_10_flips)
plt.xlabel('Head-count')
plt.ylabel('Number of coin-flip combinations with x heads')
plt.show()
```

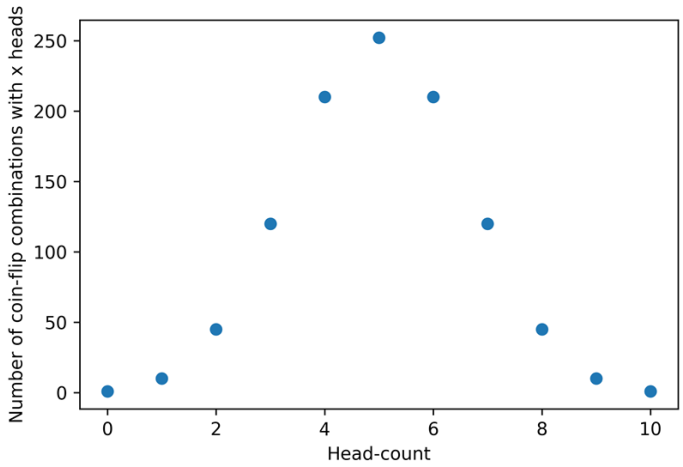


**Figure 2.6 A scatter-plot representation of the sample space for 10 flipped coins. The symmetric plot is centered around a peak at 5-of-10 counted heads.**

The visualized sample space takes on a symmetric shape. The symmetry is set around peak head-count of 5. Therefore, head-count combinations closer to 5 occur more frequently than those that are further from 5. As we learned in the previous section, such frequencies correspond to probabilities. Thus, a head-count is more probable if its value is closer to 5. Lets emphasize

this by plotting the probabilities directly on the y-axis. The probability plot will allow us to replace our lengthy y-axis label with a more concisely stated `'Probability'`. We can compute the y-axis probabilities by taking our existing combination counts and dividing them by the total sample space size.

**Listing 2.8 Plotting the coin-flip probabilities.**

```
sample_space_size = sum(weighted_sample_space.values())
prob_x_10_flips = [float(value)/ sample_space_size for value in y_10_flips]
plt.scatter(x_10_flips, prob_x_10_flips)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

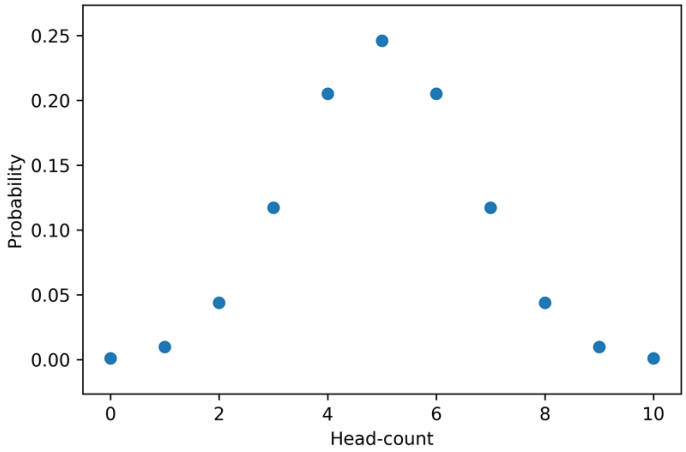


**Figure 2.7 A scatter-plot mapping head-counts to their probability of occurrence. Probabilities can be inferred by looking directly at the plot.**

We've directly visualized the relationship between head-counts and probabilities. Our plot permits us to visually estimate the probability of any head-count. Thus, just by glancing at the plot, we can determine that the probability of observing 5 heads is approximately .25. This mapping between x-values and probabilities is referred to as a **probability distribution**. Probability distributions exhibit certain mathematically consistent properties that make them useful for likelihood analysis. For instance, consider the x-values of any probability distribution. These correspond to all the possible values of some random variable `r`. The probability that `r` falls within some interval is equal to the area beneath the probability curve over the span of that interval. Therefore, the total area beneath a probability distribution always equals 1.0. This holds for any distribution, including our head-count plot. Lets confirm by executing `sum(prob_x_10_flips)`.

**Listing 2.9 Confirming that all probabilities sum to 1.0**

```
assert sum(prob_x_10_flips) == 1.0
```

Consequently, the area beneath the head-count interval of 8 through 10 is equal to the probability

of observing 8 heads or more. We can visualize that area using the `plt.fill_between` method. Within the visualization plot, a smooth-line representation will be employed to properly outline the shaded area boundary. Lets visualize that interval using a combined `plt.plot` and `plt.scatter` representation so as to shade the area while also emphasizing invidiual head-count values.

**Listing 2.10 Shading the interval under a probability curve**

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [is_in_interval(value, 8, 10) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

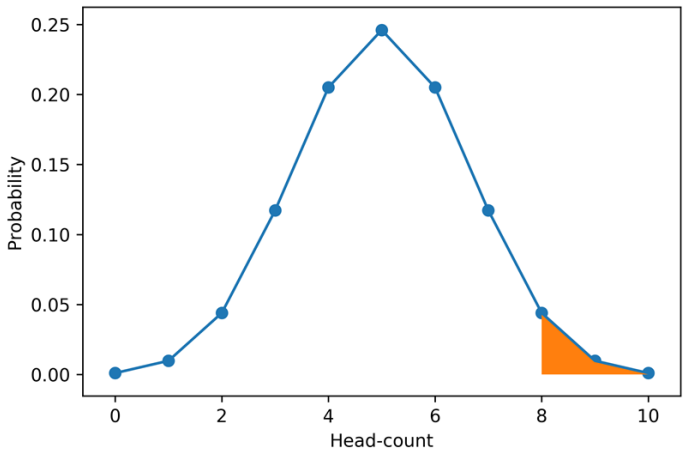


**Figure 2.8 An overlaid smooth plot and scatter-plot representation of the coin-flip probability distribution. A shaded interval covers head-counts 8 through 10. The shaded area equals the probability of observing 8 or more heads.**

**NOTE** We've purposefully smoothed the shaded interval to make a visually appealing plot. However, the true interval area is not smooth. It is composed of discrete, rectangular blocks, which resemble steps. The steps are discrete because the head-counts are indivisible integers. If we wish to visualize the actual step-shaped area, we can pass a `steps=pre` parameter into `plt.plot` and `plt.fill_between`.

Now, lets also the shade the interval demarcating the probability of observing 8 tails or more. The code below will highlight the extremes along both tail-ends of our probability distribution.

**Listing 2.11 Shading the interval under the extremes of a probability curve**

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

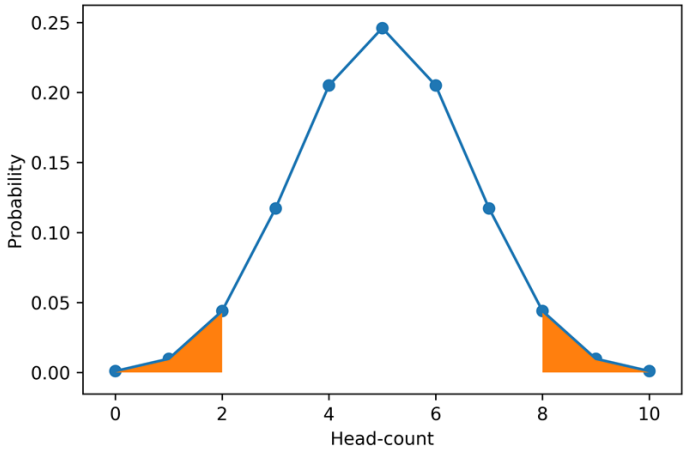


**Figure 2.9 An overlaid smooth plot and scatter-plot representation of the coin-flip probability distribution. 2 shaded intervals span an extreme number of heads and tails. The intervals are symmetric, visually implying that their probabilities are equal.**

The 2 symmetrically shaded intervals cover the right and left tail-ends of the coin-flip curve. Based on our previous analysis, we know that the probability of observing more than 7 heads or tails is approximately 10%. Therefore, each of the symmetrically shaded tail segments should cover approximately 5% of the total area under the curve.

### *2.2.1 Comparing Multiple Coin-Flip Probability Distributions*

Plotting the 10 coin-flip distribution makes it easier to visually comprehend the associated interval probabilities. Lets extend our plot to also encompass the distribution for 20 flipped coins. We'll plot both distributions on a single figure, though first we must compute the x-axis head-counts and y-axis probabilities for the 20 coin-flip distribution.

**Listing 2.12 Computing probabilities for a 20 coin-flip distribution**

```
x_20_flips = list(weighted_sample_space_20_flips.keys())
y_20_flips = [weighted_sample_space_20_flips[key] for key in x_20_flips]
sample_space_size = sum(weighted_sample_space_20_flips.values())
prob_x_20_flips = [value / sample_space_size for value in y_20_flips]
```

Now we are ready to visualize the 2 distributions simultaneously. We'll do this by executing `plt.plot` and `plt.scatter` on both probability distributions. We will also pass a few

style-related parameters into these method-calls. One of the parameters is `color`. We want the second distribution to be visually distinct from our first. Therefore, we'll set its color to black by passing `color='black'`. Alternatively, we can avoid typing out the entire color name by passing `'k'`; Matplotlib's single-character code for black. Thus, inputting `color='k'` achieves the same result with less typed text. Furthermore, we can make the 20-coin distribution plot stand out in other ways. Passing `linestyle=='--'` into `plt.plt` will ensure that the distribution points are connected using dashed lines instead of regular lines. Also, we can distinguish the individual points using x-shaped markers rather than filled circles by passing `marker='x'` into `plt.scatter`. Finally, we'll add a legend to our figure. The legend will help distinguish between each distribution style. We'll generate the legend by passing a `label` parameter into each of our 2 `plt.plot` calls. Afterwards, we'll execute the `plt.legend()` method, in order to display the legend. Within our legend, the 10 coin-flip distribution and the 20 coin-flip distribution will be labeled as *A*, and *B*, respectively.

**SIDEBAR** **Common Matplotlib style parameters**

- `color`:
  Determines the color of the plotted output. This setting can be a color name or a single-character code. Both `color=black` and `color=k` will generate a black plot. Both `color=red` and `color=r` will generate a red.
- `'linestyle'`:
  Determines the style of the plotted line that that connects the data-points. Its default value equals `'-'. Inputting `linestyle='-'` will generate a connected line. Inputting `linestyle='--'` will generate a dashed line. Inputting `linestyle=':'` will generate a dotted line. Inputting `linestyle='.'` will generate a line composed of alternating dots and dashes.
- `'marker'`:
  Determines the style of markers assigned to individually-plotted points. Its default value equals `'o'`. Inputting `maker='o'` will generate a circular marker. Inputting `'maker='x'` will generate an x-shaped marker. Inputting `'marker='s'` will generate a square-shaped marker. Inputting `'marker='p'` will generate a pentagon-shaped marker.
- `'label'`:
  Maps a label to the specified color and style. This mapping will appear in the legend of the plot. A subsequent call to `plt.legend()` is required to make the legend visible.

## Listing 2.13 Plotting 2 simultaneous distributions

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
plt.plot(x_20_flips, prob_x_20_flips, color='black', linestyle='--')
plt.scatter(x_20_flips, prob_x_20_flips, color='k', marker='x')
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

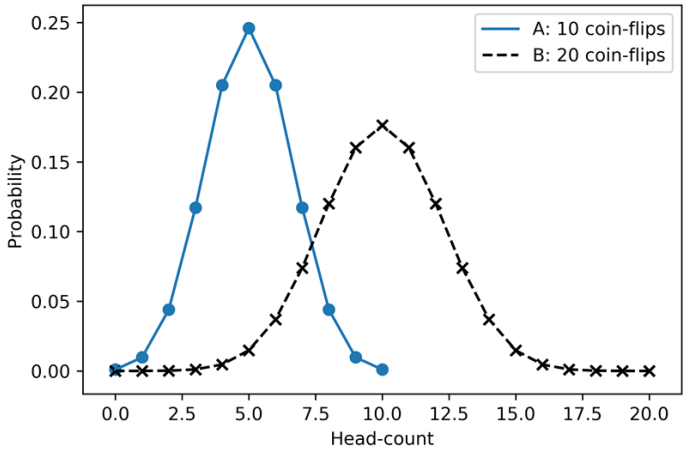


**Figure 2.10 The probability distributions for 10 coin-flips (A) and 20 coin-flips (B). The 20 coin-flip distribution is marked by dashed lines and x-shaped scattered points.**

We've visualized our 2 distributions. Next, we'll proceed to highlight our interval of interest (80% of heads or tails) across each of the 2 plotted curves. Please note that the area beneath the tail-ends of Distribution B is very small. We'll thus remove the scatter-points in order to highlight the tail-end intervals more clearly. We'll also substitute the line-style of Distribution B with the more transparent `linestyle=':'`.

## Listing 2.14 Highlighting intervals beneath 2 plotted distributions.

```
plt.plot(x_10_flips, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_flips, prob_x_20_flips, color='k', linestyle='--',
         label='B: 20 coin-flips')
plt.legend()

where_10 = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where_10)
where_20 = [not is_in_interval(value, 5, 15) for value in x_20_flips]
plt.fill_between(x_20_flips, prob_x_20_flips, where=where_20)

plt.xlabel('Head-Count')
plt.ylabel('Probability')
plt.show()
```

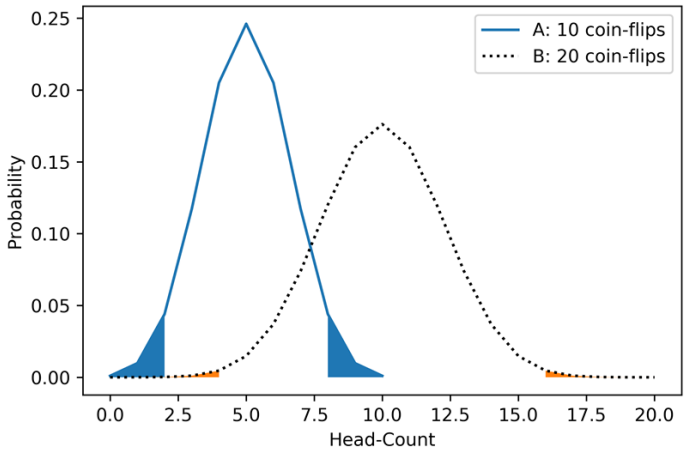


**Figure 2.11 The probability distributions for 10 coin-flips (A) and 20 coin-flips (B). Shaded intervals beneath both distributions represent an extreme number of heads and tails. The shaded interval beneath B occupies one-tenth the area of the shaded interval beneath A.**

The shaded area beneath the tail-ends of Distribution B is much lower than the shaded interval beneath Distribution A. This is because Distribution A has fatter, more elevated tail-ends that cover a thicker area quantity. Geometric thickness in the tails helps account for differences in interval probabilities between Distributions A and B.

The visualization is informative, but only if we highlight the interval areas beneath both curves. Without the calls to `plt.between` we cannot directly answer the question we posed earlier: Why does the probability of observing 80% or more heads decrease as the fair coin gets flipped more times? The answer is hard to extrapolate because the 2 distributions show little overlap, making it difficult to do a direct visual comparison. Perhaps we can improve the visualization by aligning the distribution peaks. Distribution A is centered at 5 head-counts (out of 10 coin-flips), and Distribution B is centered at 10 head-counts (out of 20 coin-flips), but if we convert the head-counts into frequencies (by dividing by the total coin flips), then both the distribution peaks should align at a frequency of 0.5. The conversion should also align our head-count intervals of 8-to-10 and 16-to-20 so that they both lie on the interval 0.8-to-1.0. Lets actually execute this x-value conversion by dividing the head-counts in Distribution A by 10, and head-counts in Distribution B by 20. Afterwards, we'll plot the frequencies against the y-axis probabilities.

### Listing 2.15 Converting head-counts into frequencies

```
x_10_frequencies = [head_count/10 for head_count in x_10_flips]
x_20_frequencies = [head_count/20 for head_count in x_20_flips]

plt.plot(x_10_frequencies, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_frequencies, prob_x_20_flips, color='k', linestyle=':', label='B: 20 coin-flips')
plt.legend()

plt.xlabel('Head-Frequency')
plt.ylabel('Probability')
plt.show()
```

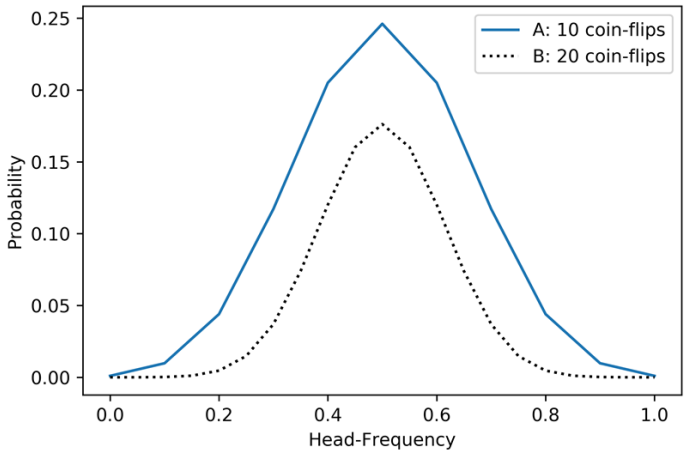


**Figure 2.12 The head-count frequencies for 10 coin-flips (A) and 20 coin-flips (B) have been plotted against their probabilities. Both y-axis peaks align at a frequency of 0.5. The area of A fully covers the area of B because the total area of each plot no longer sums to 1.0.**

As expected, the 2 peaks now both align at the head-frequency of 0.5. However, our division by the head-counts has reduced the areas beneath the 2 curves by 10-fold and 20-fold, respectively. The total area beneath each curve no longer equals 1.0. This is a problem, because as we've discussed, the total area beneath a curve must sum to 1.0 if we wish infer a probability from its shaded intervals. Thus, we'll need to force the aligned curve-areas to equal 1.0 prior to doing interval comparison. To do this, we'll simply multiply the y-axis values of curves A and B by 10 and 20. The adjusted y-values will no longer refer to probabilities, so we'll have to name them something else. The appropriate term to use is **relative likelihood**, which mathematically refers to a y-axis value within a curve whose total area is 1.0. We'll therefore name our new y-axis variables `relative_likelihood_10` and `relative_likelihood_20`.

## Listing 2.16 Computing relative likelihoods of frequencies

```
relative_likelihood_10 = [10 * prob for prob in prob_x_10_flips]
relative_likelihood_20 = [20 * prob for prob in prob_x_20_flips]
```

The conversion is complete. Its time to plot our 2 new curves, while also highlighting the intervals associated with our `where_10` and `where_20` Boolean arrays.

## Listing 2.17 Plotting aligned relative likelihood curves

```
plt.plot(x_10_frequencies, relative_likelihood_10, label='10 coin-flips')
plt.plot(x_20_frequencies, relative_likelihood_20, color='k',
         linestyle=':', label='20 coin-flips')

plt.fill_between(x_20_frequencies, relative_likelihood_20, where=where_20)
plt.fill_between(x_10_frequencies, relative_likelihood_10, where=where_10)


plt.legend()
plt.xlabel('Head-Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

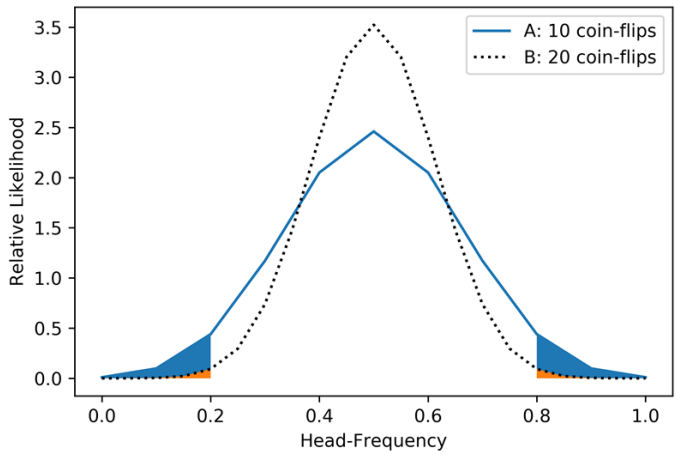


**Figure 2.13 The head-count frequencies for 10 coin-flips (A) and 20 coin-flips (B) have been plotted against their relative likelihoods. Shaded intervals beneath both plots represent an extreme number of heads and tails. The areas of these intervals correspond to probabilities because the total area of each plot sums up to 1.0.**

Within the plot, Curve A resembles a short yet muscular bodybuilder; the curve's maximum height is not that impressive even though its horizontal bulk covers a noticeable amount of area. Meanwhile, Curve B could be compared to a taller and thinner individual; the girth doesn't cover that much area even though its peak towers over highest point within Curve A. Since Curve A is wider, its area over more extreme head-frequency intervals is larger. Hence, observed recordings of such frequencies are more likely to occur when the coin-flip count is 10 and not 20. Meanwhile, the thinner more vertical Curve B covers more area around the central 0.5 frequency.

What will happen when more than 20 coins are flipped? How will the increased flip-count influence our frequency distribution? According to probability theory, each additional coin-flip will the cause the frequency curve to grow even taller and thinner. The curve will transform like a stretched rubber band that's being pulled vertically upward. It will lose thickness in exchange for vertical length. As the total number of coin flips extends into the millions and billions, the curve will completely lose its girth; becoming just single, very long vertical peak whose center lies at a frequency of 0.5. Beyond that frequency, the non-existent area beneath vertical line will

approach zero. It follows that the area beneath the peak will approach 1.0, because our total area must always equal 1.0. The area of 1.0 corresponds to a probability of 1.0. Thus, as the number of coin flips approaches infinity, the frequency of heads will come to equal the actual probability of heads with absolute certainty.

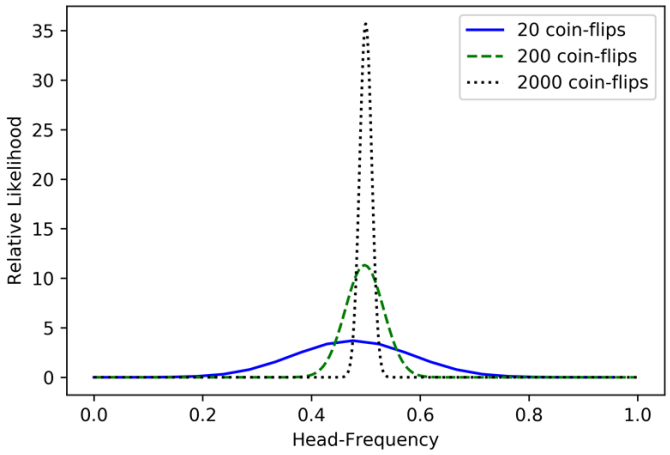


**Figure 2.14 Hypothetical head-count frequencies plotted over an increasing number of coin-flips. All y-axis peaks align at a frequency of 0.5. The peaks grow higher and more narrow as the coin-flip count goes up. At 2000 coin-flips, the constricted area of the peak is centered almost entirely at 0.5. With infinite coin-flips, the resulting peak should stretch into a single, vertical line that's perfectly positioned at 0.5.**

The relationship between infinite coin flips and absolute certainty is guaranteed by a fundamental theorem in probability theory: **the Law of Large Numbers**. According to that law, the frequency of an observation becomes virtually indistinguishable from the probability of that observation when the number of observations grows high. Therefore, with enough coin-flips, our frequency of heads will equal the actual probability of heads, which is 0.5. Beyond mere coin-flips; we can apply the law to more complex phenomena, such as card games. If we run enough card game simulations, then our frequency of a win will equal the actual probability of a win.

In the subsequent section, we will show how the Law of Large numbers can be combined with random simulations to approximate complex probabilities. Eventually, we will execute simulations to find the probabilities of randomly drawn cards. However, as the Law of "Large" numbers indicates, these simulations must be run on a large, computationally expensive scale. Therefore, efficient simulation implementation will require us to familiarize ourselves with the NumPy numeric computation library. That library will be discussed in Section Three.

## 2.3 Summary

- By plotting every possible numeric observation vs its probability, we generate a probability distribution. The total area beneath a probability distribution sums to 1. The area beneath a specific interval of the distribution equals the probability of observing some value within that interval.
- The y-axis values of a probability distribution do not necessarily need to equal probabilities, as long as the plotted area sums to 1.
- The probability distribution of a fair coin-flip sequence resembles a symmetric curve. Its x-axis head-counts can be converted into frequencies. During that conversion, we can maintain an area of 1 by converting y-axis probabilities into relative likelihoods. The peak of the converted curve is centered at a frequency of 0.5. If the coin-flip count is raised, then the peak will also rise as the curve becomes more narrow on its sides.
- According to the Law of Large Numbers, the frequency of any observation will approach the probability of that observation as the observation-count grows large. Thus, a fair-coin distribution becomes dominated by its central frequency of 0.5 as the coin-flip count goes up. se

# 3 Running Random Simulations in NumPy

***This section covers:***

- Basic usage of the NumPy library.
- Simulating random observation using NumPy.
- Visualizing simulated data.
- Estimating unknown probabilities from simulated observations.

NumPy, which stands for Numerical Python, is the engine that powers Pythonic data science. Python, despite its many virtues, is simply not suited for large-scale numeric analysis. Hence, data scientists must rely on the external NumPy library to efficiently manipulate and store numeric data. NumPy is an incredibly powerful tool for processing large collections of raw numbers. Thus, many of Python's external data-processing libraries are NumPy-compatible. One such library is Matplotlib, which we introduced in the previous section. Other NumPy-driven libraries will be discussed in later portions of the book. This section focuses on randomized numerical simulations. We will leverage NumPy to analyze billions of random data-points. These random observations will allow us to learn hidden probabilities.

## 3.1 Simulating Random Coin-Flips and Dice-Rolls Using NumPy

NumPy should already be installed within your working environment as one of the Matplotlib requirements. Lets proceed to import NumPy as `np`, based on common NumPy usage convention.

**NOTE** NumPy can also be installed independently of Matplotlib by calling "pip install numpy" from the command-line terminal.

### Listing 3.1 Importing NumPy

```
import numpy as np
```

Now that NumPy is imported, we can carry out random simulations using the `np.random` module. That module is useful for generating random values and simulating random processes. For instance, calling `np.random.randint(1, 7)` will produce a random integer between 1 and 6. The method choses from the six possible integers with equal likelihood, thus simulating a single roll of a standard die.

### Listing 3.2 Simulating a randomly rolled die

```
dice_roll = np.random.randint(1, 7)
assert 1 <= dice_roll <= 6
```

The generated `dice_roll` value is random, and its assigned value will vary amongst the readers of this book. The inconsistency could make it difficult to perfectly recreate certain random simulations in this section. We'll need a way of ensuring that all our random outputs can be reproduced at home. Conveniently, consistency can easily be maintained by calling `np.random.seed(0)`. The method-call makes sequences of randomly chosen values reproducible. After the call, we can directly guarantee that our first three dice-rolls will land on values 5, 6, and 1.

### Listing 3.3 Seeding reproducible random dice-rolls

```
np.random.seed(0)
dice_rolls = [np.random.randint(1, 7) for _ in range(3)]
assert dice_rolls == [5, 6, 1]
```

We'll now use `np.random.randint(0, 2)` to simulate a single flip of an unbiased coin.The method-call returns a random value equal either to 0 or 1. Within the simulation, we'll assume that 0 stands for tails, and 1 stands for heads.

### Listing 3.4 Simulating one fair coin-flip

```
np.random.seed(0)
coin_flip = np.random.randint(0, 2)
print(f"Coin landed on {'heads' if coin_flip == 1 else 'tails'}")
```

```
Coin landed on tails
```

Next, we'll simulate a sequence of 10 coin-flips, and then compute the observed frequency of heads.

### Listing 3.5 Simulating 10 fair coin-flips

```
np.random.seed(0)
def frequency_heads(coin_flip_sequence):
    total_heads = sum(coin_flip_sequence)
    return float(total_heads) / len(coin_flip_sequence)

coin_flips = [np.random.randint(0, 2) for _ in range(10)]
freq_heads = frequency_heads(coin_flips)
print(f"Frequency of Heads is {freq_heads}")
```

```
Frequency of Heads is 0.8
```

The observed frequency is 0.8, which is quite disproportionate from the actual probability of heads. However, as we have learned, 10 coin-flips will produce such extreme frequencies approximately 10% of the time. More coin-flips are required to estimate the actual probability.

Lets see what happens when we flip the coin 1000 times. After each flip, we will record the total frequency of heads observed in the sequence. Once the coin-flips are completed, we will visualize our output by plotting coin-flip count vs frequency count. Our plot will also include a horizontal line along the actual probability of .5. We'll generate that line by calling `plt.axhline(.5, color='k')`.

### Listing 3.6 Plotting simulated fair coin-flip frequencies

```
np.random.seed(0)
coin_flips = []
frequencies = []
for _ in range(1000):
    coin_flips.append(np.random.randint(0, 2))
    frequencies.append(frequency_heads(coin_flips))

plt.plot(list(range(1000)), frequencies)
plt.axhline(.5, color='k')
plt.xlabel('Number of Coin Flips')
plt.ylabel('Head-Frequency')
plt.show()
```

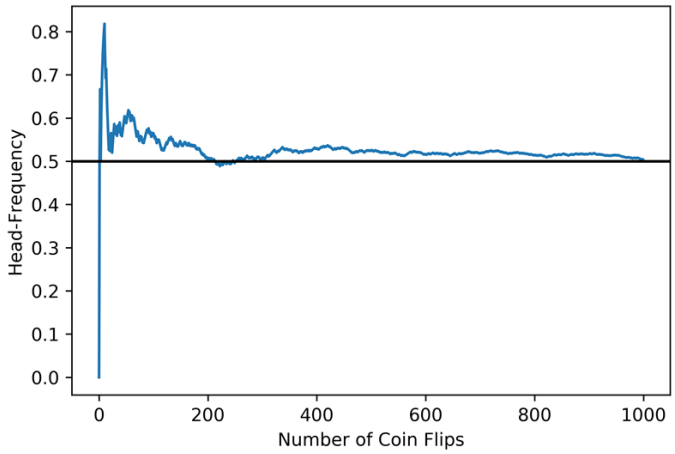


**Figure 3.1 The number of fair coin-flips plotted against the observed head-count frequency. The frequency fluctuates wildly before stabilizing at around 0.5.**

The probability of heads slowly converges to 0.5. Thus, the Law of Large Numbers appears to hold up.

### 3.1.1 Analyzing Biased Coin-Flips

We've simulated a sequence of unbiased coin-flips, but what if we wish to simulate a coin that falls on heads 70% of the time? Well, we can generate that biased output by calling `np.random.binomial(1, .7)`. The binomial method name refers to the generic coin-flip distribution, which mathematicians call the **Binomial distribution**. The method takes as input 2 parameters; the number of coin-flips, and the probability of the desired coin-flip outcome. The method executes the specified number of biased coin-flips. It then counts the instances when the desired outcome was observed. When the number of coin-flips is set to one, the method will return a binary of value of 0 or 1. In our case, a value of 1 represents our desired observation of heads.

**Listing 3.7 Simulating biased coin-flips**

```
np.random.seed(0)
print("Lets flip the biased coin once.")
coin_flip = np.random.binomial(1, .7)
print(f"Biased coin landed on {'heads' if coin_flip == 1 else 'tails'}.")

print("\nLets flip the biased coin 10 times.")
number_coin_flips = 10
head_count = np.random.binomial(number_coin_flips, .7)
print((f"{head_count} heads were observed out of "
       f"{number_coin_flips} biased coin flips"))
```

```
Lets flip the biased coin once.
Biased coin landed on heads.

Lets flip the biased coin 10 times.
6 heads were observed out of 10 biased coin flips
```

Lets generate a sequence of 1000 biased coin-flips. We'll then check if the frequency converges to 0.7.

### Listing 3.8 Computing coin-flip frequency convergence

```
np.random.seed(0)
head_count = np.random.binomial(1000, .7)
frequency = head_count / 1000
print(f"Frequency of Heads is {frequency}")
```

```
Frequency of Heads is 0.697
```

The frequency of heads approximates 0.7, but is not actually equal to 0.7. In fact, the frequency-value is .03 units smaller than the true probability of heads. Suppose we re-compute the frequency of 1000 coin-flips five more times. Will all the frequencies be lower than 0.7? Will certain frequencies hit the exact value of 0.7? We'll find out by executing `np.random.binomial(1000, .7)` over 5 looped iterations.

### Listing 3.9 Re-computing coin-flip frequency convergence

```
np.random.seed(0)
assert np.random.binomial(1000, .7) / 1000 == 0.697   ❶
for i in range(1, 6):
    head_count = np.random.binomial(1000, .7)
    frequency = head_count / 1000
    print(f"Frequency at iteration {i} is {frequency}")
    if frequency == 0.7:
        print("Frequency equals the probability!\n")
```

❶ As a reminder, we seeded our random number generator to maintain consistent output. Thus, our first pseudorandom sampling will return the previously observed frequency of 0.67. We'll skip over this result in order to generate 5 fresh frequencies.

```
Frequency at iteration 1 is 0.69
Frequency at iteration 2 is 0.7
Frequency equals the probability!

Frequency at iteration 3 is 0.707
Frequency at iteration 4 is 0.702
Frequency at iteration 5 is 0.699
```

Just one out the 5 iterations produced a measurement that equaled the real probability. Twice the measured frequency was slightly too low, and twice it was slightly too high. The observed frequency appears to fluctuate over every sampling of 1000 coin-flips. It seems that even though the Law of Large Numbers allows us to approximate the actual probability, some uncertainty still remains. Data science is somewhat messy, and we cannot always be certain of the conclusions we draw from our data. Nevertheless, our uncertainty can be measured and contained using what mathematicians call a confidence interval.

## 3.2 Computing Confidence Intervals Using Histograms and NumPy Arrays

Suppose we're handed a biased coin whose bias we don't know. We flip the coin 1000 times and observe a measured frequency of 0.709. We know the measured frequency approximates the actual probability, but by how much? More precisely, how confident are we that the actual probability falls within an interval of 0.691 to 0.709? Or that it falls between 0.609 and 0.709? Or between 0.4 and 0.8? Or between 0.0 and 1.0? Well, in the latter case, we are 100% confident that the probability falls within the all-encompassing interval of 0.0 to 1.0. As for the other intervals, we cannot compute their confidence without carrying out additional sampling.

We've previously sampled our coin over 5 iterations of 1000 coin-flips each. The sampling produced some fluctuations in the frequency. Lets explore these fluctuations by increasing our frequency count from 5 to 500. One way to execute this supplementary sampling is to run `[np.random.binomial(1000, .7) for _ in range(500)]`.

**Listing 3.10 Computing frequencies with 500 flips-per-sample**

```
np.random.seed(0)
head_count_list = [np.random.binomial(1000, .7) for _ in range(500)
```

However, we can more efficiently sample over 500 iterations by running `np.random.binomial(coin_flip_count, 0.7, size=500)`. The optional size parameter allows us to execute `np.random.binomial(coin_flip_count, 0.7)` 500 times while leveraging NumPy's internal optimizations.

**Listing 3.11 Optimizing coin-flip frequency computation**

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 500)
```

The output is not a Python list, but a NumPy array data-structure. As previously noted, NumPy arrays can more efficiently store numeric data. The actual numeric quantities stored with both `head_count_array` and `head_count_list` remain the same. We'll prove this by converting the array into a list using the `head_count_array.tolist()` method.

**Listing 3.12 Coverting a NumPy array to a Python list**

```
assert head_count_array.tolist() == head_count_list
```

Conversely, we can also convert our Python list into a value-equivalent NumPy array by calling `np.array(head_count_list)`. The equality between the converted array and `head_count_array` can be confirmed using the `np.array_equal` method.

**Listing 3.13 Coverting a Python list to a NumPy array**

```
new_array = np.array(head_count_list)
assert np.array_equal(new_array, head_count_array) == True
```

Why should we prefer to use a NumPy array over a standard Python list? Well, besides the aforementioned memory optimizations and analysis speed-ups, NumPy arrays offer certain interface improvements that make it easier for to implement clean code. One such improvement involves multiplication and division; dividing a NumPy array by some number will automatically divide all array elements by that number. The procedure will create a new array whose elements are properly divided. Thus, executing `head_count_array / 1000` will automatically transform our head-counts into frequencies associated with each coin-flip sampling. By contrast, computing frequencies in `head_count_list` requires that we either iterate over all elements in the list, or leverage Python's convoluted built-in `map` function.

**SIDEBAR** **Useful NumPy method-calls for running random simulations**

- `np.random.randint(x, y)`:
  Returns an random integer between `x` and `y-1`, inclusive.
- `np.random.binomial(1, p)`:
  Returns a single random value equal to 0 or 1. The probability that the value equals 1 is `p`.
- `np.random.binomial(x, p)`:
  Runs `x` instances of `np.random.binomial(1, p)` and returns the summed result. The returned value represents the number of non-zero observations across `x` samples.
- `np.random.binomial(x, p, size=y)`:
  Returns an array of `y` elements. Each array element is equal to a random output of `np.random.binomial(x, p)`.
- `np.random.binomial(x, p, size=y) / x`:
  Returns an array of `y` elements. Each element represents the frequency of non-zero observations across `x` samples.

**Listing 3.14 Computing frequencies using NumPy**

```
frequency_array = head_count_array / 1000
assert frequency_array.tolist() == [head_count / 1000
                                    for head_count in head_count_list]
assert frequency_array.tolist() == list(map(lambda x: x / 1000,
                                        head_count_list))
```

We've converted our head-count array into a frequency array using a simple division operation. Lets explore the contents of `frequency_array` in greater detail. We'll start by outputting the

first 20 sampled frequencies using the same : index-slicing delimiter that is utilized by Python lists. Please note that unlike a printed list, the NumPy array will not contain commas in its output.

### Listing 3.15 Printing a NumPy frequency array

```
print(frequency_array[:20])
```

```
[ 0.697  0.69   0.7    0.707  0.702  0.699  0.723  0.67   0.702  0.713
  0.721  0.689  0.711  0.697  0.717  0.691  0.731  0.697  0.722  0.728]
```

The sampled frequencies fluctuate from 0.69 to approximately 0.731. Of course, we have 480 additional frequencies to choose from within `frequency_array`. Lets extract our minimum and maximum frequencies values by calling the `frequency_array.min()` and `frequency_array.max()` array methods.

### Listing 3.16 Finding the largest and smallest frequency values

```
min_freq = frequency_array.min()
max_freq = frequency_array.max()
print(f"Minimum frequency observed: {min_freq}")
print(f"Maximum frequency observed: {max_freq}")
print(f"Difference across frequency range: {max_freq - min_freq}")
```

```
Minimum frequency observed: 0.656
Maximum frequency observed: 0.733
Difference across frequency range: 0.07699999999999996
```

Somewhere between the frequency range of 0.656 to 0.733 lies the true probability of heads. That interval span is noticeably large, with a more than 7% difference between the largest and smallest sampled frequencies. How can we rationally narrow the frequency range? Plotting the data could help. We'll plot all unique frequencies against their occurrence counts.

### Listing 3.17 Plotting measured frequencies

```
frequency_counts = defaultdict(int)
for frequency in frequency_array:
    frequency_counts[frequency] += 1

frequencies = list(frequency_counts.keys())
counts = [frequency_counts[freq] for freq in frequencies]
plt.scatter(frequencies, counts)
plt.xlabel('Frequency')
plt.ylabel('Count')
plt.show()
```

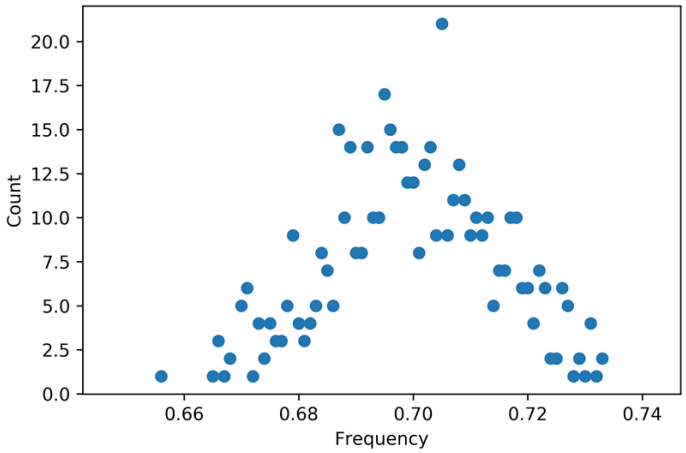


**Figure 3.2 A scatter-plot of 500 head-count frequencies plotted against the frequency counts. The frequencies are centered around 0.7. Certain proximate frequencies appear as overlapping dots within the plot.**

The visualization is informative; frequencies close to 0.7 occur more commonly than frequencies located further away. However, our plot is also somewhat flawed, since values that are very close get counted separately. These nearly identical frequencies appear as overlapping dots within the plot. Perhaps instead of treating them as individual points, we should group such proximate frequencies together.

### 3.2.1 Binning Similar Points in Histogram Plots

Lets try a more nuanced visualization by binning together frequencies that are in close vicinity of each other. We'll sub-divide our frequency range into N equally spaced bins, and then place all frequency values into one of those bins. By definition, the values in any given bin will be at most 1/N units apart. Afterwards, we'll count the total values in each bin, and visualize the counts using a plot.

The binned-based plot we just described is called a **histogram**. Histograms are easy to display in Matplotlib using the `plt.hist` method. The method takes as input the sequence of values to be binned, as well an optional `bins` parameter. That parameter specifies the number of bins used to group the data. Thus, calling `plt.hist(frequency_array, bins='77')` will split our data across 77 bins, each covering a width of .01 units. Also, we can optionally pass in `bin=auto`, and Matplotlib will select an appropriate bin-width using a widely accepted optimization technique (the details of which are beyond the scope of this book). Lets plot a histogram while optimizing bin-width by calling `plt.hist(frequency_array, bins='auto')`.

**NOTE** Within the code below, we also include an `edgecolor='black'` parameter. This helps us visually distinguish the boundaries between bins by coloring the bin-edges in black.

### Listing 3.18 Plotting a frequency histogram using `plt.hist`

```
plt.hist(frequency_array, bins='auto', edgecolor='black')
plt.xlabel('Binned Frequency')
plt.ylabel('Count')
plt.show()
```

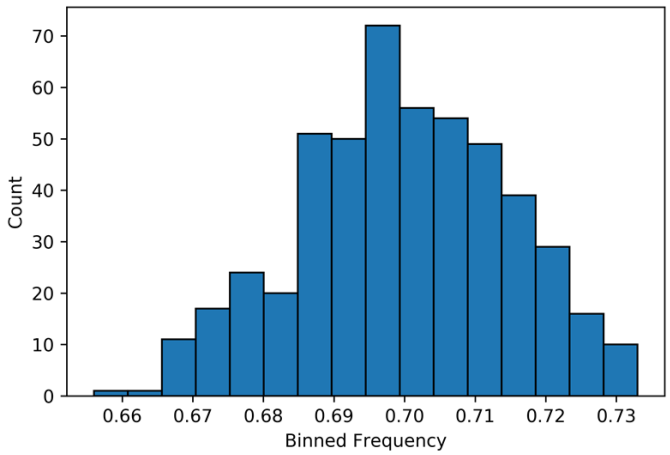


**Figure 3.3 A histogram of 500 binned frequencies plotted against the number of elements in each bin. The bin with the most elements is centered around a frequency of 0.7.**

In our plotted histogram, the bin with the highest frequency count appears somewhere between 0.69 and 0.70. This bin rises noticeably higher than the dozen-or-so other bins in histogram. The total count of bins is not yet known. However, we can obtain that information using `counts`, which is a NumPy array returned by `plt.hist`. The array holds the y-axis frequency counts for each binned group. We'll now call `plt.hist` to return `counts`. Afterwards, we will output `counts.size` to find the total number of binned groups.

### Listing 3.19 Counting bins in a plotted histogram

```
counts, _, _ = plt.hist(frequency_array, bins='auto',   ❶
                        edgecolor='black')

print(f"Number of Bins: {counts.size}")
```

❶ `counts` is one of three variables returned by `plt.hist`. The other variables are discussed later in the section.

```
Number of Bins: 16
```

There are 16 bins within in the histogram. How wide is each bin? One way to find out is to divide the total frequency range by 16. However, there's a simpler way to extract the bin-width. We just need to leverage the `bin_edges` array, which is the second variable returned by `plt.hist`. This array holds the x-axis positions of the vertical bin-edges in the plot. Thus, the difference between any 2 consecutive edge positions will equal the bin-width.

**Listing 3.20 Finding the width of bins within a histogram**

```
counts, bin_edges, _ = plt.hist(frequency_array, bins='auto',
                                edgecolor='black')

bin_width = bin_edges[1] - bin_edges[0]
assert bin_width == (max_freq - min_freq) / counts.size
print(f"Bin width: {bin_width}")
```

```
Bin width: 0.004812499999999997
```

**NOTE** The size of `bin_edges` is always one greater than the size of the `counts`. Why is that the case? Imagine if we only had one rectangular bin; it would be bounded by 2 vertical lines. Adding an additional bin would also increase the boundary size by one. If we extrapolate that logic to *N* bins, then we'd expect to see *N + 1* boundary lines.

The `bin_edges` array can be used in tandem with `counts` to output the element-count and coverage-range for any specified bin. We'll leverage these arrays to define an `output_bin_coverage` function. The function will print count and coverage values for any bin at position `i`.

**Listing 3.21 Getting a bin's frequency and size**

```
def output_bin_coverage(i):
    count = int(counts[i])   ❶
    range_start, range_end = bin_edges[i], bin_edges[i+1]   ❷
    range_string = f"{range_start} - {range_end}"
    print((f"The bin for frequency range {range_string} contains "
           f" {count} element{'' if count == 1 else 's'}"))

output_bin_coverage(0)
output_bin_coverage(5)
```

❶ A bin at position `i` contains `counts[i]` frequencies.

❷ A bin at position `i` covers a frequency range of `bin_edges_[i]` through `bin_edges[i+1]`.

```
The bin for frequency range 0.656 - 0.6608125 contains 1 element
The bin for frequency range 0.6800625 - 0.684875 contains 20 elements
```

Let's compute the count and frequency range for the highest peak within our histogram. For this, we'll need the index of `counts.max()`. Conveniently, NumPy arrays have a built-in `argmax` method, which returns the index of the maximum value within a given array.

**Listing 3.22 Finding the index of an array's maximum value**

```
assert counts[counts.argmax()] == counts.max()
```

Thus, calling `output_bin_coverage(counts.argmax())` should provide us with the output

we've requested.

**Listing 3.23 Using `argmax` to return a histogram's peak**

```
output_bin_coverage(counts.argmax())
```

```
The bin for frequency range 0.6945 - 0.6993125 contains  72 elements
```

### 3.2.2 Deriving Probabilities from Histograms

The most occupied bin within the histogram contains 72 elements. That bin covers a frequency range of approximately 0.694 - 0.699. At this point we should ask ourselves; does the actual probability of heads fall within that range? Well obviously it doesn't, since we know the true probability equals 0.7. However, let's imagine that we lacked this information. What would we do then? One option would be to calculate the likelihood that a randomly measured frequency falls within 0.694 - 0.699. If that likelihood were 1.0, than 100% of measured frequencies would be covered by the range. These measured frequencies would occasionally include the actual probability of heads. Therefore, we would be 100% confident that our true probability lies somewhere between 0.694 and 0.699. Even if the likelihood were lower, at 95%, we would still be fairly confident that the range enclosed our true probability value.

How should we calculate the likelihood? Earlier, we showed that the likelihood of an interval equals its area under a curve, but only when the total plotted area sums up to 1.0. Unfortunately, the area beneath our plotted histogram does not sum to 1.0. The histogram must be modified prior to our likelihood computation. This can be done by passing `density=True` into `plt.hist`. The passed parameter will maintain the histogram's shape while forcing its summed area to equal 1.0.

**Listing 3.24 Plotting a histogram's relative likelihoods**

```
likelihoods, bin_edges, _ = plt.hist(frequency_array, bins='auto', edgecolor='black', density=True)
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

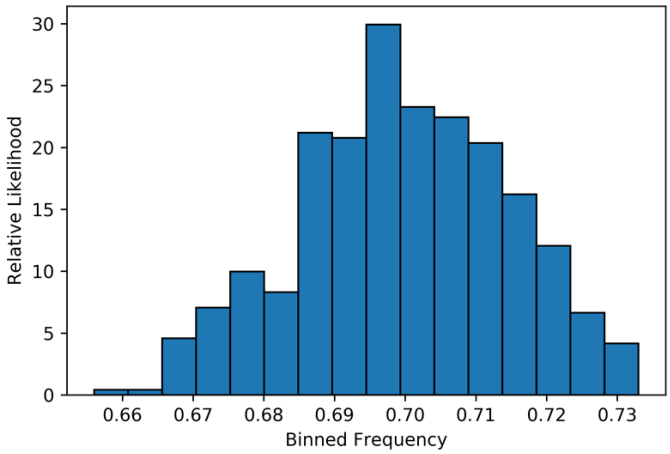


**Figure 3.4 A histogram of 500 binned frequencies plotted against their associated relative likelihoods. The area of the histogram sums to 1.0. That area can be computed by summing over the rectangular areas of each bin.**

In our new histogram, the counts have been replaced by relative likelihoods, which are stored within the `likelihoods` array. As mentioned previously, *relative likelihood* is a term applied to the y-values of a plot whose area sums to 1.0. Of course, the area beneath our histogram now sums to 1.0. We can compute that area by summing the area of each bin. The rectangular area of each bin is equal to its vertical likelihood-value multiplied by `bin_width`. Hence, the area beneath the histogram is equal to the summed likelihoods multiplied by `bin_width`. We can calculate the summed likelihoods by calling `likelihoods.sum()`. Consequently, the area equals `likelihoods.sum() * bin_width`, which equals 1.0.

### Listing 3.25 Computing the total area under a histogram

```
assert likelihoods.sum() * bin_width == 1.0
```

The histogram's total area sums to 1.0. Thus, the area beneath the histogram's peak is now a probability. As previously discussed, this is the probability of a randomly sampled frequency falling with the 0.694 - 0.699 interval range. We can compute that probability by calculating the area of the bin positioned at `likelihoods.argmax()`.

### Listing 3.26 Computing the probability of the peak frequencies

```
index = likelihoods.argmax()
area = likelihoods[index] * bin_width
range_start, range_end = bin_edges[index], bin_edges[index+1]
range_string = f"{range_start} - {range_end}"
print(f"Sampled frequency falls within interval {range_string} with probability {area}")
```

```
Sampled frequency falls within interval 0.6945 - 0.6993125 with probability 0.144
```

The probability is approximately 14%. That value is low, but we can raise it by expanding our interval range beyond one bin. We'll stretch the range to cover neighboring bins at indices

`likelihoods.argmax() - 1` and `likelihoods.argmax() + 1`.

### Listing 3.27 Raising the probability of a frequency range

```
peak_index = likelihoods.argmax()
start_index, end_index = (peak_index - 1, peak_index + 1)
area = likelihoods[start_index: end_index + 1].sum() * bin_width
range_start, range_end = bin_edges[start_index], bin_edges[end_index]
range_string = f"{range_start} - {range_end}"
print(f"Sampled frequency falls within interval {range_string} with probability {area}")
```

```
Sampled frequency falls within interval 0.6896875 - 0.6993125 with probability 0.35600000000000004
```

The three bins cover a frequency range of approximately 0.698 - 0.699. Their associated probability is 0.356. Thus, the three bins represent what statisticians call a 35.6% **confidence interval**. Basically, this means we are 35.6% confident that our true probability falls within the three-bin range. The calculated confidence percentage is still too low. Ideally, we'd prefer a confidence interval of 95% or more. We'll reach that confidence interval by iteratively expanding our left-most bin and right-most bin until the interval area stretches past 0.95.

### Listing 3.28 Computing a high confidence interval

```
def compute_high_confidence_interval(likelihoods, bin_width):
    peak_index = likelihoods.argmax()
    area = likelihoods[peak_index] * bin_width
    start_index, end_index = peak_index, peak_index
    while area < 0.95:
        if start_index > 0:
            start_index -= 1
        if end_index < likelihoods.size - 1:
            end_index += 1

        area = likelihoods[start_index: end_index + 1].sum() * bin_width

    range_start, range_end = bin_edges[start_index], bin_edges[end_index]
    range_string = f"{range_start:.6f} - {range_end:.6f}"
    print((f"The frequency range {range_string} represents a "
           f"{100 * area:.2f}% confidence interval"))
    return start_index, end_index

compute_high_confidence_interval(likelihoods, bin_width)
```

```
The frequency range 0.665625 - 0.723375 represents a 97.60% confidence interval
```

The frequency range of roughly 0.665 - 0.723 represents a 97.6% confidence interval. Thus, a sampled sequence of 1000 biased coin-flips should fall within that range 97.6% of the time. From our analysis, we're fairly confident that the true probability lies somewhere between 0.665 and 0.725. However, that actual probability range is too ambiguous to be informative. Is the true probability equal to 0.67? Or 0.69? Or 0.72? Based on the data, we cannot say for sure. We'll need to somehow narrow down that range in order to obtain a more certain probability estimation.

### 3.2.3 Shrinking the Range of a High Confidence Interval

How can we taper down our range while still maintaining a 95% confidence interval? Perhaps we should try elevating the frequency count from 500 to something noticeably larger. Previously, we've sampled 500 frequencies, where each frequency represented 1000 biased coin flips. Instead, lets sample 100,000 frequencies while keeping the flip-count constant at 1000.

**Listing 3.29 Sampling 100,000 frequencies**

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 100000)
frequency_array = head_count_array / 1000
assert frequency_array.size == 100000
```

We will re-compute the histogram on the updated `frequency_array`, which now holds 200-fold more frequency elements. Afterwards, we'll visualize that histogram while also searching for a high confidence interval. Lets incorporate the confidence interval into our visualization by coloring the histogram bars within its range. The histogram bars can be visually modified by relying on `patches`, which is the third variable returned by `plt.hist`. The graphical details of each bin at index `i` are accessible through the `patches[i]`. If we wish to color the ith bin yellow, we can simply call `patches[i].set_facecolor('yellow')`. In this manner, we can highlight all the specified histogram bars that fall within the updated interval range.

**Listing 3.30 Coloring histogram bars over an interval**

```
likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
     patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()
```

```
The frequency range 0.670429 - 0.727000 represents a 95.03% confidence interval
```

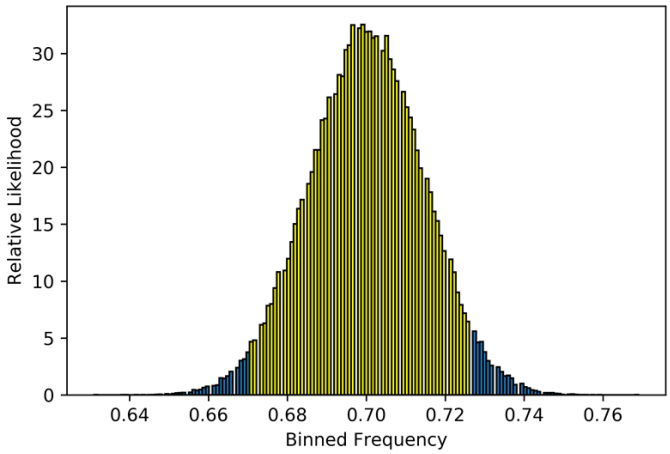


**Figure 3.5 A histogram of 100,000 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval, which represents 95% of the histogram's area. That interval covers a frequency range of roughly 0.670 - 0.727.**

The recomputed histogram resembles a symmetric bell-shaped curve. Many of its bars have been highlighted using the `set_facecolor` method. The highlighted bars represent a 95% confidence interval. The interval covers a frequency range of roughly 0.670 - 0.727. This new frequency range is nearly identical to the one we saw before. Raising the sampling size from 500 to 100,000 appears to have done little to reduce the range. How disappointing!

What should we do to narrow the range? Well, we've already tried raising the frequency count, and that didn't work. Perhaps we should've also raised the number of coin-flips per frequency-sample. Previously that value was 1000. Lets increase it 50-fold to 50,000 coin-flips per sampled frequency. We'll keep the frequency sample-size steady at 100,000. Thus, our total sampling will amount to 5 billion flipped coins.

## Listing 3.31 Sampling 5 billion flipped coins

```
np.random.seed(0)
head_count_array = np.random.binomial(50000, 0.7, 100000)
frequency_array = head_count_array / 50000

likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
     patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()
```

```
The frequency range 0.695651 - 0.703708 represents a 95.33% confidence interval
```

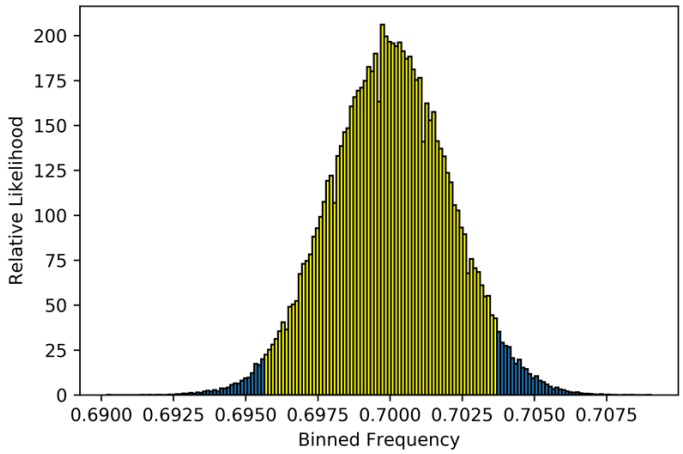


**Figure 3.6 A histogram of 100,000 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval, which represents 95% of the histogram's area. That interval covers a frequency range of roughly 0.695 - 0.703.**

The new 95.33% confidence interval covers a frequency range of roughly 0.695 - 0.703. If we round the range to 2 decimal places, it will equal 0.70 - 0.70. We are thus exceedingly confident that our true probability is approximately 0.70. By raising the coin-flips per sample, we've successfully narrowed the range of our 95% confidence interval.

On a separate note, the shape of our updated histogram once again resembled a bell-shaped curve. That bell-shaped probability distribution is called a **Gaussian distribution**. It is more commonly known as the **Bell curve**, or the **Normal distribution**. The Normal distribution is incredibly important to probability theory and statistics. The importance arises from a mathematical law known as the **Central Limit Theorem**. According to the theorem, sampled frequency-distributions will take the shape of a Normal distribution, when the number of samples is large. Furthermore, the theorem predicts a narrowing of likely frequencies as the size of each frequency-sample goes up. This is perfectly consistent with our observations, which are summarized below.

1. Initially, we sampled 1000 coin-flips 500 times.
2. Each sequence of 1000 coin-flips was converted to a frequency.
3. We plotted the histogram of 500 frequencies representing 50,000 total coin-flips.
4. The histogram shape was not symmetric. It peaked at approximately 0.7.
5. We raised the frequency count from 500 to 100,000.
6. We plotted the histogram of 100,000 frequencies representing 1 million total coin-flips.
7. The new histogram's shape resembled a normal curve. It continued peaking at 0.7.
8. We summed the rectangular area of bins around the peak. We stopped once the added bins covered 95% of the area under the histogram.
9. These bins represented a frequency-range of approximately 0.670 - 0.723.
10. We raised the coin-flips per sample from 1000 to 50,000.
11. We plotted the histogram of 100,000 frequencies representing 5 billion total coin-flips.

12. The updated histogram's shape continued to resemble a normal curve.
13. We recomputed the range covering 95% of the histogram's area.
14. The range width had shrunk to approximately 0.695 - 0.703.
15. Thus, when we raised our per-frequency flip-count, the range of likely frequencies began to narrow at around 0.7.

### 3.2.4 Computing Histograms in NumPy

Calling the `plt.hist` method will automatically generate a histogram plot. Can we obtain the histogram likelihoods and bin-edges without creating a plot? Yes; because `plt.hist` does not actually rely on Matplotlib to compute the histogram. Instead, it passes certain histogram parameters into Numpy's `np.histogram` method. This method takes as input all parameters that don't relate to histogram visualization. These include `frequency_arrays`, `bins='auto'`, and `density=True`. The method outputs the 2 variables not associated with plot manipulation: `likelihoods`, and `bin_edges`. Therefore, we can run `compute_high_confidence_interval` while bypassing visualization. We simply need to call `np.histogram`.

**Listing 3.32 Computing a histogram using `np.histogram`.**

```
np.random.seed(0)

likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval()
```

```
The frequency range 0.695651 - 0.703708 represents a 95.33% confidence interval
```

**SIDEBAR** **Useful histogram inputs and outputs.**

- `plt.hist(data, num_bins=10)`:
  Plots a histogram in which the elements of `data` are distributed across 10 equally spaced bins.
- `plt.hist(data, num_bins=auto)`:
  Plots a histogram whose bin-count is determined automatically, based on the data distribution. `auto` is the default setting of `num_bins`.
- `plt.hist(data, edges=black)`:
  In the plotted histogram, the edges of each bin get marked by black vertical lines.
- `counts, _, _ = plt.hist(data)`:
  The `counts` array is the first of the 3 variables returned by `plt.hist`. It holds the count of elements contained within each bin. These counts appear in y-axis of the histogram plot.
- `_, bin_edges, _ = plt.hist(data)`:
  The `bin_edges` array is the second of the 3 variables returned by `plt.hist`. It holds the x-axis positions of the vertical bin edges in the plot. Subtracting `bin_edges[i]` from `bin_edges[i + 1]` will return the width of every bin. Multiplying the width by `counts[i]` will return the area of the rectangular bin at position `i`.
- `likelihoods, _, _ = plt.hist(data, density=True)`:
  The binned counts are transformed into likelihoods, so that area beneath the histogram sums to 1.0. Thus, the histogram is transformed into a probability distribution. Multiplying the bin-width by `likelihoods[i]` returns the probability of a random outcome falling within a range of `bin_edges[i]` - `bin_edges[i +1]`.
- `_, _, patches = plt.hist(data)`: The `patches` list is the third of the 3 variables returned by `plt.hist`. The graphical settings of each bin at index i are stored in `patches[i]`. Calling `patches[i].set_facecolor('yellow')` will change the color of the histogram bin at position `i`.

  `likelihoods, bin_edges = np.histogram(data, density=True)`:
  Returns the histogram likelihoods and bin-edges without actually plotting the results.

## 3.3 Leveraging Confidence Intervals to Analyze a Biased Deck of Cards

Suppose you're shown a biased deck of cards. The deck holds 52 cards total. Each card is either red or black. How many red cards are present in the deck? You can answer this by counting all the red cards one-by-one, but that would be too easy. Lets add a constraint to make the problem more interesting. You are only allowed to see the first card in the deck! None of the other cards are accessible. If you wish to see a new card, you must first reshuffle the deck. You're permitted to reshuffle as many times as you like, and to view the top card after each shuffle.

Given these constraints, we must solve the problem using random sampling. Lets begin by modelling a deck with a hidden quantity of red cards. The total size of that deck is 52. The number of red cards in the deck is some unknown integer between zero and 52. Lets generate that integer using the `np.random.randint` method. We'll keep the value of our random `red_card_count` hidden from view until we've found a solution using sampling.

**Listing 3.33 Generating a random red card count**

```
np.random.seed(0)
total_cards = 52
red_card_count = np.random.randint(0, total_cards + 1)
```

Now lets assign a value to `black_card_count`. We'll leverage the fact that `red_card_count` and `black_card_count` must sum to 52 cards total. We'll also maintain bias by ensuring that the two counts are not equal.

**Listing 3.34 Generating a black card count**

```
black_card_count = total_cards - red_card_count
assert black_card_count != red_card_count
```

During the modeling phase, we'll shuffle the deck and flip over the first card. What is the probability the card will be red? Well, a red card represents one of 2 possible outcomes; red or black. These outcomes can be characterized by the sample space `{'red_card', 'black_card'}`, but only when the 2 outcomes are equally likely. However, our card deck is biased, and the outcomes are not equally likely. Their likelihoods are weighted by `red_card_count` and `black_card_count`. A weighted sample space dictionary is therefore required. The values of the dictionary will equal `red_card_count` and `black_card_count`. We'll label the associated keys as `'red_card'` and `'black_card'`. Passing `weighted_sample_space` into `compute_event_probability` will allow us to compute the probability of drawing a red card.

**Listing 3.35 Computing card probabilities using a sample space**

```
weighted_sample_space = {'red_card': red_card_count,
                         'black_card': black_card_count}
prob_red = compute_event_probability(lambda x: x == 'red_card',
                                     weighted_sample_space)
```

As a reminder, the `compute_event_probability` function divides the `red_card_count` by the sum of `red_card_count` and `black_card_count`. This division produces our probability. The sum of `red_card_count` and `black_card_count` equals `total_cards`. Therefore, the probability of drawing a red card is equal to `red_card_count` divided by `total_cards`.

**Listing 3.36 Computing card probabilities using division**

```
assert prob_red == red_card_count / total_cards
```

How should we utilize `prob_red` to model a flipped-over first card? Well, the card-flip will produce one of 2 possible outputs; red or black. Conceptually, these 2 outputs are no different from a coin-flip. We've simply replaced heads and tails with red and black. Therefore, we can model the flipped card using the Binomial distribution. Calling `np.random.binomial(1, prob_red)` will return 1 if the first card is red, and 0 otherwise.

**Listing 3.37 Simulating a random card**

```
np.random.seed(0)
color = 'red' if np.random.binomial(1, prob_red) else 'black'
print(f"The first card in the shuffled deck is {color}")
```

```
The first card in the shuffled deck is red
```

We'll proceed to shuffle the deck 10 times, and flip over the first card after each shuffle.

**Listing 3.38 Simulating 10 random cards**

```
np.random.seed(0)
red_count = np.random.binomial(10, prob_red)
print(f"In {red_count} of out 10 shuffles, a red card came up first.")
```

```
In 8 of out 10 shuffles, a red card came up first.
```

A red card appeared at the top of the deck in 8 out of 10 random shuffles. Does this mean that 80% of the cards are red? Of course not. We've previously shown how such outcomes are common when the sampling size is low. Instead of shuffling the deck 10 times, lets shuffle it 50,000 times. Afterwards, lets compute the frequency and then re-do the shuffling procedure another 100,000 times. We'll execute these steps by calling `np.random.binomial(50000, prob_red, 100000)` and dividing by 50,000. The resulting frequency array can transformed

into a histogram. That histogram will allow us to compute a 95% confidence interval for flipping over a red card. We'll compute the confidence interval by expanding the range of bins around the histogram's peak until that range covers 95% of the histogram's area.

### Listing 3.39 Computing card-probability confidence intervals

```
np.random.seed(0)
red_card_count_array = np.random.binomial(50000, prob_red, 100000)   ❶
frequency_array = red_card_count_array / 50000   ❷

likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)   ❸
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,   ❹
                                                          bin_width)
```

❶ Count the observed red-cards out of 50,000 shuffles. Repeat 100,000 times.

❷ Convert 100,000 red-counts into 100,000 frequencies.

❸ Compute the frequency histogram.

❹ Compute the 95% confidence interval for the histogram.

```
The frequency range 0.842771 - 0.849139 represents a 95.45% confidence interval
```

We are very confident that `prob_red` lies some between 0.842771 and 0.849139. We also know that `prob_red` equals `red_card_count / total_cards`, and therefore `red_card_count` equals `prob_red * total_cards`. Thus, we are highly confident that `red_card_count` lies between `0.842771 * total_cards` and `0.849139 * total_cards`. Let's compute the likely range of `red_card_count`. We'll round the end-points of the range to nearest integers because `red_card_count` corresponds to an integer value.

### Listing 3.40 Estimating the red card count

```
range_start = round(0.842771 * total_cards)
range_end = round(0.849139 * total_cards)
print(f"The number of red cards in the deck is between {range_start} and {range_end}")
```

```
The number of red cards in the deck is between 44 and 44
```

We are very confident that there are 44 red cards in the deck. Lets check if our solution is correct.

### Listing 3.41 Validating the red card count

```
if red_card_count == 44:
    print('We are correct! There are 44 red cards in the deck')
else:
    print('Oops! Our sampling estimation was wrong.')
```

```
We are correct! There are 44 red cards in the deck
```

There are indeed 44 red cards in the deck. We were able to determine this without manually counting all the cards. Our use of random card-shuffle sampling and confidence interval calculations proved sufficient to uncover the solution.

## 3.4 Using Permutations to Shuffle Cards

Card-shuffling requires us to randomly re-order the elements of a card-deck. That random re-ordering can be carried out using the `np.random.shuffle` method. The method takes as input an ordered array or list, and shuffles its elements in place. The code below will randomly shuffle a deck of cards containing 2 red cards (represented by ones) and 2 black cards (represented by zeros).

**Listing 3.42 Shuffling a 4-card deck**

```
np.random.seed(0)
card_deck = [1, 1, 0, 0]
np.random.shuffle(card_deck)
print(card_deck)
```

```
[0, 0, 1, 1]
```

The shuffle method has re-arranged the elements within `card_deck`. If we prefer to carry out the shuffle while retaining a copy of the original unshuffled deck, we can do so using `np.random.permutation`. The method returns a NumPy array containing a random ordering of cards. Meanwhile, the elements of the original inputted deck remain unchanged.

**Listing 3.43 Returning a copy of the shuffled deck**

```
np.random.seed(0)
unshuffled_deck = [1, 1, 0, 0]
shuffled_deck = np.random.permutation(unshuffled_deck)
assert unshuffled_deck == [1, 1, 0, 0]
print(shuffled_deck)
```

```
[0 0 1 1]
```

The random ordering of elements returned by `np.random.permutation` is mathematically called a **permutation**. Random permutations will vary from the original ordering most of the time. On rare occasions, they might equal the original, unshuffled permutation. What is the probability that a shuffled permuation will exactly equal `unshuffled_deck`?

We can of course find out through sampling. However, the four-element deck is small enough for us to analyze the problem using sample spaces. Composing the sample space will require us to cycle through all possible permutations of the deck. We can do so using the `itertools.permutations` method. Calling `itertools.permutations(unshuffled_deck)` will return an iterable over every possible permutation of the deck. Lets use the method to output

the first 3 permutations. Please note that these 3 permutions will be printed as Python tuples, not as arrays or lists. Tuples, unlike arrays or lists, cannot be modified in place. They are represented using parentheses.

**Listing 3.44 Iterating over card permutations**

```
import itertools
for permutation in list(itertools.permutations(unshuffled_deck))[:3]:
    print(permutation)
```

```
(1, 1, 0, 0)
(1, 1, 0, 0)
(1, 0, 1, 0)
```

The first 2 generated permutations are identical to each other. Why is that the case? Well, the first permutation is just the original `unshuffled_deck` with no rearranged elements. Meanwhile, the second permutation was generated by swapping the third and fourth elements of the first permutation. However, both those elements were zeroes, so the swap did not impact the list. We can confirm the swap actually took place by examining the first three permutations of `[0, 1, 2, 3]`.

**Listing 3.45 Monitoring permutation swaps**

```
for permutation in list(itertools.permutations([0, 1, 2, 3]))[:3]:
    print(permutation)
```

```
(0, 1, 2, 3)
(0, 1, 3, 2)
(0, 2, 1, 3)
```

Certain permutations of the 4-card deck occur more than once. Thus, its reasonable to assume that certain permutations might occur more frequently than others. Let's find out if this is the case by storing the permutation counts within a `weighted_sample_space` dictionary.

**Listing 3.46 Computing permutation counts**

```
weighted_sample_space = defaultdict(int)
for permutation in itertools.permutations(unshuffled_deck):
    weighted_sample_space[permutation] += 1

for permutation, count in weighted_sample_space.items():
    print(f"Permutation {permutation} occurs {count} times")
```

```
Permutation (1, 1, 0, 0) occurs 4 times
Permutation (1, 0, 1, 0) occurs 4 times
Permutation (1, 0, 0, 1) occurs 4 times
Permutation (0, 1, 1, 0) occurs 4 times
Permutation (0, 1, 0, 1) occurs 4 times
Permutation (0, 0, 1, 1) occurs 4 times
```

All the permutations occur with equal frequency. Consequently, all the arrangements of the four cards are equally likely to occur. A weighted sample space is therefore not required. An unweighted sample space equal to `set(itertools.permutations(unshuffled_deck))`

should be sufficient to resolve the problem.

### Listing 3.47 Computing permutation probabilities

```
sample_space = set(itertools.permutations(unshuffled_deck))  ❶
event_condition = lambda x: list(x) == unshuffled_deck  ❷
prob = compute_event_probability(event_condition, sample_space)  ❸
assert prob == 1 / len(sample_space)
print(f"Probability that a shuffle does not alter the deck is {prob}")
```

❶ The unweighted sample space equals the set of all the unique permutations of the deck.

❷ We define a lambda function that takes as input some `x`, and returns `True` if `x` equals our unshuffled deck. This one-line lambda function serves as our event condition.

❸ We compute the probability of observing an event that satisfies our event condition.

```
Probability that a shuffle does not alter the deck is 0.16666666666666666
```

Suppose we are handed a generic `unshuffled_deck` of size N containing N/2 red cards. Mathematically, it can be shown that all the color permutations of the deck will occur with equal likelihood. Thus, we can compute probabilities directly from the unweighted sample space of the deck. Unfortunately, creating this sample space is not feasible for a deck of 52 cards, since its number of possible permutations is astronomically large. In fact, that permutation count is equal to $8.06 * 10^{67}$. This enormous quantity is larger than the count of atoms on Earth. If we attempted to compute a 52-card sample space, then our program would run for many days before eventually running out of memory. However, such a sample space could easily be computed for smaller deck of size 10.

### Listing 3.48 Computing a 10-card sample space

```
red_cards = 5 * [1]
black_cards = 5 * [0]
unshuffled_deck = red_cards + black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
print(f"Sample space for a 10-card deck contains {len(sample_space)} elements")
```

```
Sample space for a 10-card deck contains 252 elements
```

We have been tasked with finding the best strategy for drawing a red card. The 10-card `sample_space` set could prove useful in these efforts. The set allows us to compute the probabilities of various competing strategies directly. We can thus rank our strategies based on their 10-card deck performance and then apply the top-ranking strategies to a 52 card deck.

## 3.5 Summary

- The `np.random.binomial` method can simulate random coin-flips. The method gets its name from the Binomial distribution, which is a generic distribution that captures coin-flip probabilities.
- When a coin is flipped repeatedly, its frequency of heads converges towards the actual probability of heads. However, the final frequency might differ slightly from the actual probability.
- We can visualize the variability of recorded coin-flip frequencies by plotting a histogram. A histogram shows binned counts of observed numeric values. The counts can be transformed into relative likelihoods, so that the area beneath the histogram sums to 1. Effectively, the transformed histogram becomes a probability distribution. The area around the distribution's peak represents a confidence interval. A confidence interval is the likelihood that an unknown probability falls within a certain frequency range. Generally, we prefer a confidence interval that is at 95% or higher.
- The shape of a frequency histogram will resemble a bell-shaped curve when the number of sampled frequencies is high. This curve is commonly referred to as the Bell curve, or the Normal distribution. According to the Central Limit Theorem, the 95% confidence interval associated with the Bell curve will grow more narrow as the size of each frequency sample goes up.
- Simulated card shuffles can be carried out using the `np.random.permutation` method. The method returns a random permutation of the inputted deck of cards. The permutation represents a random ordering of card elements. We can iterate over every possible permutation by calling `itertools.permutations`. Iterating over all the permutations for a 52 card deck is computationally impossible. However, we can easily capture all the permutations of a smaller 10-card deck. These permutations can be used to compute the small deck's sample space.

# 4 Case Study 1 Solution

## 4.1 Overview

Our aim is to play a card-game in which the cards are iteratively flipped until we tell the dealer to stop. Afterwards, one additional card is flipped. If that card is red, we win a dollar. Otherwise, we lose a dollar. Our goal is to discover a strategy that best predicts a red card in the deck. We will do so by:

**A.** Developing multiple strategies for predicting red cards in a randomly shuffled deck.

**B.** Applying each strategy across multiple simulations to compute its probability of success, within a high confidence interval. If these computations prove to be intractable, we will instead focus on those strategies that best perform across a 10-card sample space.

**C.** Returning the simplest strategy associated with the highest probability of success.

> **WARNING** Spoiler alert! The solution to Case Study 1 is about to be revealed. We strongly encourage you to try and solve the problem prior to reading the solution. The original problem statement is available for reference at the beginning of Part 1.

## 4.2 Predicting Red Cards within a Shuffled Deck

We'll start by creating a deck holding 26 red cards and 26 black cards. Black cards are represented by zeroes and red cards are represented by ones.

**Listing 4.1 Modeling a 52-card deck**

```
red_cards = 26 * [1]
black_cards = 26 * [0]
unshuffled_deck = red_cards + black_cards
```

We'll proceed to shuffle the deck.

**Listing 4.2 Shuffling a 52-card deck**

```
np.random.seed(0)
shuffled_deck = np.random.permutation(unshuffled_deck)
```

Now we'll iteratively flip over the cards within the deck, stopping when the next card is more likely to be red. Afterwards, we'll flip over the next card. We will win if that card is red.

How do we decide when we should stop? One simple strategy is to terminate the game when the number of red cards remaining in the deck is greater than the number of black cards remaining in the deck. Lets execute that strategy on the shuffled deck.

**Listing 4.3 Coding a card-game strategy**

```
remaining_red_cards = 26
for i, card in enumerate(shuffled_deck[:-1]):
    remaining_red_cards -= card
    remaining_total_cards = 52 - i
    if remaining_red_cards / float(remaining_total_cards) > 0.5:
        break

print(f"Stopping the game at index {i}.")
final_card = shuffled_deck[i + 1]
color = 'red' if final_card else 0
print(f"The next card in the deck is {'red' if final_card else 'black'}.")
print(f"We have {'won' if final_card else 'lost'}!")
```

```
Stopping the game at index 1.
The next card in the deck is red.
We have won!
```

The strategy yielded a win on our very first try. Lets examine the strategy in more detail. Essentially, we halt when the fraction of remaining red cards is greater than half of the remaining total cards. We can generalize that fraction to equal some arbitrary `min_red_fraction` value. Consequently, we can generalize the strategy to halt when the fraction of remaining red cards is greater than the inputted `min_red_fraction` parameter. The generalized strategy is implemented below, with `min_red_fraction `preset to 0.5.

**Listing 4.4 Generalizing the card-game strategy**

```
np.random.seed(0)
total_cards = 52
total_red_cards = 26
def execute_strategy(min_fraction_red=0.5, shuffled_deck=None,
                     return_index=False):
    if shuffled_deck is None:

        shuffled_deck = np.random.permutation(unshuffled_deck)   ❶
    remaining_red_cards = total_red_cards

    for i, card in enumerate(shuffled_deck[:-1]):
        remaining_red_cards -= card
        fraction_red_cards = remaining_red_cards / float(total_cards - i)
        if fraction_red_cards > min_fraction_red:
            break

    return (i+1, shuffled_deck[i+1]) if return_index else shuffled_deck[i+1]   ❷
```

❶ Shuffle the unshuffled deck if no input deck is provided.

❷ Optionally return the card index along with the final card.

### 4.2.1 Estimating the Probability of Strategy Success

Lets apply our basic strategy to a series of 1000 random shuffles.

**Listing 4.5 Running strategy over 100K shuffles**

```
observations = np.array([execute_strategy() for _ in range(1000)])
```

The total fraction of ones in observations corresponds to the observed fraction of red cards, and therefore to the fraction of wins. We can compute this fraction by summing the ones in observations and dividing by the array size. As an aside, that computation can also be carried out by calling `observations.mean()`.

**Listing 4.6 Computing the frequency of wins**

```
frequency_wins = observations.sum() / 1000.0
assert frequency_wins == observations.mean()
print(f"The frequency of wins is {frequency_wins}")
```

```
The frequency of wins is 0.524000
```

We've won 52.4 percent of total games! Our strategy appears to be working. 524 wins and 476 losses will net us a total profit of $48.

**Listing 4.7 Computing total profit**

```
dollars_won = frequency_wins * 1000
dollars_lost = (1 - frequency_wins) * 1000
total_profit = dollars_won - dollars_lost
print(f"Total profit is ${total_profit}")
```

```
Total profit is $48
```

The strategy worked well for a sample size of 1000 shuffles. We'll now plot the strategy's win-frequency convergence over a series of sample sizes ranging from 1 to 10,000.

## Listing 4.8 Plotting simulated frequencies of wins

```
np.random.seed(0)
def repeat_game(number_repeats):   ❶
    observations = np.array([execute_strategy()
                             for _ in range(number_repeats)])
    return observations.mean()

frequencies = []
for i in range(1, 1000):
    frequencies.append(repeat_game(i))

plt.plot(list(range(1, 1000)), frequencies)
plt.axhline(.5, color='k')
plt.xlabel('Number of Card Shuffles')
plt.ylabel('Win-Frequency')
plt.show()
print(f"The win-frequency for 10,000 shuffles is {frequencies[-1]}")
```

❶ Returns the frequency of wins for specified number of games.

```
The win-frequency for 10,000 shuffles is 0.513514
```

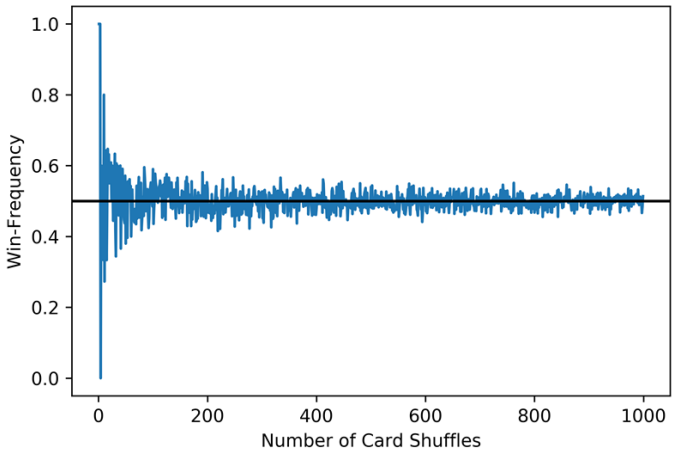


**Figure 4.1 The number of played-games plotted against the observed win-count frequency. The frequencies fluctuate at around a value of 0.5. We cannot tell if probability of winning is above or below 0.5.**

The strategy yields a win-frequency of over 50% when 10,000 card shuffles are sampled. However, the strategy also fluctuates above and below 50% through-out the entire sampling process. How confident are we that the probability of a win is actually greater than 0.5? We can find out using confidence interval analysis.

We'll compute the confidence interval by sampling 10,000 card-shuffles 300 times, for a total of

3 million shuffles. Shuffling an array is a computationally expensive procedure, so the code below will take approximately 40 seconds to run.

**Listing 4.9 Computing the confidence interval for 3 million shuffles**

```
np.random.seed(0)
frequency_array = np.array([repeat_game(10000) for _ in range(300)])

likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods, bin_width)   ❶

for i in range(start_index, end_index):
     patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()
```

❶ As a reminder, we defined the `compute_high_confidence_interval` function in Section Three.

```
The frequency range 0.488687 - 0.508312 represents a 95.67% confidence interval
```

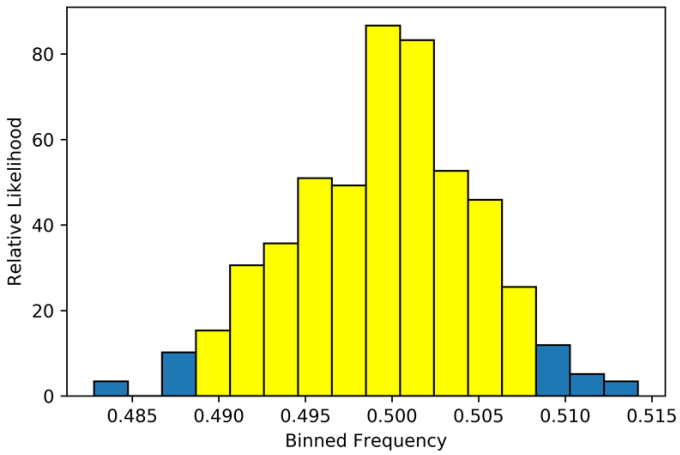


**Figure 4.2 A histogram of 300 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval. That interval covers a frequency range of roughly 0.488 - 0.508.**

We are quite confident that the actual probability lies somewhere between 0.488 and 0.508. However, we lack enough context to determine if the probability is above 0.5 or below 0.5. This is a problem. Even a minor misinterpretation of true probability could cause us to lose money.

Imagine that the true probability is 0.5001. If we apply our strategy to one billion shuffles, we should expect to win 200,000 dollars. Now suppose that we were wrong, and the actual probability is 0.4999. In this scenario, we will lose 200K. A simple estimation error over the fourth decimal space could cost us hundreds of thousands of dollars.

We must be absolutely certain that the true probability lies above 0.5. What we need is a more narrow 95% confidence interval. One way to achieve this confidence interval is to increase the sample size, at the expense of running time. The code below will sample 50,000 shuffles over 3,000 iterations. It will take approximately one hour to run.

**WARNING** The following code will take an hour to run.

**Listing 4.10 Computing the confidence interval for 150 million shuffles**

```
np.random.seed(0)

frequency_array = np.array([repeat_game(50000) for _ in range(3000)])
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] - bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)
```

```
The frequency range 0.495016 - 0.503462 represents a 95.03% confidence interval
```

We've executed our sampling. Unfortunately, the new confidence interval still does not discern whether the true probability lies above 0.5. So what should we do? Raising the number of samples is not computationally feasible (unless we're willing to let the simulation run for a couple of days). Perhaps raising `min_red_fraction` from 0.5 to 0.75 will yield an improvement? Lets update our strategy and go for a long walk as our simulation takes another hour to run.

**WARNING** The following code will take an hour to run.

**Listing 4.11 Computing the confidence interval for an updated strategy**

```
np.random.seed(0)
def repeat_game(number_repeats, min_red_fraction):
    observations = np.array([execute_strategy(min_red_fraction)
                             for _ in range(number_repeats)])
    return observations.mean()

frequency_array = np.array([repeat_game(50000, 0.75) for _ in range(3000)])
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] - bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)
```

```
The frequency range 0.495140 - 0.503995 represents a 95.63% confidence interval
```

Nope! The span of our confidence interval remains unresolved. The interval still covers both profitable and unprofitable probabilities.

We could perhaps gain more insight by applying our strategies to a 10-card deck. The sample space of such a deck can be explored in its entirety. Therefore, we should be left with no doubt about the actual probability of a strategy win.

## 4.3 Optimizing Strategies using the Sample Space for a 10-Card Deck

The code below computes the sample space for a 10-card deck. Afterwards, it applies our basic strategy to that sample space. The final output is the probability that the strategy will yield a win.

**Listing 4.12 Applying a basic strategy to a 10-card deck**

```
total_cards = 10
total_red_cards = int(total_cards / 2)
total_black_cards = total_red_cards
unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
win_condition = lambda x: execute_strategy(shuffled_deck=np.array(x))   ❶
prob_win = compute_event_probability(win_condition, sample_space)   ❷
print(f"Probability of a win is {prob_win}")
```

❶ The event condition where our basic strategy yields a win.

❷ As a reminder, we defined the `compute_event_probability` function in Section One.

```
Probability of a win is 0.5
```

Surprisingly, our basic strategy yields a win only 50% of the time. This is no better than selecting the first card at random! Maybe our `min_red_fraction` parameter was insufficiently low? We can find out by sampling all the 2 decimal `min_red_fraction` values between 0.50 and 1.0. The code below computes the win-probabilities over a range of `min_red_fraction` values. It then returns the minimum and maximum probabilities.

**Listing 4.13 Applying multiple strategies to a 10-card deck**

```
def scan_strategies():
    fractions = [value / 100 for value in range(50, 100)]
    probabilities = []
    for frac in fractions:
        win_condition = lambda x: execute_strategy(frac,
                                                   shuffled_deck=np.array(x))
        probabilities.append(compute_even_probability(win_condition,
                                                      sample_space))
    return probabilities

probabilities = scan_strategies()
print(f"Lowest probability of win is {min(probabilities)}")
print(f"Highest probability of win is {max(probabilities)}")
```

```
Lowest probability of win is 0.5
Highest probability of win is 0.5
```

Both the lowest and highest probabilities are equal to 0.5! None of our strategies have outperformed a random card choice. Perhaps adjusting the deck size will yield some

improvement. Let's analyze the sample spaces of decks containing 2, 4, 6, and 8 cards. We'll apply all strategies to each sample space and return their probabilities of winning. Afterwards, we'll search for a probability that isn't equal to 0.5.

**Listing 4.14 Applying multiple strategies to multiple decks**

```
for total_cards in [2, 4, 6, 8]:
    total_red_cards = int(total_cards / 2)
    total_black_cards = total_red_cards
    unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards

    sample_space = set(itertools.permutations(unshuffled_deck))
    probabilities = scan_strategies()
    if all(prob == 0.5 for prob in probabilities):
        print(f"No winning strategy found for deck of size {total_cards}")
    else:
        print(f"Winning strategy found for deck of size {total_cards}")
```

```
No winning strategy found for deck of size 2
No winning strategy found for deck of size 4
No winning strategy found for deck of size 6
No winning strategy found for deck of size 8
```

All of the strategies yield a probability of 0.5 across the small decks. Each time we increase the card-deck size, we add 2 additional cards into the deck, but this fails to improve strategy performance. A strategy that fails on a 2-card deck will continue to fail on a 4-card deck. A strategy that fails on an 8-card deck will continue to fail on a 10-card deck. Its easy to extrapolate this logic even further. A strategy that fails on a 10-card deck will probability fail on a 12-card deck, and thus on a 14-card deck, and on a 16-card deck. Eventually, it will fail on a 52-card deck. Qualitatively, this inductive argument makes sense. Mathematically, it can be proven to be true. Right now we don't need to concern ourselves with the math. What's important is that our instincts about our strategy's success has been proven wrong. Our strategies don't work on a 10-card deck, and we have little reason to believe that they will work on a 52-card deck. Why is this the case? Our observed failures provide no insight into why each strategy fails. What exactly is going wrong?

On the surface our strategy makes sense; if there are more red cards than black cards in the deck, then we are more likely to pick a red card from the deck. However, we failed to take into account those scenarios when the red cards never outnumber the black cards. For instance; suppose the first 26 cards are red and the remainder are black. In these circumstances, our strategies will fail to halt and we will lose. Also, lets consider a shuffled deck where the first 25 cards are red, the next 26 cards are black, and the final card is red. Here, our strategy will fail to halt but we will still win. It seems each strategy can lead to one of four possible scenarios:

**A.** Strategy halts and the next card is red. We win.

**B.** Strategy halts and the next card is black. We lose.

**C.** Strategy doesn't halt and the final card is red. We win.

**D.** Strategy doesn't halt and the final card is black. We lose.

Lets sample how frequently the four scenarios occur across 50,000 card shuffles. We'll record these frequencies over our range of two-digit `min_red_fraction` values. We'll then plot each `min_red_fraction` value against the occurrence rates observed from the four scenarios.

### Listing 4.15 Plotting strategy outcomes across a 52-card deck

```
np.random.seed(0)
total_cards = 52
total_red_cards = 26
unshuffled_deck = red_cards + black_cards

def repeat_game_detailed(number_repeats, min_red_fraction):

    observations = [execute_strategy(min_red_fraction, return_index=True)
                    for _ in range(num_repeats)]   ❶
    successes = [index for index, card, in observations if card == 1]   ❷
    halt_success = len([index for index in successes if index != 51])   ❸
    no_halt_success = len(successes) - halt_success   ❹

    failures = [index for index, card, in observations if card == 0]   ❺
    halt_failure = len([index for index in failures if index != 51])   ❻
    no_halt_failure = len(failures) - halt_failure   ❼
    result = [halt_success, halt_failure, no_halt_success, no_halt_failure]
    return [r / float(number_repeats) for r in result]   ❽

fractions = [value / 100 for value in range(50, 100)]
num_repeats = 50000
result_types = [[], [], [], []]

for fraction in fractions:   ❾
    result = repeat_game_detailed(num_repeats, fraction)
    for i in range(4):
        result_types[i].append(result[i])

plt.plot(fractions, result_types[0],
         label='A) Strategy Halts. We Win.')
plt.plot(fractions, result_types[1], linestyle='--',
         label='B) Strategy Halts. We Lose.')
plt.plot(fractions, result_types[2], linestyle=':',
         label='C) No Halt. We Win.')
plt.plot(fractions, result_types[3], linestyle='-.',
         label='D) No Halt. We Lose.')
plt.xlabel('min_red_fraction')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

❶ We execute a strategy across `num_repeats` simulations.

❷ This list contains all instances of wins.

❸ Scenario where our strategy halts and we win.

❹ Scenario where our strategy doesn't halt and we win.

❺ This list contains all instances of losses.

❻ Scenario where our strategy halts and we lose.

❼ Scenario where our strategy doesn't halt and we lose.

❽ We return the observed frequencies for all four scenarios.

❾ We scan the scenario frequencies across multiple strategies.

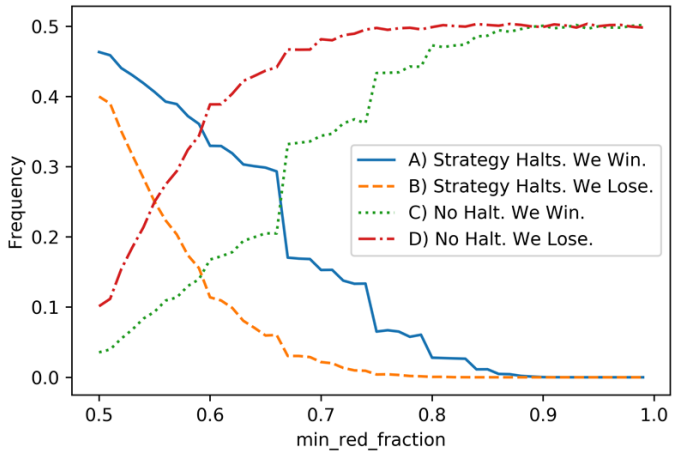


**Figure 4.3 The min_red_fraction parameter is plotted against the sampled frequencies for all four possible scenarios. Scenario A initially has a frequency of roughly 0.45, but eventually it drops to 0. Scenario C has a frequency of roughly 0.05, but eventually it rises to 0.5. The frequency sum for A and C remain at approximately 0.5, thus reflecting a 50% chance of winning the game.**

Lets examine the plot at the `min_red_fraction` value of 0.5. Of the four scenarios, Scenario A (*Strategy Halts. We Win*) occurs most frequency. Its frequency is approximately 0.45. Meanwhile, a halt leads to a loss approximately 40% of the time. (Strategy B). So why do we have a 50% chance of winning the game? Well, in 5% of the cases, our strategy fails to halt but we still win (Scenario C). The weakness of our strategy is counterbalanced by random chance.

Within the plot, as the `min_red_fraction` goes up, the frequency of Scenario A goes down. The more conservative we are, the less likely we are to stop the game prematurely and yield a win. Meanwhile, the success rate of Scenario C goes up. The more conservative we are, the higher the likelihood of reaching the final card and winning by chance.

As `min_red_fraction` goes up, the frequency of Scenario A drops to zero and the frequency of Scenario C goes to 0.5. What's important is that the added frequencies of A and B appear to fluctuate at around 0.5. The probability of a win remains at 50%, despite the fact that Scenario A occurs less than 50% of the time. Sometimes our strategy halts and we do win. Other times, the strategy halts and we still lose. Any advantage that each strategy offers is automatically wiped out by these loses. However, we occasionally get lucky; our strategy fails to halt yet we win the game. These lucky wins amend our loses; and our probability of winning remains steady at 50%. No matter what we do, our likelihood of winning remains 50-50. Therefore, the most optimal strategy we can offer is to pick the first card in the shuffled deck.

**Listing 4.16 The most optimal winning strategy**

```
def optimal_strategy(shuffled_deck):
    return shuffled_deck[0]
```

## 4.4 Key Takeaways

- Probabilities can be counterintuitive. Innately, we assumed that our planned card-game strategy would perform better than random. However, this proved not to be the case. We must be careful when dealing with random processes. Its best to rigorously test all our intuitive assumptions prior to wagering on any future outcome.
- Sometimes, even large-scale simulations fail to find a probability within the required level of precision. However, by simplifying our problem, we can utilize sample spaces in order to yield insights. Sample spaces allow us to test our intuition. If our intuitive solution fails on a toy-version of the problem, then it is also likely to fail on the actual version of the problem.

# Case Study 2: Assessing Online Ad-Clicks for Significance

## CS2.1 Problem Statement

Fred is a loyal friend, and he needs your help. Fred just launched a burger bistro in the city of Brisbane. The bistro is open for business, but business is slow. Fred wants to entice new customers to come and try his tasty burgers. To do this, Fred will run an online advertising campaign directed at Brisbane residents. Every weekday, between 11:00am - 1:00pm, Fred will purchase 3,000 ads aimed at hungry locals. Every ad will be viewed by a single Brisbane resident. The text of every ad will read, "Hungry? Try the Best Burger in Brisbane. Come to Fred's". Clicking on the text will take potential customers to Fred's site. Each displayed ad will cost our friend 1-cent, but Fred believes that the investment will be worth it. Fred is getting ready to execute his ad-campaign. However, he runs into a problem. Fred previews his ad, and its text is blue. Fred believes that blue is a boring color. He feels that other colors could yield more clicks. Fortunately, Fred's advertising software allows him to choose from 30 different colors. Is there a text-color that will bring more clicks than blue? Fred decides to find out.

Fred instigates an experiment. Every weekday, for a month, Fred purchases 3,000 online ads. The text of every ad is assigned to one of 30 possible colors. The advertisements are distributed evenly by color. Thus, 100 ads of identical color are viewed by 100 people every day. For example, 100 people view a blue ad, and another 100 people view a green ad. These numbers add up to 3,000 views that are distributed across the 30 colors. Fred's advertising software automatically tracks all daily views. It also records the daily clicks associated with each of the 30 colors. The software stores this data in a table. That table holds the clicks-per-day and views-per-day for every specified color. Each table row maps a color to the views and clicks for all analyzed days.

Fred has carried out his experiment. He obtained ad-click data for all 20 days. That data is organized by color. Now, Fred wants know if there exists a color that draws significantly more ad-clicks than blue. Unfortunately, Fred doesn't know to properly interpret the results. He's not sure which clicks are meaningful, and which clicks have occurred purely by random. Fred is

brilliant at broiling burgers, but has no training in data analysis. This is why Fred has turned to you for help. Fred asks you to analyze his table and to compare the counts of daily clicks. He's searching for a color that draws significantly more ad-clicks than blue. Are you willing to help Fred? If so, he's promised you free burgers for a year!

## CS2.2 Dataset Description

Fred's ad-click data is stored within the file `colored_ad_click_table.csv`. The csv file extension is an acronym for **Comma Separated Values**. Our csv file is a table stored as text. The table columns are separated by commas. The first line in the file contains the comma-separated labels for the columns. The first 99 characters of that line are *Color,Click Count: Day 1,View Count: Day 1,Click Count: Day 2,View Count: Day 2,Click Count: Day 3,*.

Let's briefly clarify the column labels.

- Column 1: *Color*
  - Each row within the column corresponds to one of 30 possible text colors.
- Column 2: *Click Count: Day 1*
  - The column tallies the times each colored ad was clicked on Day 1 of Fred's experiment.
- Column 3: *View Count: Day 1*
  - The column tallies the times each ad was viewed on Day 1 of Fred's experiment.
  - According to Fred, all daily views are expected to equal 100.
- The remaining 38 columns contain the clicks-per-day and views-per-day for the other 19 days of the experiment.

## CS2.3 Overview

In order to address the problem at hand we will need to know how to:

**A.** Measure the centrality and dispersion of sampled data.

**B.** Interpret the significance of two diverging means through p-value calculation.

**C.** Minimize mistakes associated with misleading p-value measurements.

**D.** Load and manipulate data stored in tables using Python.

# 5 Basic Probability and Statistical Analysis Using SciPy

***This section covers:***

- Analysis of Binomials using the SciPy library.
- Defining dataset centrality.
- Definining dataset dispersion.
- Computing the centrality and dispersion of probability distributions.

Statistics is a branch of mathematics dealing with the collection and interpretation of numeric data. It is the precursor of all modern data science. The term Statistic originally signified "the science of the state", because statistical methods were first developed to analyze the data of state governments. Since ancient times, government agencies have gathered data pertaining to their populace. That data would be used to levy taxes and to organize large military campaigns. Hence, critical state decisions depended on the quality of data. Poor record-keeping could lead to potentially disastrous results. That is why state bureaucrats were very concerned by any random fluctuations in their records. Probability theory eventually tamed these fluctuations, making the randomness interpretable. Ever since then, statistics and probability theory have been closely intertwined.

Statistics and probability theory are closely related, but in some ways they are very different. Probability theory studies random processes over a potentially infinite number of measurements. It is not bound by real-world limitations. This allows us to model the behavior of a coin by imagining millions of coin-flips. In real life, flipping a coin millions of times is a pointlessly time-consuming endeavor. Surely we can sacrifice some data instead of flipping coins all day and night. Statisticians acknowledge these constraints placed upon us by the data-gathering process. Real-world data collection is costly and time-consuming. Every data-point carries a price. We cannot survey a country's population without employing government officials. We cannot test our online ads without paying for every ad that's clicked. Thus, the size of our final

dataset usually depends on the size of our initial budget. If the budget is constrained, then the data will also be constrained. This trade-off between data and resourcing lies at the heart of modern statistics. Statistics helps us understand exactly how much data is sufficient to draw insights and make impactful decisions. The purpose of statistics is to find meaning in data even when that data is limited in size.

Statistics is highly mathematical, and is usually taught using math equations. Nevertheless, direct exposure to equations is not a prerequisite for statistical understanding. In fact, many data scientists do not write formulas when running statistical analyses. Instead, they leverage Python libraries such as SciPy, which handle all the complex math calculations. However, proper library usage still requires an intuitive understanding of statistical procedures. In this section, we will cultivate our understating of statistics by applying probability theory to real-world problems.

## 5.1 Exploring the Relationships between Data and Probability Using SciPy

SciPy, which is shorthand for Scientific Python, provides many useful methods for scientific analysis. The SciPy library includes an entire module for addressing problems in probability and statistics; `scipy.stats`. Let's install the library. Afterwards, we'll import the `stats` module.

> **NOTE** Call "pip install scipy" from the command-line terminal in order to install the SciPy library.

**Listing 5.1 Importing the `stats` module from SciPy**

```
from scipy import stats
```

The `stats` module will greatly aid our efforts in assessing the randomness of data. For example, in Section One we computed the probability of a fair coin producing at-least 16 heads after 20 flips. Our calculations required us to examine all possible combinations of 20 flipped coins. Afterwards, we computed the probability of observing 16 or more heads or 16 or more tails, in order to measure the randomness of our observations. SciPy allows us to measure this probability directly using the `stats.binomial_test` method. The method is named after the Binomial distribution, which governs how a flipped coin might fall. The method requires 3 parameters: the number of heads, the total number of coin flips, and the probability of a coin landing on heads. Let's apply the Binomial test to 16 heads observed from 20 coin-flips. Our output should equal the previously computed value of approximately 0.011.

**NOTE** SciPy and standard Python handle low-value decimal points differently. In Section One, when we computed the probability, the final value was rounded to 17 significant digits. SciPy, on the other hand, returns a value containing 18 significant digits. Thus, for consistency's sake, we'll round our SciPy output to 17 digits.

**Listing 5.2 Analyzing extreme head-counts using SciPy**

```
num_heads = 16
num_flips = 20
prob_head = 0.5
prob = stats.binom_test(num_heads, num_flips, prob_head)
print(f"Probability of observing more than 15 heads or 15 tails is {prob:.17f}")
```

```
Probability of observing more than 15 heads or 15 tails is 0.01181793212890625
```

It's worth emphasizing that `stats.binomial_test` did not compute the probability of observing 16 heads. Rather, it returned the probability of seeing a coin-flip sequence where 16 or more coins fell on the same face. If we want the probability seeing exactly 16 heads, then we must utilize the `stats.binom.pmf` method. That method represents the **probability mass function** of the Binomial distribution. A probability mass function maps inputted integer values to their probability of occurrence. Thus, calling `stats.binom.pmf(num_heads, num_flips, prob_heads)` will return the likelihood of a coin yielding `num_heads` number of heads. Under current settings, this will equal the probability of a fair coin falling on heads 16 out of 20 times.

**Listing 5.3 Computing an exact probability using `stats.binom.pmf`**

```
prob_16_heads = stats.binom.pmf(num_heads, num_flips, prob_head)
print(f"The probability of seeing {num_heads} of {num_flips} heads is {prob_16_heads}")
```

```
The probability of seeing 16 of 20 heads is 0.004620552062988271
```

We've used `stats.binom.pmf` to find the probability of seeing exactly 16 heads. However, that method is also able to compute multiple probabilities simultaneously. Multiple head-count probabilities can be processed by passing in a list of head-count values. For instance, passing `[4, 16]` will return a 2-element NumPy array containing the probabilities of seeing 4 heads and 16 heads, respectively. Conceptually, the probability of seeing 4 heads and 16 tails equals the probability of seeing 4 tails and 16 heads. Thus, executing `stats.binom.pmf([4, 16], num_flips, prob_head)` should return a 2-element array whose elements are equal. Let's confirm.

**Listing 5.4 Computing an array of probabilities using `stats.binom.pmf`**

```
probabilities = stats.binom.pmf([4, 16], num_flips, prob_head)
assert probabilities.tolist() == [prob_16_heads] * 2
```

List-passing allows us to easily compute probabilities across intervals. For example, if we pass

`range(21)` into `stats.binom.pmf`, then the outputted array will contain all probabilities across the interval of every possible head-count. As we learned in Section One, the sum of these probabilities should equal 1.0.

**NOTE** Summing low-value decimals is computationally tricky. Over the course of the summation, tiny errors will accumulate. Due to the these errors, our final summed probability will marginally diverge from 1.0, unless we round it to 14 significant digits. We do this rounding in the code below.

**Listing 5.5 Computing an interval probability using `stats.binom.pmf`**

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, prob_head)
print(f"Total sum of probabilities equals {total_prob:.14f}")
```

```
Total sum of probabilities equals 1.00000000000000
```

Also, as discussed in Section Two, plotting `interval_all_counts` versus `probabilities` will reveal the shape of our 20 coin-flip distribution. Thus, we can generate the distribution plot without having to iterate through possible coin-flip combinations.

**Listing 5.6 Plotting a 20 coin-flip Binomial Distribution**

```
import matplotlib.pyplot as plt
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

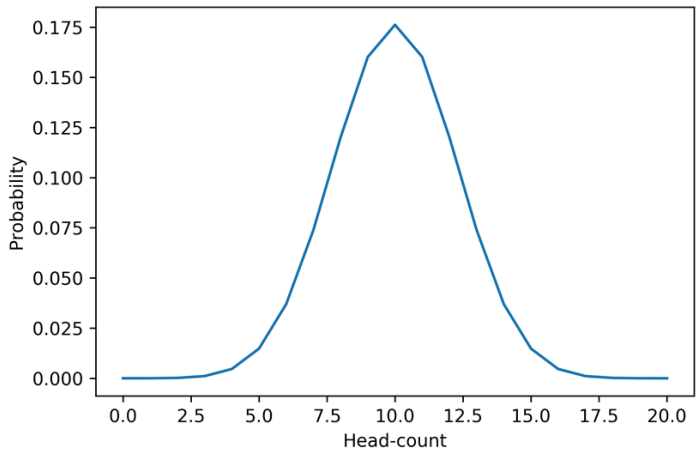


**Figure 5.1 The probability distribution for 20 coin-flips, generated using SciPy.**

In Section Two our ability to visualize the Binomial was limited by the total number of coin-flip combinations that we needed to compute. Now, this is no longer the case. The `stats.binom.pmf` methods allows to display any distribution associated with an arbitrary

coin-flip count. Let's use our new-found freedom to simultaneously plot the distributions for 20, 80, 140, and 200 coin-flips.

**Listing 5.7 Plotting 5 different Binomial distributions**

```
flip_counts = [20, 80, 140, 200]
linestyles = ['-', '--', '-.', ':']
colors = ['b', 'g', 'r', 'k']

for num_flips, linestyle, color in zip(flip_counts, linestyles, colors):
    x_values = range(num_flips + 1)
    y_values = stats.binom.pmf(x_values, num_flips, 0.5)
    plt.plot(x_values, y_values, linestyle=linestyle, color=color,
             label=f'{num_flips} coin-flips')
plt.legend()
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

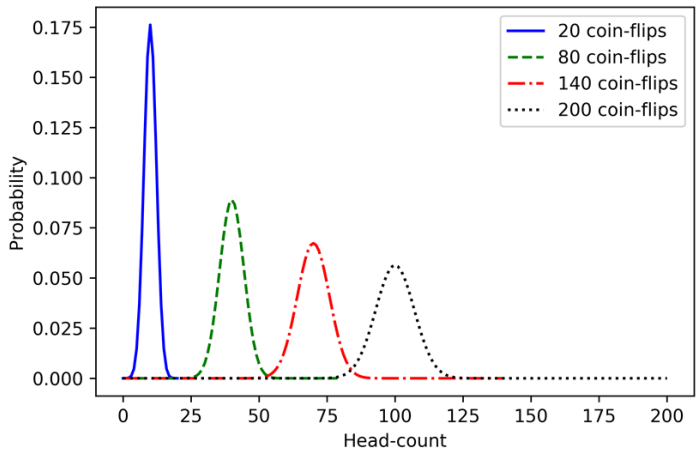


**Figure 5.2 Multiple Binomial probability distributions across 20, 80, 140, and 20 coin-flips. The distribution centers shift right as the coin-flip count goes up. Also, every distribution becomes more dispersed around its center as the coin-flip increases.**

Within the plot, the central peak of each Binomial appears to shift right-ward as the coin-flip count goes up. Also, the 20 coin-flip distribution is noticeably thinner than the 200 coin-flip distribution. In other words, the plotted distributions grow more dispersed around their central positions as these central positions relocate right.

Such shifts in centrality and dispersion are commonly encountered in data analysis. We've previously observed these shifts in Section Three. In that section, we used randomly sampled data to visualize several histogram distributions. Subsequently, we observed that the plotted histogram thickness was dependent on our sample size. At the time, our observations were purely qualitative. We lacked a rigorous metric for comparing the thickness of two plots. However, simply noting that one plot appears "thicker" than another is insufficient. Likewise, stating that one plot is more "right-ward" than another is also insufficient. We need to quantitate our distribution differences. We must assign specific numbers to centrality and dispersion in order to

discern how these numbers change from plot to plot. Doing so requires that we familiarize ourselves with the concepts of variance and mean.

## 5.2 Mean as a Measure of Centrality

Suppose we wish to study our local temperature over the first week of summer. When summer comes around, we glance at the thermometer outside our window. At noon-time, the temperature is exactly 80 degrees. We proceed to repeat our noon-time measurements over the course of the next 6 days. Our subsequent measurements are 80, 77, 73, 61, 74, 79, and 81 degrees. Let's store these measurements in a NumPy array.

**Listing 5.8 Storing recorded temperatures in a NumPy array**

```
import numpy as np
measurements = np.array([80, 77, 73, 61, 74, 79, 81])
```

We'll now attempt to summarize our measurements using a single central value. First, we'll sort the measurement in-place by calling `measurements.sort()`. Afterwards, we'll plot the sorted temperatures in order to evaluate their centrality.

**Listing 5.9 Plotting the recorded temperatures**

```
measurements.sort()
number_of_days = measurements.size
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.ylabel('Temperature')
plt.show()
```

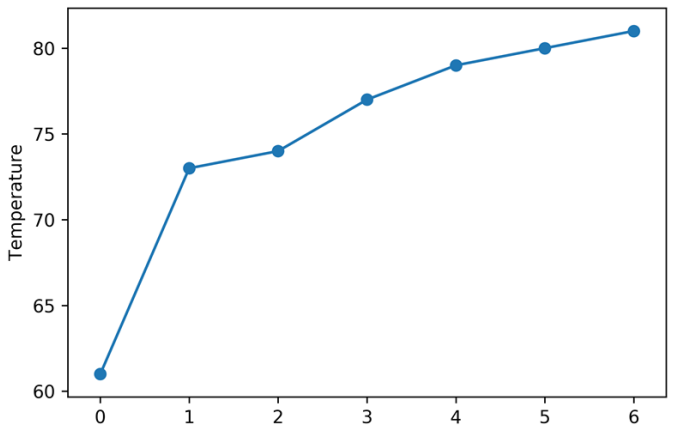


**Figure 5.3 A plot containing 7 sorted temperatures. A central temperature exists somewhere between 60 and 80 degrees.**

Based on the plot, a central temperature exists somewhere between 60 degrees and 80 degrees. Therefore, we can naively estimate the center as approximately 70 degrees. Let's quantitate our

estimate as the mid-point between the lowest value and the highest value in the plot. We'll compute that midpoint by taking half the difference between the minimum and maximum temperatures, and adding it to the minimum temperature. As an aside, we can obtain that same value by summing the minimum and maximum directly, and afterwards dividing that sum by 2.

### Listing 5.10 Finding the midpoint temperature

```
difference = measurements.max() - measurements.min()
midpoint = measurements.min() + difference / 2
assert midpoint == (measurements.max() + measurements.min()) / 2
print(f"The midpoint temperature is {midpoint} degrees")
```

```
The midpoint temperature is 71.0 degrees
```

The midpoint temperature is 71 degrees. Let's mark that midpoint in our plot using a horizontal line. We'll draw the horizontal line by calling `plt.axhline(midpoint)`.

### Listing 5.11 Plotting the midpoint temperature

```
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--')
plt.ylabel('Temperature')
plt.show()
```

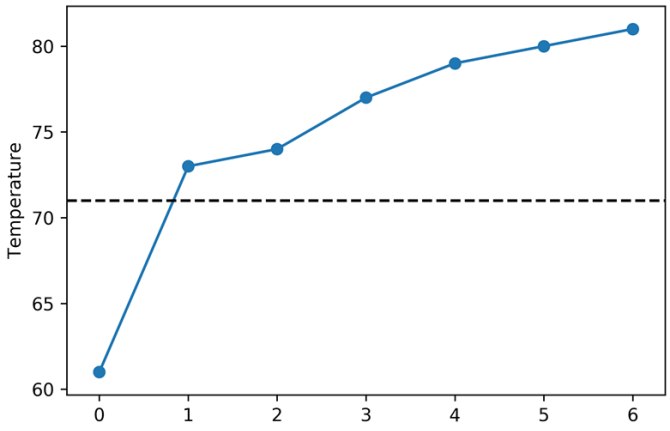


**Figure 5.4 A plot containing 7 sorted temperatures. A temperature of 71 degrees marks the midpoint between the highest and lowest temperature. That midpoint seems low; 6 of 7 temperatures are present above the midpoint value.**

Our plotted midpoint seems a little low; 6 of our 7 measurements are higher than the midpoint. Intuitively, our central value should split the measurements more evenly. The number of temperatures above and below the center should be approximately equal. That sense of equality can be immediately achieved by selecting the middle element in our sorted 7-element array. The middle element, which statisticians call the **median**, will split our measurements into exactly equal parts. 3 measurements will be appear below the median, and 3 measurements will appear

below it. 3 is also the index in the `measurements` array where the median is present. Let's add the median to our plot.

### Listing 5.12 Plotting the median temperature

```
median = measurements[3]
print(f"The median temperature is {median} degrees")
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.legend()
plt.ylabel('Temperature')
plt.show()
```

```
The median temperature is 77 degrees
```

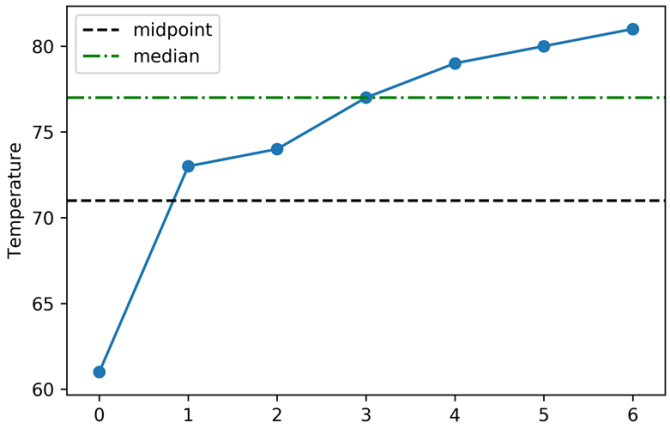


**Figure 5.5 A plot containing 7 sorted temperatures. A median of 77 degrees splits the temperatures in half. The median appears slightly off balance. It is closer to the 3 upper temperatures than to the 3 lower temperatures.**

Our median of 77 degrees splits the temperatures in half. However, one might argue that the split is not as balanced as it could be. The median is closer to the 3 upper temperatures than it is to the 3 lower temperatures. In particular, the median is noticeably far from our minimum measure of 61 degrees. Perhaps we can balance the split by penalizing the median for being too far from the minimum. We'll carry out this penalty by introducing the concept of **squared distance**. The squared distance is simply the squared difference between two given values. It will grow in a quadratic, non-linear fashion as the two values grow further apart. Thus, if we penalize our central value based on its distance to 61, then the squared distance penalty will grow noticeably larger as it drifts away from 61 degrees.

## Listing 5.13 Penalizing centers using squared distance from minimum

```
def squared_distance(value1, value2):
    return (value1 - value2) ** 2

possible_centers = range(measurements.min(), measurements.max() + 1)  ❶
penalties = [squared_distance(center, 61) for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()
```

❶ We are using the range of values between the minimum and maximum measured temperatures as our set of possible centers.

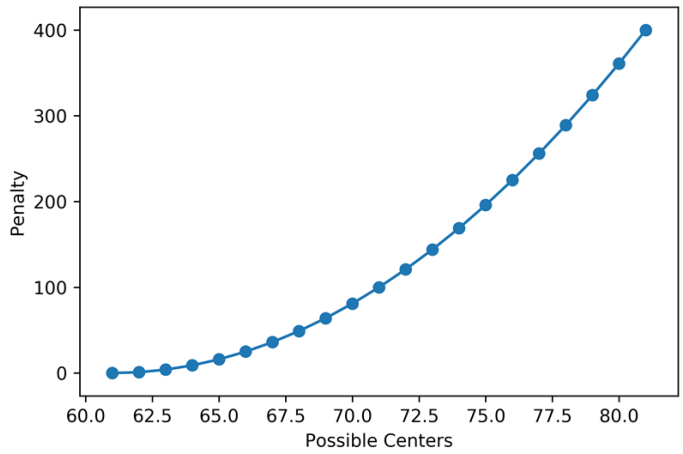


**Figure 5.6 A plot of possible centers penalized based on their squared distance relative to the minimum temperature of 61 degrees. Not surprisingly, the minimum penalty occurs at 61 degrees. Unfortunately, the penalty doesn't take into account the distance to the remaining 6 recorded temperatures.**

Our plot displays the penalty across a range of possible centers based on their distance to our minimum. As the centers shift towards the 61, the penalty will drop, but their distance to the remaining 6 measurements will increase. Thus, we ought to penalize each potential center based on its squared distance to all 7 recorded measurements. We'll do so by defining a sum of squared distances function, which will add up the squared distances between some value and the measurement array. That function will serve as our new penalty. Plotting the possible centers against their penalties will allow us to find the center whose penalty is minimized.

## Listing 5.14 Penalizing centers using total sum of squared distances

```
def sum_of_squared_distances(value, measurements):
    return sum(squared_distance(value, m) for m in measurements)

penalties = [sum_of_squared_distances(center, measurements)
             for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()
```

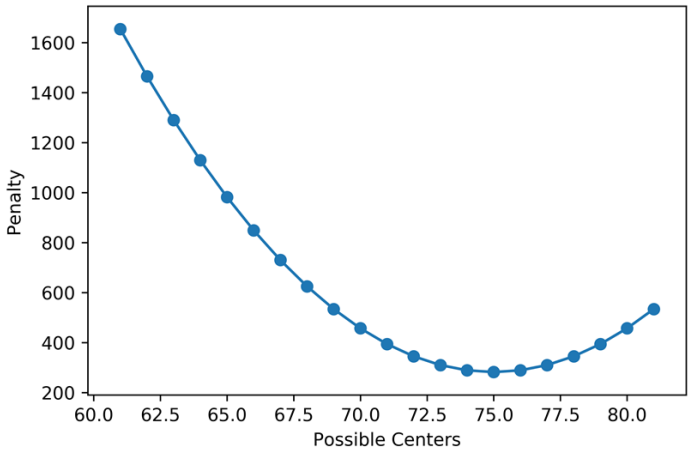


**Figure 5.7 A plot of possible centers penalized based on the sum of their squared distance relative to all recorded temperatures. The minimum penalty occurs at 75 degrees.**

Based on our plot, the temperature of 75 degrees incurs the lowest penalty. We'll informally refer to this temperature value as our "least-penalized center". Let's demarcate it using a horizontal line within our temperature plot.

## Listing 5.15 Plotting the least-penalized temperature

```
least_penalized = 75
assert least_penalized == possible_centers[np.argmin(penalties)]

plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.axhline(least_penalized, color='r', linestyle='-',
            label='least penalized center')
plt.legend()
plt.ylabel('Temperature')
plt.show()
```

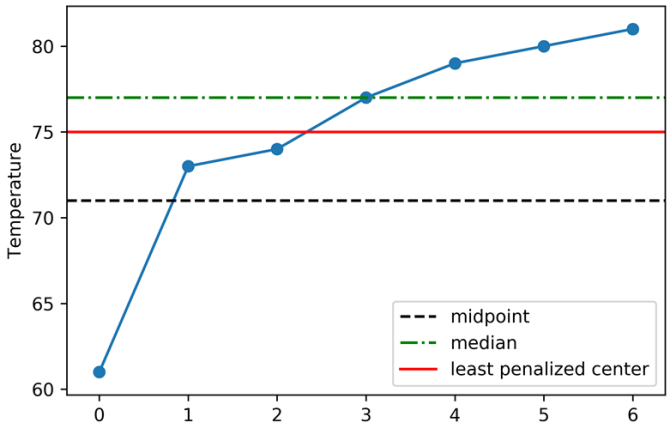


**Figure 5.8 A plot containing 7 sorted temperatures. The least-penalized center of 75 degrees splits the temperatures in a balanced manner.**

The least-penalized center splits the measured temperatures fairly evenly; 4 measurements appear above it and 3 measurements appear below it. Thus, this center maintains a balanced data-split while providing a closer distance to the coldest recorded temperature relative to the median.

The least-penalized center is a good measure of centrality. It minimizes all the penalties incurred for being too far from any given point. This leads to balanced distances between the center and every data-point. Unfortunately, our computation of that center was very inefficient. Scanning all possible penalties is not a scalable solution. Is there a more efficient way to compute the center? Yes! Mathematicians have shown that sum of squared distances error is always minimized by the **average** value of a dataset. Thus, we can compute the least-penalized center directly. We simply need to sum all the elements in `measurements` and then divide that sum by the array size.

**Listing 5.16 Computing the least-penalized center using an average value**

```
assert measurements.sum() / measurements.size == least_penalized
```

A summed array of values divided by array size is formally called the **arithmetic mean**. Informally, the value is referred to as the **mean** or the average of the array. As discussed in Section Three, the mean can be computed by calling the `mean` method of a NumPy array. We can also compute the mean by calling the `np.mean` and `np.average` methods.

**Listing 5.17 Computing the mean using NumPy**

```
mean = measurements.mean()
assert mean == least_penalized
assert mean == np.mean(measurements)
assert mean == np.average(measurements)
```

The `np.average` method differs from the `np.mean` method because it takes as input an optional

`weights` parameter. The `weights` parameter is a list of numeric weights that capture the importance of the measurements relative to each other. When all the weights are equal, the output of `np.average` is no different from `np.mean`. However, adjusting the weights will lead to a difference in the outputs.

**Listing 5.18 Passing weights into `np.average`**

```
equal_weights = [1] * 7
assert mean == np.average(measurements, weights=equal_weights)

unequal_weights = [100] + [1] * 6
assert mean != np.average(measurements, weights=unequal_weights)
```

The `weights` parameter is useful for computing the mean across duplicate measurements. Suppose we analyze 10 temperature measurements where 75 degrees appears 9 times, and 77 degrees appears just once. The full list of measurements is represented by `9 * [75] + [1]`. We can compute the mean by calling `np.mean` on that list. We can also compute the mean by calling `np.average([75, 77], weights=[9, 1])`. Both these computations will be equal.

**Listing 5.19 Computing the weighted mean of duplicate values**

```
weighted_mean = np.average([75, 77], weights=[9, 1])
print(f"The mean is {weighted_mean}")
assert weighted_mean == np.mean(9 * [75] + [77]) == weighted_mean
```

```
The mean is 75.2
```

Computing the weighted mean serves a shortcut for computing the regular mean when duplicates are present. In the computation, the relative ratio of unique measurement counts is represented by the ratio of the weights. Thus, even if we convert our absolute counts of 9 and 1 into relative weights of 900 and 100, the final value of `weighted_mean` should remain the same. This will also be true if the weights are converted into relative probabilities of 0.9 and 0.1.

**Listing 5.20 Computing the weighted mean of relative weights**

```
assert weighted_mean == np.average([75, 77], weights=[900, 100])
assert weighted_mean == np.average([75, 77], weights=[0.9, 0.1])
```

We can treat probabilities as weights. Consequently, this allows us to compute the mean of any probability distribution.

### *5.2.1 Finding the Mean of a Probability Distribution*

At this point in the book, we are intimately familiar with the 20 coin-flip Binomial distribution. The distribution's peak is symmetrically centered at 10 heads. How does that peak compare to the distribution's mean? Lets find out. We'll compute the mean by passing a `probabilities` array to into the `weights` parameter of `np.average`. Afterwards, we'll plot the mean as a vertical line that cuts across the distribution.

## Listing 5.21 Computing the mean of a Binomial distribution

```
num_flips = 20
interval_all_counts = range(num_flips + 1)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
mean_binomial = np.average(interval_all_counts, weights=probabilities)
print(f"The mean of the Binomial is {mean_binomial:.2f} heads")
plt.plot(interval_all_counts, probabilities)
plt.axvline(mean_binomial, color='k', linestyle='--')   ❶
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

❶ The `axvline` method plots a vertical line at a specified x coordinate.

```
The mean of the Binomial is 10.00 heads
```

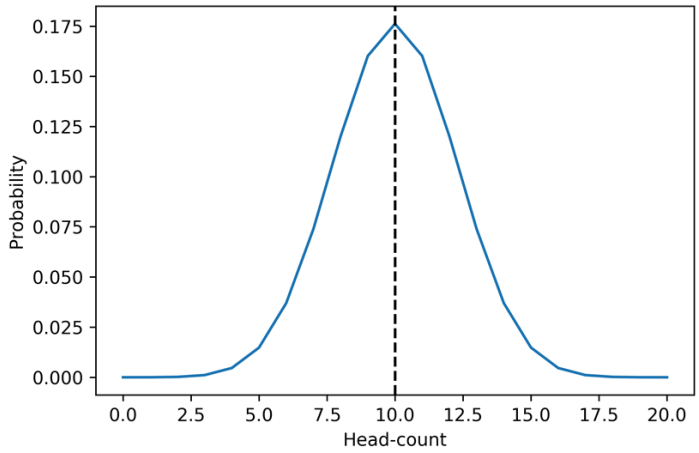


**Figure 5.9 A 20 coin-flip Binomial distribution bisected by its mean. The mean is positioned directly in the distribution's center.**

The mean of the Binomial is 10 heads. It cuts across the distribution's central peak. The mean perfectly captures the Binomial's centrality. That is why SciPy permits us to obtain the mean of any Binomial simply by calling `stats.binom.mean`. The `stats.binom.mean` method takes as input 2 parameters: number of coin flips and the probability of heads.

## Listing 5.22 Computing the Binomial mean using SciPy

```
assert stats.binom.mean(num_flips, 0.5) == 10
```

Using the `stats.binom.mean` method, we can rigorously analyze the relationship between Binomial centrality and coin-flip count. Let's plot the Binomial mean across a range of coin-flip counts spanning from 0 to 500.

## Listing 5.23 Plotting multiple Binomial means

```
means = [stats.binom.mean(num_flips, 0.5) for num_flips in range(500)]
plt.plot(range(500), means)
plt.xlabel('Coin Flips')
plt.ylabel('Mean')
plt.show()
```

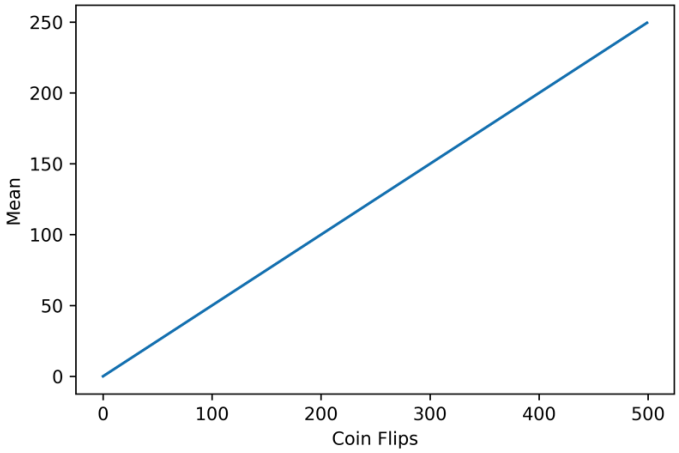


**Figure 5.10 Coin-flip count plotted against Binomial mean. The relationship is linear. The mean of each Binomial is mean is equal to half its coin-flip count.**

The coin-flip count and mean appear to share a linear relationship. Based on the plot, the fair coin's mean is always equal to half the coin-flip count. With this in mind, let's consider the mean of the single coin-flip Binomial distribution (which is commonly called the **Bernoulli distribution**). The Bernoulli distribution has a coin-flip count of 1, so its mean is equal to 0.5. Not surprisingly, the probability of a fair coin landing on heads is equal to the Bernoulli mean.

## Listing 5.24 Predicting the mean of a Bernoulli distribution

```
num_flips = 1
assert stats.binom.mean(num_flips, 0.5) == 0.5
```

We can leverage the observed linear relationship to predict the mean of a 1000 coin-flip distribution. We expect that mean to equal 500, and for it to be positioned in the distribution's center. Let's confirm this is the case.

**Listing 5.25 Predicting the mean of a 1000 coin-flip distribution**

```
num_flips = 1000
assert stats.binom.mean(num_flips, 0.5) == 500

interval_all_counts = range(num_flips)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, 0.5)
plt.axvline(500, color='k', linestyle='--')
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

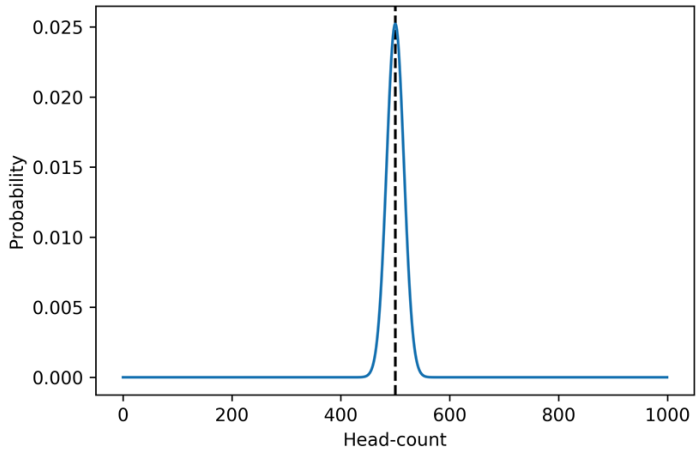


**Figure 5.11 A 1000 coin-flip Binomial distribution bisected by its mean. The mean is positioned directly in the distribution's center.**

A distribution's mean serves as an excellent measure of centrality. Let's now explore the use of variance as a measure of dispersion.

## 5.3 Variance as a Measure of Dispersion

Dispersion is the scattering of data-points around some central value. A smaller dispersion is indicative of more predictable data. A larger dispersion is indicative of greater data fluctuations. Consider a scenario where we measure the summer temperatures in California and Kentucky. We gather 3 measurements for each state. Our measured California temperatures are 52, 77, and 96 degrees. Our measured Kentucky temperatures are 71, 75, and 79 degrees. We'll store these measured temperatures and compute their means.

**Listing 5.26 Measuring the means of multiple temperature arrays**

```
california = np.array([52, 77, 96])
kentucky = np.array([71, 75, 79])

print(f"Mean California temperature is {california.mean()}")
print(f"Mean Kentucky temperatures is {california.mean()}")
```

```
Mean California temperature is 75.0
```

```
Mean Kentucky temperatures is 75.0
```

The means of the 2 measurement arrays both equal 75. California and Kentucky appear to share the same central temperature value. Despite this, the 2 measurement arrays are far from equal. The California temperatures are much more dispersed and unpredictable. They range from 52 to 96 degrees. Meanwhile, the stable Kentucky temperatures range from the low 70s to high 70s. They are more closely centered around the mean. We'll visualize this difference in dispersion by plotting the 2 measurement arrays. Additionally, we'll demarcate the mean by plotting a horizontal line.

### Listing 5.27 Visualizing the difference in dispersion

```
plt.plot(range(3), california, color='b', label='California')
plt.scatter(range(3), california, color='b')
plt.plot(range(3), kentucky, color='r', linestyle='-.', label='Kentucky')
plt.scatter(range(3), kentucky, color='r')
plt.axhline(75, color='k', linestyle='--', label='Mean')
plt.legend()
plt.show()
```

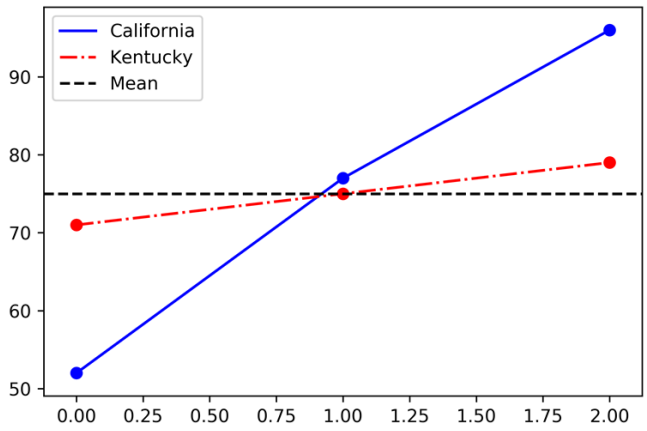


**Figure 5.12 A plot of sorted temperatures for California and Kentucky. Temperatures in both states share a mean of 75 degrees. The California temperatures are more dispersed around that mean.**

Within the plot, the 3 Kentucky temperatures nearly overlap with the flat mean. Meanwhile, the majority of California temperatures are noticeably more distant from the mean. Let's quantify these observations. We'll aim to penalize the California measurements for being too distant from their center. Previously, we computed such penalties using the sum of squared distances function. Now, we will compute the sum of squared distances between the California measurements and their mean. Statisticians refer to the sum of squared distances from the mean as simply the **sum of squares**. We'll now define a `sum_of_squares` function, and then apply it to our California temperatures.

### Listing 5.28 Computing California's sum of squares

```
def sum_of_squares(data):
    mean = np.mean(data)
    return sum(squared_distance(value, mean) for value in data)

california_sum_squares = sum_of_squares(california)
print(f"California's sum of squares is {california_sum_squares}")
```

```
California's sum of squares is 974.0
```

California's sum of squares is 974. We expect Kentucky's sum of squares to be noticeably lower. Let's confirm.

### Listing 5.29 Computing Kentucky's sum of squares

```
kentucky_sum_squares = sum_of_squares(kentucky)
print(f"Kentucky's sum of squares is {kentucky_sum_squares}")
```

```
Kentucky's sum of squares is 32.0
```

Kentucky's sum of squares is 32. Thus, we see a 30-fold difference between our California results and our Kentucky calculations. This isn't surprising, because the Kentucky data are much less dispersed. The sum of squares helps measure that dispersion. However, the measurement is not perfect. Suppose we duplicate the temperatures within the `California` array by recording each temperature twice. The level of dispersion will remain the same even though the sum of squares will double.

### Listing 5.30 Computing sum of squares after array duplication

```
california_duplicated = np.array(california.tolist() * 2)
duplicated_sum_squares = sum_of_squares(california_duplicated)
print(f"Duplicated California sum of squares is {duplicated_sum_squares}")
assert duplicated_sum_squares == 2 * california_sum_squares
```

```
Duplicated California sum of squares is 1948.0
```

The sum of squares is not a good measure of dispersion because it's influenced by the size of an inputted array. Fortunately, that influence is easy to eliminate. We simply divide the sum of squares by the array size. Dividing `california_sum_squares` by `california.size` will produce a value equal to `duplicated_sum_squares / california_duplicated.size`.

### Listing 5.31 Dividing sum of squares by array size

```
value1 = california_sum_squares / california.size
value2 = duplicated_sum_squares / california_duplicated.size
assert value1 == value2
```

Dividing sum of squares by the number of measurements produces what statisticians call the **variance**. Conceptually, the variance is equal to the average squared distance from the mean.

**Listing 5.32 Computing the variance from mean squared distance**

```
def variance(data):
    mean = np.mean(data)
    return np.mean([squared_distance(value, mean) for value in data])

assert variance(california) == california_sum_squares / california.size
```

The variances for the `california` and `california_duplicated` arrays will equal, since their levels of dispersion are identical.

**Listing 5.33 Computing the variance after array duplication**

```
assert variance(california) == variance(california_duplicated)
```

Meanwhile, the variances for the `California` and `Kentucky` arrays will retain their 30-fold ratio caused by a difference in dispersion.

**Listing 5.34 Comparing the variances of California and Kentucky**

```
california_variance = variance(california)
kentucky_variance = variance(kentucky)
print(f"California Variance is {california_variance}")
print(f"Kentucky Variance is {kentucky_variance}")
```

```
California Variance is 324.6666666666667
Kentucky Variance is 10.666666666666666
```

Variance is a good measure of dispersion. It can be computed by calling `np.var` on a Python list or NumPy array. The variance of a NumPy array can also be computed using the array's built-in `var` method.

**Listing 5.35 Computing the variance using NumPy**

```
assert california_variance == california.var()
assert california_variance == np.var(california)
```

Variance is dependent on the mean. If we compute a weighted mean, then we must also compute a weighted variance. Computing the weighted variance is easy. As stated earlier, the variance is simply the average of all the squared distances from the mean. Therefore the weighted variance is simply the weighted average of all the squared distances from the weighted mean. Let's define a `weighted_variance` function. The function will take as input 2 parameters: a data-list and weights. It will then compute the weighted mean. Afterwards, it will use the `np.average` method to computed the weighted average of the squared distances from that mean.

**Listing 5.36 Computing the weighted variance using `np.average`**

```
def weighted_variance(data, weights):
    mean = np.average(data, weights=weights)
    squared_distances = [squared_distance(value, mean) for value in data]
    return np.average(squared_distances, weights=weights)

assert weighted_variance([75, 77], [9, 1]) == np.var(9 * [75] + [77])   ❶
```

❶ `weighted_variance` allows to treat duplicated elements as weights.

The `weighted_variance` function can take as its input an array of probabilities. This allows us to compute the variance of any probability distribution.

### 5.3.1 Finding the Variance of a Probability Distribution

Lets compute variance of the Binomial distribution associated with 20 fair coin-flips. We'll run the computation by assigning a `probabilities` array to the `weights` parameter of `weighted_variance`.

**Listing 5.37 Computing the variance of a Binomial distribution**

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
variance_binomial = weighted_variance(interval_all_counts, probabilities)
print(f"The variance of the Binomial is {variance_binomial:.2f} heads")
```

```
The variance of the Binomial is 5.00 heads
```

The Binomial's variance is 5, which is equal to half the Binomial's mean. That variance can be computed more directly using SciPy's `stats.binom.var` method.

**Listing 5.38 Computing the Binomial variance using SciPy**

```
assert stats.binom.mean(num_flips, 0.5) == 10
```

Using the `stats.binom.var` method, we can rigorously analyze the relationship between Binomial dispersion and coin-flip count. Let's plot the Binomial variance across a range of coin-flip counts spanning from 0 to 500.

**Listing 5.39 Plotting multiple Binomial variances**

```
variances = [stats.binom.var(num_flips, prob_head)
             for num_flips in range(500)]
plt.plot(range(500), variances)
plt.xlabel('Coin Flips')
plt.ylabel('Variance')
plt.show()
```

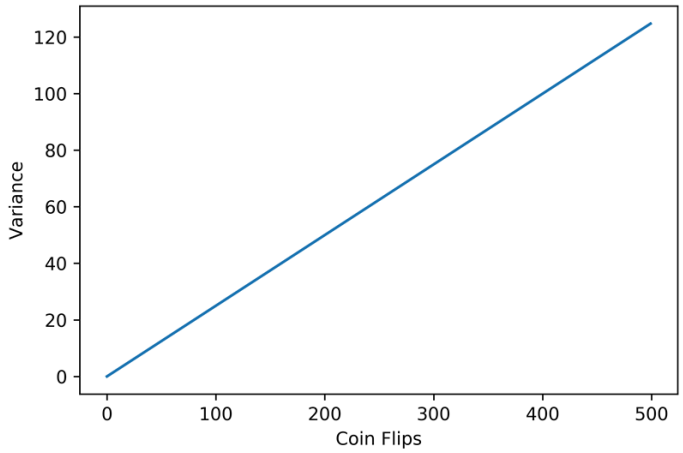


**Figure 5.13 Coin-flip count plotted against Binomial variance. The relationship is linear. The variance of each Binomial is equal to one-fourth its coin-flip count.**

The Binomial's variance, like its mean, is linearly related to the coin-flip count. The variance is equal to one-fourth the coin-flip count. Thus, the Bernoulli distribution has a variance of 0.25, because its coin-flip count is 1. By this logic, we can expect a variance of 250 for a 1000 coin-flip distribution.

## Listing 5.40 Predicting Binomial variances

```
assert stats.binom.var(1, 0.5) == 0.25
assert stats.binom.var(1000, 0.5) == 250
```

**SIDEBAR** **Common SciPy methods for Binomial analysis**

- `stats.binom.mean(num_flips, prob_heads)`:
  Returns the mean of a Binomial where the flip-count equals `num_flips`, and probability of heads equals `prob_heads`.
- `stats.binom.var(num_flips, prob_heads)`:
  Returns the variance of a Binomial where the flip-count equals `num_flips`, and probability of heads equals `prob_heads`.
- `stats.binom.pmf(head_count_int, num_flips, prob_head)`:
  Returns the probability of observing `head_count_int` heads out of `num_flips` coin-flips. A single coin-flip's probability of heads is set to `prob_heads`.
- `stats.binom.pmf(head_count_array, num_flips, prob_head)`:
  Returns an array of Binomial probabilities. These are obtained by executing `stats.binom.pmf(e, num_flips, prob_head)` on each element `e` of `head_count_array`.
- `stats.binom_test(head_count_int, num_flips, prob_head)`:
  Returns the probability of `num_flips` coin-flips generating at-least `head_count_int` heads or `head_count_int` tails. A single coin-flip's probability of heads is set to `prob_heads`.

The variance is powerful measure of data dispersion. However, statisticians often use an alternative measure, which they call the **standard deviation**. The standard deviation is equal to the square-root of the variance. It can be computed by calling `np.std`. Squaring the output of `np.std` will naturally return the variance.

**Listing 5.41 Computing the standard deviation**

```
data = [1, 2, 3]
standard_deviation = np.std(data)
assert standard_deviation ** 2 == np.var(data)
```

Why use the standard deviation instead of variance? The answer has to do with unit-tracking. All measurements have units. Our measured temperatures had units of Fahrenheit. When we squared the distances of the temperature to their mean, we also squared their units. Therefore, our variance was in units of Fahrenheit-squared. Such squared units are very tricky to conceptualize. Taking the square root converts the units back to Fahrenheit again. Thus, a standard deviation in units of Fahrenheit is more easily interpretable than the variance.

The mean and standard deviation are incredibly useful values. They allow us to:

**A.** Compare numeric datasets.

- Suppose we're given two arrays of recorded temperatures for two consecutive summers. We can quantify the differences between these summer records using mean and standard deviation.

**B.** Compare probability distributions.

- Suppose two climate research labs publish two probability distributions. Each distribution captures all temperature probabilities across a standard summer day. We can summarize the differences between the two distributions by comparing their means and standard deviations.

**C.** Compare a numeric dataset to a probability distribution.

- Suppose a well-known probability distribution captures a decade's worth of temperature probabilities. However, recently recorded summer temperatures appear to contradict these probability outputs. Is this a sign of climate change, or simply a random anomaly? We can find out by juxtaposing the centrality and dispersion for the distribution and the temperature dataset.

The third use-case underlies much of statistics. In the subsequent sections, we will learn how to compare datasets to distribution likelihoods. Many of our comparisons will focus on the Normal distribution, which commonly arises in data analysis. Conveniently, that distribution's bell-shaped curve is a direct function of mean and standard deviation. We'll soon leverage SciPy, along with these two parameters, to better grasp the Normal curve's significance.

## 5.4 Summary

- A probability mass function maps inputted integer values to their probability of occurrence.
- The probability mass function for the Binomial distribution can be generated by calling `stats.binomial.pmf`.
- Mean is a good measure of a dataset's centrality. It minimizes the sum of squares relative to all the dataset. We can compute an unweighted mean by summing the dataset values and dividing by the dataset size. We can also compute a weighted mean by inputting`weights` array into `np.average`. The weighted mean of the Binomial distribution increases linearly with coin-flip count.
- Variance is a good measure of a dataset's dispersion. It equals the average squared distance of data point from the mean. The weighted variance of the Binomial distribution increases linearly with coin-flip count.
- The standard deviation is an alternate measure of dispersion. It equals the square-root of the variance. The standard deviation maintains the units used within a dataset.

# 6 *Making Predictions Using the Central Limit Theorem and SciPy*

***This section covers:***

- Analysis of the Normal curve using the SciPy library.
- Predicting mean and variance using the Central Limit Theorem.
- Predicting population properties using the Central Limit Theorem.

The Normal distribution is a bell-shaped curve that we introduced in Section Three. The curve arises naturally from random data sampling, due to the Central Limit Theorem. Previously, we noted how according to that theorem, repeatedly sampled frequencies will take the shape of a Normal curve. Furthermore, the theorem predicts a narrowing of that curve as the size of each frequency-sample goes up. In other words, the distributionâ€™s standard deviation should decrease as the sampling size grows larger.

The Central Limit Theorem lies at the heart of all classic statistics. In this section, we probe the theorem in great detail, using the computational power of SciPy. Eventually, we will learn how to leverage the theorem to make predictions from limited data.

## 6.1 Manipulating the Normal Distribution Using SciPy

In Section Three, we showed how random coin-flip sampling will produce that Normal curve. Let's generate a Normal distribution by plotting a histogram of coin-flip samples. Our input into the histogram will contain 100,000 head-count frequencies. Computing the frequencies will require us to sample a series of coin-flips 100,000 times. Each sample will contain an array of zeroes and ones representing 10,000 flipped coins. We'll refer to the array length as our sample size. If we use the sample size to divide the sum of values in the sample, we will compute the observed head-count frequency. Conceptually, this frequency is equal to simply taking the sample's mean. Below, we compute the head-count frequency of a single random sample and confirm its relationship to the mean. Please note that every data-point in the sample is drawn from the Bernoulli distribution.

**Listing 6.1 Computing head-count frequencies from the mean**

```
np.random.seed(0)
sample_size = 10000
sample = np.array([np.random.binomial(1, 0.5) for _ in range(sample_size)])
head_count = sample.sum()
head_count_frequency = head_count / sample_size
assert head_count_frequency == sample.mean()   ❶
```

❶ The head-count frequency is identical to the sample mean.

Of course, we can compute all 100,000 head-count frequencies in just a single line of code, as discussed in Section Three.

**Listing 6.2 Computing 100,000 head-count frequencies**

```
np.random.seed(0)
frequencies = np.random.binomial(sample_size, 0.5, 100000) / sample_size
```

Each sampled frequency equals the mean of 10,000 randomly flipped coins. Therefore, we'll rename our frequencies variable as `sample_means`. We'll then proceed to visualize our `sample_means` data as a histogram.

**Listing 6.3 Visualizing sample means in a histogram**

```
sample_means = frequencies
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

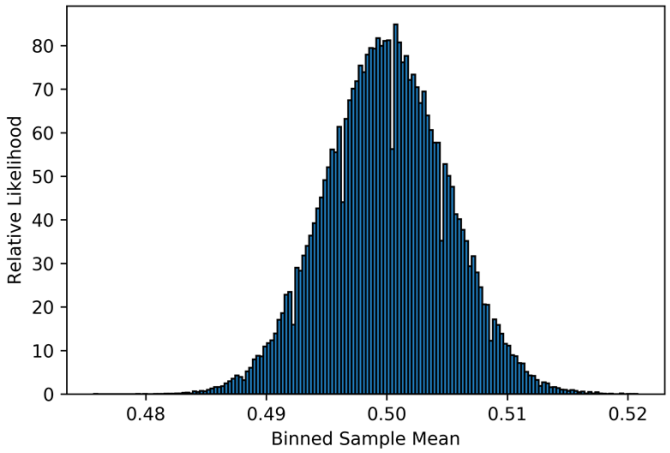


**Figure 6.1 A histogram of 100,000 sampled means plotted against their relative likelihoods. The histogram resembles a bell-shaped Normal distribution.**

The histogram is shaped like a Normal distribution. Let's calculate the distribution's mean and standard deviation.

## Listing 6.4 Computing mean and standard deviation of a histogram

```
mean_normal = np.average(bin_edges[:-1], weights=likelihoods)
var_normal = weighted_variance(bin_edges[:-1], likelihoods)
std_normal = var_normal ** 0.5
print(f"Mean is approximately {mean_normal:.2f}")
print(f"Standard deviation is approximately {std_normal:.3f}")
```

```
Mean is approximately 0.50
Standard deviation is approximately 0.005
```

The distribution's mean is approximately 0.5, and its standard deviation is approximately 0.005. In a Normal distribution, these values can be computed directly from the distribution's peak. We just need the peak's x-value and y-value coordinates. The x-value equals the distribution's mean. Meanwhile, the standard deviation is equal to the inverse of the y-value multiplied by $2Ї€^{1/2}$. These properties are derived from the mathematical analysis of the Normal curve. Let's re-compute the mean and standard deviation using just the coordinates of the peak.

## Listing 6.5 Computing mean and standard deviation from peak coordinates

```
import math
peak_x_value = bin_edges[likelihoods.argmax()]
print(f"Mean is approximately {peak_x_value:.2f}")
peak_y_value = likelihoods.max()
std_from_peak = (peak_y_value * (2* math.pi) ** 0.5) ** -1
print(f"Standard deviation is approximately {std_from_peak:.3f}")
```

```
Mean is approximately 0.50
Standard deviation is approximately 0.005
```

Additionally, we can compute the mean and standard deviation simply by calling

`stats.norm.fit(sample_means)`. This SciPy method returns the two parameters required to recreate the normal distribution formed by our data.

### Listing 6.6 Computing mean and standard deviation using `stats.norm.fit`

```
fitted_mean, fitted_std = stats.norm.fit(sample_means)
print(f"Mean is approximately {fitted_mean:.2f}")
print(f"Standard deviation is approximately  {fitted_std:.3f}")
```

```
Mean is approximately 0.50
Standard deviation is approximately 0.005
```

The computed mean and standard deviation can be used to reproduce our Normal curve. We can regenerate the curve calling `stats.norm.pdf(bin_edges, fitted_mean, fitted_std)`. SciPy's `stats.norm.pdf` method represents the **probability density function** of a Normal distribution. A probability density function is like a probability mass function but with one key difference: it does not return probabilities. Instead, it returns relative likelihoods. As discussed in Section Two, relative likelihoods are the y-axis values of a curve whose total area equals 1.0. Unlike probabilities, these likelihoods can equal values that are greater than 1.0. Despite this, the total area beneath a plotted likelihood interval still equals the probability of observing a random value within that interval.

Let's compute the relative likelihoods using `stats.norm.pdf`. Afterwards, we'll plot the likelihoods together with the sampled coin-flip histogram.

### Listing 6.7 Computing Normal likelihoods using `stats.norm.pdf`

```
normal_likelihoods = stats.norm.pdf(bin_edges, fitted_mean, fitted_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
         label='Normal Curve')
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True)   ❶
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

❶ The alpha parameter is used to make the histogram more transparent in order to better contrast the histogram with the plotted likelihood curve.

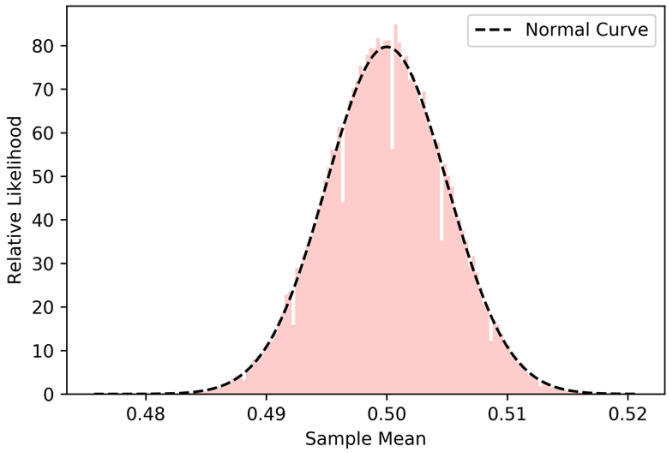


**Figure 6.2 A histogram overlaid with a Normal probability density function. The parameters defining the plotted Normal curve have been computed using SciPy. The plotted Normal curve fits nicely over the histogram.**

The plotted curve fits nicely over the histogram. The curve's peak sits at an x-axis position of 0.5, and rises to a y-axis position of approximately 80. As a reminder, the peak's x and y coordinates are a direct function of `fitted_mean` and `fitted_std`. In order to emphasize this import relationship, lets do a simple exercise. We'll shift the peak 0.01 units to the right, while also doubling the peak's height. How do we execute the shift? Well, the peak's axis is equal to the mean, so we'll adjust the input mean to `fitted_mean + 0.01`. Also, the peak's height is inversely proportional to the standard deviation. Therefore, inputting `fitted_std / 2` should double the height of the peak.

## Listing 6.8 Manipulating a Normal curve's peak coordinates

```
adjusted_likelihoods = stats.norm.pdf(bin_edges, fitted_mean + 0.01,
                                      fitted_std / 2)
plt.plot(bin_edges, adjusted_likelihoods, color='k', linestyle='--')
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

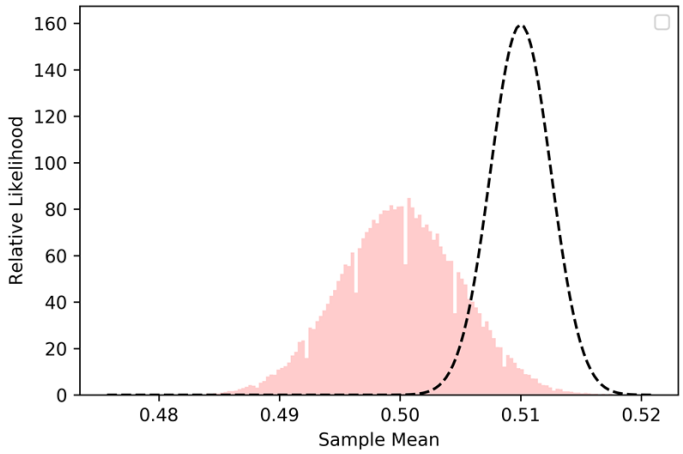


**Figure 6.3 A modified Normal curve whose center is .01 units to the right of the histogram. The peak of the curve is twice the height of the histogram's peak. These modifications were achieved by manipulating the histogram's mean and standard deviation.**

### 6.1.1 Comparing Two Sampled Normal Curves

SciPy allows us to explore and adjust the shape of the Normal distribution based on the inputted parameters. Also, the values of these input parameters depend on how we sample random data. Let's quadruple the coin-flip sample size to 40,000 and plot the resulting distribution changes. Below, we compare the plotted shapes of the old and updated Normal distributions, which we'll label as A and B, respectively.

**Listing 6.9 Plotting two curves with different samples sizes**

```
np.random.seed(0)
new_sample_size = 40000
new_head_counts = np.random.binomial(new_sample_size, 0.5, 100000)
new_mean, new_std = stats.norm.fit(new_head_counts / new_sample_size)
new_likelihoods = stats.norm.pdf(bin_edges, new_mean, new_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
         label='A: Sample Size 10K')
plt.plot(bin_edges, new_likelihoods, color='b', label='B: Sample Size 40K')
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

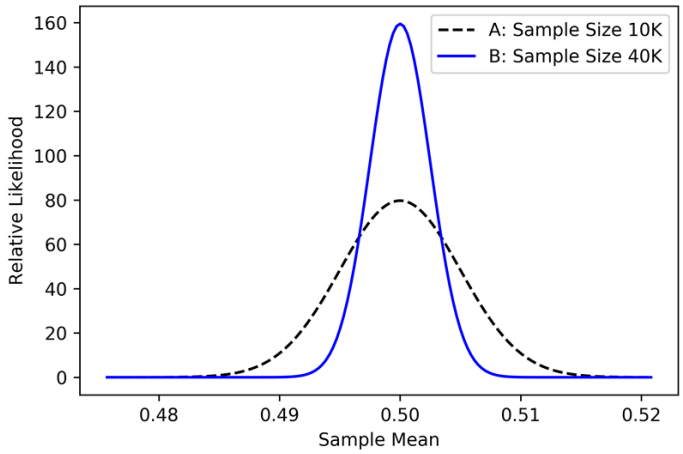


**Figure 6.4 Two Normal distributions generated using coin-flip data. Distribution A was derived using a sample size of 10,000 coin-flips per sample. Distribution B was derived using a sample size of 40,000 coin-flips per sample. Both distributions are centered around a mean-value of 0.5. However, Distribution B is much more narrowly dispersed around its center. Also, the peak of Distribution B is twice as high as the peak of Distribution A. Given the relationship between peak-height and variance, we can infer that the variance of Distribution B is one-fourth the variance of Distribution A.**

Both Normal distributions are centered around the sample mean value of 0.5. However, the distribution with the larger sample size is more narrowly centered around its peak. This is consistent with what we saw in Section Three. In that section, we observed that as the sample size increases, the peak-location stays constant while area around the peak contracts in width. The narrowing of the peak leads to a drop in the confidence interval range. A confidence interval represents the likely value-range covering the true probability of heads. Previously, we used confidence intervals to estimate the probability of heads from the x-axis head-count frequencies. Now, our x-axis represents the sample means, where every sample mean is identical to a head-count frequency. Thus, we can use our sample means to find the probability of heads. Also, as a reminder, all coin-samples were drawn from the Bernoulli distribution. We've recently shown that the mean of the Bernoulli distribution equals the probability of heads. Thus, not surprisingly, each sample's mean serves as an estimate of the true Bernoulli mean. Subsequently, we can interpret the confidence interval as likely value-range covering the true Bernoulli mean.

Let's calculate the 95% confidence interval for the true Bernoulli mean, using Normal Distribution B. Previously, we manually computed the 95% confidence interval by exploring the curve-area around the peak. However, SciPy allows us to automatically extract that range by calling `stats.norm.interval(0.95, mean, std)`. The method will return an interval that covers 95% of the area beneath the Normal distribution defined by `mean` and `std`.

**Listing 6.10 Computing a confidence interval using SciPy**

```
mean, std = new_mean, new_std
start, end = stats.norm.interval(0.95, mean, std)
print(f"The true mean of the sampled binomial distribution is between {start:.3f} and {end:.3f}")
```

```
The true mean of the sampled binomial distribution is between 0.495 and 0.505
```

We are 95% confident that the true mean of our sampled Bernoulli distribution is between 0.495 and 0.505. In fact, that mean is equal to exactly 0.5. We can confirm this using SciPy.

**Listing 6.11 Confirming the Bernoulli mean**

```
assert stats.binom.mean(1, 0.5) == 0.5
```

**SIDEBAR** **Common SciPy methods for Normal curve analysis**

- `stats.norm.fit(data)`:
  Returns the mean and standard deviation required to fit a Normal curve to `data`.
- `stats.norm.pdf(observation, mean, std)`:
  Returns the likelihood mapped to a single value of Normal curve defined by mean `mean` and standard deviation `std`.
- `stats.norm.pdf(observation_array, mean, std)`:
  Returns an array of Normal likelihoods. These are obtained by executing `stats.norm.pdf(e, mean, std)` on each element `e` of `observation_array`.
- `stats.norm.interval(x_percent, mean, std)`:
  Returns the `x_percent` confidence interal defined by mean `mean` and standard deviation `std`.

Let's now attempt to estimate the variance of the Bernoulli distribution based on the plotted Normal curves. At first glance, this seems like a difficult task. Though the means of the two plotted distributions remain constant at 0.5, their variances noticeably shift. The relative shift in variance can be estimated by comparing peaks. The peak of Distribution B is twice as high as the peak of Distribution A. This height is inversely proportional to the standard deviation. Therefore the standard deviation of Distribution B is half the standard deviation of Distribution A. Since the standard deviation is the square root of the variance, we can infer that the variance of Distribution B is one-fourth the variance of Distribution A. Thus, increasing the sample size four-fold from 10,000 to 40,000 leads to a four-fold decrease in the variance.

**Listing 6.12 Assessing shift in variance after increased sampling**

```
variance_ratio = (new_std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.2f}")
```

```
The ratio of variances is approximately 0.25
```

It appears that variance is inversely proportional to sample size. If so, than a four-fold decrease

in sample size from 10,000 to 2500 should generate a four-fold increase in the variance. Let's generate some head-counts using a sample size of 2500 and confirm if this is the case.

**Listing 6.13 Assessing shift in variance after decreased sampling**

```
np.random.seed(0)
reduced_sample_size = 2500
head_counts = np.random.binomial(reduced_sample_size, 0.5, 100000)
_, std = stats.norm.fit(head_counts / reduced_sample_size)
variance_ratio = (std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.1f}")
```

```
The ratio of variances is approximately 4.0
```

Yes! A four-fold decrease in the sample size leads to a four-fold increase in the variance. Thus, if we decrease the sample size from 10,000 to 1, we can expect 10,000-fold increase in the variance. That variance for a sample size of 1 should be equal to `(fitted_std ** 2) * 10000`.

**Listing 6.14 Predicting variance for a sample size of 1**

```
estimated_variance = (fitted_std ** 2) * 10000
print(f"Estimated variance for a sample size of 1 is {estimated_variance:.2f}")
```

```
Estimated variance for a sample size of 1 is 0.25
```

Our estimated variance for a sample size of 1 is 0.25. However, if the sample size were 1, then our `sample_means` array would simply be a sequence of randomly recorded ones and zeroes. By definition, that array would represent the output of the Bernoulli distribution. Therefore, running `sample_means.var` would approximate the variance of the Bernoulli distribution. Consequently, our estimated variance for a sample size of 1 equals the variance of the Bernoulli distribution. In fact, the Bernoulli variance does equal 0.25.

**Listing 6.15 Confirming the predicted variance for a sample size of 1**

```
assert stats.binom.var(1, 0.5) == 0.25
```

We have just used the Normal distribution to compute variance and mean of the Bernoulli distribution from which we sampled. Let's review the chain of steps that led to our results.

**A.** We sampled random ones and zeros from the Bernoulli distribution.

**B.** Each sequence of `sample_size` ones and zeros was grouped a single sample.

**C.** We computed a mean for every sample.

**D.** The sample means formed a Normal cuve. We found its mean and standard deviation.

**E.** The mean of the Normal curve equaled the mean of the Bernoulli distribution.

**F.** The variance of the Normal curve multiplied by the sample size equaled the variance of the

Bernoulli distribution.

What if we had sampled from some other non-Bernoulli distribution? Would we still be able to estimate the mean and variance through random sampling? Yes we would! According the Central Limit Theorem, sampling mean-values from almost any distribution will produce a Normal curve. This includes distributions such as:

- The **Poisson distribution** (`stats.poisson.pmf`). Commonly used to model:
  - Number of customers who visit a store per hour.
  - Number clicks on an online-ad per second.
- The **Gamma distribution** (`scipy.stats.gamma.pdf`). Commonly used to model:
  - Monthly rainfall in a region.
  - Banking loan defaults based on loan size.
- The **Log-Normal** distribution (`scipy.stats.lognorm.pdf`). Commonly used to model:
  - Fluctuating stock prices.
  - Incubation periods of infectious diseases.
- Countless distributions occurring in nature that haven't yet been assigned a name.

**WARNING** Under edge-case circumstances, sampling will not produce a Normal curve. This is occasionally true of the Pareto distribution, which is used to model income inequality.

Once we've sampled a Normal curve, we can use it to analyze the underlying distribution. The mean of the Normal curve will approximate the mean of the underlying distribution. Also, the variance of the Normal curve multiplied by the sample size will approximate the variance of the underlying distribution.

**NOTE** In other words, if we sample from a distribution with variance `var`, we will obtain a Normal curve with variance `sample_size / var`. As the sample size approaches infinity, the variance of the Normal curve will approach zero. At zero-variance the Normal will collapse into single vertical line positioned at the mean. This property can be used to derive the Law of Large Numbers, which we introduced in Section Two.

The relationship between a Normal distribution produced by sampling and the properties of the underlying distribution serves as a foundation for all statistics. Using that relationship, we can leverage the Normal curve to estimate both mean and variance of almost any distribution through random sampling.

## 6.2 Determining Mean and Variance of a Population through Random Sampling

Suppose we are tasked with finding the average age of people living in a town. The town's population is exactly 50,000 people. Below, we'll simulate the ages of the townsfolk using the `np.random.randint` module.

**Listing 6.16 Generating a random population**

```
np.random.seed(0)
population_ages = np.random.randint(1, 85, size=50000)
```

How do we compute the average age of the residents? One cumbersome approach would be to take a census of every resident in the town. We could record all 50,000 ages and afterwards compute their mean. That mean would be exact because it covers the entire population. Statisticians define this exact mean as the **population mean**. Furthermore, the variance of an entire population is referred to as the **population variance**. Let's quickly compute the population mean and population variance of our simulated town.

**Listing 6.17 Computing population mean and variance**

```
population_mean = population_ages.mean()
population_variance = population_ages.var()
```

Computing the population mean is easy when we have simulated data. However, obtaining that data in real life would be incredibly time consuming. We would have to interview all 50,000 people. Without more resources, interviewing the whole town would be borderline impossible.

A simpler approach would be to interview 10 randomly chosen people in the town. We'd record the ages from this random sample, and afterwards compute the sample mean. Let's simulate the sampling process by drawing 10 random ages from the `np.random.choice` module. Executing `np.random.choice(age, size=sample_size)`, will return an array of 10 randomly sampled ages. After sampling is complete, we will compute the mean of the resulting 10 element array.

**Listing 6.18 Simulating 10 interviewed people**

```
np.random.seed(0)
sample_size = 10
sample = np.random.choice(population_ages, size=sample_size)
sample_mean = sample.mean()
```

Of course, our sample mean is likely to be noisy and inexact. We can measure that noise by finding the percent difference between `sample_mean` and `population_mean`.

**Listing 6.19 Comparing sample mean to population mean.**

```
percent_diff = lambda v1, v2: 100 * abs(v1 - v2) / v2
percent_diff_means = percent_diff(sample_mean, population_mean)
print(f"There is a {percent_diff_means:.2f} percent difference between means.")
```

```
There is a 27.59 percent difference between means
```

There is approximately a 27% difference between the sample mean and the population mean. Clearly, our sample is insufficient to estimate the mean. We'll need to gather more samples. Perhaps we should raise our sampling to cover 1,000 residents of the town. This seems like a reasonable objective that is preferable to surveying all 50,000 residents. Unfortunately, interviewing 1,000 people will still be very time consuming. Even if we assume an idealistic interview rate of 2 people per minute, it will still take us 8 hours to reach our interview goal. Conceivably, we can optimize our time by parallelizing the interview process. We can post an ad in the local paper asking for 100 volunteers. Each volunteer will survey 10 random people, in order to sample their ages. Afterwards, every volunteer will send us a computed sample mean. Thus, we will receive 100 sample means, representing 1,000 interviews total.

**NOTE** Each volunteer will send us a sample mean. Conceivably, the volunteers could send full data instead. However, the sample means are preferable, for the following reasons. First of all, the means don't require as much memory storage as the full data. Second of all, the means can be plotted as a histogram, in order to check the quality of our sample size. If that histogram does not approximate a Normal curve, then additional samples will be required.

Lets simulate our surveying process below.

**Listing 6.20 Computing sample means across 1,000 people**

```
np.random.seed(0)
sample_means = [np.random.choice(population_ages, size=sample_size).mean()
                for _ in range(100)]
```

According to the Central Limit Theorem, a histogram of sample means should resemble the Normal distribution. Furthermore, the mean of the Normal distribution should approximate the population mean. We can confirm this is the case by fitting the sample means to a Normal distribution.

### Listing 6.21 Fitting sample means to a Normal curve

```
likelihoods, bin_edges, _  = plt.hist(sample_means, bins='auto', alpha=0.2,
                                      color='r', density=True)
mean, std = stats.norm.fit(sample_means)
normal_likelihoods = stats.norm.pdf(bin_edges, mean, std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--')
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

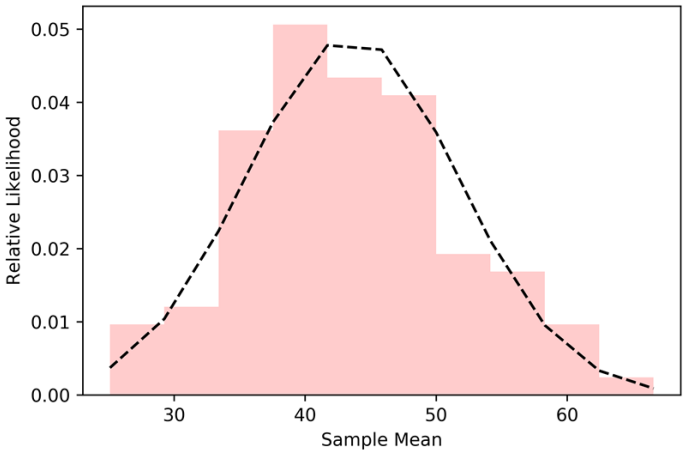


**Figure 6.5 A histogram computed from 100 age-samples. That histogram is overlaid with its associated Normal distribution. The Normal distribution's mean and standard deviation parameters have been derived from the plotted histogram data.**

Our histogram is not very smooth, because we've only processed 100 data points. However, the histogram's shape still approximates a Normal distribution. We'll print that distribution's mean and compare it to the population mean.

### Listing 6.22 Comparing Normal mean to population mean

```
print(f"Actual population mean is approximately {population_mean:.2f}")
percent_diff_means = percent_diff(mean, population_mean)
print(f"There is a {percent_diff_means:.2f}% difference between means.")
```

```
Actual population mean is approximately 42.53
There is a 2.17% difference between means.
```

Our estimated mean of the ages is roughly 43. The actual population mean is roughly 42.5. There is an approximately 2% difference between the estimated mean and the actual mean. Thus our result, while not perfect, is still a very good approximation of the actual average age within the town.

Now, we'll briefly turn our attention to the standard deviation computed from the Normal distribution. Squaring the standard deviation will produce the distribution's variance. According to the Central Limit Theorem, we can leverage that variance to estimate the variance of ages in

the town. We simply need to multiply the computed variance by sample size.

**Listing 6.23 Estimating the population variance**

```
normal_variance = std ** 2
estimated_variance = normal_variance * sample_size
```

Let's compare the estimated variance to the population variance.

**Listing 6.24 Comparing estimated variance to population variance**

```
print(f"Estimated variance is approximately {estimated_variance:.2f}")
print(f"Actual population variance is approximately {population_variance:.2f}")
percent_diff_var = percent_diff(estimated_variance, population_variance)
print(f"There is a {percent_diff_var:.2f} percent difference between variances.")
```

```
Estimated variance is approximately 576.73
Actual population variance is approximately 584.33
There is a 1.30 percent difference between variances.
```

There is approximately a 1.3% difference between the estimated variance and the population variance. We've thus approximated the town's variance to a relative accurate degree, while only sampling 2% of the people living in the town. Our estimates may not be 100% perfect. However, the amount of time we saved more than makes up for that miniscule drop in accuracy.

So far, we've only used the Central Limit Theorem to estimate the population mean and variance. However, the power of the theorem goes beyond mere estimation of distribution parameters. We can use the Central Limit Theorem to make predictions about people.

## 6.3 Making Predictions Using Mean and Variance

Let us now consider a new scenario, in which we analyze a fifth grade classroom. Mrs. Mann is a brilliant fifth grade teacher. She has spent 25 years inspiring a love of learning her students. Her classroom holds 20 students. Thus, over the years, she has taught 500 students total.

> **NOTE** We are assuming that each year, Mrs. Mann teaches exactly 20 students. Of course, in real life, classroom size might fluctuate from year-to-year.

Her students frequently outperform other fifth-graders in the state. That performance is measured using scholastic assessment exams, which are administered to all fifth graders every year. These exams are graded, from 0 to 100. All grades can be accessed by querying the state assessment database. However, due to poor database design, the queryable exam records do not specify the year when each exam was taken.

Imagine we're tasked with addressing the following question; has Mrs. Mann ever taught a class that collectively aced the assessment exam? More specifically, has she ever taught a class of 20 students whose mean assessment grade was above 89%?

To answer that question, assume that we've queried the state database. We've obtained grades for all of Mrs. Mann's past students. Of course, a lack of temporal information prevents us from grouping the grades by year. Thus, we cannot simply scan the records for a yearly mean above 89%. However, we can still compute the mean and variance across the 500 total grades. Lets suppose the mean is equal to 84, and the variance is equal to 25. We'll refer to these values as the population mean an population variance, since they cover the entire population of students who've ever been taught by Mrs. Mann.

**Listing 6.25 Population mean and variance of recorded grades**

```
population_mean = 84
population_variance = 25
```

Lets model the yearly test results of Mrs. Mann's class as a collection of 20 grades randomly drawn from a distribution with mean `population_mean` and variance `population` variance. This model is simplistic. It makes several extreme assumptions, such as:

- Performance of each student in the class does not depend on any other student.
  - In real life, this assumption doesn't always hold. For instance, disruptive students can negatively impact the performance of others.
- Exams are equally difficult every year.
  - In real life, standardized exams can be adjusted by government officials.
- Local economic factors are negligable.
  - In real life, fluctuating economies impact school district budgets, and as well as student home environments. These external factors can affect the quality of grades.

Our simplifications might impact prediction accuracy. However, given our limited data, we have little choice in the matter. Statisticians are frequently forced to make such compromises in order to address otherwise intractable problems. Most of the time, their simplified predictions still reasonably reflect real-world behaviors.

Given our simple model, we can sample a random batch of 20 grades. What is the probability that the grades will have a mean of at-least 90? This probability can easily be computed using the Central Limit Theorem. According the theorem, the likelihood distribution of mean grades will resemble a Normal curve. The mean of the Normal curve will equal `population_mean`. The variance of the Normal curve will equal `population_variance` divided by our sample size of 20 students. Taking the square root of that variance will produce the standard deviation of the curve, which statisticians call the **Standard Error of the Mean**, or **SEM** for short. By definition, the SEM equals the population standard deviation divided by the square root of the sample size.

We'll compute the curve parameters, and plot the Normal curve below.

**Listing 6.26 Plotting a Normal curve using mean and SEM**

```
mean = population_mean
sem = (population_variance / 20) ** 0.5
grade_range = range(101)
normal_likelihoods = stats.norm.pdf(grade_range, mean, sem)
plt.plot(grade_range, normal_likelihoods)
plt.xlabel('Mean Grade of 20 Students (%)')
plt.ylabel('Relative Likelihood')
plt.show()
```

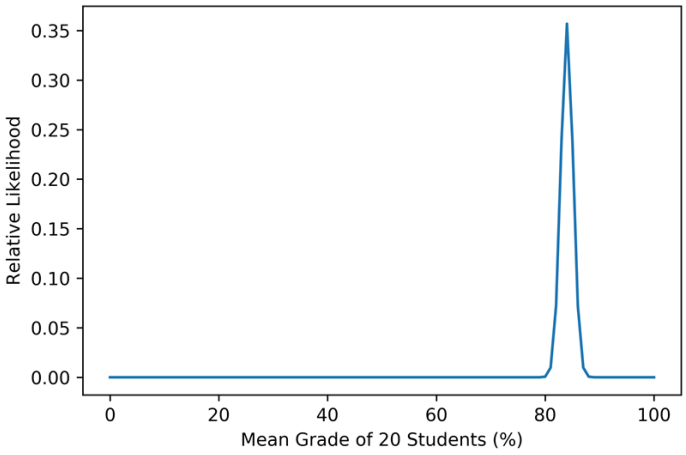


**Figure 6.6 A Normal distribution derived from the population mean and the standard error from the mean (SEM). The SEM is equal to the standard deviation divided by the square root of the sample size. The area beneath the plotted curve can be used to calculate probabilities.**

The area beneath the plotted curve approaches zero at values higher than 89%. That area is also equal to the probability of a given observation. Therefore, the probability of observing a mean grade that's at or above 90% is incredibly low. Still, to be sure, we'll need to compute the actual probability. Thus, we'll need to somehow accurately measure the area under the Normal distribution.

### *6.3.1 Computing the Area Beneath a Normal Curve*

In Section Three, we had computed areas under histograms. Determining these areas proved easy. All histograms, by definition, are comprised of small rectangular units. Thus, we could simply sum the areas of rectangles composing a specified interval. The total sum equaled the interval's area. Unfortunately, our smooth Normal curve does not decompose into rectangles. So how do find its area? One simple solution is to subdivide the Normal curve into small, trapezoidal units. This ancient technique referred to as the **trapezoid rule**. A trapezoid is a four-sided polygon with 2 parallel sizes. An trapezoid's area is equal to the sum of these parallel sides, multiplied half the distance between them. Summing over multiple consecutive trapezoid areas will approximate the area over an interval.

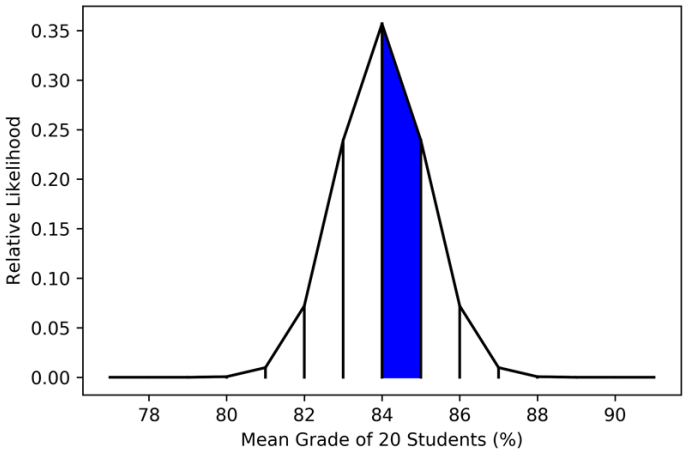


**Figure 6.7 A Normal distribution, subdivided into trapezoidal regions. The lower-left corner of each trapezoid is located at an x-coordinate of i. The parallel sizes of each trapezoid are defined by stats.norm.pdf(i), and stats.norm.pdf(i + 1). These parallel sizes are 1 unit apart. The area of the trapezoid at position 84 has been shaded in. That area is equal to (stats.norm.pdf(84) + stats.norm.pdf(85) / 2). Summing trapezoid areas across an interval-range will approximate the total area over that interval.**

The trapezoid rule is very easy to execute in just a few lines of code. Alternatively, we can utilize NumPy's `np.trapz` method to take the area of an inputted array. Lets apply the trapezoid rule to our Normal distribution. We'll want to test how well the rule approximates the total area covered by `normal_likelihoods`. Ideally, that area will approximate 1.0.

## Listing 6.27 Approximating the area using the trapezoid rule

```
total_area = np.sum([normal_likelihoods[i: i + 2].sum() / 2   ❶
                     for i in range(normal_likelihoods.size - 1)])

assert total_area == np.trapz(normal_likelihoods)   ❷
print(f"Estimated area under the curve is {total_area}")
```

❶ The area of each trapezoid equals the sum of 2 consecutive likelihoods divided by 2. The x-coordinate distance between the trapezoid sides is 1, so it doesn't factor into our calculations.

❷ Please note that NumPy executes the trapezoid rule in mathematically more efficient manner.

```
Estimated area under the curve is 1.0000000000384808
```

The estimated area is very close to 1.0, but its not exactly equal to 1.0. In fact, it is slightly greater than 1.0. If we're willing to tolerate this minor imprecision, then our trapezoid rule output is acceptable. Otherwise, we'll need a precise solution for the area of a Normal distribution. That precision is provided by SciPy. We can access a mathematically exact solution using the `stats.norm.sf` method. The method represents the **survival function** of the Normal curve. The survival function equals the distribution's area over an interval that's greater than some `x`. In

other words, the survival function is the exact solution to the area approximated by `np.trapz(normal_likelihoods[x:])`. Thus, we can expect `stats.norm.sf(0, mean, sem)` to equal 1.0.

**Listing 6.28 Computing the total area using SciPy**

```
assert stats.norm.sf(0, mean, sem) == 1.0   ❶
```

❶ Theoretically, the lower bound x-value of a Normal curve stretches into negative infinity. Therefore, this actual area is microscopically smaller than 1.0. However, the difference is so negligible, that SciPy is unable to detect it. For our intents and purposes, we can treat the precise area as 1.0.

Similarly, we expect `stats.norm.sf(mean, mean, sem)` to equal 0.5. This is because the mean perfectly splits the Normal curve into 2 equal halves. Thus, the interval of values beyond the mean will cover half the area of the Normal curve. Meanwhile, we expect `np.trapz(normal_likelihoods[mean:])` to approximate but not fully equal 0.5. Lets confirm below.

**Listing 6.29 Inputting the mean into the survival function**

```
assert stats.norm.sf(mean, mean, sem) == 0.5
estimated_area = np.trapz(normal_likelihoods[mean:])
print(f"Estimated area beyond the mean is {estimated_area}")
```

```
Estimated area beyond the mean is 0.5000000000192404
```

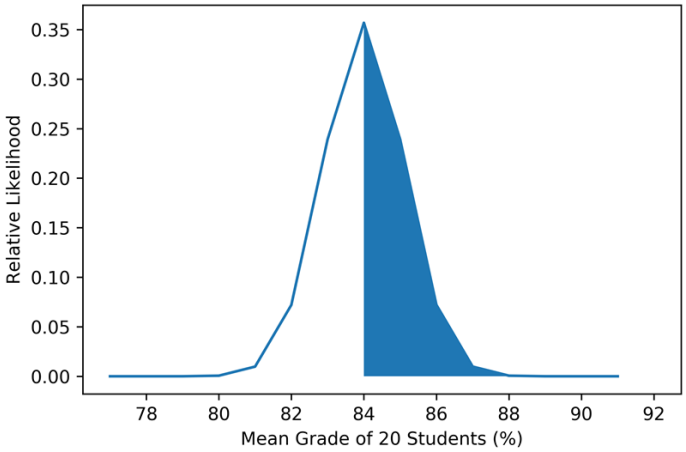


**Figure 6.8 We've highlighted the area denoted by stats.norm.sf(mean, mean, sem).That area covers an interval of values that are greater than or equal to the mean. The shaded area equals half the total area of the curve. Its exact value is 0.5.**

**SIDEBAR** **Common Methods for Curve Area Measurement**

- `numpy.trapz(array)`:
  Executes the trapezoid rule to estimate the area of `array`. The x-coordinate difference the array elements is set to 1.
- `numpy.trapz(array, dx=dx)`:
  Executes the trapezoid rule to estimate the area of `array`. The x-coordinate difference between thee array elements is set to `dx`.
- `stats.norm.sf(x_value, mean, std)`:
  Returns the area beneath a Normal curve, covering an interval that's greater than or equal to `x_value`. The mean and standard deviation of the Normal curve are set to `mean` and `std`, respectively.
- `stats.norm.sf(x_array, mean, std)`:
  Returns an array of areas. These are obtained by executing `stats.norm.sf(e, mean, std)` on each element `e` of `x_array`.

Now, lets execute `stats.norm.sf(90, mean, sem)`. This will return the area over an interval of values lying beyond 90%. The area represents the likelihood of 20 students jointly acing an exam.

**Listing 6.30 Computing the probability of a good collective grade**

```
area  = stats.norm.sf(90, mean, sem)
print(f"Probability of 20 students acing the exam is {area}")
```

```
Probability of 20 students acing the exam is 4.012555633463782e-08
```

As expected, the probability is low.

### 6.3.2 Interpreting the Computed Probability

The probability of the students acing the exam is approximately 1 in 25 million. Also, the exam is held just once a year. Consequently, it would take about 25 million years for a random arrangement of students to achieve that level of performance. Meanwhile, Mrs. Mann has only been teaching for 25 years. This represents a million-fold difference in magnitude. What are the odds of her presiding over a classroom with an average grade of at-least 90%? Practically zero. We can conclude that such a classroom never existed!

**NOTE** The actual odds can be computed by running `1 - stats.binom.pmf(0, 25, stats.norm.sf(90, mean, sem))`. Can you figure out why?

Of course, we could be wrong. Perhaps a group of a very talented fifth-graders randomly wound

up in the same classroom. This is highly unlikely, but nonetheless its possible. Also, our simple calculations didn't factor in shifts in exam difficultly. What if the exam gets easier every year? This would invalidate our treatment of the grades as a randomly drawn sample.

It seems our final conclusion is imperfect. We did the best we could, given what we knew. Still, some uncertainty remains. In order to eliminate that uncertainty, we'd need the missing dates for the graded exams. Unfortunately, that data was not provided. Quite commonly, statisticians are forced to make consequential decisions from limited records. Consider briefly the following 2 scenarios:

- A coffee farm ships 500 tons of coffee beans per year, in 5 pound bags. On average, 1% of the beans are moldy, with a standard deviation of 0.2%. The FDA permits a maximum of 3% moldy beans per bag. Does there exist a bag that violates the FDA's requirements?
    - We can apply the Central Limit Theorem if we assume that mold growth is independent of time. However, mold could grow more rapidly in the humid summer months. Regrettably, we lack the records to confirm.
- A seaside town is building a seawall to defend against tsunamis. According to historical data, the average tsunami height is 23 feet, with a standard deviation of 4 feet. The planned wall height is 33 feet. Is that height sufficient to protect the town?
    - Its tempting to assume that the tsunami average height will remain unchanged from year-to-year. However, certain studies indicate that climate change is causing sea-levels to rise. Climate change might lead to more powerful tsunamis in the future. Regrettably, the scientific data is not conclusive enough to know for sure.

In both scenarios, we must make important decisions by relying on statistical techniques. These techniques depend on certain assumptions that might not actually hold. Consequently, we must exercise great caution when we draw conclusions from incomplete information.

In the coming section, we'll continue to explore both the risks and advantages of making decisions based on limited data.

## 6.4 Summary

- A Normal distribution's mean and standard deviation are determined by the position of its peak. The mean is equal to the x-coordinate of the peak.Meanwhile, the standard deviation is equal to the inverse of the y-coordinate multiplied by 2Ï€$^{1/2}$.
- A probability density function maps inputted float values to their likelihood weights. Taking the area underneath that curve will produce a probability.
- Repeatedly sampling the mean from almost any distribution will produce a Normal curve. The mean the Normal curve will approximate the mean of underlying distribution. Also, the variance of the Normal curve multiplied by the sample size will approximate the variance of the underlying distribution.
- The Standard Error of the Mean (SEM) equals the population standard deviation divided by the square root of the sample size. Consequently, dividing the population variance by the sample size, and subsequently taking the square root will also generate the SEM. The SEM, coupled together with the population mean, allows us to compute the probability of observing certain sample combinations.
- The trapezoid rule allows us to estimate the area under a curve, by decomposing that curve into trapezoidal units. Afterwards, we simply sum over the areas of each trapezoid.
- A survival function measures a distribution's area over an interval that's greater than some x.
- We must cautiously consider our assumptions while making inferences from limited data.

# 7 Statistical Hypothesis Testing

***This section covers:***

- Comparing sample means to population means.
- Comparing means of 2 distinct samples.
- What is statistical significance?
- Common statistical errors, and how to avoid them.

Countless ordinary people are forced to make hard choices every day. This is especially true of jurors in the American justice system. Jurors preside over a defendant's fate during a trial. They consider the evidence, and then decide between 2 competing hypotheses:

**A.** The defendant is innocent.

**B.** The defendant is guilty.

The 2 hypotheses are not weighted equally. The defendant is presumed to be innocent until proven guilty. Thus, the jurors assume that the innocence hypothesis is true. They can only reject the innocence hypothesis if the prosecution's evidence is convincing. Yet rarely is the evidence is 100% perfect. Some doubt to defendant's guilt remains. That doubt is factored into the legal process. The jury is instructed to accept the innocence hypothesis if there is "reasonable doubt" of the defendant's guilt. They can only reject the innocence hypothesis if the defendant appears guilty "beyond a reasonable doubt".

Reasonable doubt is an abstract concept that's hard to precisely define. Nonetheless, we can distinguish between reasonable and unreasonable doubt across a range of real-world scenarios. Consider the following 2 trial cases:

**A.** DNA evidence links the defendant directly to the crime. There is a 1 in a billion chance that

the DNA does not belong to the defendant.

**B.** Blood-type evidence links the defendant directly to the crime. The is a 1 in 15 chance that the blood does not belong to the defendant.

In the first scenario, the jury cannot be 100% certain of the defendant's guilt. There is a 1 in a billion chance that an innocent defendant is on trial. Such circumstances, however, are incredibly unlikely. Its not reasonable to assume that they actually happened. Some doubt to the defendant's guilt might linger, but it will not be reasonable doubt. Thus, the jury should reject the innocence hypothesis.

Meanwhile, in the second scenario, the doubt is much more prevalent. 1 in 15 people share the same blood-type as the defendant. Its reasonable to assume that someone else could have been present at the crime-scene. While the jurors might doubt the defendant's innocence, they will also reasonably doubt the defendant's guilt. Thus, the jurors can't reject the innocence hypothesis unless additional proof of guilt is offered.

In our 2 scenarios, the jurors carrying out a **statistical hypothesis test**. Generally, hypothesis tests allow statisticians to choose between 2 competing hypotheses, both of which arise from uncertain data. One of the hypotheses gets accepted or rejected based on a measured level of doubt. In this section, we'll explore several well-known statistical hypothesis testing techniques. We'll begin with a very simple test. The test will measure whether a sample mean noticeably deviates from an existing population.

## 7.1 Assessing the Divergence Between Sample Mean and Population Mean

In Section Six, we used statistics to analyze a single fifth grade classroom. Now, lets imagine a scenario where we analyze every fifth grade classroom in North Dakota. One spring day, all fifth graders in the state are given the same exact assessment exam. The exam grades are fed into North Dakota's assessment database. The population mean and variance are computed across all grades in the state. According to the records, the population mean is 80, and the population variance is 100. Let's quickly store these values for later use.

**Listing 7.1 Population mean and variance of North Dakota's grades**

```
population_mean = 80
population_variance = 100
```

Now, suppose we travel to South Dakota. In South Dakota, we hear rumors of a single fifth-grade class whose mean exam grade equaled 84%. We investigate and confirm that a South Dakotan class of 18 students has outperformed North Dakota's population mean by 4 percentage points. This is pretty impressive. Perhaps fifth graders in South Dakota were somehow better prepared than their North Dakota counterparts? If so, then North Dakota should really

incorporate Southern teaching methods into the curriculum. The change in curriculum would be a long, expensive process, but the pay-off to the students would be worth it. Of course, it's also possible that the observed exam difference is a mere statistical fluke. Which is it? Is that one South Dakotan classroom somehow special? Or was their high performance mean a random anomaly? We'll the probe the question using hypothesis testing.

Right now, we face 2 rival possibilities. First, its possible that overall student population is identical across the neighboring states. In other words, a typical South Dakota classroom is no different from a typical North Dakota classroom. Under such circumstances, South Dakota's population mean and variance values would be indistinguishable from the distribution parameters of its northern neighbor. Statisticians refer to this hypothetical parameter equivalency as the **null hypothesis**. If the null hypothesis is true, than our high-performing South Dakota classroom is simply an outlier, and doesn't represent the actual mean.

Alternatively, its feasible that our South Dakota is not an outlier. If that's the case, than the high performance mean is representative of South Dakota's general population. Consequently, the state's mean and variance values would differ from North Dakota's population parameters. Statisticians call this the **alternative hypothesis**. If the null hypothesis is false, then the alternative hypothesis must be true. If our alternative hypothesis is true, then we'll invest our resources into improving North Dakota's education system. However, we must first show that the null hypothesis is unlikely to be true. We will attempt to show that by leveraging the Central Limit Theorem.

Let's temporarily assume that the null hypothesis is true. If the hypothesis is true, then South Dakota's population mean is equal to North Dakota's population mean. Also, South Dakota's population variance is equal to North Dakota's population variance. Consequently, we can model our 18-student classroom as a random sample taken from a Normal distribution. That distribution's mean will equal `population_mean`. Meanwhile, its standard deviation will equal the SEM, defined as `(population_variance / 18) ** 0.5`.

**Listing 7.2 Normal curve parameters if the null hypothesis is true**

```
mean = population_mean
sem = (population_variance / 18) ** 0.5
```

If the null hypothesis is true, then the probability of encountering an average exam grade of at-least 84% is equal to `stats.norm.sf(84 mean, sem)`. We'll now print out that probability.

**Listing 7.3 Finding the probability of a high-performance grade**

```
prob_high_grade = stats.norm.sf(84, mean, sem)
print(f"Probability of an average grade >= 84 is {prob_high_grade}")
```

```
Probability of an average grade >= 84 is 0.044843010885182284
```

Our computed probability approximately 0.044. Therefore, if the null hypothesis is true, then a random South Dakotan classroom will obtain an average grade of at-least 84% approximately 4.4% of the time. That probability is quite low. The grade difference relative to the population mean of 80% appears to be extreme. Yet is it really extreme? In Section One, we had asked a similar question when we examined the likelihood of observing 8 heads out 10 coin-flips. In our coin analysis, we summed the probability of over-performance with the probability of under-performance. In other words, we summed the probability of observing 8 or more heads with the probability of observing 2 heads or less. Here, our dilemma is identical. The probability of observing a high grade-average is insufficient to evaluate extremeness. We must also compute the probability of observing an equally extreme low grade-average. Therefore, we need to compute the probability of observing a sample mean that is at-least 4 percentage points below the population mean of 80%.

We will now compute the probability of observing an exam-average that's less than or equal to 76%. The calculation can be carried out with SciPy's `stats.norm.cdf` method. The method computes the **cumulative distribution function** of the Normal curve. A cumulative distribution function is the direct opposite of the survival function. Applying `stats.norm.cdf` to `x` returns the area under a Normal curve that ranges from negative infinity to x.

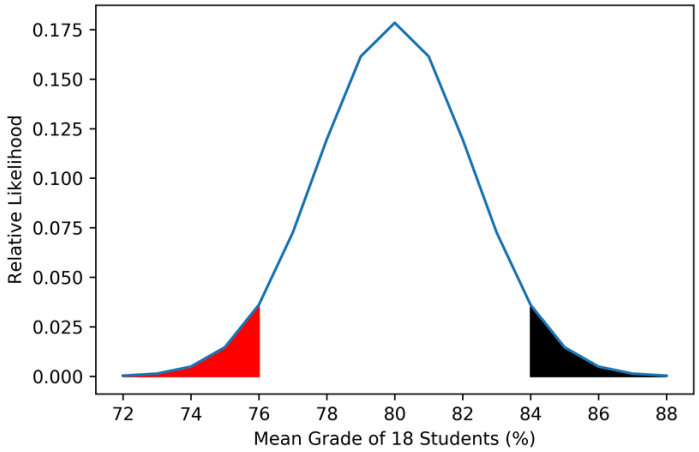


**Figure 7.1 Two areas are highlighted beneath a Normal curve. The left-most area covers all x-values that are less than or equal to 76%. We can compute that area using the cumulative distribution function. To execute the function, we simply need to call stats.norm.cdf(76, mean, sem). Meanwhile, the right-most area covers all x-values that are at-least 84%. We can compute that area using the survival function. To execute the function, we simply need to call stats.norm.sf(84, mean, sem).**

We'll now use `stats.norm.cdf` to find the probability of observing an unusually low average grade.

### Listing 7.4 Finding the probability of a low-performance grade

```
prob_low_grade = stats.norm.cdf(76, mean, sem)
print(f"Probability of an average grade <= 76 is {prob_low_grade}")
```

```
Probability of an average grade <= 76 is 0.044843010885182284
```

It appears that `prob_low_grade` is exactly equal to `prob_high_grade`. This equality arises from the symmetric shape of the Normal curve. The cumulative distribution and the survival function are mirror images of each other. Their reflection occurs across the mean. Thus, `stats.norm.sf(mean + x, mean, sem)` will always equal `stats.norm.cdf(mean - x, mean, sem)` for any input `x`. Below, we will demonstrate this symmetry. We will also visualize both functions in order to reveal their reflection across a vertically-plotted mean.

### Listing 7.5 Comparing the survival and the cumulative distribution functions

```
for x in range(-100, 100):
    sf_value = stats.norm.sf(mean + x, mean,  sem)
    assert sf_value == stats.norm.cdf(mean - x, mean, sem)

plt.axvline(mean, color='k', label='Mean', linestyle=':')
x_values = range(60, 101)
plt.plot(x_values, stats.norm.cdf(x_values, mean, sem),
         label='Cumulative Distribution')
plt.plot(x_values, stats.norm.sf(x_values, mean, sem),
         label='Survival Function', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()

plt.axvline(mean, color='k', label='Mean', linestyle=':')
x_values = range(60, 101)
plt.plot(x_values, stats.norm.cdf(x_values, mean, sem),
         label='Cumulative Distribution')
plt.plot(x_values, stats.norm.sf(x_values, mean, sem),
         label='Survival Function', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

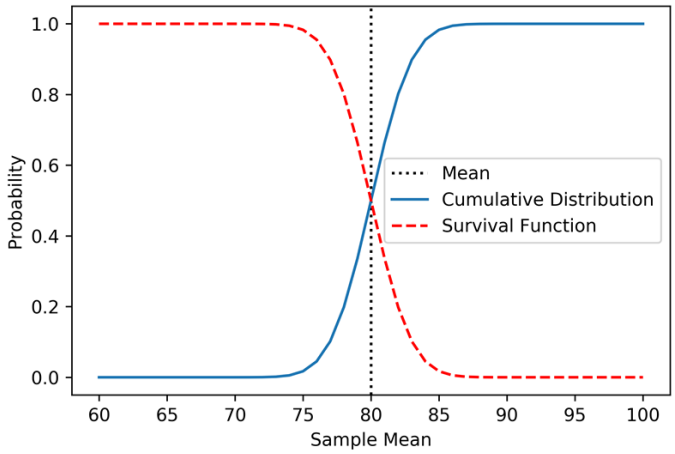


**Figure 7.2 A cumulative distribution function of a Normal distribution plotted together with the survival function. The cumulative distribution function and the survival function are mirror images of each other. They are reflected across the Normal curve's mean, which is plotted as a vertical line.**

Now we are ready to sum `prob_high_grade` and `prob_low_grade`. Due to symmetry, that sum equals `2 * prob_high_grade`. Conceptually, the sum represents the probability of observing an extreme deviation from the population mean when the null hypothesis is true. Statisticians refer to this null hypothesis-driven probability as the **p-value**. Let's print the p-value arising from our data.

### Listing 7.6 Computing the null hypothesis driven p-value

```
p_value = prob_low_grade + prob_high_grade
assert p_value == 2 * prob_high_grade
print(f"The p-value is {p_value}")
```

```
The p-value is 0.08968602177036457
```

Our p-value is approximately 0.089. If the null hypothesis is true, than there is approximately a 9% chance that the observed extreme grade-average will appear by random. While 9% is not a particularly high value, it is not particularly low either. Its therefore reasonable to suppose that the null hypothesis might be true and that the extreme test-average occurred purely by random chance. We haven't definitively proved this, but our calculations raise serious doubts about whether the total restructuring of North Dakota's educational system is worth the resource investment. What if the average of the South Dakotan class had equaled 85%, not 84%? Will that 1-percent shift influence our p-value output? Let's find out.

**Listing 7.7 Computing the p-value for an adjusted sample mean**

```
def compute_p_value(observed_mean, population_mean, sem):
    mean_diff = abs(population_mean - observed_mean)
    prob_high = stats.norm.sf(population_mean + mean_diff, population_mean, sem)
    return 2 * prob_high

new_p_value = compute_p_value(85, mean, sem)
print(f"The updated p-value is {new_p_value}")
```

```
The updated p-value is 0.03389485352468927
```

A tiny increase in the average grade has caused 3-fold decrease in the p-value. The updated p-value equals approximately 0.033. Hence, there's only a 3.3% chance of observing an average test grade that's at-least as extreme as 85%, when the null hypothesis is true. That 3.3% chance is quite low. We might therefore be tempted to reject the null hypothesis, and to assume that alternative hypothesis better represents our data. Should we do this? Should we invest our time and money into revamping North Dakota's school system?

This is not an easy question to answer. Generally, statisticians tend to reject the null hypothesis if the p-value is less than or equal to 0.05. The threshold of 0.05 is called the **significance level**, and p-values below that threshold are deemed to be **statistically significant.** However, that 0.05 threshold is just an arbitrary value that serves as a rule-of-thumb heuristic for discovering interesting data. The threshold was first introduced in 1935 by famed statistician Ronald Fisher. Later, Fisher exclaimed that the significance level should not remain static, and should be manually adjusted based on the nature of the analysis. Regrettably, by then it was too late; the 0.05 threshold had been adopted as our standard measure of significance. Today, most statisticians agree that a p-value below 0.05 implies an interesting signal in the data. Thus, a p-value of 0.033 is sufficient to temporarily reject the null hypothesis, and to get one's data published in a respectable scientific journal. Unfortunately, that threshold of 0.05 doesn't actually arise from the laws of mathematics and statistics. Instead, it is a social construct. Academic researchers have settled on 0.05 as the p-value required for research publication. As a consequence, the scientific community has recently experienced a flood of publications with **type I errors**. A type I error is defined as an erroneous rejection of the null hypothesis. Such errors occur when random data fluctuations get interpreted as genuine deviations from the population mean. Scientific articles containing type I errors falsely assert a difference between means, where none exists.

How do we limit the prevalence of type I errors? Well, some scientists claim that a p-value cutoff of 0.05 is unreasonably high. These scientists believe that we should only reject the null hypothesis if that p-value is noticeably lower than 0.05. However, there is currently no consensus on whether a lower p-value threshold is appropriate. This is because a lower threshold will lead to an increase in **type II errors**, in which we wrongly reject the alternative hypothesis. When scientists commit a type II error, they fail to notice a legitimate discovery.

Selecting an optimal significance level is difficult. Nevertheless, let us temporarily set the significance level to a very stringent value of 0.001. What would be the minimum grade-average required to trigger this p-value threshold? Let's find out. We'll loop through all grade averages above 80%, computing the p-value as we go. We'll stop when we encounter a p-value that's less than or equal to 0.001.

**Listing 7.8 Scanning for a stringent p-value result**

```
for grade in range(80, 100):
    p_value = compute_p_value(grade, mean, sem)
    if p_value < 0.001:
        break

print(f"An average grade of {grade} leads to a p-value of {p_value}")
```

```
An average grade of 88 leads to a p-value of 0.0006885138966450773
```

Given the new threshold, we would require an average grade of at-least 88% in order to reject the null hypothesis. Thus, an average grade of 87% would not pass our threshold of significance, despite being 7 percentage points higher than the population mean. Our lowering of the cutoff has inevitably exposed us to an increased risk of type II errors. Consequently, in this book, we'll maintain the commonly accepted p-value cutoff of 0.05. However, we will also proceed with excessive caution in order to avoid erroneously rejecting the null hypothesis. In particular, we'll do our best to minimize the most common cause of type I errors, and the topic of the next section: data-dredging.

## 7.2 Data Dredging: Coming to False Conclusions through Oversampling

Sometimes, statistics students utilize the p-value incorrectly. Consider the following simple scenario. Two roommates pour out a bag of candy. The bag contains multiple candy pieces, in 5 different colors. There are more blue candies in the bag than any other individual color. The first roommate assumes that blue is the dominant color in any candy bag. The second roommate disagrees. She computes the p-value based on the null hypothesis that all colors occur with equal likelihood. That p-value is greater than 0.05. However, the first roommate refuses to back down. He opens up another bag of candy. The p-value is recomputed from the contents of that bag. This time, the p-value is equal to 0.05. The first roommate claims victory. He asserts that given the low p-value, the null hypothesis is probably false. Yet he is wrong.

The first roommate fundamentally misconstrued the meaning of the p-value. He wrongly assumed it represents the probability of the null hypothesis being true. In fact, the p-value actually represents the probability of observing deviations if the null hypothesis is true. The difference between the definitions is subtle but very important. The first definition implies that the null hypothesis is likely to be false if the p-value is low. The second definition guarantees that we'll eventually observe a low p-value by repeatedly counting candies, even when null

hypothesis is true. Furthermore, the frequency of low p-value observations will equal the p-value itself. Hence, if we open 100 bags of candy, we should expect to observe a p-value of 0.05 approximately 5 times. By taking random measurements repeatedly, we will eventually obtain a statistically significant result, even if no statistical significance exists!

Running the same experiment too many times increases our risk of type I errors. Let's explore this notion in the context of our fifth-grade exam analysis. Suppose that North Dakota's state-wide test performance does not diverge from the exam results in the other 49 states. More precisely, we'll assume that the national mean and variance equals North Dakota's `population_mean` and `population_variance ` exam-grade results. Thus, the null hypothesis is true for all the states in the United States.

Furthermore, let's assume we don't yet know that null hypothesis is always true. The only thing we know for sure is North Dakota's population mean and variance. We set on a road-trip in search of state whose grade-distribution differs from North Dakota's distribution. Unfortunately, our search is bound to be futile, because no such state exists.

Our first stop is Montana. There, we choose a random fifth-grade classroom of 18 students. We then compute the classroom's average grade. Since the null hypotheses is secretly true, we can simulate the value of that average grade by sampling from a Normal distribution defined by `mean` and `sem`. Let's simulate the exam-performance of the class by calling `np.random.normal(mean, sem)`. The method call will sample from a Normal distribution defined by the inputted variables.

### Listing 7.9 Randomly sampling Montana's exam performance

```
np.random.seed(0)
random_average_grade = np.random.normal(mean, sem)
print(f"Average grade equals {random_average_grade:.2f}")
```

```
Average grade equals 84.16
```

The average exam grade in the class equaled approximately 84.16. We can determine if that average is statistically significant by checking if its p-value is less than or equal to 0.05.

### Listing 7.10 Testing significance of Montana's exam performance

```
if compute_p_value(random_average_grade, mean, sem) <= 0.05:
    print("The observed result is statistically significant")
else:
    print("The observed result is not statistically significant")
```

```
The observed result is not statistically significant
```

The average-grade is not statistically significant. We will continue our journey. We'll visit a single 18-student classroom in each of the remaining 48 states. The grade-average for each classroom will be computed. The p-value will also be computed. Once we discover a statistically

significant p-value, our journey will end.

The code below will simulate our travels. It iterates through the remaining 48 states, randomly drawing a grade-average for each state. Once a statistically significant grade-average is discovered, the iteration loop will stop.

**Listing 7.11 Randomly searching for a significant state result**

```
np.random.seed(0)
for i in range(1, 49):
    print(f"We visited state {i + 1}")
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
    if p_value <= 0.05:
        print("We found a statistically significant result.")
        print(f"The average grade was {random_average_grade:.2f}")
        print(f"The p-value was {p_value}")
        break

if i == 48:
    print("We visited every state and found no significant results.")
```

```
We visited state 2
We visited state 3
We visited state 4
We visited state 5
We found a statistically significant result.
The average grade was 85.28
The p-value was 0.025032993883401307
```

The fifth state that we visited produced a statistically significant result! A classroom in the state had a grade-average of 85.28! The p-value for that average equaled approximately 0.025. This value is noticeably below our p-value cutoff of 0.05. We thus can reject the null hypothesis! We've discovered a state that outperforms North Dakota to a statistically significant degree. However, our conclusions are erroneous. The null hypothesis has remained true the entire time. What went wrong? Well, as stated earlier, the frequency of low p-value observations will equal the p-value itself. Therefore, we expect to encounter a p-value of 0.025 approximately 2.5% of the time, even if the null hypothesis is true. Since we are travelling across 49 states, and 2.5% of 49 is 1.225, we should expect to visit approximately one state with a random p-value of roughly 0.025.

Our quest to find a statistically significant result was doomed from the start, because we have misused statistics. We have indulged in the cardinal statistical sin of **data dredging**, also known as **data fishing** or **p-hacking**. In data-dredging, experiments are repeated over and over again, until a statistically significant result is found. Afterwards, the statistically significant result is presented to others, while the remaining failed experiments are discarded. Data dredging is the most common cause of type I errors present in scientific publications. Sadly, sometimes researchers formulate a hypothesis and repeat an experiment until the particular false hypothesis is validated as true. For instance, a researcher might hypothesize that certain candies cause cancer in mice. The researcher proceeds to feed a specific candy brand to a group of mice, but no

cancer link is found. The researcher then switches the brand of candy, and repeats the experiment again. And again. And again. Years later, a brand of candy linked to cancer is finally found. Of course, the actual experiment outcome is borderline fraudulent. No real statistical link exists between cancer and candy. The researcher has simply run the experiment way too many times, until a low p-value was randomly measured.

Avoiding data dredging is not difficult. We must simply choose in advance a finite number of experiments to run. Afterwards, we should set our significance level to 0.05 divided by the planned experiment count. This simple technique is known as the **Bonferonni correction**. Let's repeat our analysis of US exam performance using the Bonferonni correction. The analysis requires us to visit 49 states in order to evaluate 49 classrooms. Therefore, our significance level should be set to `0.05 / 49`.

### Listing 7.12 Using the Bonferonni correction to adjust significance

```
num_planned_experiments = 49
significance_level = .05 / num_planned_experiments
```

We'll proceed to re-run our previous analysis. The analysis will terminate if we encounter a p-value that's less than or equal to `significance_level`.

### Listing 7.13 Re-running an analysis using an adjusted significance level

```
np.random.seed(0)
for i in range(49):
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
    if p_value <= significance_level:
        print("We found a statistically significant result.")
        print(f"The average grade was {random_average_grade:.2f}")
        print(f"The p-value was {p_value}")
        break

if i == 48:
    print("We visited every state and found no significant results.")
```

```
We visited every state and found no significant results.
```

We've visited 49 states and found no statistically significant deviations from North Dakota's population mean and variance. The Bonferonni correction thus allows us to avoid a potential type I error.

As a final word of caution, the Bonferonni correction only works if we divide 0.05 by the number of planned experiments. It noticeably less effective if we simply divide by the count of completed experiments. For instance, if we plan to run 1000 experiments, but the p-value of our very first experiment equals 0.025, we should still not alter the our significance level to `0.05 / 1`. Similarly, if the p-value of the second completed experiment equals 0.025, we should still

maintain a significance level of `0.05 / 1000` rather than adjusting it to `0.05 / 2`. Otherwise, we risk wrongly biasing our conclusions towards our first few experimental outcomes. All experiments must be treated equally for us to draw a fair, correct conclusion.

The Bonferonni correction is a useful technique for more accurate hypothesis testing. It can be applied to all kinds of statistical hypothesis tests, beyond just simple tests that exploit both population mean and variance. This is fortunate, because statistical tests vary in their levels of complexity. In the next section, we will explore a more complicated test, which does not depend on knowing the population variance.

## 7.3 Bootstrapping with Replacement: Testing a Hypothesis When the Population Variance is Unknown

We are easily able to compute a p-value using population mean and variance. Regrettably, in many real-life circumstances, the population variance is not known. Consider the following scenario, in which we own a very large aquarium. It holds 20 tropical fish of varying lengths. The fish lengths range from 2 cm to nearly 120 cm. The average fish-length equals 27 cm. We'll represent these fish lengths using the `fish_length` array below.

**Listing 7.14 Defining lengths of fish in an aquarium**

```
fish_lengths = np.array([46.7, 17.1, 2.0, 19.2, 7.9, 15.0, 43.4,
                         8.8, 47.8, 19.5, 2.9, 53.0, 23.5, 118.5,
                         3.8, 2.9, 53.9, 23.9, 2.0, 28.2])
assert fish_lengths.mean() == 27
```

Does our aquarium accurately capture the distributed lengths of real tropical fish? We would like to find out. A trusted source informs us that the population mean-length of wild, tropical fish equals 37 cm. There is a sizable 10 cm difference between the population mean and our sample mean. That difference feels significant, but feelings have no place in rigorous statistics. We must determine if the difference is statistically significant. Only then can we draw a valid conclusion.

Measuring statistical significance will require hypothesis testing. Previously, we've carried out hypothesis testing using our `compute_p_value` function. We would like to apply this function to our data. However, we run into a problem. We don't know the population variance! Without the population variance, we cannot compute the SEM, which is a variable required to run `compute_p_value`. So what should we do? How do we find the standard error of the mean when population variance is not known?

At first glance, it appears we have no way of finding the SEM. We could naively treat our sample variance as an estimate of the population variance, by executing `fish_lengths.var()`. Unfortunately, small samples are prone to random variance fluctuations, and so any such estimate is highly unreliable. Thus, we are stuck. We face a seemingly impenetrable problem, and must rely on a seemingly impossible solution: **Bootstrapping with Replacement**. The term

"Bootstrapping" originates from the phrase "pull yourself up by your bootstraps". The phrase refers to lifting yourself into the air by pulling on the laces of your boots. Of course, doing so is impossible. In Bootstrapping with Replacement, we'll attempt something equally impossible by computing a p-value directly from our limited data! Fortunately, sometimes the seemingly impossible is actually possible. We will be successful in our efforts.

We'll begin the Bootstrapping procedure by removing a random fish from the aquarium. The length of the selected fish will be measured for later use.

**Listing 7.15 Sampling a random fish from the aquarium**

```
np.random.seed(0)
random_fish_length = np.random.choice(fish_lengths, size=1)[0]
sampled_fish_lengths = [random_fish_length]
```

Now, we will place the chosen fish back into the aquarium. This replacement step is where Bootstrapping with Replacement gets its name. After we've returned the fish, we will reach into the aquarium again. We will then choose a fish purely at random. There is a 1-in-20 chance that we'll select the same fish as before. Such repetition is acceptable. The Bootstrapping technique depends on it. We'll record the length of the fish and place it back into the waters. Next, we will repeat the procedure 18 more times until 20 random fish lengths have been measured.

**Listing 7.16 Sampling 20 random fish with repetition**

```
np.random.seed(0)
for _ in range(20):
    random_fish_length = np.random.choice(fish_lengths, size=1)[0]
    sampled_fish_lengths.append(random_fish_length)
```

The `sampled_fish_lengths` list contains 20 measurements. All of these measurements were taken directly from 20-element `fish_lengths` array. However, the elements of `fish_lengths` and `sampled_fish_lengths` are not identical. Due to random sampling, the mean values of the array and list are likely to differ.

**Listing 7.17 Comparing the sample mean to the aquariuam mean**

```
sample_mean = np.mean(sampled_fish_lengths)
print(f"Mean of sampled fish-lengths is {sample_mean:.2f} cm")
```

```
Mean of sampled fish-lengths is 26.03 cm
```

The mean of sampled fish-lengths is 26.03 cm. It deviates from our original mean by 0.97 cm. Thus, sampling with replacement has introduced some degree of variance into our observations. If we sample another 20 measurements from the aquarium, we can expect the resulting sample mean to also deviate from 27 cm. Let's confirm this by repeating our sampling using a single line of code; `np.random.choice(fish_lengths, size=20, replace=True)`. Setting the replace parameter to `True` will ensure that we sample with replacement from the `fish_lengths array`.

**Listing 7.18 Sampling with replacement using NumPy**

```
new_sampled_fish_lengths = np.random.choice(fish_lengths, size=20,
                                            replace=True)  ❶
new_sample_mean = new_sampled_fish_lengths.mean()
print(f"Mean of the new sampled fish-lengths is {sample_mean:.2f} cm")
```

❶ As a side note, the `replace` parameter is currently set to `True` by default within the method..

The new sample mean equals 26.16 cm. Thus, mean-values will fluctuate when we sample with replacement. Where there is fluctuation, there is randomness. Our mean-values are randomly distributed Let's explore the shape of this random distribution by repeating our sampling process 150,000 times. During each of 150,000 iterations we will measure 20 random fish. Afterwards, we will compute the sample mean. Finally, we will plot a histogram of the 150,000 sampled means.

**Listing 7.19 Plotting the distribution of 150k sampled means**

```
np.random.seed(0)
sample_means = [np.random.choice(fish_lengths,
                                 size=20,
                                 replace=True).mean()
                for _ in range(150000)]
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

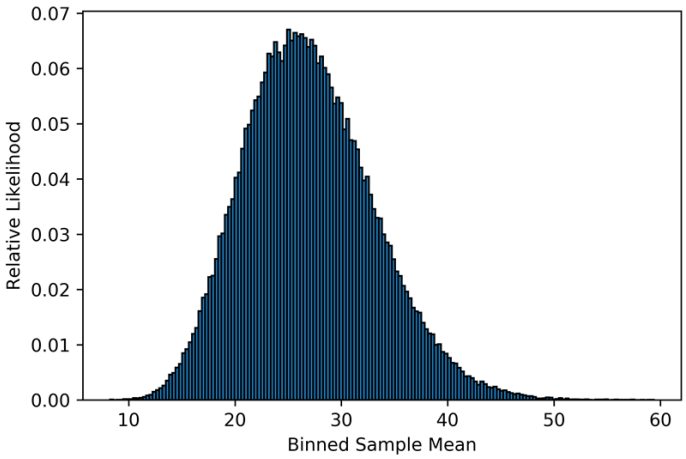


**Figure 7.3 A histogram of sample-means computed using sampling with replacement. The histogram is not bell-shaped; its asymmetric.**

The histogram we've generated is not a Normal curve. The shape is not symmetric, its left side rises steeper than its right side. Mathematicians refer to this asymmetry as a **skew**. We can confirm the skew within our histogram, by calling `stats.skew(sample_means))`. The

`stats.skew` method returns a non-zero value when the inputted data is asymmetric.

**Listing 7.20 Computing the skew of an asymmetric distribution**

```
assert abs(stats.skew(sample_means)) > 0.4   ❶
```

❶ No data is ever perfectly symmetric, and the skew is rarely zero, even if the data is sampled from a Normal curve. However, Normal data tends to have a skew that is exceedingly close to zero. Any data with a skew whose absolute value is greater than 0.04 is very unlikely to come from a Normal distribution.

Our asymmetric histogram cannot be modeled using a Normal distribution. Nevertheless, the histogram represents a continuous probability distribution. Like all continuous distributions, the histogram can be mapped to a probability density function, a cumulative distribution function, and a survival function. Knowing the outputs of these functions could be useful. We could, for instance, leverage the survival function to compute the probability of observing a sample mean whose value is greater than equal to our population mean. Yet how do we obtain the function values? One option is to manually write code that computes curve-area using the `bin_edges` and `likelihoods` arrays. Alternatively, we could just utilize SciPy. The library provides us with a method for obtaining all three functions directly from the histogram. That method is `stats.rv_histogram`. It takes as input a tuple by defined by the `bin_edges` and likelihoods arrays. Calling `stats.rv_histogram((likelihoods, bin_edges))` will return a `random_variable` SciPy object. The object will contain `pdf`, and `cdf`, and `sf` methods, just like `stats.norm`. The random_variable.pdf method will output the probability density for the histogram. Likewise, the `random_variable.cdf` and `random_variable.sf` methods will output the cumulative distribution function and the survival function, respectively.

Below, we will compute the random_variable object arising from the histogram. Afterwards, we will plot the probability density function by calling `random_variable.pdf(bin_edges)`.

**Listing 7.21 Fitting to data to a generic distribution using SciPy**

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
plt.plot(bin_edges, random_variable.pdf(bin_edges))
plt.hist(sample_means, bins='auto', alpha=0.1,  color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

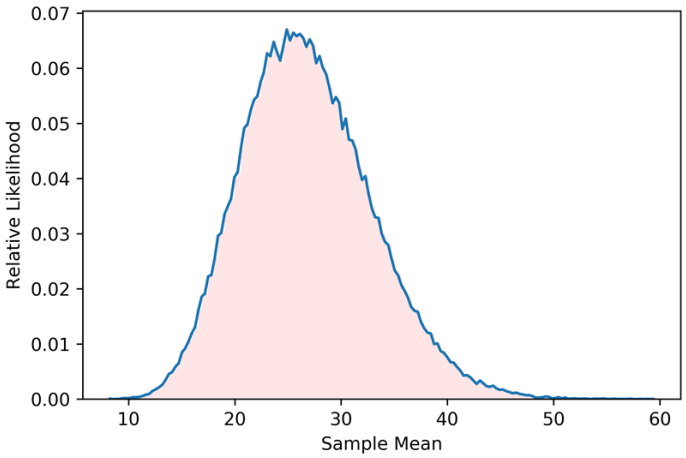


**Figure 7.4 An asymmetric histogram overlaid with its probability density function. SciPy was utilized to learn the probability density function from the histogram.**

As expected, the probability density function perfectly resembles the histogram shape. Let's now plot both the cumulative distribution function and the survival function associated with `random_variable`. We should anticipate that the two plotted functions will not be symmetric around the mean. In order to check for this asymmetry, we will plot the distribution's mean using a vertical line. We can compute that mean by calling `random_variable.mean()`.

## Listing 7.22 Plotting mean and interval-areas for a generic distribution

```
rv_mean = random_variable.mean()
print(f"Mean of the distribution is approximately {rv_mean:.2f} cm")

plt.axvline(random_variable.mean(), color='k', label='Mean', linestyle=':')
plt.plot(bin_edges, random_variable.cdf(bin_edges),
         label='Cumulative Distribution')
plt.plot(bin_edges, random_variable.sf(bin_edges),
         label='Survival', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

```
Mean of the distribution is approximately 27.00 cm
```

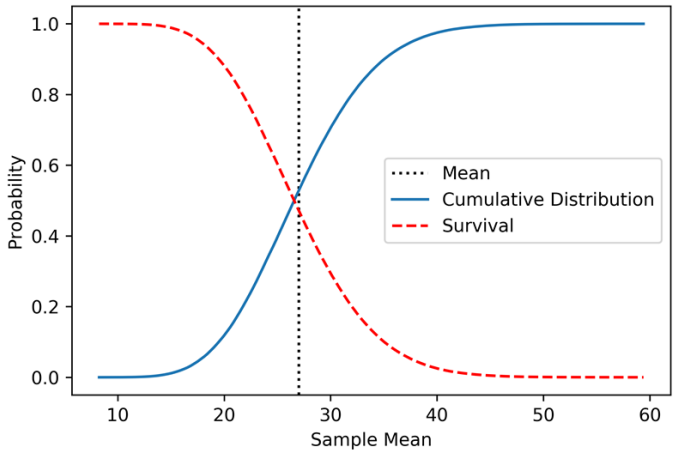


**Figure 7.5 A cumulative distribution function of an asymmetric distribution plotted together with the survival function. The two functions no longer symmetrically reflect across the mean, like they did in our Normal curve analysis. Therefore, we can no longer compute the p-value simply by doubling the survival function output.**

The mean of the distribution is approximately 27 cm. Not surprisingly, 27 cm is also the mean-length of the fish in our aquarium. A random sampling of the fish is likely produce value that is close to the aquarium's mean. However, sampling with replacement will sometimes produce a value that's greater than 37 cm, or less than 17 cm. The probabilities of observing these extremes can be computed from our two plotted functions. Let's examine these two functions in more detail.

Based on our plot, the cumulative distribution function and the survival function are not mirror images of each other. Nor do they intersect directly at the mean, like they did in our Normal curve analysis. Clearly, our distribution does not behave like a symmetric Normal curve. This asymmetry leads to certain consequences. For the symmetric Normal curve, doubling the survival function output had allowed us calculate a p-value. In our asymmetric distribution, the survival function itself is insufficient to compute tail-end probabilities. Fortunately, we can leverage both the survival function and the cumulative distribution function to uncover probabilities of extreme observations. These probabilities should allow us to evaluate the statistical significance of our data.

We can measure significance by answering this question: What is probability that 20 fish sampled with replacement produce a mean as extreme as the population mean? As a reminder, the population mean is 37 cm. The population mean is 10 cm away from our distribution mean. Therefore, extremeness is defined as a sampled output that's at-least 10 cm away from `rv_mean`. Based on our previous discussions, the problem can be broken down into computing two distinct values. First, we must compute the probability of observing a sample mean that's at-least 37 cm. Next, we must compute the probability of observing a sample mean that's less than or equal to

17 cm. The former probability equals `random_variable.sf(37)`. The latter probabilities `random_variable.cdf(17)`. Summing the two probabilities will give us our answer. Let's compute that probability sum.

**Listing 7.23 Computing the probability of an extreme sample mean**

```
prob_extreme= random_variable.sf(37) + random_variable.cdf(17)
print(f"Probability of observing an extreme sample mean is approximately {prob_extreme:.2f}")
```

```
Probability of observing an extreme sample mean is approximately 0.10
```

The probability of observing an extreme value from our sampling is approximately 0.10. In other words, one-tenth of random aquarium samplings will produce a mean that's at-least as extreme as the population mean. Our population mean is not so far off from aquarium mean as we had thought. In fact, a mean-discrepancy of 10 cm or more will appear in 10% of sampled fish outputs. Thus, the difference between our sample mean of 27 cm and our population mean of 37 cm is not statistically significant.

By now, all this should seem familiar. The `prob_extreme` value is just the p-value in disguise. When the null hypothesis is true, the difference between sample mean and population mean will be at-least 10 cm in 10% of sampled cases. This p-value of 0.1 is greater than our cutoff of 0.05. Thus, we cannot reject the null hypothesis. There is no statistically significant difference between our sample mean and population mean. We've compute a p-value in a round-about way. Some readers might be suspicious of our methods. After all, sampling from our limited collection of 20 fish seems like a strange way to draw statistical insights. Nevertheless, the described technique is legitimate. Bootstrapping with Replacement is a reliable procedure for extracting the p-value, especially when dealing with limited data.

The Bootstrapping technique has been rigorously studied for more than four decades. Statisticians have shown that variations of the technique provide multiple methods for accurate p-value computation. We've just reviewed one such method; now we will briefly introduce another. It has been shown that sampling with replacement approximates a dataset's standard error from the mean. Basically, the standard deviation of the sampled distribution is equal to the SEM, when the null hypothesis is true. Thus, if the null hypothesis is true, then our missing SEM is equal to `random_variable.std`. This give us yet another way of finding the p-value. We simply need to execute `compute_p_value(27, 37, random_variable.std)`. That computed p-value should equal approximate 0.1. Let's confirm below.

**Listing 7.24 Using Bootstrapping to estimate the SEM**

```
estimated_sem = random_variable.std()
p_value = compute_p_value(27, 37, estimated_sem)
print(f"P-value computed from estimated SEM is approximately {p_value:.2f}")
```

```
P-value computed from estimated SEM is approximately 0.10
```

As expected, the computed p-value is approximately 0.1. We've shown how Bootstrapping with Replacement provides us with two divergent approaches for computing the p-value. The first approach requires us to:

**A.** Sample with replacement from data. Repeat tens of thousands of times to obtain a list of sample means.

**B.** Generate a histogram from the sample means.

**C.** Convert the histogram to a distribution using the `stats.rv_histogram` method.

**D.** Take the area beneath the left and right extremes of the distribution curve using the survival function and the cumulative distribution function.

Meanwhile, the second approach appears to be slightly simpler. It requires us to:

**A.** Sample with replacement from data. Repeat tens of thousands of times to obtain a list of sample means.

**B.** Compute the standard deviation of the means. It approximated the SEM.

**C.** Utilize the estimated SEM to carry out basic hypothesis testing using our `compute_p_value` function.

As a third alternative, lets briefly introduce a new approach, which is even easier to implement in Python. The third approach does not require us to generate histogram, nor does it require us to use a custom `compute_value_function`. Instead, the technique leverages the Law of Large Numbers, introduced in Section Two. According to that law, the frequency of observed events approximates the probability of event occurrence, if the sample count is sufficiently large. Thus, we can estimate the p-value simply by computing the frequency of extreme observations. Let's quickly apply this simple technique to `sample_means` by counting means that do not fall between 17 cm and 37 cm. We will divide the count by `len(sample_means)` in order to compute the p-value.

**Listing 7.25 Computing the p-value from direct counts**

```
number_extreme_values = 0
for sample_mean in sample_means:
    if not 17 < sample_mean < 37:
        number_extreme_values += 1

p_value =  number_extreme_values / len(sample_means)
print(f"P-value is approximately {p_value:.2f}")
```

```
P-value is approximately 0.10
```

Bootstrapping with Replacement is a simple but powerful technique for making inferences from limited data. However, the technique still presupposes the knowledge of a population mean.

Unfortunately, in real-life situations, the population mean is rarely known. For instance, our assigned online-ad table does not include a population mean. This missing information will not stop us. In the next section, we learn how to compare collected samples when both the population mean and the population variance are unknown.

## 7.4 Permutation Testing: Comparing Means of Samples when the Population Parameters are Unknown

Sometimes in statistics, we need to compare 2 distinct sample means, while the population parameters remain unknown. Let's explore one such scenario.

Suppose our neighbor also owns an aquarium. Her aquarium contains 10 fish, whose average length is 46 cm. We'll represent these new fish-lengths using the `new_fish_length` array below.

**Listing 7.26 Defining lengths of fish in a new aquarium**

```
new_fish_lengths = np.array([51, 46.5, 51.6, 47, 54.4, 40.5, 43, 43.1,
                             35.9, 47.0])
assert new_fish_lengths.mean() == 46
```

We want to compare the contents of our neighbor's aquarium with our own. We'll begin by measuring the difference between `new_fish_length.mean()` and `fish_length.mean()`.

**Listing 7.27 Computing difference between 2 sample means**

```
mean_diff = abs(new_fish_lengths.mean() - fish_lengths.mean())
print(f"There is a {mean_diff:.2f} cm difference between the two means")
```

```
There is a 19.00 cm difference between the two means
```

There is an 19 cm difference between the two aquarium means. That difference is substantial, but is it statistically significant? We want to find out. However, we have a problem. All of our previous analyses have required a population mean. Now we find ourselves with two sample means but no population mean. Without a population mean, it's difficult to evaluate the null hypothesis. After all, the null hypothesis assumes that fish from both aquariums share a population mean. Under current circumstances, that presumed shared value is unknown. What should we do?

We need to reframe the null hypothesis, so that it doesn't directly depend on the population mean. If the null hypothesis is true, then the 20 fish in the first aquarium and the 10 fish in the second aquarium all get drawn from the same population. Thus, under the null hypothesis, it doesn't really matter which 20 fish wind up in aquarium A and which 10 fish wind up in aquarium B. The arrangements of fish between the two aquariums will have little effect. Still, random rearrangements of the fish will cause the mean_diff variable to fluctuate. However, if the null hypothesis is true, then difference between means should fluctuate in a predictable manner.

For that reason, we don't need to know the sample mean in order to evaluate the null hypothesis. We can simply focus on the random permutations of fish between the two aquariums. This will allow us to carry out a **Permutation test**, where `mean_diff` is leveraged to compute statistical significance. Like Bootstrapping with Replacement, the Permutation test will rely on random sampling of data.

We'll begin the Permutation test by placing all 30 fish into a single aquarium. The unification of our fish can be modeled using the `np.hstack` method. The method takes as input a list of NumPy arrays, which are afterwards merged together into a single NumPy array.

### Listing 7.28 Merging 2 arrays using `np.haystack`

```
total_fish_lengths = np.hstack([fish_lengths, new_fish_lengths])
assert total_fish_lengths.size == 30
```

Once the fish are grouped together, we will allow them to swim in random directions. This will fully randomize the positions of the fish in the aquarium. We'll use the `np.random.shuffle` method to shuffle the positions of the fish.

### Listing 7.29 Shuffling the positions of merged fish

```
np.random.seed(0)
np.random.shuffle(total_fish_lengths)
```

Afterwards, we'll choose 20 of our randomly shuffled fish. These 20 fish will be moved to a separate aquarium. The other 10 fish will remain. Once more, we'll have 20 fish in aquarium A and 10 fish in aquarium B. However, the mean-lengths of the fish in each aquarium will probably differ from `fish_lengths.mean()` and `new_fish_lengths.mean()`. Consequently, the difference between mean fish-lengths will also change. Let's confirm below.

### Listing 7.30 Computing the difference between 2 random sample means

```
random_20_fish_lengths = total_fish_lengths[:20]
random_10_fish_lengths = total_fish_lengths[20:]
mean_diff = random_20_fish_lengths.mean() - random_10_fish_lengths.mean()
print(f"The sampled difference between mean fish lengths is {mean_diff}")
```

```
The sampled difference between mean fish lengths is 14.329999999999998
```

The sampled difference between fish lengths is no longer 19 cm. Now, it is 14.33 cm. Not surprisingly, `mean_diff` is fluctuating random variable. We therefore can proceed to find its distribution through random sampling. Below, will repeat our fish-shuffling procedure 30,000 times in order to obtain a histogram of `mean_diff` values.

### Listing 7.31 Plotting the distribution of the fluctuating difference between means

```
np.random.seed(0)
mean_diffs = []
for _ in range(30000):
    np.random.shuffle(total_fish_lengths)
    mean_diff = total_fish_lengths[:20].mean() - total_fish_lengths[20:].mean()
    mean_diffs.append(mean_diff)

likelihoods, bin_edges, _ = plt.hist(mean_diffs, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Mean Difference')
plt.ylabel('Relative Likelihood')
plt.show()
```

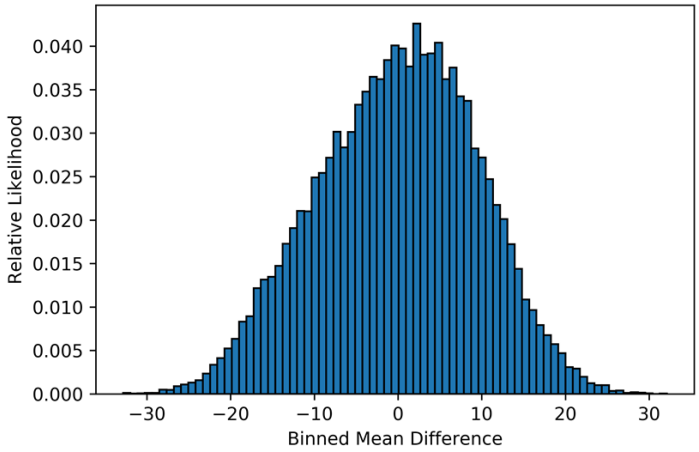


**Figure 7.6 A histogram of sample-mean differences computed using random rearrangements of samples into 2 distinct groups.**

Next, we will fit the histogram to a random variable using the `stats.rv_hist` method.

### Listing 7.32 Fitting the histogram to a random variable

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
```

Finally, we will use the `random_variable` object to carry out hypothesis testing. We want to know the probability of observing an extreme value when the null hypothesis is true. We'll define extremeness as a difference between means whose absolute value is at-least 19 cm. Thus, our p-value will equal `random_variable.cdf(-19) + random_variable.sf(19)`.

### Listing 7.33 Computing the Permutation p-value

```
p_value = random_variable.sf(19) + random_variable.cdf(-19)
print(f"P-value is approximately {p_value:.2f}")
```

```
P-value is approximately 0.04
```

The p-value is approximately 0.04. It is below our significance threshold of 0.05. Hence, the

mean-difference between fish-length the two aquariums is statistically significant. The fish in the two aquariums do not originate from a shared distribution.

As an aside, we can simplify our Permutation test by leveraging the Law of Large Numbers. We simply need to compute the frequency of extreme recorded samples, just like we did with Bootstrapping with Replacement. Let's use this alternative method to re-compute our p-value of approximately 0.04.

**Listing 7.34 Computing the Permutation p-value from direct counts**

```
number_extreme_values = 0.0
for min_diff in mean_diffs:
    if not -19 < min_diff < 19:
        number_extreme_values += 1

p_value =  number_extreme_values / len(mean_diffs)
print(f"P-value is approximately {p_value:.2f}")
```

```
P-value is approximately 0.04
```

The Permutation test allows to statistically compare differences between 2 lists of collected samples. The nature of these samples isn't important; they could be fish lengths, or they could be ad-click counts. Hence, the Permutation test could be very useful when we compare our recorded ad-click counts.

## 7.5 Summary

- Statistical hypothesis testing requires us to choose between 2 competing hypotheses. According to the null hypothesis, a pair of populations are identical. According to the alternate hypothesis, the pair of populations are not identical.
- In order to evaluate the null hypothesis, we must compute a p-value. The p-value equals the probability of observing our data when the null hypothesis is true. The null hypothesis is rejected if the p-value lies above a specified significance level. Typically, the significance level is set to 0.05.
- If we reject the null hypothesis, and the null hypothesis is true, than we commit a type I error. If we fail to reject the null hypothesis, and the alternative hypothesis is true, then we commit a type II error.
- Data dredging increases our risk of type I errors. In data dredging, an experiment is repeated until the p-value falls below the significance level. We can minimize data dredging by carrying out a Bonferroni correction, in which the significance level is divided by the experiment count.
- We can compare a sample mean to a population mean and variance by relying on the Central Limit Theorem. The population variance is needed to compute the SEM. If we're not provided with the population variance, then we can estimate the SEM using Bootstrapping with Replacement.
- We can compare the means of 2 distinct samples by running a Permutation test.

# 8 Analyzing Tables Using Pandas

***This section covers:***

- Storing 2D tables using the Pandas library
- Summarizing 2D table content
- Manipulating row and column content
- Visualizing tables using the Seaborn library

The ad-click data for Case Study Two is saved in a 2-dimensional table. Data tables are commonly used to store information. The tables may be stored in different formats. Some tables are saved as spreadsheets in Excel. Other tables are text-based CSV files where the columns are separated by commas. The formatting of a table isn't important. What is important is its structure. All tables have structural features in common. Every table contains horizontal rows and vertical columns. Quite often, column headers also hold explicit column names.

## 8.1 Storing Tables Using Basic Python

Let's define a sample table in Python. The table will store measurements for various species of fish. The measurements will cover both length and width, in centimeters. Our measurement table will contain three columns: *Fish*, *Length*, and *Width*. The *Fish* column will store a labeled species of fish. The *Length* and *Width* columns will specify the length and width of each fish species. We'll represent this table as a dictionary. The column names will serve as dictionary keys. These keys will map to lists of column values. Let's proceed to build our `fish_measures` table.

**Listing 8.1 Storing a table using Python data-structures**

```
fish_measures = {'Fish': ['Angelfish', 'Zebrafish', 'Killifish', 'Swordtail'],
                 'Length':[15.2, 6.5, 9, 6],
                 'Width': [7.7, 2.1, 4.5, 2]}
```

Suppose we want to know the length of a zebrafish. To obtain the length, we must first access the index of the `'Zebrafish'` element in `fish_measures['Fish']`. Afterwards, we'll need to check index in `fish_measures['Length']`. The process is slightly convoluted, as is illustrated in the code below.

**Listing 8.2 Accessing table columns using a dictionary**

```
zebrafish_index = fish_measures['Fish'].index('Zebrafish')
zebrafish_length = fish_measures['Length'][zebrafish_index]
print(f"The length of a zebrafish is {zebrafish_length:.2f} cm")
```

```
The length of a zebrafish is 6.50 cm
```

Our dictionary representation is functional, but is also difficult to use. A better solution is required. That solution is provided by the external Pandas library. Pandas is very useful tool for manipulating numeric tables in Python.

## 8.2 Exploring Tables Using Pandas

We'll proceed to install the Pandas library. Once Pandas is installed, we will import it as `pd`, using common Pandas usage convention.

> **NOTE** Call "pip install pandas" from the command-line terminal in order to install the Pandas library.

**Listing 8.3 Importing the Pandas library**

```
import pandas as pd
```

We'll now load our `fish_measures` tables into Pandas. This can be done by calling `pd.DataFrame(fish_measures)`. The method-call will construct a Pandas DataFrame object. The term **data frame** is just a common synonym for table that arises in statistical software jargon. Basically, the `DataFrame` object will convert our dictionary into a 2-dimensional table. According to convention, Pandas `DataFrame` objects are assigned to a variable df. Below, we will execute `df = pd.DataFrame(fish_measures)`. Afterwards, we'll print out the contents of `df`.

**Listing 8.4 Loading a table into Pandas**

```
df = pd.DataFrame(fish_measures)
print(df)
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
1  Zebrafish     6.5    2.1
2  Killifish     9.0    4.5
3  Swordtail     6.0    2.0
```

The alignments between table rows and columns are clearly visible in the printed output. The complete table contents are also visible. The table is rather small, so displaying it is not a serious issue. However, for larger tables, we might prefer to only print the first few rows. Calling `print(df.head(x))` will print out just the first `x` rows within a table. Let's print out the first 2 rows by calling `print(df.head(2))`.

**Listing 8.5 Accessing the first 2 rows of a table**

```
print(df.head(2))
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
1  Zebrafish     6.5    2.1
```

Sometimes, the best way to summarize a larger Pandas table is to execute the `pandas.describe()` method. By default, the method will generate statistics for all numeric columns within the table. The statistical output will include minimum and maximum column values, as well as the mean and standard deviation. When we print `pandas.describe()` we should expect to see information for the numeric *Length* and *Width* column, but not for the string-based *Fish* column.

**Listing 8.6 Summarizing the numeric columns**

```
print(df.describe())
```

```
          Length     Width
count   4.000000  4.000000
mean    9.175000  4.075000
std     4.225616  2.678775
min     6.000000  2.000000
25%     6.375000  2.075000
50%     7.750000  3.300000
75%    10.550000  5.300000
max    15.200000  7.700000
```

**SIDEBAR** **The outputs of the Pandas `describe` method**

`count`: The number of elements in each column.

`mean`: The mean of elements in each column. `std`: The standard deviation of elements in each column.

`min`: The minimum value in each column.

`25%`: 25% of column elements fall below this value.

`50%`: 50% the column elements fall below this value. The value is identical to the median.

`75%`: 75% of column elements fall below this value.

`max`: The maximum value in each column.

According to the summary, the mean of *Length* is 9.175 cm, and the mean of *Width* is 4.075 cm.

Additional statistical information is also included in the output. Sometimes, that additional information is not very useful. If all we care about is the mean, then we can omit all other outputs by calling `df.mean()`.

### Listing 8.7 Computing the column mean

```
print(df.mean())
```

```
Length    9.175
Width     4.075
dtype: float64
```

The `df.describe()` method is primarily intended to be executed on numeric columns. However, we can force it to process strings by calling `df.describe(include=[np.object])`. Setting the `include` parameter to `[np.object]` will instruct Pandas to search for table columns that are built on-top of NumPy string arrays. Since we cannot run statistical analysis on strings, the resulting output will not contain statistical information. Instead, the description will count the total number of unique strings, and the frequency with which the most common string occurs. The most frequent string will also be included. Our *Fish* column contains 4 unique strings, and each string is only mentioned once. Therefore, we expect the most frequent string to be chosen at random, with a frequency of 1.

### Listing 8.8 Summarizing the string columns

```
print(df.describe(include=[np.object]))
```

```
count           4  ❶
unique          4  ❷
top     Zebrafish  ❸
freq            1  ❹
```

❶ The number of strings in each column.

❷ The mean of unique strings in each column.

❸ The most frequently occurring string in each column.

❹ The frequency with which the most frequent string occurs.

**SIDEBAR** **Pandas Summarization Methods**

`df.head()`: Returns the first 5 rows in data frame `df`.

`df.head(x)`: Returns the first `x` rows in data frame `df`.

`df.describe()`: Returns statistics relating to numeric columns in `df`.

`df.describe(include=[np.object]`: Returns statistics relating to string columns in `df`.

`df.mean()`: Returns the mean of all numeric columns in `df`.

As alluded to in the previous paragraph, the *Fish* column is built on-top of NumPy string array. In fact, the entire data frame is built on-top of a 2-dimensional NumPy array. Pandas stores all data in NumPy for quick manipulation. We can easily retrieve the underlying NumPy array by accessing `df.values`.

**Listing 8.9 Retrieving the table as a 2D NumPy array**

```
print(df.values)
assert type(df.values) == np.ndarray
```

```
[['Angelfish' 15.2 7.7]
 ['Zebrafish' 6.5 2.1]
 ['Killifish' 9.0 4.5]
 ['Swordtail' 6.0 2.0]]
```

## 8.3 Retrieving Table Columns

Let's turn our attention to retrieving individual columns. The columns can be accessed using their column names. We can output all possible column names by calling `print(df.columns)`.

**Listing 8.10 Accessing all column names**

```
print(df.columns)
```

```
Index([u'Fish', u'Length', u'Width'], dtype='object')
```

Now let's print all data stored in the column *Fish*. We'll do this by accessing `df.Fish`.

**Listing 8.11 Accessing an individual column**

```
print(df.Fish)
```

```
0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

Please note that the printed output is not a NumPy array. Instead, `df.Fish` is Pandas object that represents a 1-dimensional array. In order to print a NumPy array, we must run `print(df.Fish.values)`.

**Listing 8.12 Retrieving a column as a NumPy array**

```
print(df.Fish.values)
assert type(df.Fish.values) == np.ndarray
```

```
['Angelfish' 'Zebrafish' 'Killifish' 'Swordtail']
```

We've accessed the *Fish* column using `df.Fish`. We can also access *Fish* using a dictionary-style bracket representation. Below, we'll print `df['Fish']`.

### Listing 8.13 Accessing a column using brackets

```
print(df['Fish'])
```

```
0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

The bracket representation allows us to retrieve multiple columns at once. If we wish to retrieve multiple columns, we simply execute `df[name_list]`, where `name_list` represents a list of column names. Suppose we want to retrieve both the *Fish* column and the *Length* column. Running `df[['Fish', 'Length']]` will return a truncated table containing only those two columns.

### Listing 8.14 Accessing multiple column using brackets

```
print(df[['Fish', 'Length']])
```

```
        Fish  Length
0  Angelfish    15.2
1  Zebrafish     6.5
2  Killifish     9.0
3  Swordtail     6.0
```

We can analyze data stored within `df` in variety of useful ways. We could, for instance, sort our rows based on a value of single column. Calling `df.sort_values('Length')` will return a new table whose rows are sorted based on length.

### Listing 8.15 Sorting rows by column value

```
print(df.sort_values('Length'))
```

```
        Fish  Length  Width
3  Swordtail     6.0    2.0
1  Zebrafish     6.5    2.1
2  Killifish     9.0    4.5
0  Angelfish    15.2    7.7
```

Furthermore, we can leverage values within columns to filter out unwanted rows. For example, calling `df[df.Width >= 3]` will return a table whose rows contain a width of at-least 3 cm.

### Listing 8.16 Filtering rows by column value

```
print(df[df.Width >= 3])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
2  Killifish     9.0    4.5
```

**SIDEBAR** **Pandas Column Retrieval Methods**

`df.columns`: Returns the column names in data frame `df`.

`df.x`: Returns column `x`.

`df[x]`: Returns column `x`.

`df[[x,y]]`: Returns columns `x` and `y`

`df.x.values`: Returns column `x` as a NumPy array.

`df.sort_values(x)`: Returns a data frame sorted by the values in column `x`.

`df[df.x > y]`: Returns a data frame filtered by the values in column `x` that are > `y`.

## 8.4 Retrieving Table Rows

Now let's turn our attention to retrieving rows within `df`. Unlike columns, our rows do not have preassigned label values. To compensate, Pandas assigns a special index value for each row. These indices appear on the left-most side of the printed table. Based on the printed output, the index for the *Angelfish* row is 0, and the index for the *Swordtail* row is 3. We can access these rows by calling `df.loc[[0, 3]]`. As a general rule, executing `df.loc[[index_list]]` will locate all the rows whose indices appear in `index_list`. Let's now locate the rows that align with the *Swordtail* and *Angelfish* indices.

**Listing 8.17 Accessing rows by index**

```
print(df.loc[[0, 3]])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

Suppose we wish to retrieve rows using species names and not numeric indices. More precisely, we want to retrieve those rows whose Fish column contains either `'Angelfish'` or `'Swordtail'`. In Pandas, that retrieval process is a bit tricky. We need to execute `df[booleans]`, where `booleans` is list of `True` or `False` values that are `True` if they match a row of interest. Basically, the indices of True values must correspond to rows that match either `'Angelfish'` or `'Whitefish'`. How do we obtain the `booleans` list? One naïve approach is to iterate over `df.Fish`, returning `True` if a column-value appears in `['Angelfish', 'Swordtail']`. Let's run the naïve approach below.

**Listing 8.18 Accessing rows by column value**

```
booleans = [name in ['Angelfish', 'Swordtail']
            for name in df.Fish]
print(df[booleans])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

Our code has located the rows of interest. However, we can more concisely locate these rows by leveraging the Pandas `isin` method. Calling `df.Fish.isin(['Angelfish', Swordtail'])` will return an analogue of our previously computed `booleans` list. Thus, we can retrieve all rows in single line of code by running `df[df.Fish.isin(['Angelfish', 'Swordtail'])]`.

**Listing 8.19 Accessing rows by column value using `isin`**

```
print(df[df.Fish.isin(['Angelfish', 'Swordtail'])])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

The `df` table stores 2 measurements across 4 species of fish. We can easily access measurements in the columns. Accessing rows by species, however, is harder. This is because row indices do not directly equal the species names. Let's remedy the situation by replacing the row indices with species. We'll swap numbers for species-names using the `df.set_index` method. Calling `df.set_index('Fish', inplace=True)` will set our indices to equal the species in the Fish column. The `inplace=True` parameter will modify the indices internally, rather than returning a modified copy of `df`.

**Listing 8.20 Swapping row indices for column values**

```
df.set_index('Fish', inplace=True)
print(df)
```

```
           Length  Width
Fish
Angelfish    15.2    7.7
Zebrafish     6.5    2.1
Killifish     9.0    4.5
Swordtail     6.0    2.0
```

The left-most index column is no longer numeric. It has been replaced with species-names. We can now access the *Angelfish* and *Swordtail* columns by running `df.loc[['Angelfish', 'Swordtail']`.

**Listing 8.21 Accessing rows by string index**

```
print(df.loc[['Angelfish', 'Swordtail']])
```

```
           Length  Width
Fish
Angelfish    15.2    7.7
Swordtail     6.0    2.0
```

**SIDEBAR** **Pandas Row Retrieval Methods**

- `df.loc[[x, y]]`:
  Returns the rows located at indices `x` and `y`.
- `df[booleans]`:
  Returns the rows where `booleans[i]` is `True` for column `i`.
- `df[name in array for name in df.x]`:
  Returns rows where column name `x` is present in `array`.
- `df[df.x.isin(array)])`:
  Returns rows where column name `x` is present in `array`.
- `df.set_index('x', inplace=True)`:
  Swaps numeric row indices for the column values in column `x`.

## 8.5 Modifying Table Rows and Columns

Currently, each table-row contains the length and width of a specified fish. What will happen if we swap our rows and columns? We can find out by running `df.T`. The `T` stands for **transpose**. In a transpose operation, the elements of a table are flipped around its diagonal so that the rows and columns are switched. Let transpose our table and print the results.

**Listing 8.22 Swapping rows and columns**

```
df_transposed = df.T
print(df_transposed)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail
Length       15.2        6.5        9.0        6.0
Width         7.7        2.1        4.5        2.0
```

We've modified the table. Each column now refers to an individual species of fish. Meanwhile, each row refers to a particular measurement type. The first row holds length, and the second row holds width. Thus, calling `print(df_transposed.Swordtail)` will print out the swordtail's length and width.

**Listing 8.23 Printing a transposed column**

```
print(df_transposed.Swordtail)
```

```
Length    6.0
Width     2.0
Name: Swordtail, dtype: float64
```

Let's try to modify our transposed table. We'll add the measurements of a clownfish to `df_transposed`. The length and width of a clownfish are 10.6 cm and 3.7 cm, respectively. We'll add these measurements by running `df_transposed['Clownfish'] = [10.6, 3.7]`.

**Listing 8.24 Adding a new column**

```
df_transposed['Clownfish'] = [10.6, 3.7]
print(df_transposed)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length       15.2        6.5        9.0        6.0       10.6
Width         7.7        2.1        4.5        2.0        3.7
```

Alternatively, we can assign new columns using the `df_transposed.assign` method. The method lets us add multiple columns by passing in more than one column name. For instance, calling `df_transposed.assign(Clownfish2=[10.6, 3.7], Clownfish3=[10.6, 3.7])` return a table with two new columns; *Clownfish2* and *Clownfish3*. Please note that the assign method will never add new columns directly to a table. Instead, it will return a copy of the table containing the new data.

**Listing 8.25 Adding multiple new columns**

```
df_new = df_transposed.assign(Clownfish2=[10.6, 3.7], Clownfish3=[10.6, 3.7])
assert 'Clownfish2' not in df_transposed.columns
assert 'Clownfish2' in df_new.columns
print(df_new)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish  Clownfish2  \
Length       15.2        6.5        9.0        6.0       10.6        10.6
Width         7.7        2.1        4.5        2.0        3.7         3.7

Fish    Clownfish3
Length        10.6
Width          3.7
```

Our newly added columns are redundant. We'll delete these columns by calling `df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)`. The `df_new.drop` method will drop all specified columns from a table.

**Listing 8.26 Deleting multiple columns**

```
df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)
print(df_new)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length       15.2        6.5        9.0        6.0       10.6
Width         7.7        2.1        4.5        2.0        3.7
```

Let's do something productive with our table. We'll utilize the stored measurements in order to compute the surface area of each fish. We can treat every fish as an ellipse. The area of an ellipse equals `math.pi * length * width / 4`. In order to find each area, we must iterate over the values in every column. Iterating over columns within a data frame is just like iterating over

elements within a dictionary. We simply execute `df_new.items()`. This returns an iterable of tuples, containing column names and also column values. Let's iterate over the columns in `df_new` to get the area of every fish.

### Listing 8.27 Iterating over column values

```
areas = []
for fish_species, (length, width) in df_new.items():
    area = math.pi * length * width / 4
    print(f"Area of {fish_species} is {area}")
    areas.append(area)
```

Let's add the computed areas to our table. We can augment a new *Area* row by executing `df_new.loc['Area'] = areas`. Afterwards, we'll need to run `df_new.reindex()` to update the row indices with the added *Area* name.

### Listing 8.28 Adding a new row

```
df_new.loc['Area'] = areas
df_new.reindex()
print(df_new)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length  15.200000   6.500000   9.000000   6.000000  10.600000
Width    7.700000   2.100000   4.500000   2.000000   3.700000
Area    91.923001  10.720685  31.808626   9.424778  30.803316
```

Our updated table contains 3 rows and 5 columns. We can confirm this using `df_new.shape`.

### Listing 8.29 Checking table shape

```
row_count, column_count = df_new.shape
print(f"Our table contains {row_count} rows and {column_count} columns")
```

```
Our table contains 3 rows and 5 columns
```

**SIDEBAR** **Modifying Data Frames in Pandas**

- `df.T`:
  Returns a transposed data frame, where rows and columns are swapped.
- `df[x] = array`:
  Creates a new column `x`. `df.x` will map to values in `array`.
- `df.assign(x=array)`:
  Returns a data frame containing all the elements of `df`, and also a new column `x`. `df.x` will map to values in `array`.
- `df.assign(x=array, y=array2)`:
  Returns a data frame containing 2 new columns, `x` and `y`.
- `df.drop(columns=[x, y])`:
  Returns a data frame in which columns `x` and `y` have been deleted.
- `df.drop(columns=[x, y], inplace=True)`:
  Deletes columns `x` and `y` in place, thus modifying `df`.
- `df.loc[x]`:
  Adds a new row at index `x`. We'll need to run `df.reindex()` for that row to be accessible.

## 8.6 Saving and Loading Table Data

We've finished making changes to the table. Let's store the table for later use. Calling `df_new.to_csv('Fish_measurements.csv')` will save the table to a CSV file. Within that CSV file, columns will be delimited by commas.

**Listing 8.30 Saving a table to a CSV file**

```
df_new.to_csv('Fish_measurements.csv')
with open('Fish_measurements.csv') as f:
    print(f.read())
```

```
,Angelfish,Zebrafish,Killifish,Swordtail,Clownfish
Length,15.2,6.5,9.0,6.0,10.6
Width,7.7,2.1,4.5,2.0,3.7
Area,91.92300104403735,10.720684930375171,31.808625617596654,9.42477796076938,30.80331596844792
```

The CSV file can be loaded into Pandas using the `pd.read_csv` method. Calling `pd.read_csv('Fish_measurements.csv', index_col=0)` will return a data frame containing all our table information. The optional `index_col` parameter will specify which column holds the row-index names. If no column is specified, then numeric row indices will be automatically assigned.

**Listing 8.31 Loading a table from a CSV file**

```
df = pd.read_csv('Fish_measurements.csv', index_col=0)
print(df)
print("\nRow index names when column is assigned:")
print(df.index.values)

df_no_assign = pd.read_csv('Fish_measurements.csv')
print("\nRow index names when no column is assigned:")
print(df_no_assign.index.values)
```

```
        Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length  15.200000   6.500000   9.000000   6.000000  10.600000
Width    7.700000   2.100000   4.500000   2.000000   3.700000
Area    91.923001  10.720685  31.808626   9.424778  30.803316

Row index names when column is assigned:
['Length' 'Width' 'Area']

Row index names when no column is assigned:
[0 1 2]
```

Using `pd.csv`, we can load our ad-click table into Pandas. Afterwards, we'll be able to efficiently analyze that table.

**SIDEBAR** **Saving and Loading Data Frames in Pandas**

- `pd.DataFrame(dictionary)`:
  Converts the data in `dictionary` to a data frame.
- `pd.read_csv(filename')`:
  Converts a CSV file into to a data frame.
- `pd.read_csv(filename, index_col=i)`:
  Converts a CSV file into to a data frame. The ith column provides row-index names.
- `df.to_csv(filename)`:
  Saves the contents of `df` to a CSV file.

## 8.7 Visualizing Tables Using Seaborn

We can view the contents of a Pandas table using a simple print command. However, some numeric tables are too large to be viewed as printed output. Such tables are more easily displayed using heatmaps. A **heatmap** is graphical representation of a table, in which numeric cells are colored by value. The color-shades shift continuously, depending on the value size. The end result is a bird's-eye view of value differences within the table.

The easiest way to create a heatmap is to use the external Seaborn library. Seaborn is visualization library that is built on-top of Matplotlib, and is closely integrated with Pandas data frames. Lets install the library. Afterwards, we'll import seaborn as `sns`.

**NOTE** Call "pip install seaborn" from the command-line terminal in order to install the Seaborn library.

**Listing 8.32 Importing the Seaborn library**

```
import seaborn as sns
```

Now, we'll visualize our data frame as heatmap by calling `sns.heatmap(df)`.

**Listing 8.33 Visualizing a heatmap using Seaborn**

```
sns.heatmap(df)
plt.show()
```


**Figure 8.1 A heatmap of fish measurements. Its color-legend specifies the mapping between measurements and colors. Darker colors correspond to lower measurement values. Lighter colors correspond to higher measurement values.**

We've plotted a heatmap of fish measurements. The displayed colors correspond with measurement values. The mappings between color-shades and values are visualized in a legend, to the right of the plot. Lighter colors map to higher measurement values. Thus, we can immediately tell that area of an angelfish is the highest measurement in the plot.

We can alter that color-pallet within the heatmap plot by passing in a `cmap` parameter. Below, we'll execute `sns.heatmap(df, cmap='YlGnBu')`. This will create a heatmap where the color-shades transition from yellow to green, and then to blue.

**Listing 8.34 Adjusting heatmap colors**

```
sns.heatmap(df, cmap='YlGnBu')
plt.show()
```


**Figure 8.2 A heatmap of fish measurements. Darker colors correspond to higher measurement values. Lighter colors correspond to lower measurement values.**

Within the updated heatmap, the color-tones have flipped. Now, lighter colors correspond to higher measurements. We can confirm this by annotating the plot with the actual measurement values. We'll annotate the heatmap by passing `anot=True` into the `sns.heatmap` method.

## Listing 8.35 Annotating the heatmap

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.show()
```

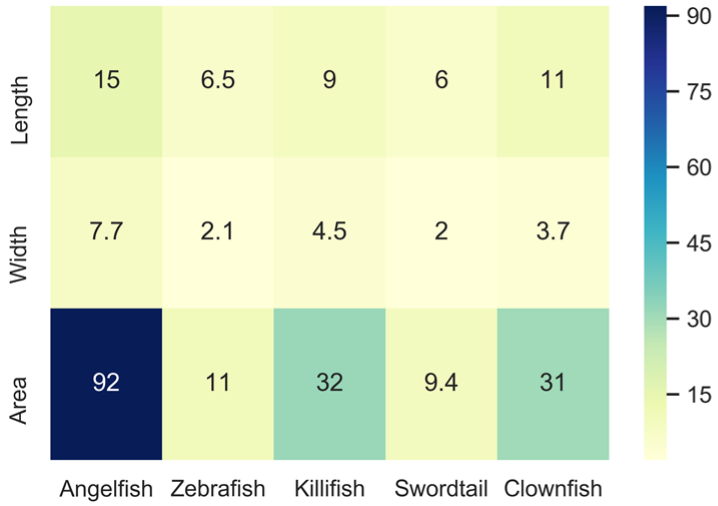


**Figure 8.3 A heatmap of fish measurements. The actual measurement values are included in the plot.**

As mentioned previously, the Seaborn library is built on top of Matplotlib. Consequently, we can use Matplotlib commands to modify elements of the heatmapt. For example, calling `plt.yticks(rotation=0)` will rotate the y-axis measurement labels. This will make the labels easier to read.

## Listing 8.36 Rotating heatmap labels using Matplotlib

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.show()
```

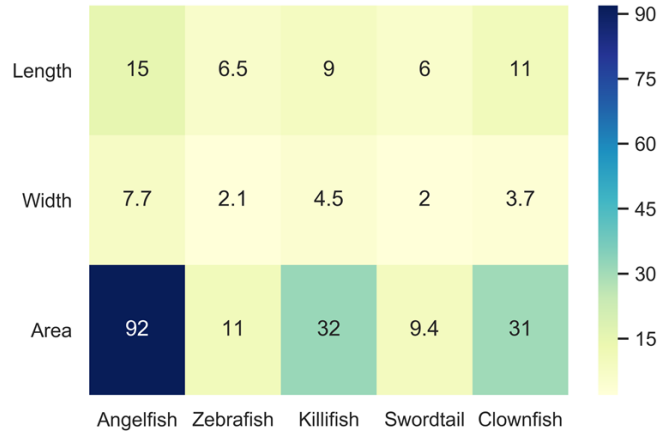


**Figure 8.4 A heatmap of fish measurements. The y-axis measurement labels have been rotated horizontally, for easier viewing.**

Finally, we should note that the `sns.heatmap` method can also process 2D lists and arrays. Thus, running `sns.heatmap(df.values)` will also create a heatmap plot. However, the y-axis and x-axis labels will be missing from that plot. In order to specify the labels, we will need to set the `xticklabels` and `yticklabels` parameters within the `sns.heatmap` method.

The following code will use our table's array representation to replicate the contents of Figure 4.

## Listing 8.37 Visualizing a heatmap from a NumPy array

```
sns.heatmap(df.values,    ❶
            cmap='YlGnBu', annot=True,
            xticklabels=df.columns,    ❷
            yticklabels=df.index)    ❸
plt.yticks(rotation=0)
plt.show()
```

❶ As a reminder, `df.values` returns a 2D NumPy array underlying the data frame.

❷ The x-axis fish labels are manually set to equal column names.

❸ The y-axis measurement labels are manually set to equal row indices.

**SIDEBAR** **Seaborn Heatmap Visualization Commands**

- `sns.heatmap(array)`:
  Generates a heatmap from the contents of the 2D `array`.
- `sns.heatmap(array, xticklabels=x, yticklabels)`:
  Generates a heatmap from the contents of the 2D `array`. The x-labels and y-labels are set to equal `x` and `y`, respectively.
- `sns.heatmap(df)`:
  Generates a heatmap from the contents of the data frame `df`. The x-labels and y-labels are automatically set to equal `df.columns` and `df.index`, respectively.
- `sns.heatmap(df, cmap=m)`:
  Generates a heatmap where the color-scheme is specified by `m`.
- `sns.heatmap(df, annot=True)`:
  Generates a heatmap where the annotated values are included in the plot.

## 8.8 Summary

- 2D table structures can easily be processed using Pandas. We can load the data into Pandas using dictionaries, or external files.
- Pandas stores each table in a data frame, which is built on top of a NumPy array.
- Columns in a data frame have a name. We can use these names to access the columns. Meanwhile, the rows within a data frame are assigned numeric indices, by default. We can use these indices to access the rows. Also, It is possible to swap the numeric row indices for string names.
- We can the summarize the contents of a data frame using the `describe` method. The method returns valuable statistics, such as mean and standard deviation.
- We can visualize the contents of data frame using a colored heat map.

# 9 Case Study 2 Solution

## 9.1 Overview

Our advertising data-table monitors ad-clicks across 30 different colors. We aim is to discover an ad-color that generates significantly more clicks than blue. We will do so by:

**A.** Loading and cleaning our advertising data using Pandas.

**B.** Running a Permutation test between blue and the other recorded colors.

**C.** Checking computed p-values for statistical significance using a properly determined significance level.

> **WARNING** Spoiler alert! The solution to Case Study 2 is about to be revealed. We strongly encourage you to try and solve the problem prior to reading the solution. The original problem statement is available for reference at the beginning of Part 2.

## 9.2 Processing the Ad-Click Table in Pandas

Let's begin by loading our ad-click table into Pandas. Afterwards, we'll check the number of rows and columns in the table.

**Listing 9.1 Loading ad-click table into Pandas**

```
df = pd.read_csv('colored_ad_click_table.csv')
num_rows, num_cols = df.shape
print(f"Table contains {num_rows} rows and {num_cols} columns")
```

```
Table contains 30 rows and 41 columns
```

Our table contains 30 rows and 41 columns. The rows should correspond to clicks-per-day and views-per-day associated with individual colors. Let's confirm by checking the column names.

### Listing 9.2 Checking column names

```
print(df.columns)
```

```
Index(['Color', 'Click Count: Day 1', 'View Count: Day 1',
       'Click Count: Day 2', 'View Count: Day 2', 'Click Count: Day 3',
       'View Count: Day 3', 'Click Count: Day 4', 'View Count: Day 4',
       'Click Count: Day 5', 'View Count: Day 5', 'Click Count: Day 6',
       'View Count: Day 6', 'Click Count: Day 7', 'View Count: Day 7',
       'Click Count: Day 8', 'View Count: Day 8', 'Click Count: Day 9',
       'View Count: Day 9', 'Click Count: Day 10', 'View Count: Day 10',
       'Click Count: Day 11', 'View Count: Day 11', 'Click Count: Day 12',
       'View Count: Day 12', 'Click Count: Day 13', 'View Count: Day 13',
       'Click Count: Day 14', 'View Count: Day 14', 'Click Count: Day 15',
       'View Count: Day 15', 'Click Count: Day 16', 'View Count: Day 16',
       'Click Count: Day 17', 'View Count: Day 17', 'Click Count: Day 18',
       'View Count: Day 18', 'Click Count: Day 19', 'View Count: Day 19',
       'Click Count: Day 20', 'View Count: Day 20'],
      dtype='object')
```

The columns are consistent with our expectations. The first column contains all analyzed colors. The remaining 40 columns hold the click counts and the view counts for each day of the experiment. As a sanity check, let's examine the quality of data stored within our table. We'll start by outputting the analyzed color-names.

### Listing 9.3 Checking color names

```
print(df.Color.values)
```

```
['Pink' 'Gray' 'Sapphire' 'Purple' 'Coral' 'Olive' 'Navy' 'Maroon' 'Teal'
 'Cyan' 'Orange' 'Black' 'Tan' 'Red' 'Blue' 'Brown' 'Turquoise' 'Indigo'
 'Gold' 'Jade' 'Ultramarine' 'Yellow' 'Virdian' 'Violet' 'Green'
 'Aquamarine' 'Magenta' 'Silver' 'Bronze' 'Lime']
```

30 common colors are present in the *Color* column. The first letter of every color-name is capitalized. Thus, we can confirm that the color blue is present by executing `assert 'Blue' in df.Color`.

### Listing 9.4 Checking for blue color

```
assert 'Blue' in df.Color.values
```

The string-based Color column looks good. Let's turn our attention to the remaining 40 numeric columns. Outputting all 40 columns would lead to an overwhelming amount of data. Instead, we'll examine columns for the first day of the experiment; *Click Count: Day 1* and *View Count: Day 1*. We'll select these 2 columns and use `describe()` to summarize their contents.

### Listing 9.5 Summarizing Day 1 of the experiment

```
selected_columns = ['Color', 'Click Count: Day 1', 'View Count: Day 1']
print(df[selected_columns].describe())
```

```
       Click Count: Day 1  View Count: Day 1
count           30.000000              30.0
```

```
mean            23.533333             100.0
std              7.454382               0.0
min             12.000000             100.0
25%             19.250000             100.0
50%             24.000000             100.0
75%             26.750000             100.0
max             49.000000             100.0
```

The values in the *Click Count: Day 1* column range from 12 clicks to 49 clicks. Meanwhile, the minimum and maximum values in *View Count: Day 1* are both equal to 100 views. Therefore, all the values in that column are equal to 100 views. This behavior is expected. We had been specifically informed that each color receives 100 daily views. Let's confirm that all the daily views equal 100.

### Listing 9.6 Confirming equivalent daily views

```
view_columns = [column for column in df.columns if 'View' in column]
assert np.all(df[view_columns].values == 100)   ❶
```

❶ Efficient NumPy code to ensure that values in a NumPy array equal 100.

All view-counts equal 100. Therefore, all 20 *View Count* columns are redundant. We can delete them from our table.

### Listing 9.7 Deleting view-counts from the table

```
df.drop(columns=view_columns, inplace=True)
print(df.columns)
```

```
Index(['Color', 'Click Count: Day 1', 'Click Count: Day 2',
       'Click Count: Day 3', 'Click Count: Day 4', 'Click Count: Day 5',
       'Click Count: Day 6', 'Click Count: Day 7', 'Click Count: Day 8',
       'Click Count: Day 9', 'Click Count: Day 10', 'Click Count: Day 11',
       'Click Count: Day 12', 'Click Count: Day 13', 'Click Count: Day 14',
       'Click Count: Day 15', 'Click Count: Day 16', 'Click Count: Day 17',
       'Click Count: Day 18', 'Click Count: Day 19', 'Click Count: Day 20'],
      dtype='object')
```

The redundant columns have been removed. Only the color and click-count data remains. Our 20 *Click Count* columns correspond to number of clicks per 100 daily views. Consequently, we can treat these counts as percentages. Effectively, the color in each row is mapped to the percentage of daily ad-clicks. Let's summarize the percentage of daily ad-clicks for blue ads. To generate that summary, we'll index each row by color, and then call `df.T.Blue.describe()`.

### Listing 9.8 Summarizing daily blue-click statistics

```
df.set_index('Color', inplace=True)
print(df.T.Blue.describe())
```

```
count    20.000000
mean     28.350000
std       5.499043
min      18.000000
25%      25.750000
```

```
50%      27.500000
75%      30.250000
max      42.000000
Name: Blue, dtype: float64
```

The daily click-percentages for blue range from 18% to 42%. The mean percent of clicks is 28.35%. On average, 28.35% of blue ads receive a click per every view. This average click-rate is pretty good. How does it compare to the other 29 colors? We are ready to find out.

## 9.3 Computing P-values from Differences in Means

Let's start by filtering the data. We'll delete blue, leaving behind the other 29 colors. Afterwards, we'll transpose our table in order to access colors by column name.

**Listing 9.9 Creating a non-blue table**

```
df_not_blue = df.T.drop(columns='Blue')
print(df_not_blue.head(2))
```

```
Color               Pink  Gray  Sapphire  Purple  Coral  Olive  Navy  Maroon  \
Click Count: Day 1    21    27        30      26     26     26    38      21
Click Count: Day 2    20    27        32      21     24     19    29      29

Color               Teal  Cyan  ...  Ultramarine  Yellow  Virdian  Violet  \
Click Count: Day 1    25    24  ...           49      14       27      15
Click Count: Day 2    25    22  ...           41      24       23      22

Color               Green  Aquamarine  Magenta  Silver  Bronze  Lime
Click Count: Day 1     14          24       18      26      19    20
Click Count: Day 2     25          28       21      24      19    19

[2 rows x 29 columns]
```

Our `df_not_blue` table contains the percent-clicks for 29 colors. We would like to compare these percentages to our blue percentages. More precisely, we want to know if there exists a color whose mean click-rate is statistically different from the mean-click rate of blue. How will we compare these means? The sample mean for every color is easily obtainable, but we do not have a population mean. Thus, our best option is to run a Permutation test. In order to run the test, we will need to define a reusable Permutation test function. The function will take as input 2 NumPy arrays. It will return a p-value as its output.

### Listing 9.10 Defining a Permutation test function

```
def permutation_test(data_array_a, data_array_b):
    data_mean_a = data_array_a.mean()
    data_mean_b = data_array_b.mean()
    extreme_mean_diff = abs(data_mean_a - data_mean_b)   ❶
    total_data = np.hstack([data_array_a, data_array_b])
    number_extreme_values = 0.0
    for _ in range(30000):
        np.random.shuffle(total_data)
        sample_a = total_data[:data_array_a.size]
        sample_b = total_data[data_array_a.size:]
        if abs(sample_a.mean() - sample_b.mean())  >= extreme_mean_diff:   ❷
            number_extreme_values += 1

    p_value = number_extreme_values / 30000
    return p_value
```

❶ The observed difference betweeen sample means.

❷ The difference between resampled means is extremely large.

We'll run a Permutation test between blue and the other 29 colors. Afterwards, we'll sort these colors based on their p-value results. Our outputs will be visualized as heatmap, to better emphasize the differences between p-values.

### Listing 9.11 Running a Permutation test across colors

```
np.random.seed(0)
blue_clicks = df.T.Blue.values
color_to_p_value = {}
for color, color_clicks in df_not_blue.items():
    p_value = permutation_test(blue_clicks, color_clicks)
    color_to_p_value[color] = p_value

sorted_colors, sorted_p_values = zip(*sorted(color_to_p_value.items(),   ❶
                                             key=lambda x: x[1]))
plt.figure(figsize=(3, 10))   ❷
sns.heatmap([[p_value] for p_value in sorted_p_values],   ❸
            cmap='YlGnBu', annot=True, xticklabels=['p-value'],
            yticklabels=sorted_colors)
plt.show()
```

❶ Efficient Python code to sort a dictionary and return 2 lists: a list of sorted values, and a list of associated keys. Each sorted p-value at position `i` aligns with the color in `sorted_colors[i]`.

❷ We adjust the width and height of the plotted heatmap to 3 inches and 10 inches, respectively. These adjustments will improve the quality of our heatmap visualization.

❸ The `sns.heatmap` method takes as its input a 2D table. We thus transform our 1D list of p-values into a 2D table, which contains 29 rows and 1 column.

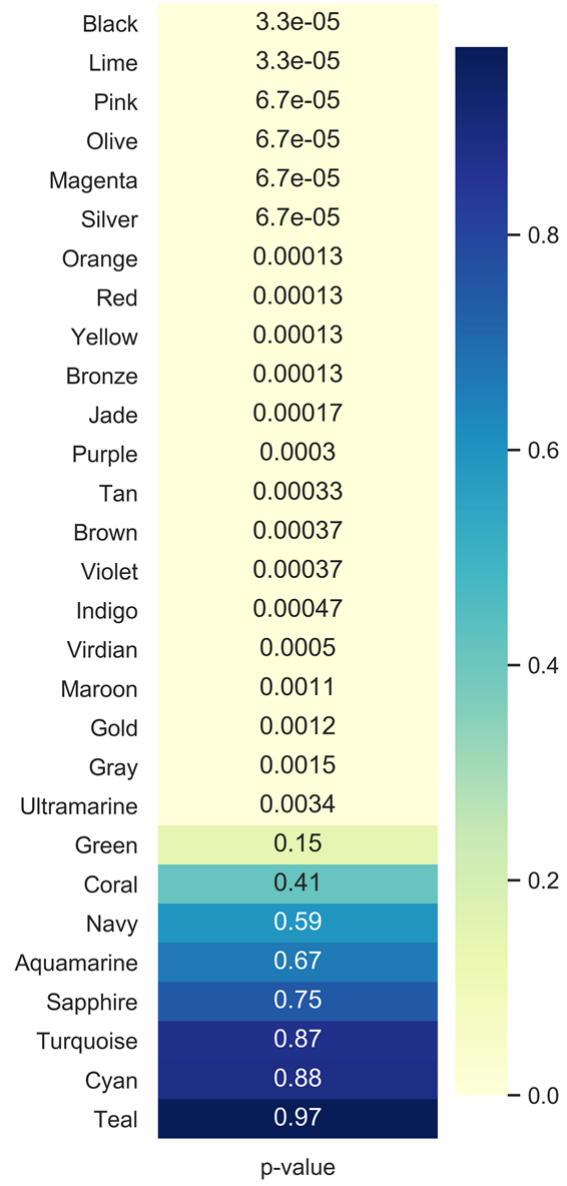


**Figure 9.1 A heatmap of p-value / color-pairs returned by the Permutation test. 21 of the colors map to a p-value that is lower than 0.05.**

The majority of colors generate a p-value that is noticeably lower than 0.05. Black has the lowest p-value. Its ad-click percentages must deviate significantly from blue. Yet, from a design perspective, black is not a very clickable color. Text-links are commonly not black, because black links are hard to distinguish from regular text. Something suspicious is going on here. What exactly is the difference between recorded clicks for black and blue? We can find by printing `df_not_blue.Black.mean()`.

**Listing 9.12 Finding the mean click-rate of black.**

```
mean_black = df_not_blue.Black.mean()
print(f"Mean click-rate of black is {mean_black}")
```

```
Mean click-rate of black is 21.6
```

The mean click-rate of black is 21.6. This value is prominently lower than the blue mean of 28.35. Hence, the statistical difference between the colors is caused by fewer people clicking on black. Perhaps other low p-values are also are also caused inferior click-rates. Let's filter out those colors whose mean is less than the mean of blue. After filtering, we will print the remaining colors.

**Listing 9.13 Filtering colors with inferior click-rates**

```
remaining_colors = df[df.T.mean().values > blue_clicks.mean()].index   ❶
size = remaining_colors.size
print(f"{size} colors have on average more clicks than Blue.")
print("These colors are:")
print(remaining_colors.values)
```

❶ Efficient, one-line code to filter the colors. First, the code creates a Boolean array. The array specifies which colors contain a mean greater than blue. The Boolean array is fed into `df` for filtering. The indices of filtered result specify the remaining color-names.

```
5 colors have on average more clicks than Blue.
These colors are:
['Sapphire' 'Navy' 'Teal' 'Ultramarine' 'Aquamarine']
```

Only 5 colors remain. Each of these colors is a different shade of blue. Let's print the sorted p-values for the 5 remaining colors. We'll also print the mean-clicks, for easier analysis.

**Listing 9.14 Printing the 5 remaining colors**

```
for color, p_value in sorted(color_to_p_value.items(), key=lambda x: x[1]):
    if color in remaining_colors:
        mean = df_not_blue[color].mean()
        print(f"{color} has a p-value of {p_value} and a mean of {mean}")
```

```
Ultramarine has a p-value of 0.0034 and a mean of 34.2
Navy has a p-value of 0.5911666666666666 and a mean of 29.3
Aquamarine has a p-value of 0.6654666666666667 and a mean of 29.2
Sapphire has a p-value of 0.7457666666666667 and a mean of 28.9
Teal has a p-value of 0.9745 and a mean of 28.45
```

## 9.4 Determining Statistical Significance

Four of the colors have large p-values. Only one color has a p-value that's small. That color is ultramarine; a special shade of blue. Its mean of 34.2 is greater than blue's mean of 28.35. Ultramarine's p-value is 0.0034. Is that p-value statistically significant? Well, the p-value is more than 10x lower than the standard significant level of 0.05. However, that significance level does not take into account our comparisons between blue and 29 other colors. Each comparison is an experiment, testing whether a color differs from blue. If we run enough experiments, then we are guaranteed to encounter a low p-value, sooner or later. The best way to correct for this is to execute a Bonferonni correction. Otherwise, we will fall victim to p-value hacking.

Let's carry out a Bonferonni correction. We'll lower the significance level to 0.05 / 29.

### Listing 9.15 Applying the Bonferonni correction

```
significance_level = 0.05 / 29
print(f"Adjusted significance level is {significance_level}")
if color_to_p_value['Ultramarine'] <= significance_level:
    print("Our p-value is statistically significant")
else:
    print("Our p-value is not statistically significant")
```

```
Adjusted significance level is 0.001724137931034483
Our p-value is not statistically significant
```

Our p-value is not statistically significant. Fred had carried out too many experiments for us to drawn a meaningful conclusion. Not all these experiments were necessary. There is no valid reason to expect that black, or brown or gray colors would outperform blue. Perhaps if Fred had disregarded some of these colors, then our analysis would have been more fruitful. Conceivably, if Fred had simply compared blue to the other 5 variants of blue, then maybe we would have obtained a statistically significant result. Let's explore the hypothetical situation where Fred instigates 5 experiments and ultramarine's p-value remains unchanged.

### Listing 9.16 Exploring a hypothetical significance level

```
hypothetical_sig_level = 0.05 / 5
print(f"Hypothetical significance level is {hypothetical_sig_level}")
if color_to_p_value['Ultramarine'] <= hypothetical_sig_level:
    print("Our hypothetical p-value would have been statistically significant")
else:
    print("Our hypothetical p-value would not not have been statistically significant")
```

```
Hypothetical significance level is 0.01
Our hypothetical p-value would have been statistically significant
```

Under these hypothetical conditions, our results would be statistically significant. Sadly, we can't use these hypothetical conditions to lower our significance level. We have no guarantee that re-running the experiments will reproduce a p-value of 0.0034. P-values will fluctuate, and superfluous experiments increase the chance of untrustworthy fluctuations. Given Fred's high experiment count, we simply cannot draw a statistically significant conclusion.

However, not all is lost. Ultramarine still represents a promising substitute for blue. Should Fred carry out that substitution? Perhaps. Let us consider our 2 alternative scenarios. In the first scenario, the null hypothesis is true. If that's the case, then both blue and ultramarine share the same population mean. Under these circumstances, swapping ultramarine for blue will not affect the ad click-rate. In the second scenario, the higher ultramarine click-rate is actually statistically significant. If that's the case, then swapping ultramarine for blue will actually yield more ad-clicks. Therefore, Fred has everything to gain and nothing to lose by setting all his ads to ultramarine.

From a logical standpoint, Fred should definitely swap blue for ultramarine. However, if he

carries out the swap, then some uncertainty will remain. Fred will never know if ultramarine truly returns more clicks than blue. What if Fred's curiosity gets the best of him? If Fred really wants an answer, then his only choice is to run another experiment. In that experiment, half the displayed ads would be blue. All other displayed ads would be ultramarine. Fred's software would exhibit the advertisements while recording all the clicks and views. Afterwards, we could re-compute the p-value. Then, we could compare the p-value to the appropriate significance level. That significance level would remain at 0.05. The Bonferonni correction would not be necessary, because only a single experiment was run. After the p-value comparison, Fred would finally know whether ultramarine outperforms blue.

## 9.5 41 Shades of Blue: A Real-Life Cautionary Tale

Fred assumed that analyzing every single color would yield more impactful results. But Fred was wrong. More data isn't necessarily better. Sometimes, more data leads to more uncertainty.

Fred is not a statistician. He can be forgiven for failing to comprehend the consequences of over-analysis. The same cannot be said of certain quantitative experts operating in business today. Take for example, an notorious incident that occurred at a well-known corporation. The corporation needed to select a color for the web-links on its site. The chief designer chose a visually-appealing shade of blue. A top-level executive distrusted this decision. Why did the designer choose this shade of blue, and not another?

The executive had come from a quantitative background, and insisted that link-color should be selected scientifically. They gave orders for a massive analytic test. The test would supposedly determine the perfect shade of blue. 41 shades of blue were assigned to company web-links, completely at random. Millions of clicks were subsequently recorded. Eventually, the "optimal" shade of blue was selected based on maximum clicks-per-view.

The executive proceeded to make their methodology public. Worldwide, statisticians cringed. The executive's decisions revealed an ignorance of basic statistics. That ignorance embarrassed both the executive and the company.

## 9.6 Key Takeaways

- More data isn't always better. Running a pointless surplus of analytic tests increases our chance of anomalous results.
- It's worth taking the time to think about a problem prior to running an analysis. If Fred had carefully considered the 31 colors, he would have realized that its pointless to test them all. Many of the colors lead to ugly links. Colors like black are very unlikely to yield more clicks than blue. Filtering the color-set would have led to a more informative test.
- Even though Fred's experiment was flawed, we still managed to extract a useful insight. Ultramarine might prove to be a reasonable substitute for blue, though more testing is required. Occasionally, data scientists are presented with flawed data. Despite those flaws, good insights might still be possible.

# Case Study 3: Tracking Disease Outbreaks Using News Headlines

## CS3.1 Problem Statement

Congratulations! You have just been hired by the American Institute of Health. The Institute monitors disease epidemics in both foreign and domestic lands. A critical component of the monitoring process involves analysis of published news data. Each day, the Institute receives hundreds of news headlines describing disease outbreaks in various locations. The news headlines are too numerous to be analyzed by hand.

Your first assignment is as follows; you will process the daily quota of news headlines and extract the locations mentioned within. You will then cluster the headlines based on their geographic distribution. Finally, you will review the largest clusters within the United States and outside of the United States. Any interesting findings should be reported to your immediate superior.

### CS3.1.1 Dataset Description

The file 'headlines.txt' contains the hundreds of headlines that you must analyze. Each headline appears separately on an individual line within the file.

## CS3.2 Overview

In order to address the problem at hand we will need to know how to:

**A.** Cluster datasets using multiple techniques and distance measures.

**B.** Measure distances between locations on a spherical globe.

**C.** Visualize locations on a map.

**D.** Extract location coordinates from headline text.

# 10 Clustering Data into Groups

***This section covers:***

- Clustering data by centrality.
- Clustering data by density.
- Trade-offs between clustering algorithms.
- Executing clustering using the Scikit-Learn Library.
- Iterating over clusters using Pandas

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.

## 10.1 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.

### Listing 10.1 Modeling dart coordinates using 2 Normal distributions

```
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}")
```

```
The x coordinate of a randomly thrown dart is 2.49
The y coordinate of a randomly thrown dart is 0.57
```

**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.

### Listing 10.2 Simulating randomly thrown darts

```
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):   ❶
        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()
```

❶ We can leverage NumPy to execute this loop in just one line of code. Running `x_coordinates, y_coordinates = np.random.multivariate_normal(bulls_eye, np.diag(2 * [variance]), 5000).T` will return 5,000 x and y coordinates sampled from the Multivariate Normal distribution.

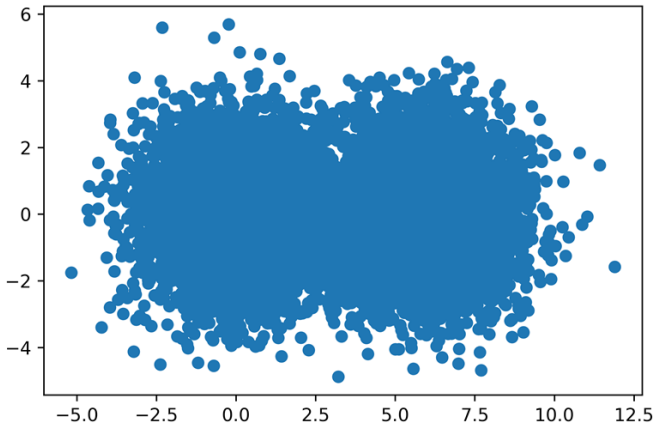


**Figure 10.1 A simulation of darts randomly scattered around 2 bull's-eye targets.**

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, $distance^2 = (x\_dart - x\_bull)^2 + (y\_dart - y\_bull)^2$. 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]`.

**Listing 10.3 Assigning darts to the nearest bull's-eye**

```
from scipy.spatial.distance import euclidean
def nearest_bulls_eye(dart):
    distances = [euclidean(dart, bulls_e) for bulls_e in bulls_eyes]  ❶
    return np.argmin(distances)  ❷

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}")
```

❶ We obtain the Euclidean distance between the dart and each bull's-eye, using the `euclidean` function imported from SciPy.

❷ Returns the index matching the shortest bull's-eye distance in `distances`.

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.

**Listing 10.4 Coloring darts based on nearest bull's-eye**

```
def color_by_cluster(darts):   ❶
    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]]   ❷
        x_coordinates, y_coordinates = np.array(selected_darts).T   ❸
        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))]   ❹
color_by_cluster(darts)
```

❶ This helper function plots the colored elements of an inputted `darts` list. Each `dart` within `darts` serves as input into `nearest_bulls_eye`.

❷ This selects the darts most proximate to `bulls_eyes[bs_index]`. We can more efficiently execute selection by running `selected_darts = np.array(darts)[np.array(closest_bulls_eyes) == bs_index]`.

❸ Separates the x and y coordinates of each dart by transposing an array of selected darts. As discussed in Section Nine, the transpose will swap the row and column positions with a 2D data structure.

❹ Combines the separate coordinates of each dart into a single list of x and y coordinates. We can also execute coordinate combination by running `darts = [list(tuple) for tuple in zip(x_coordinates, y_coordinates)]`.

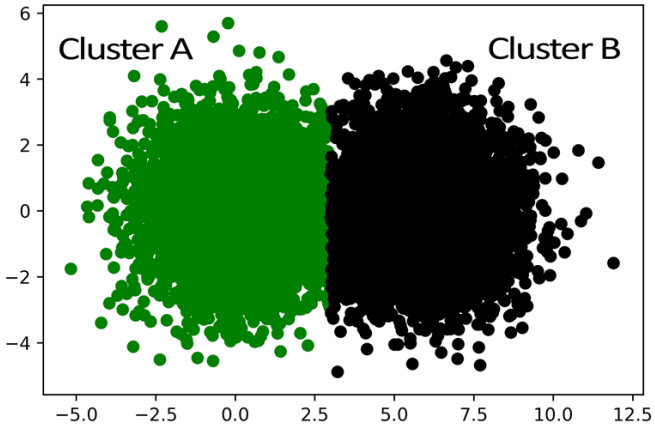


**Figure 10.2 Darts colored based on proximity to the nearest bull's-eye. Cluster A represents all points closest to the left bull`s-eye, and Cluster B represents all points closet to the right bull`s-eye.**

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.

## Listing 10.5 Assigning darts to randomly chosen centers

```
bulls_eyes = np.array(darts[:2])  ❶
color_by_cluster(darts)
```

❶ We randomly select the first two darts to be our representative bulls'-eyes.

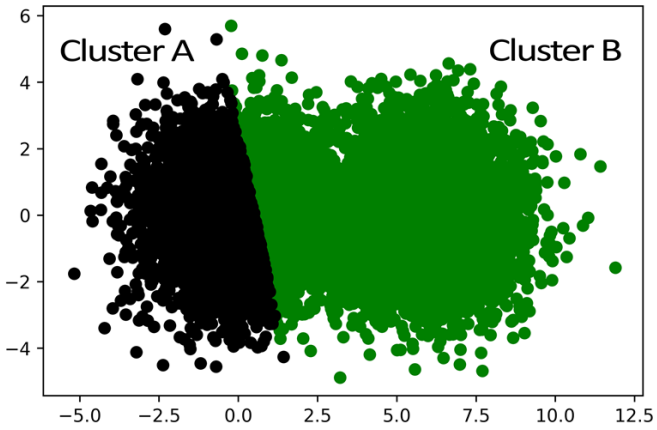


**Figure 10.3 Darts colored based on proximity to randomly selected centers. Cluster B is stretched too far out to the left.**

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.

### Listing 10.6 Assigning darts to centers based on mean

```
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)]  ❶
        updated_bulls_eyes.append(mean_center)

    return updated_bulls_eyes

bulls_eyes = update_bulls_eyes(darts)
color_by_cluster(darts)
```

❶ We take the mean of x and y coordinates for all the darts assigned to a given bull's-eye. These average coordinates are then used to update our estimated bull's-eye position. We can more efficiently run this calculation by executing `mean_center = np.mean(selected_darts, axis=0)`.

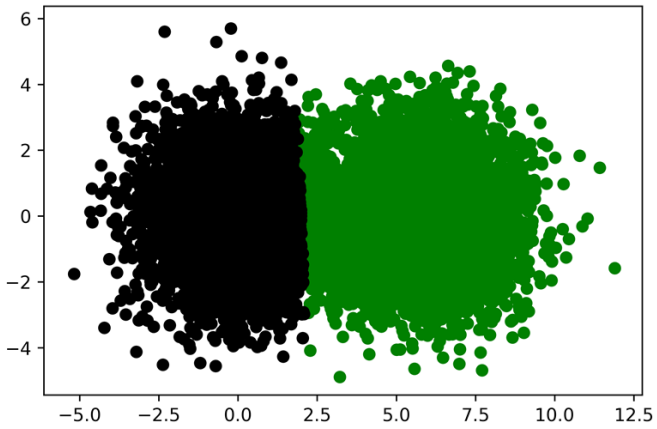


**Figure 10.4 Darts colored based on proximity to recomputed centers. The 2 clusters now appear to be more even.**

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.

## Listing 10.7 Adjusting bull's-eye positions over 10 iterations

```
for i in range(10):
    bulls_eyes = update_bulls_eyes(darts)

color_by_cluster(darts)
```

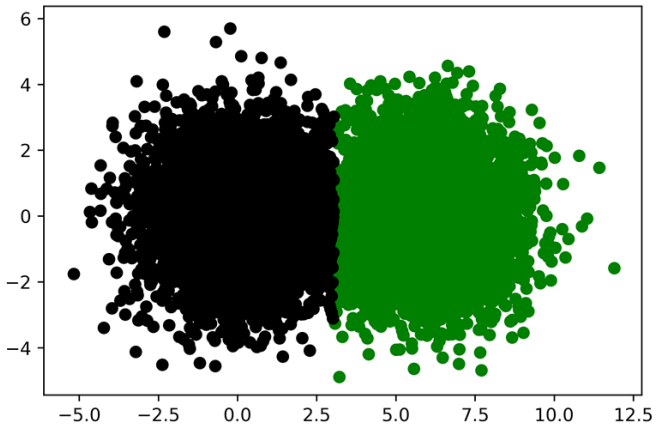


**Figure 10.5 Darts colored based on proximity to iteratively recomputed centers**

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.

## 10.2 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.

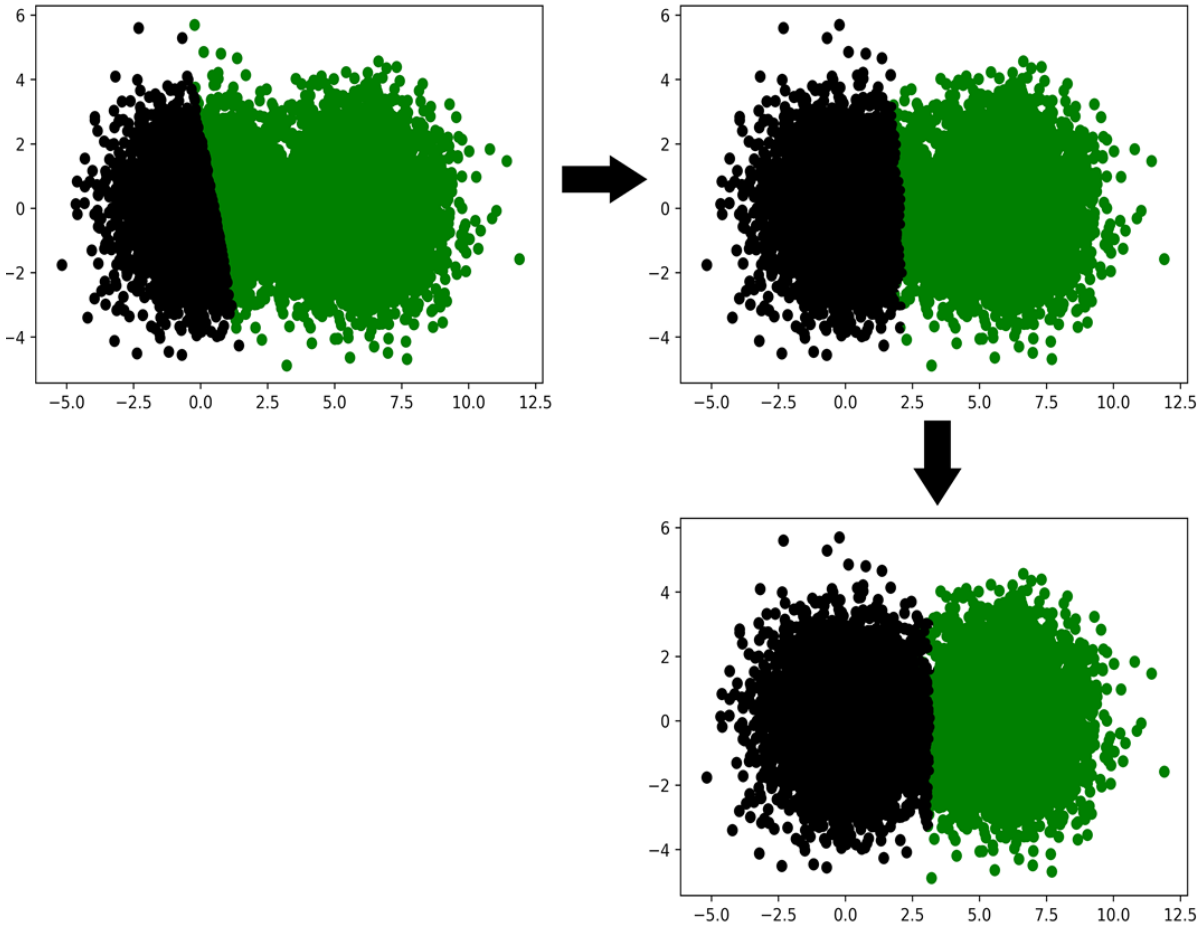


**Figure 10.6 The K-means algorithm iteratively converging from 2 randomly selected centroids to the actual bulls'-eye centroids.**

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.

### *10.2.1 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.

**Listing 10.8 Importing `KMeans` from Scikit-learn**

```
from sklearn.cluster import KMeans
```

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.

**Listing 10.9 K-means clustering using Scikit-learn**

```
cluster_model = KMeans(n_clusters=2)                    ❶
assigned_bulls_eyes = cluster_model.fit_predict(darts)  ❷

print("Bull's-eye assignments:")
print(assigned_bulls_eyes)
```

```
Bull's-eye assignments:
[0 0 0 ... 1 1 1]
```

❶ Creates a `cluster_model object` in which the number of centers is set to 2.

❷ Optimizes 2 centers using the K-means algorithm, and returns the assigned cluster for each dart.

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

**Listing 10.10 Plotting K-means cluster assignments**

```
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()
```

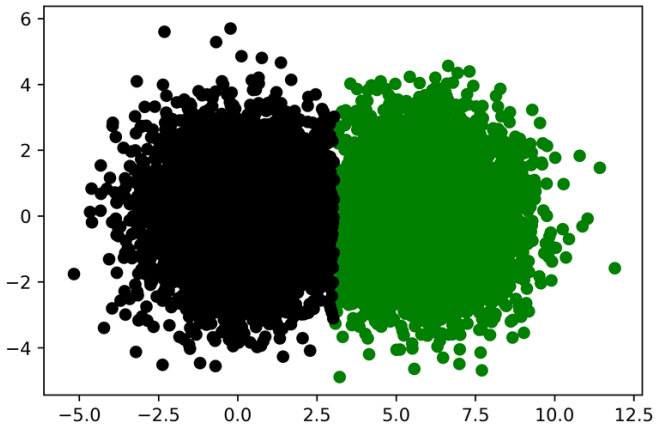


**Figure 10.7 The K-means clustering results returned by Scikit-learn are consistent with our expectations.**

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.

**Listing 10.11 Using `cluster_model` to cluster new data**

```
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}")
```

```
Dart at [500, 500] is closest to bull's-eye 0
Dart at [-500, -500] is closest to bull's-eye 1
```

### *10.2.2 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.

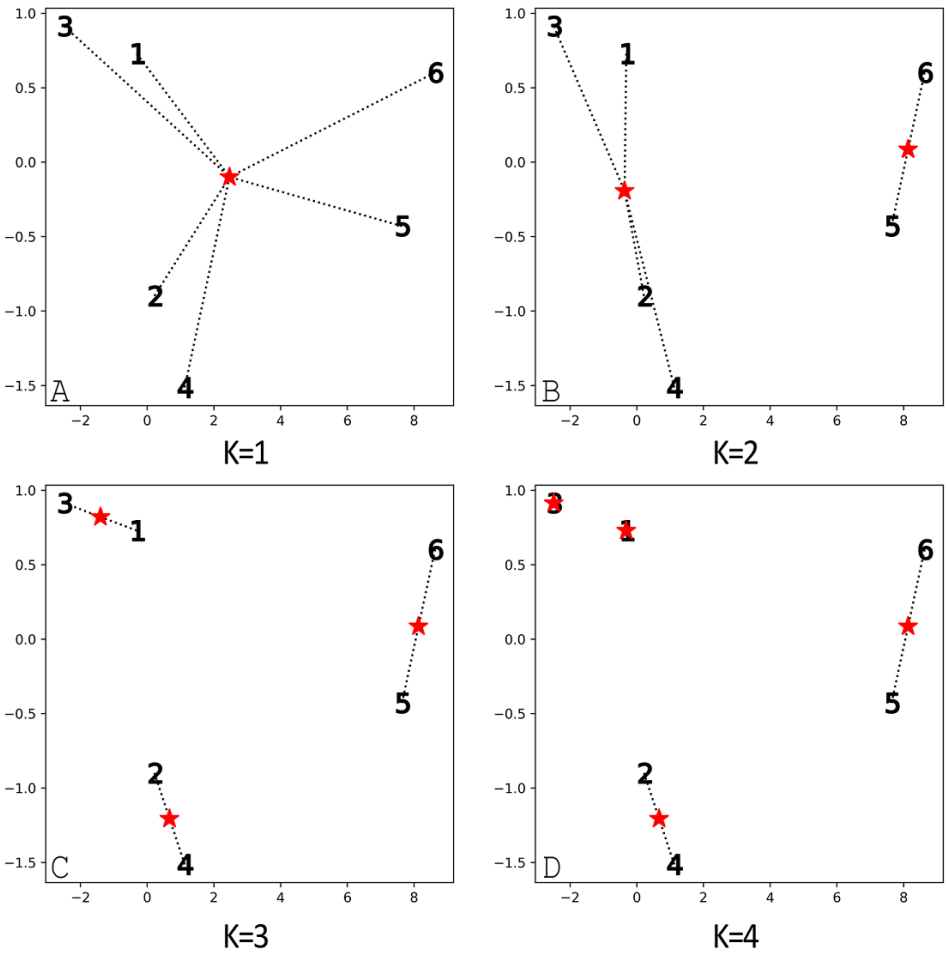


**Figure 10.8 Six points, numbered 1 through 6, are plotted in 2D space. Centers, marked by stars, are computed across various values of K. A line is drawn from every point to its nearest center. Inertia is computed by summing the squared lengths of the six lines. A) K=1. All six lines stretch out from 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.

**Listing 10.12 Plotting the K-means inertia**

```
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()
```

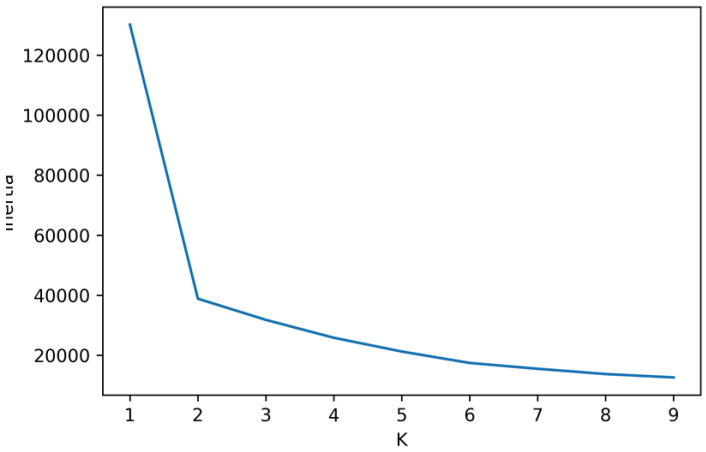


**Figure 10.9 An inertia plot for a dartboard simulation containing 2 bull's-eyes targets. The plot resembles an arm bent at the elbow. The elbow points directly to a K of 2.**

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.

**Listing 10.13 Plotting inertia for a 3-dartboard simulation**

```
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()
```

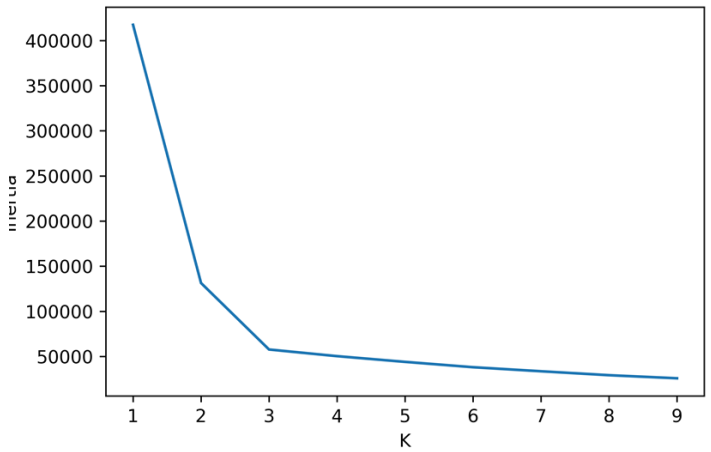


**Figure 10.10 An inertia plot for a dartboard simulation containing 3 bull's-eyes targets. The plot resembles an arm bent at the elbow. The lower-most portion of the elbow points to a K of 3.**

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.

**SIDEBAR** **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.

## 10.3 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.

## Listing 10.14 Simulating rings around a planet

```
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,    ❶
                                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()
```

❶ The `make_circles` function creates two concentric circles in 2D. The scale of the smaller circle's radius relative to the larger circle is determined by the `factor` parameter.

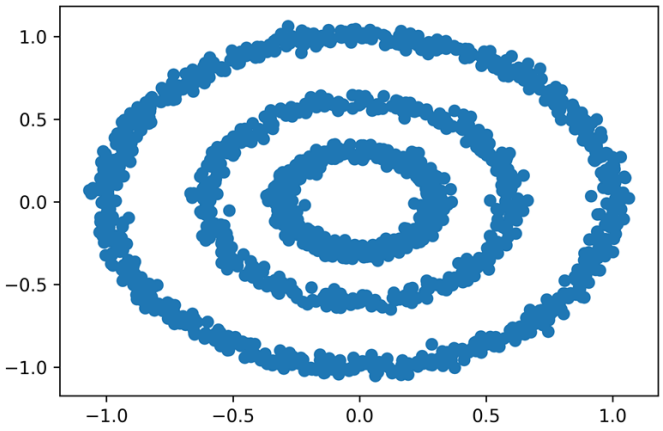


**Figure 10.11 A simulation of 3 rock rings positioned around a central point.**

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.

## Listing 10.15 Using K-means to cluster rings

```
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()
```

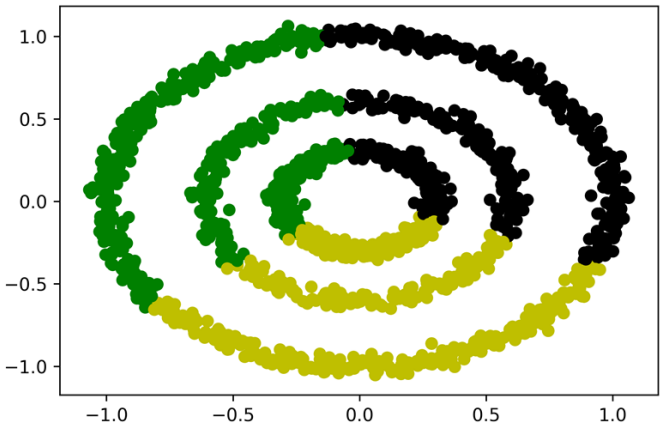


**Figure 10.12 K-means clustering fails to properly identify the 3 distinct rock rings**

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.

**Listing 10.16 Specifying density parameters**

```
epsilon=0.1
min_points = 10
```

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.

**Listing 10.17 Finding the neighbors of `rocks[0]`**

```
neighbor_indices = [i for i, rock in enumerate(rocks[1:])
                    if euclidean(rocks[0], rock) <= epsilon]
```

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.

**Listing 10.18 Checking the density of `rocks[0]`**

```
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.")
```

```
The rock at index 0 has 40 neighbors.
It lies in a dense region.
```

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:

**A.** If a point is located within an `epsilon` distance of `min_point` neighbors, then that point is in a dense region of space.

**B.** 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.

**Listing 10.19 Creating a dense cluster**

```
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")
```

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

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`.

**Listing 10.20 Expanding a dense cluster**

```
dense_region_indices = set(dense_region_indices)   ❶
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")
```

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

❶ We convert `dense_region_indices` into a set. This allows us to update the set with additional indice without worrying about duplicates.

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.

## *10.4 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:

**A.** We select a random `point` coordinate from a `data` list.

**B.** We obtain all neighbors within an `epsilon` distance of that `point`.

**C.** 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.

**D.** 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.

**E.** 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.

**Listing 10.21 Using `DBSCAN` to cluster rings**

```
from sklearn.cluster import DBSCAN
cluster_model = DBSCAN(eps=epsilon, min_samples=min_points)   ❶
rock_clusters = cluster_model.fit_predict(rocks)   ❷
colors = [['g', 'y', 'k'][cluster] for cluster in rock_clusters]
plt.scatter(x_coordinates, y_coordinates, color=colors)
plt.show()
```

❶ Creates a `cluster_model object` to carry out density clustering. An `epsilon` value of 0.1 is passed-in using the `eps` paramter. A `min_points` value of 10 is passed-in using the `min_samples` parameter.

❷ Clusters the rock-rings based on density. Returns the assigned cluster for each rock.

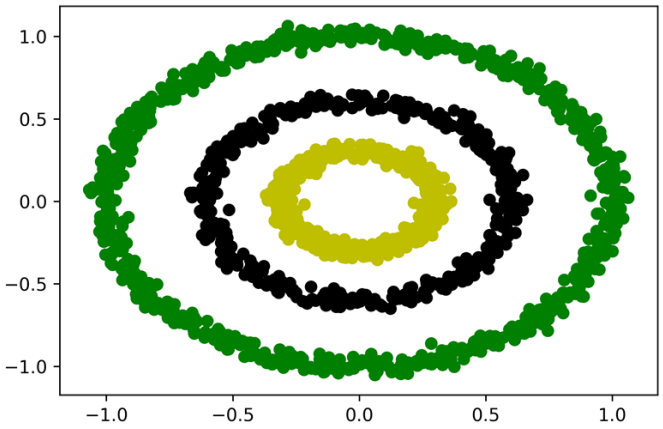


**Figure 10.13 DBSCAN clustering accurately identifies the 3 distinct rock rings.**

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

### 10.4.1 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.

**Listing 10.22 Finding outliers using DBSCAN**

```
noisy_data = rocks + [[1000, -1000]]
clusters = DBSCAN(eps=epsilon,
                  min_samples=min_points).fit_predict(noisy_data)
assert clusters[-1] == -1
```

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.

### 10.4.2 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.

**Listing 10.23 Computing the Manhattan distance**

```
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)   ❶

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

❶ We can also generate this output by importing `cityblock` from `scipy.spatial.distance` and afterwards running `.17 * cityblock(x, y)`.

```
Manhattan distance is 4.42 miles
```

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.

## Listing 10.24 Clustering using Manhattan distance

```
points = [[35, 5], [33, 6], [37, 4], [40, 7], [45, 5]]
clusters = DBSCAN(eps=1, min_samples=3,
                  metric=manhattan_distance).fit_predict(points)   ❶

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')   ❷
    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)   ❸
plt.minorticks_on()


plt.show()
```

❶ The `manhattan_distance` function is passed into DBSCAN through the `metric` paramter.

❷ Outliers are plotted using x-shaped markers.

❸ The `grid` method displays the rectangular grid across which we compute Manhattan distance.

```
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
```

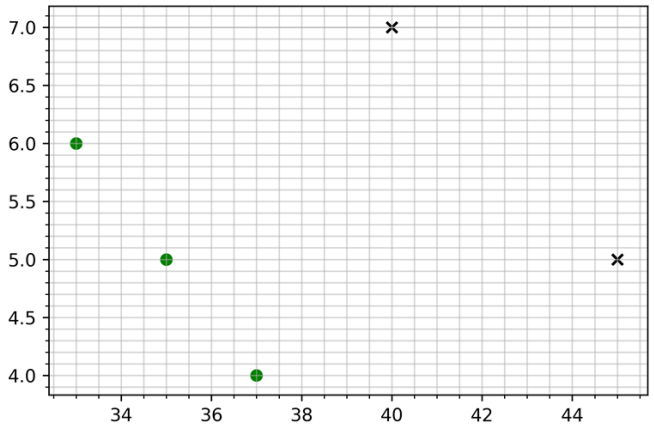


**Figure 10.14 5 points in a rectangular grid have been clustered using the Manhattan distance. The 3 points in the lower-left corner of the grid all fall within a single cluster. The remaining two points are outliers, marked by an x.**

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.

**Listing 10.25 Clustering using a ridiculous measure of distance**

```
def ridiculous_measure(point_a, point_b):
    is_negative_a = np.array(point_a) < 0   ❶
    is_negative_b = np.array(point_b) < 0
    if is_negative_a.all() and is_negative_b.all():   ❷
        return 0
    elif is_negative_a.any() and is_negative_b.any():   ❸
        return 10
    else:   ❹
        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}")
```

❶ Returns a Boolean array, where `is_negative_a[i]` is `True` if `point_a[i] < 0`.

❷ All elements of `point_a` and `point_b` are negative.

❸ A negative element exists but not all elements are negative.

❹ All elements are non-negative.

```
[-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
```

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.

**SIDEBAR** **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.

### 10.4.3 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.

## 10.5 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]`.

**Listing 10.26 Storing clustered coordinates in a table**

```
import pandas as pd
x_coordinates, y_coordinates = np.array(rocks).T
df = pd.DataFrame({'X': x_coordinates, 'Y': y_coordinates,
                   'Cluster': rock_clusters})
```

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.

**Listing 10.27 Plotting a single cluster using Pandas**

```
df_cluster = df[df.Cluster == 0]   ❶
plt.scatter(df_cluster.X, df_cluster.Y)   ❷
plt.show()
```

❶ We select just those rows where the *Cluster* column equals 0.

❷ We plot the *X* and *Y* columns of the selected rows. Please note that we also execute the scatter plot by running `df_cluster.plot.scatter(x='X', y='Y')`.

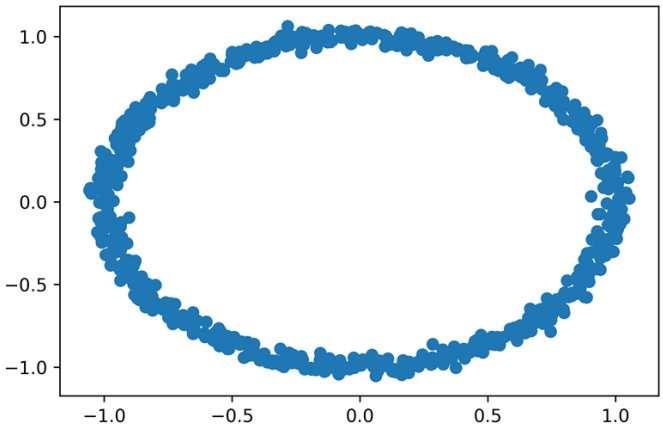


**Figure 10.15 Rocks that fall into cluster zero.**

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.

**Listing 10.28 Iterating over clusters using Pandas**

```
for cluster_id, df_cluster in df.groupby('Cluster'):   ❶
    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()
```

```
Skipping over cluster 0
Plotting cluster 1
Plotting cluster 2
```

❶ Each element of the iterable returned by `df.groupby('Cluster')` is a tuple. The first element of the tuple is the cluster id obtained from `df.Cluster`. The second element is a table composed of all those rows where `df.Cluster` equals the cluster id.

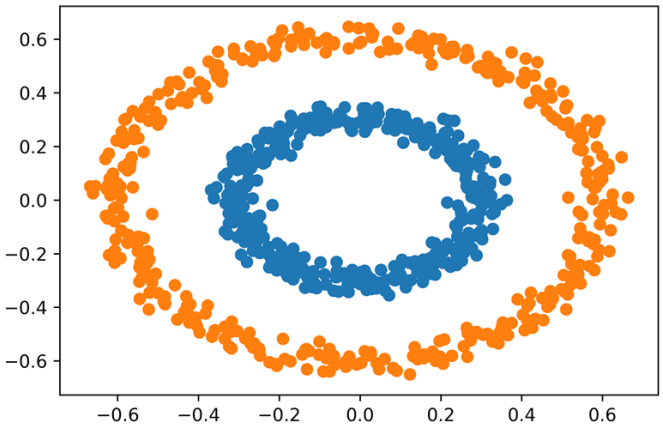


**Figure 10.16 Rocks that fall into clusters 1 and 2.**

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

## 10.6 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.

# 11 *Geographic Location Visualization and Analysis*

***This section covers:***

- Computing the distance between geographic locations.
- Plotting locations on a map using the Basemap library.
- Extracting geo-coordinates from location names.
- Finding location names in text using regular expressions.

People have relied on location information since before the dawn of recorded history. Cave-dwellers once carved maps of hunting routes into mammoth tusks. Such maps evolved as civilizations flourished. The ancient Babylonians fully mapped borders of thei vast empire. Much later, in 3000 BC, Greek scholars improved cartography using mathematical innovation. The Greeks discovered that the Earth was round, and accurately computed the Earth's circumference. Greek mathematicians laid the ground-work for measuring distances across the Earth's curved surface. Such measurements required the creation a geographic coordinate system. A rudimentary system based on latitude and longitude was introduced in 2000 BC.

Combining cartography with latitude and longitude helped revolutionize maritime navigation. Sailors could more freely travel the seas by checking their positions on a map. Roughly speaking, maritime navigation protocols followed these 3 steps:

**A.** Data Observation

- A sailor would record a series of observations. These included wind-direction, the position of the stars, and (after approximately 1300 AD) the northward direction of a compass.

**B.** Mathematical and Algorithmic Analysis of Data

- A navigator would analyze all data in order to estimate the ship's position. Sometimes,

the analysis required trigonometric calculations. More commonly, the navigator would consult a series of rule-based measurement charts. By algorithmically adhering to the rules within the charts, the navigator could figure out the ship's coordinates.

**C.** Visualizing and Decision-Making

- The Captain would examine the computed location on a map, relative to the expected destination. Afterwards, the Captain would give orders to adjust the orientation of the ship, based on the visualized results.

This navigation paradigm perfectly encapsulates the standard data science process. As data scientists, we are offered raw observations. We algorithmically analyze that data. Afterwards, we visualize the results in order to make critical decisions. Thus, data science and location analysis are linked. That link has only grown stronger through the centuries. Today, countless corporations analyze locations in ways that the ancient Greeks could have never imagined. Hedge funds study satellite photos of farmlands to make bets on the global soybean market. Transport-service providers analyze vast traffic patterns to efficiently route their fleet of cars. Epidemiologists process newspaper data to monitor the global spread of disease.

In this section, we explore a variety of techniques for analyzing and visualizing geographic locations. We'll begin with the simple task of calculating the distance between 2 geographic points.

## 11.1 The Great-Circle Distance: A Metric for Computing Distances Between 2 Global Points

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** .

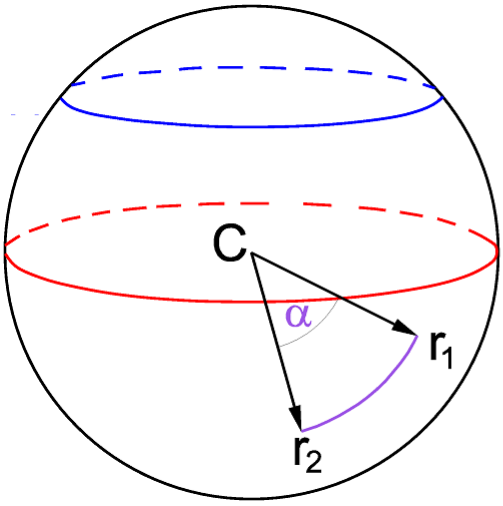


**Figure 11.1 Visualizing the great-circle distance between 2 points on the surface of a sphere. These points are labeled as r1 and r2. A curved arc designates the traveling distance between them. The arc-length is equal to the radius of the sphere multiplied by , where  is the angle between points relative 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.

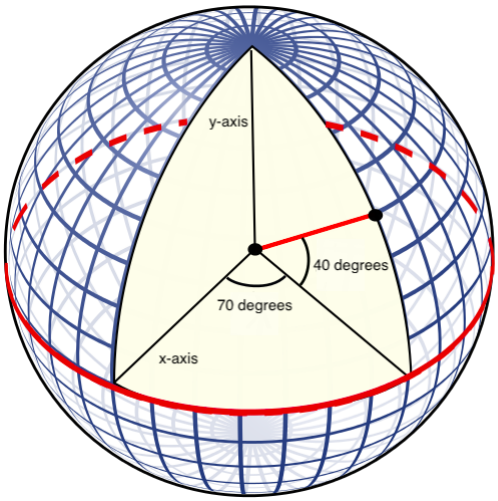


**Figure 11.2 Representing a point on the surface of a sphere using spherical coordinates. The point arises as we rotate 70 degrees away from the x-axis, and 40 degrees towards the y-axis. Hence, its spherical coordinates are (70, 40).**

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.

**Listing 11.1 Defining a great-circle distance function**

```
from math import cos, sin, asin   ❶

def great_circle_distance(x1, y1, x2, y2):
    delta_x, delta_y = x2 - x1, y2 - y2   ❷
    haversin = sin(delta_x / 2) ** 2 + np.product([cos(x1), cos(x2),   ❸
                                                   sin(delta_y / 2) ** 2])
    return 2 * asin(haversin ** 0.5)
```

❶ We import 3 common trigonometric functions from Python's `math` module.

❷ We compute the angular difference between the 2 pairs of spherical coordinates.

❸ We execute a series of well-known trigonometric operations to obtain the great-circle distance on a unit-sphere.

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 2. Traversing the circumference of a circle requires a 360 degree angle. Thus, 2 radians equal 360 degrees, and a single degree equals  / 180 radians.

**Listing 11.2 Computing the great-circle distance**

```
from math import pi
distance = great_circle_distance(0, 0, pi, pi)
print(f"The distance equals {distance} units")
```

```
The distance equals 3.141592653589793 units
```

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.

**Listing 11.3 Defining the coordinates of the Earth's poles**

```
latitude_north, longitude_north = (90.0, 0)   ❶
latitude_south, longitude_south = (-90.0, 0)
```

❶ Technically speaking, the North Pole and the South Pole do not have an official longitude coordinate. However, we're mathematically justified in assigning a zero-longitude to each pole.

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`.

**Listing 11.4 Computing the great-circle distance between poles**

```
to_radians =  np.radians([latitude_north, longitude_north,
                          latitude_south, longitude_south])
distance = great_circle_distance(*to_radians.tolist())   ❶
print(f"The unit-circle distance between poles equals {distance} units")
```

❶ As a reminder, running `func(*[arg1, arg2])` is a Python shortcut for executing `func(arg1, arg2)`.

```
The unit-circle distance between poles equals 3.141592653589793 units
```

As expected, the distance between poles on a unit-sphere is $\pi$ . 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.

**Listing 11.5 Computing the travel distance between Earth's poles**

```
earth_distance = 3956 * distance
print(f"The distance between poles equals {earth_distance} miles")
```

```
The distance between poles equals 12428.14053760122 miles
```

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.

**Listing 11.6 Defining a travel distance function**

```
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
```

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.

## 11.2 Plotting Maps Using Basemap

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.

**Listing 11.7 Initializing the Basemap mapping class**

```
from mpl_toolkits.basemap import Basemap
map_plotter = Basemap()
```

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()`.

**Listing 11.8 Visualizing the Earth using Basemap**

```
fig = plt.figure(figsize=(12, 8))   ❶
map_plotter.drawcoastlines()
plt.show()
```

❶ This method-call ensures that the plotted map will be of adequate size. Passing `figsize=(12, 8)` into `plt.figure` will create a figure that is 12 inches wide and 8 inches high.

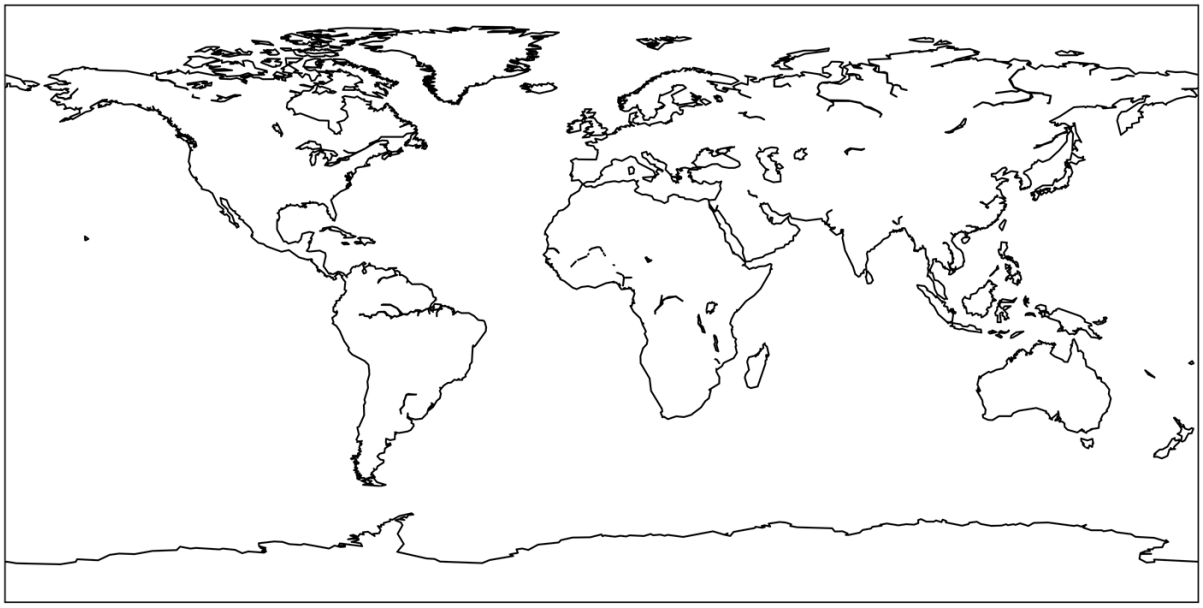

**Figure 11.3 A standard map of the Earth, in which the coastlines of the continents have been plotted.**

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 projection 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.

### Listing 11.9 Mapping coastlines and countries

```
fig = plt.figure(figsize=(12, 8))
map_plotter.drawcoastlines()  ❶
map_plotter.drawcountries()
plt.show()
```

❶ We must execute draw `drawcoastlines()` every time we generate a new plot. Otherwise the coastlines will not be included in the plot.

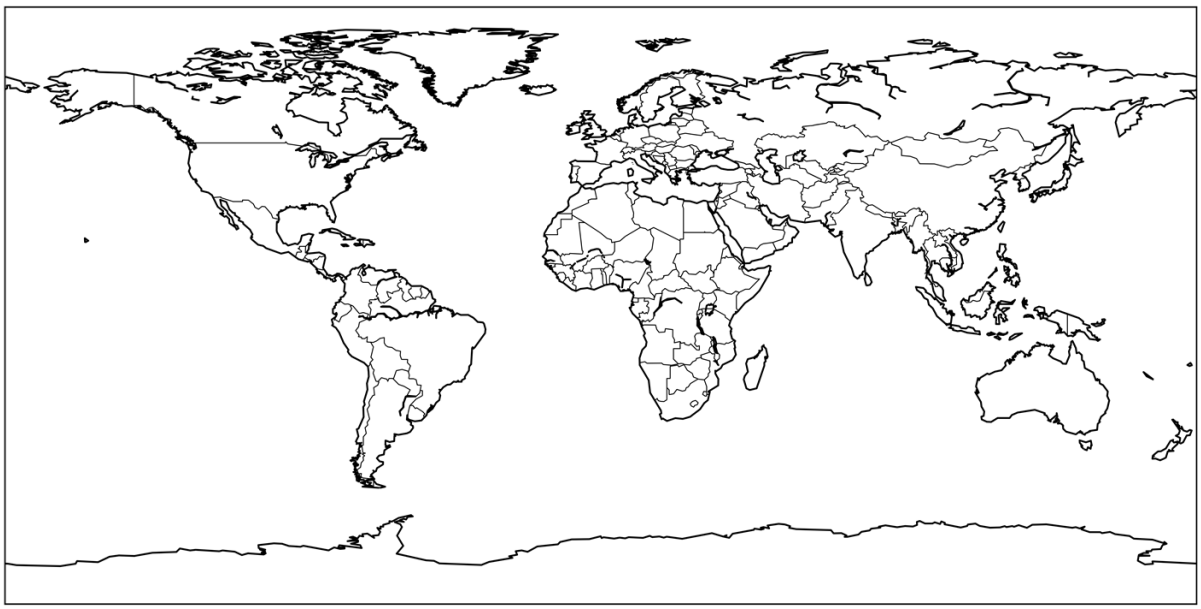

**Figure 11.4 A standard map of the Earth where coastlines and national boundary lines are present.**

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.

### Listing 11.10 Coloring a map of the Earth

```
fig = plt.figure(figsize=(12, 8))
map_plotter.shadedrelief()
plt.show()
```

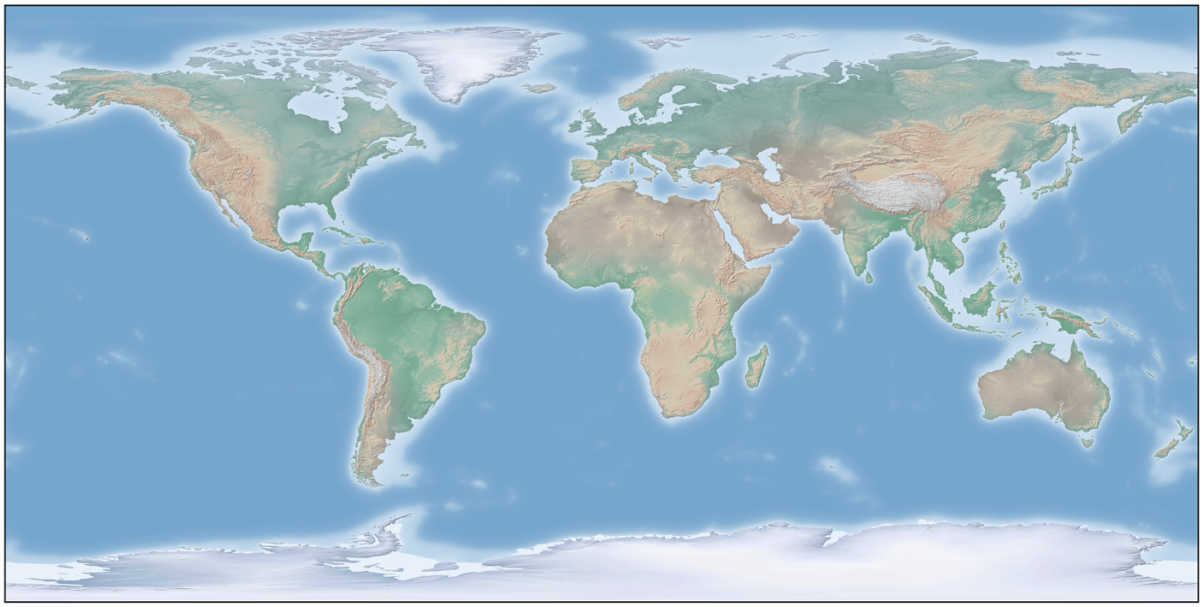

**Figure 11.5 A standard map of the Earth that has been colored to contain oceanographic and topographic details.**

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 `latlon=True` into the `map_plotter.scatter` method, so that the plotted points are treated as spherical coordinates.

### Listing 11.11 Plotting coordinates on a map

```
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()
```

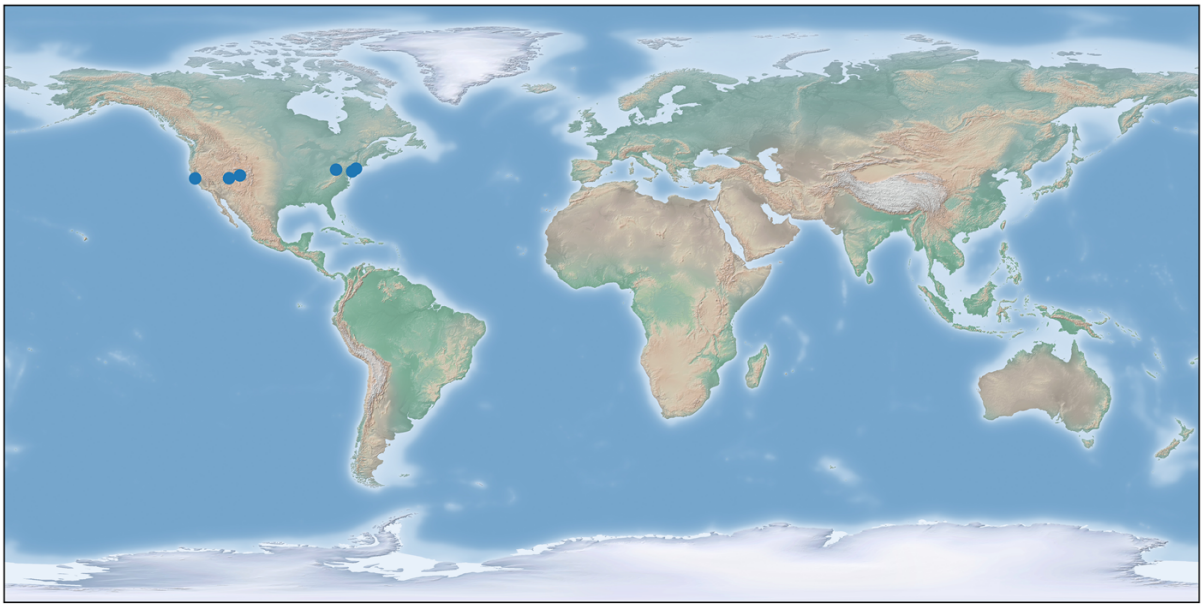

**Figure 11.6 A standard map of the Earth with plotted latitude and longitude coordinates.**

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.

## Listing 11.12 Plotting North American Coordinates

```
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)   ❶
map_ortho.drawcoastlines()
plt.show()
```

❶ The `s` parameter specifies the plotted marker size. We increase that size for better visibility.

**Figure 11.7 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.

## Listing 11.13 Plotting USA Coordinates

```
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()
```

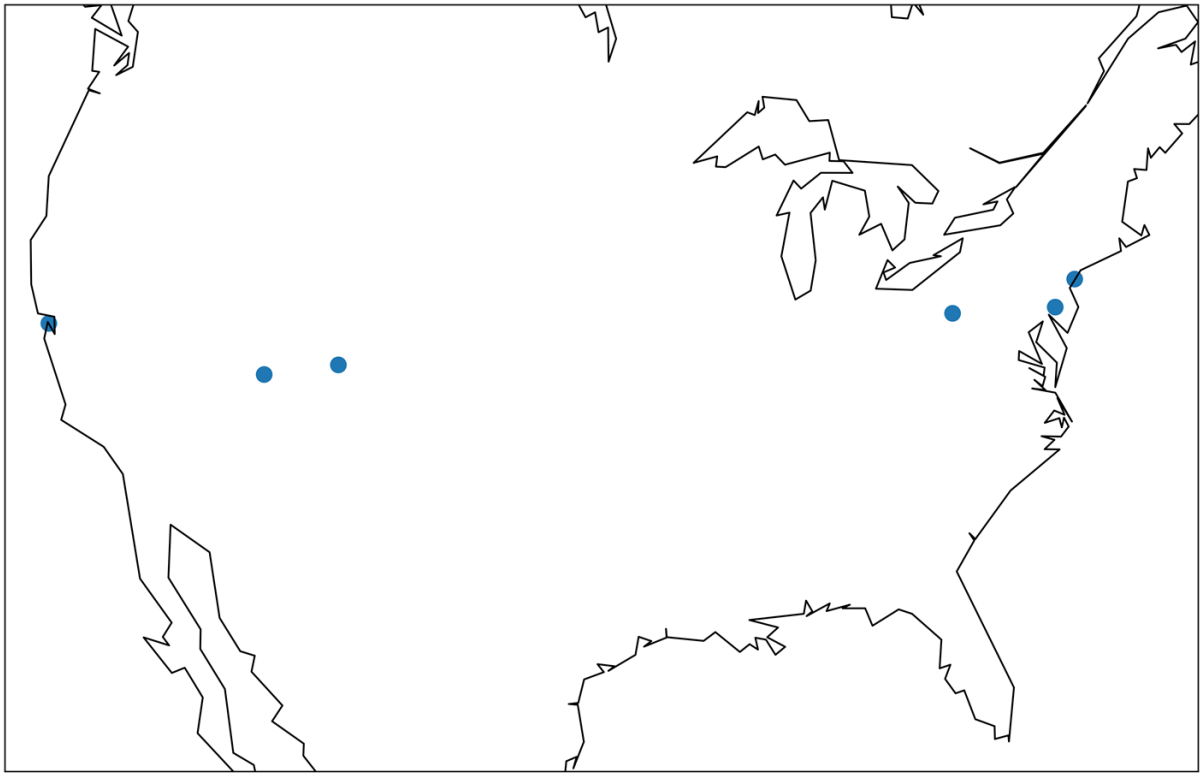

**Figure 11.8 An LCC projection of the United States with plotted latitude and longitude coordinates.**

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

## Listing 11.14 Mapping state boundaries in the USA

```
fig = plt.figure(figsize=(12, 8))
map_lcc.scatter(longitudes, latitudes, latlon=True, s=70)
map_lcc.drawcoastlines()
map_lcc.drawstates()
plt.show()
```

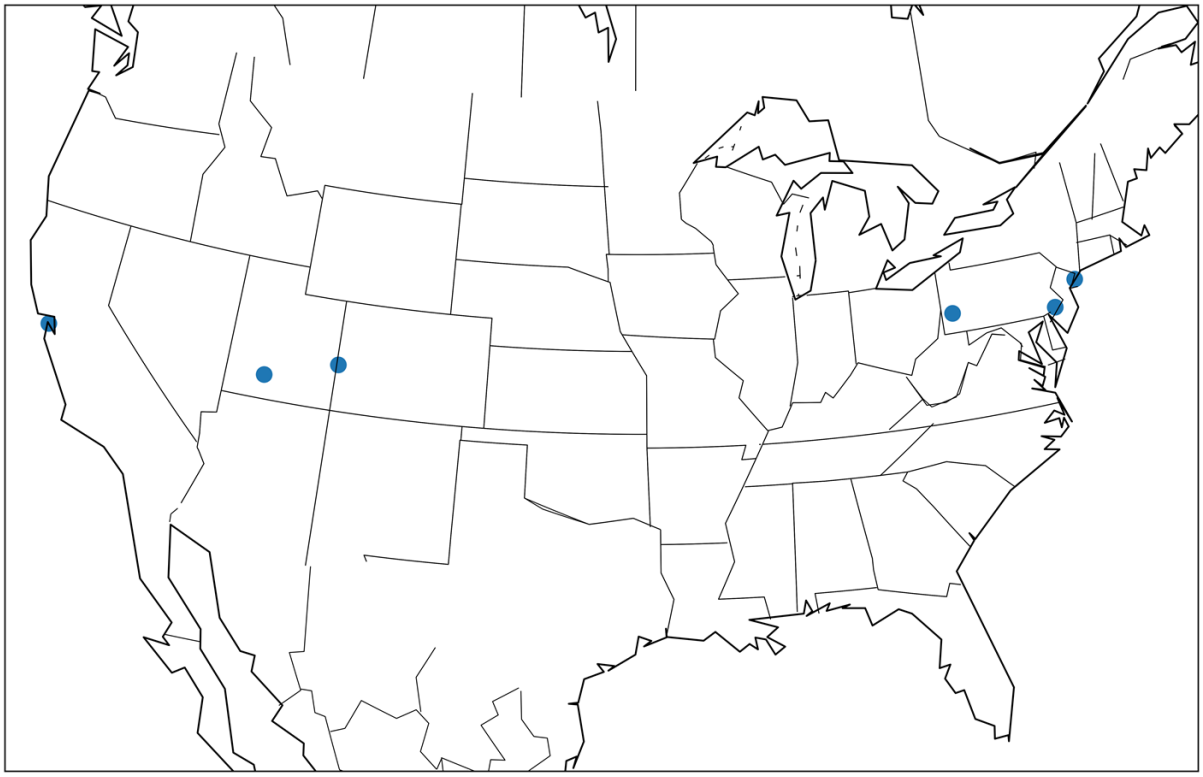

**Figure 11.9 An LCC projection of the United States where state boundaries are present..**

**SIDEBAR** **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.

## 11.3 Location Tracking Using GeoNamesCache

The GeoNames database (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/), 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.

**Listing 11.15 Initializing a `GeonamesCache` object**

```
from geonamescache import GeonamesCache
gc = GeonamesCache()
```

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.

**Listing 11.16 Fetching all 7 continents from GeoNamesCache**

```
continent_dictionary = gc.get_continents()
print(continent_dictionary.keys())
```

```
dict_keys(['AF', 'AS', 'EU', 'NA', 'OC', 'SA', 'AN'])
```

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.

**Listing 11.17 Fetching North America from GeoNamesCache**

```
north_america = continents['NA']
print(north_america.keys())
```

```
dict_keys(['lng', 'geonameId', 'timezone', 'bbox', 'toponymName', 'asciiName', 'astergdem', 'fcl', 'popu
```

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

**Listing 11.18 Printing North America's naming schemas**

```
for name_key in ['name', 'asciiName', 'toponymName']:
    print(north_america[name_key])
```

```
North America
North America
North America
```

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.

**Listing 11.19 Mapping North America's central coordinates**

```
latitude = float(north_america['lat'])    ❶
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()
```

❶ The `'lat'` and `'lng'` keys map to North American's central latitude and longitude.

**Figure 11.10 The central North American latitude and longitude plotted on a map of North America.**

### 11.3.1 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.

**Listing 11.20 Fetching US data from GeoNamesCache**

```
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)
```

```
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',   ❶
'capital': 'Washington',   ❷
'areakm2': 9629091,   ❸
'population': 310232863,   ❹
'tld': '.us',
'currencycode': 'USD',
'currencyname': 'Dollar',   ❺
'phone': '1',
'postalcoderegex': '^\\d{5}(-\\d{4})?$',
'languages': 'en-US,es-US,haw,fr',   ❻
'neighbours': 'CA,MX,CU'}   ❼
```

❶ The continent code of the United States.

❷ The capital of the United States.

❸ The square area, in kilometers, of the United States.

❹ The population size of the United States

❺ The currency of the United States.

❻ Common spoken languages within the United States.

❼ The neighboring territories of the United States.

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.

**Listing 11.21 Fetching neighboring countries**

```
us_neighbors = us_data['neighbours']
for neighbor_code in us_neighbors.split(','):
    print(countries[neighbor_code]['name'])
```

```
Canada
Mexico
Cuba
```

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.

**Listing 11.22 Fetching countries by name**

```
result = gc.get_countries_by_names()['United States']
assert result == countries['US']
```

### 11.3.2 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.

**Listing 11.23 Fetching cities from GeoNamesCache**

```
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])
```

`cities` is a dictionary mapping a unique `city_id` to geographic information.

```
{'geonameid': 3041563,   ❶
'name': 'Andorra la Vella',   ❷
'latitude': 42.50779,   ❸
'longitude': 1.52109,   ❹
'countrycode': 'AD',   ❺
'population': 20430,   ❻
'timezone': 'Europe/Andorra'}   ❼
```

❶ Unique city id.

❷ City name.

❸ Latitude.

❹ Longitude.

❺ Code of the country where the city is found.

❻ Population.

❼ Timezone

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.

**Listing 11.24 Fetching US cities from GeoNamesCache**

```
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.")
```

```
GeoNamesCache holds data for 3248 US cities
```

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.

**Listing 11.25 Approximating US central coordinates**

```
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()
```

**Figure 11.11 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.

### Listing 11.26 Fetching cities by name

```
matched_cities_by_name = gc.get_cities_by_name('Philadelphia')
print(matched_cities_by_name)
```

```
[{'4560349': {'geonameid': 4560349, 'name': 'Philadelphia', 'latitude': 39.95233, 'longitude': -75.16379
```

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.

### Listing 11.27 Fetching multiple cities with a shared name

```
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}")
```

```
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
```

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.

### Listing 11.28 Mapping the most populous San Francisco

```
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)  ❶
plt.text(x, y, ' San Francisco', fontsize=16)
plt.show()
```

❶ We convert the latitude and longitude into regular x and y coordinates in order to add a "San Francisco" label to our map.


**Figure 11.12 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.

**SIDEBAR** **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`.

### 11.3.3 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.

**Listing 11.29 Fetching New York City from GeoNamesCache**

```
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.")
```

```
'New York' is not present in GeoNamesCache database.
'New York City' is present in GeoNamesCache database.
```

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*.

**Listing 11.30 Fetching accented cities from GeoNamesCache**

```
print(gc.get_cities_by_name(u'Cañon City'))
print(gc.get_cities_by_name(u'Hagåtña'))
```

```
[{'5416005': {'geonameid': 5416005, 'name': 'Cañon City', 'latitude': 38.44098, 'longitude': -105.24245,
[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'co
```

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.

**Listing 11.31 Counting all accented cities in GeoNamesCache**

```
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")
```

```
An example accented city name is 'Khawr Fakkn'
4896 cities have accented names
```

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.

**Listing 11.32 Stripping accents in alternative city names**

```
alternative_names = {unidecode(name): name
                     for name in accented_names}
print(gc.get_cities_by_name(alternative_names['Hagatna']))
```

```
[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'co
```

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.

**Listing 11.33 Finding accent-free city-names in text**

```
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
```

```
[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña', 'latitude': 13.47567, 'longitude': 144.74886, 'co
```

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.

## 11.4 Matching Location Names in Text

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.

**Listing 11.34 Basic string matching**

```
assert 'Boston' in 'Boston Marathon'
assert 'Boston Marathon'.startswith('Boston')
assert 'Boston Marathon'.endswith('Boston') == False
```

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'`.

**Listing 11.35 Basic sub-string matching errors**

```
assert 'in a' in 'sin apple'
assert 'in a' in 'win attached'
```

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.

**Listing 11.36 String matching using regexes**

```
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'
```

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

**Listing 11.37 Case-insensitive matching using regexes**

```
for text in ['BOSTON', 'boston', 'BoSTOn']:
    assert re.search(regex, text, flags=re.IGNORECASE) is not None  ❶
```

❶ We can also achieve the same result if we pass `flags=re.I` into `re.search`.

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.

**Listing 11.38 Word boundary matching using regexes**

```
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
```

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'`.

**Listing 11.39 Multi-city matching using regexes**

```
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
```

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`.

**Listing 11.40 String matching using compiled regexes**

```
compiled_re = re.compile(regex)
text = 'I visited Boston yesterday.'
for i in range(1000):
    assert compiled_re.search(text) is not None
```

**SIDEBAR** **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.

## 11.5 Summary

- The shortest travel distance between terrestrial points is along our planet's spherical surface. This **great-circle distance** can be computed using a series of well-known trigonometric operations.
- The latitude and longitude are **spherical coordinates**. These coordinates measure the angular position of a point on the surface of the Earth, relative to the x-axis and y-axis.
- We can plot a latitude and longitude on a map, using the external Basemap library. The library can visualize mapped data using multiple projection types. Our choice of projection is dependent on the plotted data. If the data spans the Globe, then we can use the standard **Equidistant cylindrical projection**. If the data is confined to North America, then we might consider using the **Orthographic projection**. If the data-points are located within the continental United States, then we should use the **Lambart conformal conic projection**.
- We can obtain latitudes and longitudes from location names using the GeoNamesCache library. GeoCacheNames maps city-names to latitudes and longitudes. It also maps country-names to cities. Thus, given a country-name, we can approximate its central coordinates by averaging the latitudes and longitudes of its cities. However, that approximation will not be perfect, due to city-bias and also the curved shape of the Earth.
- Multiple cities commonly share an identical name. Thus, GeoNamesCache can map multiple coordinates to a single city-name. Given only a city-name, we should return the coordinates of the most populous city with that name.
- GeoNamesCache maps coordinates to accented versions of each city-name. We can strip-out these accents using the `unidecode` function from the external `Unidecode` library.
- Regular expressions can find location names in text. By combining GeoNamesCache, with Basemap and regular expressions, we can plot locations that are mentioned in the text.

# 12 Case Study 3 Solution

## 12.1 Overview

Our goal is to extract locations from disease-related headlines in order to uncover the largest active epidemics within and outside of the United States. We will do so by:

**A.** Loading the data.

**B.** Extracting locations from the text using regular expressions and the GeoNamesCache library.

**C.** Checking the location-matches for errors.

**D.** Clustering the locations based on geographic distance.

**E.** Visualizing the clusters on a map and removing any errors.

**F.** Outputting representative locations from the largest clusters in order to draw interesting conclusions.

> **WARNING** Spoiler alert! The solution to Case Study 3 is about to be revealed. We strongly encourage you to try and solve the problem prior to reading the solution. The original problem statement is available for reference at the beginning of Part 3.

## 12.2 Extracting Locations from Headline Data

We'll begin by loading the headline data.

**Listing 12.1 Loading headline data**

```
headline_file = open('headlines.txt','r')
headlines = [line.strip()
             for line in headline_file.readlines()]
num_headlines = len(headlines)
print(f"{num_headlines} headines have been loaded")
```

```
650 headines have been loaded
```

650 headlines have been loaded. Now, we need a mechanism for extracting city and country names from the headline text. One naïve solution is to match the locations in GeoNamesCache against each and every headline. This approach, however, will fail to match locations whose capitalization and accent-marks diverge from the stored GeoNamesCache data. For more optimal matching, we should transform each location name into a case-independent and accent-independent regular expression. We can execute these transformations using a custom `name_to_regex` function. That function will take as input a location name. It will return a compiled regular expression capable of identifying any location of our choosing.

Lets define our `name_to_regex` function.

**Listing 12.2 Converting names to regexes**

```
def name_to_regex(name):
    decoded_name = unidecode(name)
    if name != decoded_name:
        regex = fr'\b({name}|{decoded_name})\b'
    else:
        regex = fr'\b{name}\b'
    return re.compile(regex, flags=re.IGNORECASE)
```

Using `name_to_regex`, we can create create a mapping between regular expressions and the original names in GeoNamesCache. Lets create 2 dictionaries; `country_to_name` and `city_to_name`. The dictionaries will map regular expressions to country-names and city-names, respectively.

**Listing 12.3 Mapping names to regexes**

```
countries = [country['name']
             for country in gc.get_countries().values()]
country_to_name = {name_to_regex(name): name
                   for name in countries}

cities = [city['name'] for city in gc.get_cities().values()]
city_to_name = {name_to_regex(name): name for name in cities}
```

Next, we'll use our mappings to define a function that will look for location names in text. The function will take as input both a headline and a location dictionary. It will iterate over each regex key in the dictionary, returning the associated value if the regex pattern matches to the headline.

**Listing 12.4 Finding locations in text**

```
def get_name_in_text(text, dictionary):
    for regex, name in sorted(dictionary.items(),
                              key=lambda x: x[1]):   ❶
        if regex.search(text):
            return name
    return None
```

❶ Iterating over dictionaries gives us a non-deterministic sequence of results. A change in sequence-order could alter which locations get matched to the inputted `text`. This is especially true if multiple locations are present in `text`. Sorting by location name ensures that function output will not change from run to run.

We'll utilize `get_names_in_text` to discover the cities and countries that are mentioned in the `headlines` list. Afterwards, we'll store the results in a Pandas table for easier analysis.

**Listing 12.5 Finding locations in headlines**

```
import pandas as pd

matched_countries = [get_name_in_text(headline, country_to_name)
                     for headline in headlines]
matched_cities = [get_name_in_text(headline, city_to_name)
                  for headline in headlines]
data = {'Headline': headlines, 'City': matched_cities,
        'Country': matched_countries}
df = pd.DataFrame(data)
```

Lets explore our location table. We'll start by summarizing the contents of `df` using the `describe` method.

**Listing 12.6 Summarizing the location data**

```
summary = df[['City', 'Country']].describe()
print(summary)
```

```
       City Country
count   619      15
unique  511      10
top      Of  Brazil
freq     45       3
```

**NOTE** Multiple countries in the data share the top occurrence frequency of 3. Pandas does not have a deterministic method for selecting one top country over another. Depending on your local settings, an country other than *Brazil* could be returned as a top country. It will still have a frequency of 3.

The table contains 619 mentions of cities representing 511 unique city names. It also contains just 15 countries representing 10 unique country names. The most frequently mentioned country is *Brazil*, which appears within 3 headlines.

The most frequently mentioned city is apparently *Of, Turkey*. That doesn't seem right! The 45 instances of *Of* are more likely to match the preposition than the rarely referenced Turkish location. We will output some instances of *Of* in order to confirm the error.

### Listing 12.7 Fetching cities named *Of*

```
of_cities = df[df.City == 'Of'][['City', 'Headline']]
ten_of_cities = of_cities.head(10)
print(ten_of_cities.to_string(index=False))   ❶
```

❶ We convert `df` to a string, in which the row indices have been removed. This leads to a more concise printed output.

```
City                                          Headline
  Of              Case of Measles Reported in Vancouver
  Of  Authorities are Worried about the Spread of Br...
  Of  Authorities are Worried about the Spread of Ma...
  Of  Rochester authorities confirmed the spread of ...
  Of     Tokyo Encounters Severe Symptoms of Meningitis
  Of  Authorities are Worried about the Spread of In...
  Of            Spike of Pneumonia Cases in Springfield
  Of  The Spread of Measles in Spokane has been Conf...
  Of                    Outbreak of Zika in Panama City
  Of    Urbana Encounters Severe Symptoms of Meningitis
```

Yes, our matches to *Of* are definitely erroneous. We can easily correct the situation by requiring all matched cities to begin with a capitalization. However, the observed bug is a symptom of a much bigger issue; in all the wrongly matched headlines we matched to *Of* but not to the actual city name. The mismatches occurred because we didn't consider potential multiple matches in a headline. How frequently do headlines contain 2 or more city matches? Lets find out. We will track the list of all matched cities in a headline within a newly added *Cities* column.

### Listing 12.8 Finding multi-city headlines

```
def get_cities_in_headline(headline):   ❶
    cities_in_headline = set()
    for regex, name in city_to_name.items():
        match = regex.search(headline)
        if match:
            if headline[match.start()].isupper():   ❷
                cities_in_headline.add(name)

    return list(cities_in_headline)

df['Cities'] = df['Headline'].apply(get_cities_in_headline)   ❸
df['Num_cities'] = df['Cities'].apply(len)   ❹
df_multiple_cities = df[df.Num_cities > 1]   ❺
num_rows, _ = df_multiple_cities.shape
print(f"{num_rows} headlines match multiple cities")
```

❶ This function will return a list of all unique cities in a headline.

❷ We make sure the first letter of the city-name is capitalized.

❸ We add a `Cities` column to the table. This is achieved using the `apply` method, which applies an inputted function to all elements of a column in order to create a brand new column.

❹ We add a column counting the number of cities in a headline.

❺ We filter out those rows that do not contain multiple city matches.

```
67 headlines match multiple cities
```

67 headlines contain more than one city, representing approximately 10% of the data. Why are so many headlines matching against multiple locations? Perhaps exploring some sample matches will yield an answer.

## Listing 12.9 Sampling multi-city headlines

```
ten_cities = df_multiple_cities[['Cities', 'Headline']].head(10)
print(ten_cities.to_string(index=False))
```

```
City                                          Headline
Lima                Lima tries to address Zika Concerns
Pune                     Pune woman diagnosed with Zika
Rome  Authorities are Worried about the Spread of Ma...
Molo                Molo Cholera Spread Causing Concern
Miri                               Zika arrives in Miri
Nadi  More people in Nadi are infected with HIV ever...
Baud  Rumors about Tuberculosis Spreading in Baud ha...
Kobe                    Chikungunya re-emerges in Kobe
Waco                More Zika patients reported in Waco
Erie                       Erie County sets Zika traps
Kent                      Kent is infested with Rabies
Reno  The Spread of Gonorrhea in Reno has been Confi...
Sibu                     Zika symptoms spotted in Sibu
Baku    The Spread of Herpes in Baku has been Confirmed
Bonn  Contaminated Meat Brings Trouble for Bonn Farmers
Jaen                        Zika Troubles come to Jaen
Yuma                     Zika seminars in Yuma County
Lyon                  Mad Cow Disease Detected in Lyon
Yiwu  Authorities are Worried about the Spread of He...
Suva  Suva authorities confirmed the spread of Rotav...
```

It appears that short, invalid city names are getting matched to the headlines along with longer, more correct location names. For example, the city of `'San'` is always returned along with more legitimate city names like `'San Francisco'` and `'San Salvador'`. How do we resolve this situation? One solution is simply to assign the longest city-name as the representative location if more than one matched city is found.

## Listing 12.10 Selecting the longest city names

```
def get_longest_city(cities):
    if cities:
        return max(cities, key=len)
    return None

df['City'] = df['Cities'].apply(get_longest_city)
```

As a sanity check, we'll output those rows in the the table that contain a short city-name (4 characters or less), in order to ensure that no erroneous short name is getting assigned to one of our headlines.

### Listing 12.11 Printing the shortest city names

```
short_cities = df[df.City.str.len() <= 4][['City', 'Headline']]
print(short_cities.to_string(index=False))
```

```
City                                          Headline
 Lima                Lima tries to address Zika Concerns
 Pune                     Pune woman diagnosed with Zika
 Rome  Authorities are Worried about the Spread of Ma...
 Molo                Molo Cholera Spread Causing Concern
 Miri                               Zika arrives in Miri
 Nadi  More people in Nadi are infected with HIV ever...
 Baud  Rumors about Tuberculosis Spreading in Baud ha...
 Kobe                     Chikungunya re-emerges in Kobe
 Waco                More Zika patients reported in Waco
 Erie                        Erie County sets Zika traps
 Kent                       Kent is infested with Rabies
 Reno  The Spread of Gonorrhea in Reno has been Confi...
 Sibu                      Zika symptoms spotted in Sibu
 Baku   The Spread of Herpes in Baku has been Confirmed
 Bonn  Contaminated Meat Brings Trouble for Bonn Farmers
 Jaen                         Zika Troubles come to Jaen
 Yuma                       Zika seminars in Yuma County
 Lyon                  Mad Cow Disease Detected in Lyon
 Yiwu  Authorities are Worried about the Spread of He...
 Suva  Suva authorities confirmed the spread of Rotav...
```

The results appear to be legitimate. Let's now shift our attention from cities to countries. Only 15 of the total headlines contain actual country information. The count is low enough for us to manually examine all these headlines.

### Listing 12.12 Fetching headlines with countries

```
df_countries = df[df.Country.notnull()][['City',   ❶
                                         'Country',
                                         'Headline']]
print(df_countries.to_string(index=False))
```

❶ The `df.Country.notnull()` method will return a list of booleans. Each boolean will equal `True` only if a country is present in the associated row.

```
           City    Country                                        Headline
         Recife     Brazil            Mystery Virus Spreads in Recife, Brazil
Ho Chi Minh City    Vietnam     Zika cases in Vietnam's Ho Chi Minh City surge
        Bangkok   Thailand                      Thailand-Zika Virus in Bangkok
     Piracicaba     Brazil                 Zika outbreak in Piracicaba, Brazil
          Klang   Malaysia                    Zika surfaces in Klang, Malaysia
  Guatemala City  Guatemala  Rumors about Meningitis spreading in Guatemala...
     Belize City     Belize                  Belize City under threat from Zika
        Campinas     Brazil                    Student sick in Campinas, Brazil
     Mexico City     Mexico                Zika outbreak spreads to Mexico City
   Kota Kinabalu   Malaysia           New Zika Case in Kota Kinabalu, Malaysia
     Johor Bahru   Malaysia                 Zika reaches Johor Bahru, Malaysia
       Hong Kong  Hong Kong                    Norovirus Exposure in Hong Kong
     Panama City     Panama                    Outbreak of Zika in Panama City
```

```
      Singapore  Singapore                Zika cases in Singapore reach 393
    Panama City     Panama            Panama City's first Zika related death
```

All of the country-bearing headlines also contain city information. Thus, we can assign a latitude and longitude without relying on the country's central coordinates. Consequently, we can disregard the country names from our analysis.

### Listing 12.13 Dropping countries from the table

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

We are nearly ready to add latitudes and longitudes to our table. However, we first need to consider those rows where no locations were detected. Lets count the number of unmatched headlines, and then print a subset of that data.

### Listing 12.14 Exploring unmatched headlines

```
df_unmatched = df[df.City.isnull()]
num_unmatched = len(df_unmatched)
print(f"{num_unmatched} headlines contain no city matches.")
print(df_unmatched.head(10)[['Headline']].values)
```

```
39 headlines contain no city matches.
[['Louisiana Zika cases up to 26']
 ['Zika infects pregnant woman in Cebu']
 ['Spanish Flu Sighted in Antigua']
 ['Zika case reported in Oton']
 ['Hillsborough uses innovative trap against Zika 20 minutes ago']
 ['Maka City Experiences Influenza Outbreak']
 ['West Nile Virus Outbreak in Saint Johns']
 ['Malaria Exposure in Sussex']
 ['Greenwich Establishes Zika Task Force']
 ['Will West Nile Virus vaccine help Parsons?']]
```

Approximately 6% of the headlines did not match any cities. Some of these headlines mentioned legitimate cities, which GeoNamesCache failed to identify. How should we treat the missing cities? Well, given their low frequency, perhaps we should delete the missing mentions. Of course, the price for that deletion is a slight reduction in data quality. However, that loss will not significantly impact our results, because our coverage of matched cities is quite high.

### Listing 12.15 Dropping unmatched headlines

```
df = df[~df.City.isnull()][['City', 'Headline']]  ❶
```

❶ The ~ symbol will reverse the booleans in the list returned by the `df.City.isnull()` method. Thus, each reversed boolean will only equal `True` if a city is present in the associated row.

## 12.3 Visualizing and Clustering the Extracted Location Data

All the rows in our table contain a city-name. Now, we can assign a latitude and longitude to each row. We'll use the GeoNamesCache `get_cities_by_name` method to return the coordinates of the most populated city bearing the extracted city-name.

**Listing 12.16 Assigning geographic coordinates to cities**

```
latitudes, longitudes = [], []
for city_name in df.City.values:
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population'])   ❶
    city = list(city.values())[0]   ❷
    latitudes.append(city['latitude'])
    longitudes.append(city['longitude'])

df = df.assign(Latitude=latitudes, Longitude=longitudes)   ❸
```

❶ We choose the matched city with largest population.

❷ We extract extract city latitudes and logitudes.

❸ We add *Latitude* and *Longitude* columns to our table table.

With latitudes and longitudes assigned, we can attempt to cluster the data. Lets execute K-means across our set of 2D coordinates. We'll use the Elbow method to choose a reasonable value for K.

**Listing 12.17 Plotting a geographic elbow curve**

```
coordinates = df[['Latitude', 'Longitude']].values
k_values = range(1, 10)
inertia_values = []
for k in k_values:
    inertia_values.append(KMeans(k).fit(coordinates).inertia_)

plt.plot(range(1, 10), inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

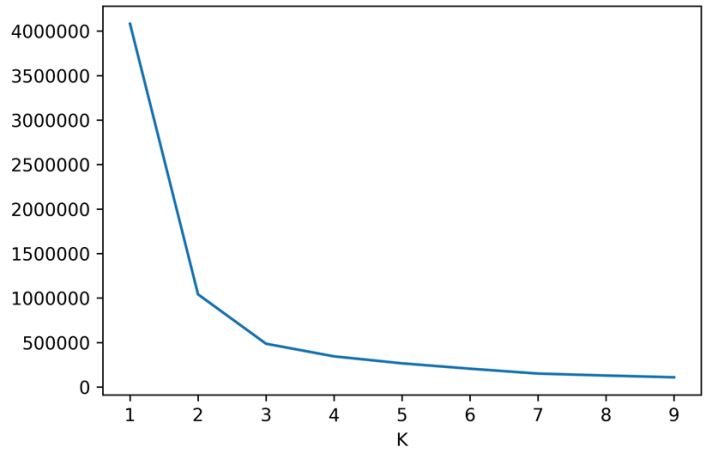


**Figure 12.1 A geographic Elbow curve points to a K of 3.**

The "elbow" within our Elbow plot points to a K of 3 That K-value is very low; limiting our scope to at-most 3 different geographic territories. Still, we should maintain some faith in our analytic methodology. We'll cluster the locations into 3 groups and plot them on a map.

**Listing 12.18 Using K-means to cluster cities into 3 groups**

```
def plot_clusters(clusters, longitudes, latitudes):
    fig = plt.figure(figsize=(12, 10))

    map_plotter = Basemap()
    map_plotter.scatter(longitudes, latitudes, c=clusters, latlon=True,
                        marker='o', alpha=1.0)
    map_plotter.drawcoastlines()
    plt.show()

df['Cluster'] = KMeans(3).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

**NOTE** The marker shapes in Figures 12.1 through 12.5 have been manually adjusted to discriminate among clusters in the black-and-white print-version of the book.

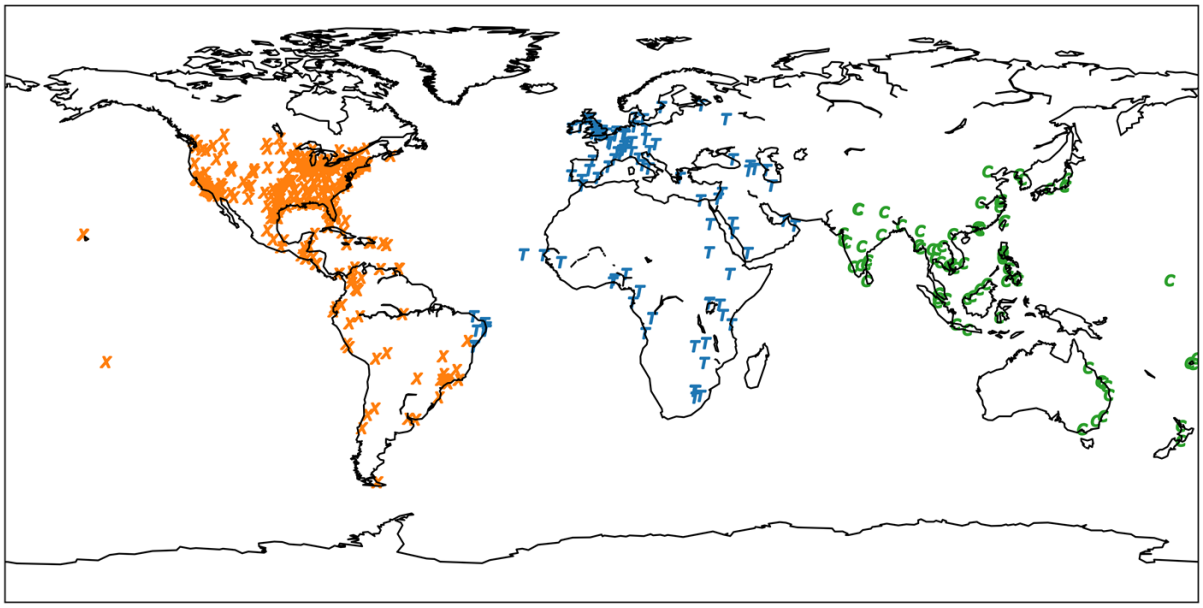

**Figure 12.2 Mapped K-means city-clusters. K is set to 3. The 3 clusters are spread thin across 6 continents.**

The results look pretty ridiculous. Our 3 clusters cover:

**A.** North and South America.

**B.** Africa and Europe.

**C.** Asia and Australia.

These continental categories are too broad to actually be useful. Furthermore, the East-most South American cities awkwardly cluster with African and European locations (despite the fact that an entire Ocean lies between them). These clusters are not helpful for understanding the data. Perhaps our K was too low after all. We'll disregard the recommended K-value from the Elbow analysis, and double the size of K to 6.

**Listing 12.19 Using K-means to cluster cities into 6 groups**

```
df['Cluster'] = KMeans(6).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

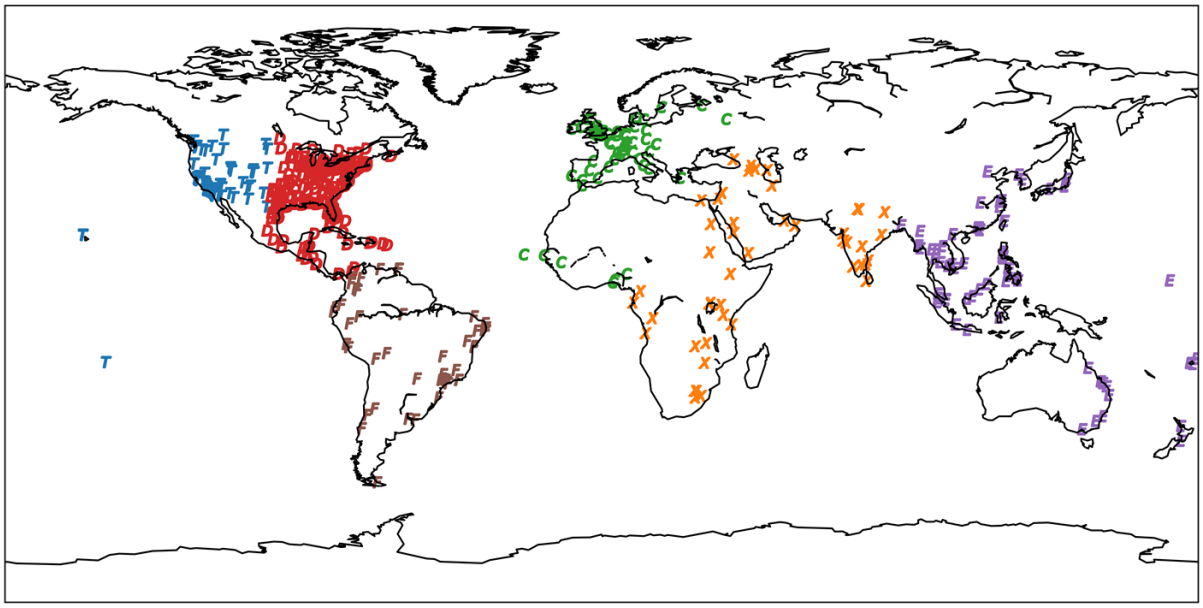

**Figure 12.3 Mapped K-Means city clusters. K is set to 6. Africa's clustered points are incorrectly split between the European and Asian continents.**

Increasing the K improves clustering within the North and South American continents. South America now falls within its own searate cluster, and North America is split between two Western and Eastern cluster groups. However, on the other side of the Atlantic, the clustering quality remains low. Africa's geolocations are incorrectly split between the European and Asian continents. K-mean's sense of centrality is unable to properly distinguish between Africa, Europe and Asia. Perhaps the algorithm's reliance on Euclidean distance is preventing it from capturing relationships between points distributed on our planet's curved surface.

As an alternative approach, we can attempt to execute DBSCAN clustering. The DBSCAN algorithm takes as input any distance metric of our choosing, allowing us to cluster on the great-circle distance between points. We'll start by coding a great-circle distance function whose inputs are a pair of NumPy arrays.

**Listing 12.20 Defining a NumPy-based great-circle metric**

```
def great_circle_distance(coord1, coord2, radius=3956):   ❶
    if np.array_equal(coord1, coord2):
        return 0.0

    coord1, coord2 = np.radians(coord1), np.radians(coord2)
    delta_x, delta_y = coord2 - coord1
    haversin = sin(delta_x / 2) ** 2 + np.product([cos(coord1[0]),
                                                   cos(coord2[0]),
                                                   sin(delta_y / 2) ** 2])
    return  2 * radius * asin(haversin ** 0.5)
```

❶ `radius` is preset to the radius of the Earth, in miles.

With our distance metric in place, we are nearly ready to run the DBSCAN algorithm. Before we do, we'll need to select reasonable values for the `eps` and `min_samples` parameters. Lets assume

the following; a global city cluster contains at least 3 cities that are on average no more than 250 miles apart. Based on these assumptions, we will input values of 250 and 3 into `eps` and `min_samples`, respectively.

**Listing 12.21 Using DBSCAN to cluster cities**

```
metric = great_circle_distance
dbscan = DBSCAN(eps=250, min_samples=3, metric=metric)
df['Cluster'] = dbscan.fit_predict(coordinates)
```

DBSCAN assigns -1 to outlier data-points that do not cluster. Lets remove these outliers from our table. Afterwards, we'll plot the remaining results.

**Listing 12.22 Plotting non-outlier DBSCAN clusters**

```
df_no_outliers = df[df.Cluster > -1]
plot_clusters(df_no_outliers.Cluster, df_no_outliers.Longitude,
              df_no_outliers.Latitude)
```

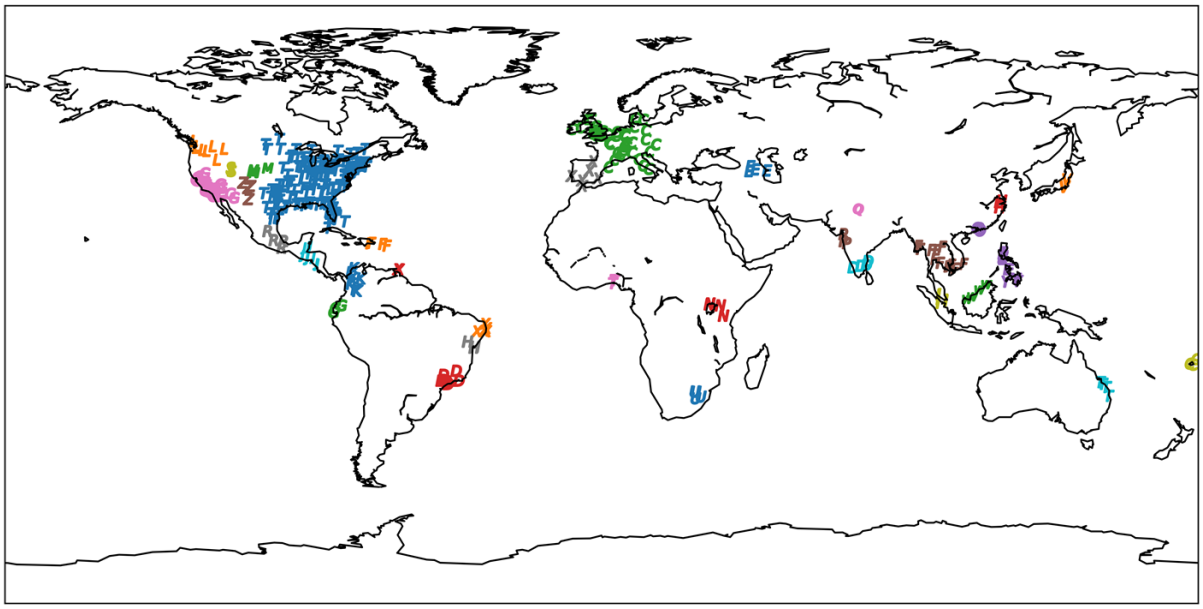

**Figure 12.4 Mapped DBSCAN city-clusters computed using the Great Circle distance metric.**

DBSCAN has does a decent job of generating discrete clusters within parts of South America, Asia, and Southern Africa. The Eastern United States however, falls into a single overly-dense cluster. Why is this the case? Partially it is due to a certain narrative bias in Western media, in which American events are more likely to get coverage. This leads to a denser spread of mentioned locations. One way to overcome the geographic bias is to re-cluster US cities using a more rigorous epsilon parameter. Such a strategy seems sensible within the context of our problem statement, which asks for separate top clusters from American and globally-grouped headlines. Thus, we'll cluster US locations independently from the rest of the World. To do so, we will first assign country-codes across each of our cities.

**Listing 12.23 Assigning country codes to cities**

```
def get_country_code(city_name):
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population'])
    return list(city.values())[0]['countrycode']

df['Country_code'] = df.City.apply(get_country_code)
```

The country-codes allow us to separate the data into 2 distinct `DataFrame` objects. The first object, `df_us`, which hold all the United States locations. The second object, `df_not_us`, will hold all the remaining global cities.

**Listing 12.24 Seperating US and global cities**

```
df_us = df[df.country_code == 'US']
df_not_us = df[df.country_code != 'US']
```

We've separated US and non-US cities. Now, we will need to re-cluster the coordinates within the 2 separated tables. The re-clustering of `df_not_us` is necessary because of the density shifts caused by the deletions of all US locations. We will maintain an `eps` of 250 for `df_not_us`. Furthermore, we will half the `eps` for `df_us` to 125, in order to acknowledge the tighter density of US locations. All outlier points will be deleted during the re-clustering process.

**Listing 12.25 Re-clustering extracted cities**

```
def re_cluster(input_df, eps):
    input_coord =  input_df[['latitude', 'longitude']].values
    metric = great_circle_distance
    dbscan = DBSCAN(eps=eps, min_samples=3, metric=metric)
    clusters = dbscan.fit_predict(input_coord)
    input_df = input_df.assign(cluster=clusters)
    return input_df[input_df.cluster > -1]

df_not_us = re_cluster(df_not_us, 250)
df_us = re_cluster(df_us, 125)
```

## *12.4 Extracting Insights from Location Clusters*

Lets investigate the clustered data within the `df_not_us` table. We'll start by grouping the clustered results using the Pandas `groupby` method.

**Listing 12.26 Grouping cities by cluster**

```
groups = df_not_us.groupby('Cluster')
num_groups = len(groups)
print(f"{num_groups} Non-US have been clusters detected")
```

```
31 Non-US have been clusters detected
```

31 global clusters have been detected. Lets sort these groups by size and count the headlines in the largest cluster.

**Listing 12.27 Finding the largest cluster**

```
sorted_groups = sorted(groups, key=lambda x: len(x[1]),
                       reverse=True)
group_id, largest_group = sorted_groups[0]
group_size = len(largest_group)
print(f"Largest cluster contains {group_size} headlines")
```

```
Largest cluster contains 51 headlines
```

The largest cluster contains 51 total headlines. Reading all these headlines individually will be a time-consuming process. We can save time by outputting just those headlines that represent the most central locations in the cluster. Centrality can be captured by calculating the average latitude and longitude of a group. Afterwards, we can compute the distance between every location and the average coordinates. Lower distances will indicate higher centrality.

**NOTE** As we've discussed in Section Eleven, the average latitude and longitude is merely an approximation of the center, since it does not consider the curvature of the Earth.

Below, we'll define a `compute_centrality` function, which assigns a *Distance_to_center* column to an inputted group.

**Listing 12.28 Computing cluster centrality**

```
def compute_centrality(group):
    group_coords = group[['Latitude', 'Longitude']].values
    center = group_coords.mean(axis=0)
    distance_to_center = [great_circle_distance(center, coord)
                          for coord in group_coords]
    group['Distance_to_center'] = distance_to_center
```

Computing the centrality allows us to sort the grouped locations based on their distance to the centers, in order to output the most central headlines. Lets print the 5 most central headlines within our largest cluster.

**Listing 12.29 Finding the central headlines in largest cluster**

```
def sort_by_centrality(group):
    compute_centrality(group)
    return group.sort_values(by=['Distance_to_center'], ascending=True)

largest_group = sort_by_centrality(largest_group)
for headline in largest_group.Headline.values[:5]:
    print(headline)
```

```
Mad Cow Disease Disastrous to Brussels
Scientists in Paris to look for answers
More Livestock in Fontainebleau are infected with Mad Cow Disease
Mad Cow Disease Hits Rotterdam
Contaminated Meat Brings Trouble for Bonn Farmers
```

The central headlines in `largest_cluster` focus on an outbreak of Mad Cow Disease within various European cities. We can confirm that the cluster's locale is centered in Europe by outputting the top countries associated with cities in the cluster.

**Listing 12.30 Finding the top 3 countries in largest cluster**

```
from collections import Counter
def top_countries(group):
    countries = [gc.countries[country_code]['name']
                 for country_code in group.country_code.values]
    return Counter(countries).most_common(3)   ❶


print(top_countries(largest_group))
```

❶ The `Counter` class tracks the mostly repeating elements within a list, along with their counts.

```
[('United Kingdom', 19), ('France', 7), ('Germany', 6)]
```

The most frequently mentioned cities within `largest_cluster` are located in the United Kingdom, France, and Germany. The majority of locations in `largest_cluster` are definitely in Europe.

Lets repeat this analysis across the next 4 largest global clusters. The following code will help determine if any other disease epidemics are currently threatening the Earth.

**Listing 12.31 Summarizing content within the largest clusters**

```
for _, group in sorted_groups[1:5]:
    sorted_group = sort_by_centrality(group)
    print(top_countries(sorted_group))
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')
```

```
[('Philippines', 16)]
Zika afflicts patient in Calamba
Hepatitis E re-emerges in Santa Rosa
More Zika patients reported in Indang
Batangas Tourism Takes a Hit as Virus Spreads
Spreading Zika reaches Bacoor


[('El Salvador', 3), ('Honduras', 2), ('Nicaragua', 2)]
Zika arrives in Tegucigalpa
Santa Barbara tests new cure for Hepatitis C
Zika Reported in Ilopango
More Zika cases in Soyapango
Zika worries in San Salvador


[('Thailand', 5), ('Cambodia', 3), ('Vietnam', 2)]
More Zika patients reported in Chanthaburi
Thailand-Zika Virus in Bangkok
Zika case reported in Phetchabun
Zika arrives in Udon Thani
More Zika patients reported in Kampong Speu
```

```
[('Canada', 10)]
Rumors about Pneumonia spreading in Ottawa have been refuted
More people in Toronto are infected with Hepatitis E every year
St. Catharines Patient in Critical Condition after Contracting Dengue
Varicella has Arrived in Milton
Rabies Exposure in Hamilton
```

Oh no! Zika is spreading through the Philippines! There are also Zika outbreaks in Southeast Asia and in Central America. The final Canadian cluster however, contains a mix of random disease headlines, which implies that no dominant outbreak is occurring in that Northern territory.

Lets turn our attention to the US clusters. We'll start by visualizing the clusters on a map of the United States.

**Listing 12.32 Plotting United States DBSCAN clusters**

```
fig = plt.figure(figsize=(12, 10))
map_lcc = Basemap(projection='lcc', llcrnrlon=-119,
                  llcrnrlat=22, urcrnrlon=-64, urcrnrlat=49,
                   lat_1=33, lat_2=45, lon_0=-95)

map_lcc.scatter(df_us.Longitude.values, df_us.Latitude.values,
                c=df_us.Cluster, latlon=True)
map_lcc.drawcoastlines()
map_lcc.drawstates()
plt.show()
```

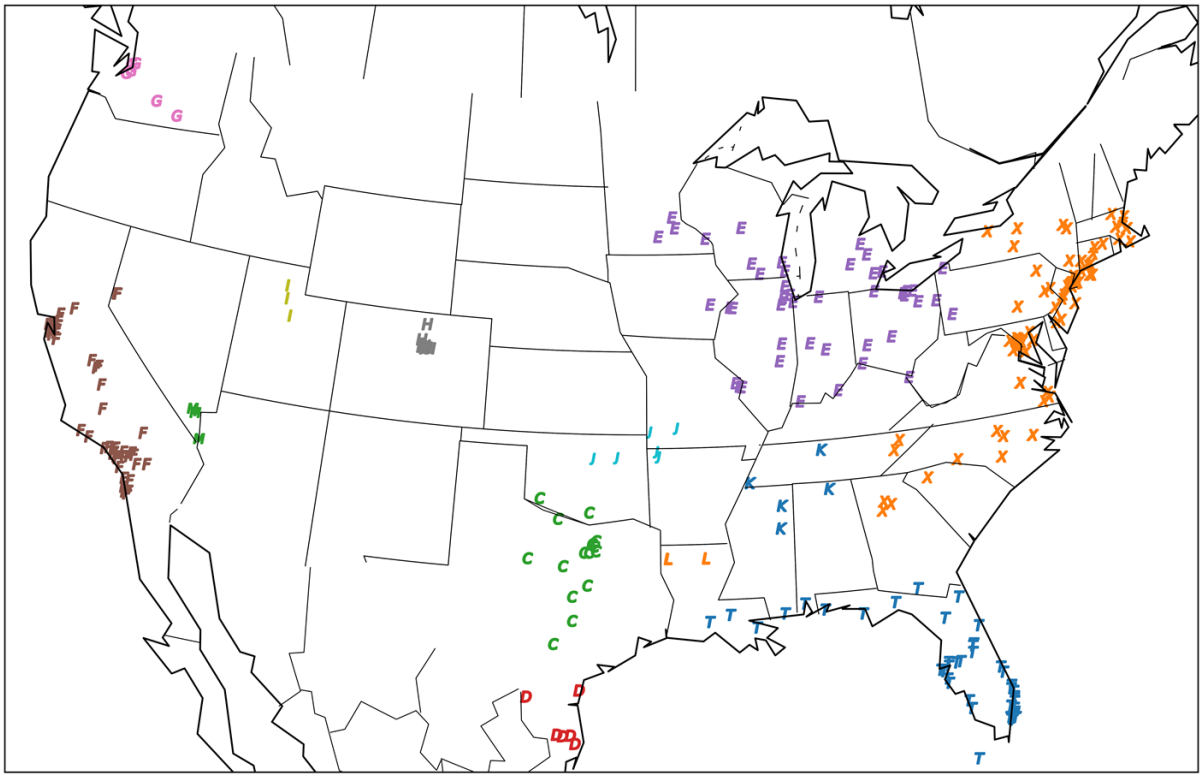

**Figure 12.5 Mapped DBSCAN location clusters within the boundaries of the United States.**

The visualized map yields reasonable outputs. The Eastern United States no longer falls into a single dense cluster.

We'll proceed to analyze the top 5 US clusters by printing their centrality-sorted headlines.

**Listing 12.33 Summarizing content within the largest US clusters**

```
us_groups = df_us.groupby('Cluster')
us_sorted_groups = sorted(us_groups, key=lambda x: len(x[1]),
                          reverse=True)
for _, group in us_sorted_groups[:5]:
    sorted_group = sort_by_centrality(group)
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')
```

```
Schools in Bridgeton Closed Due to Mumps Outbreak
Philadelphia experts track pandemic
Vineland authorities confirmed the spread of Chlamydia
Baltimore plans for Zika virus
Will Swine Flu vaccine help Annapolis?


Bradenton Experiences Zika Troubles
Tampa Bay Area Zika Case Count Climbs
Zika Strikes St. Petersburg
New Zika Case Confirmed in Sarasota County
Zika spreads to Plant City


Rhinovirus Hits Bakersfield
Schools in Tulare Closed Due to Mumps Outbreak
New medicine wipes out West Nile Virus in Ventura
Hollywood Outbreak Film Premieres
Zika symptoms spotted in Hollywood


How to Avoid Hepatitis E in South Bend
Hepatitis E Hits Hammond
Chicago's First Zika Case Confirmed
Rumors about Hepatitis C spreading in Darien have been refuted
Rumors about Rotavirus Spreading in Joliet have been Refuted


More Zika patients reported in Fort Worth
Outbreak of Zika in Stephenville
Zika symptoms spotted in Arlington
Dallas man comes down with case of Zika
Zika spreads to Lewisville
```

The global Zika epidemic has hit both Florida and Texas! This is indeed some very unnerving news. However, no discernable disease patterns are present in the other top 3 clusters. Currently, the spreading Zika outbreak is confined to just the Southern United States. We will immediately report this to our superiors so that they can take appropriate action. As we prepare to present our findings, lets plot one additional image, which will appear on the front page of our report. That image will summarize the menacing scope of the spreading Zika epidemic. It will display all US and global clusters where Zika is mentioned in more than 50% of article headlines.

## Listing 12.34 Plotting Zika clusters

```
def count_zika_mentions(headlines):   ❶
    zika_regex = re.compile(r'\bzika\b',   ❷
                            flags=re.IGNORECASE)
    zika_count = 0
    for headline in headlines:
        if zika_regex.search(headline):
            zika_count += 1

    return zika_count

fig = plt.figure(figsize=(12, 10))
map_plotter = Basemap()

for _, group in sorted_groups + us_sorted_groups:   ❸
    headlines = group.Headline.values
    zika_count = count_zika_mentions(headlines)
    if float(zika_count) / len(headlines) > 0.5:   ❹
        map_plotter.scatter(group.Longitude.values,
                            group.Latitude.values,
                            latlon=True)
map_plotter.drawcoastlines()
plt.show()
```

❶ This function counts the number of times that Zika is mentioned in a list of headlines.

❷ This regular expression will match an instance of the word "Zika" in a headline. The match will be case-insensitive.

❸ We iterate over both US and global clusters.

❹ We plot those clusters where Zika is mentioned in more than 50% of article headlines.

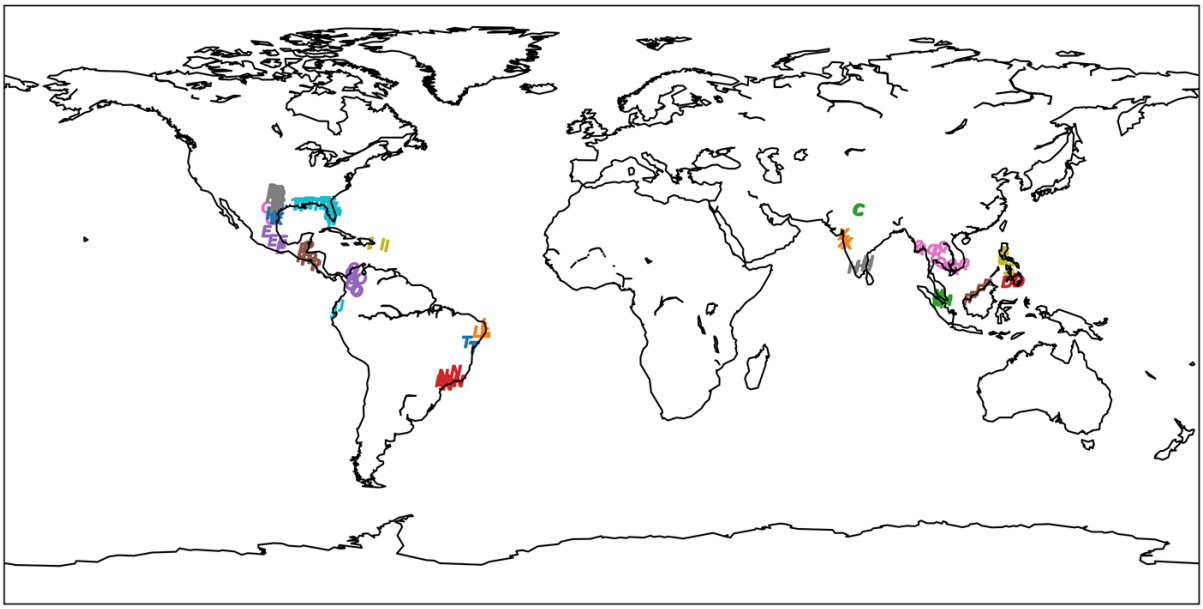

**Figure 12.6 Mapped DBSCAN location clusters where Zika is mentioned in more than 50% of article headlines.**

We have successfully clustered our headlines by location, and plotted those clusters where the word Zika is dominant. This relationship between our clusters and their textual content leads to an interesting question; is it possible to cluster the headlines based on text similarity rather than geographic distance? In other words, can we group our headlines by text overlap, so that all the references to Zika automatically appear in a single cluster? Yes we can! In the subsequent Case Study, we learn how to measure similarity between texts in order to group documents by topic.

## 12.5 Key Takeaways

- Data Science tools can fail in unexpected ways. When we ran GeoNamesCache on our news headlines, the library incorrectly matched short city-names (such as *Of* and *San*) to the inputted text. Through data exploration, we were able to account for these mistakes. If instead, we blindly clustered the locations, then our final output would have been junky. Consequently, its important to diligently explore all input-data and all output-data.
- Sometimes, problematic data-points are present in an otherwise good dataset. In our case, less than 6% of headlines wrongly lacked a city assignment. Correcting for these headlines would have been difficult. Instead, we choose to delete the headlines from the dataset. Occasionally, its okay to delete problematic examples, if their impact on the dataset is minimal. However, we should still weigh the pros and cons of the deletion prior to making a final decision.
- The Elbow method heuristically picks K for K-means clustering. Heuristic tools are not guaranteed to work correctly every time. In our analysis, an Elbow plot returned a K of 3. Obviously, this value was too low. Thus, we intervened and attempted to choose different K. If we had indiscriminately trusted the Elbow output, then our final clustering would have been worthless.
- Common sense should dictate our analysis of clustering outputs. Earlier, we examined a K-means output where K equaled 6. We observed the clustering of Central African and European cities. This result was clearly wrong. Europe and Central Africa are very different locations. Thus, we transitioned to a different clustering approach. When common sense dictates that the clustering is wrong, then we should try an alternate approach.
- Sometimes its acceptable to break-up a dataset into parts, and individually analyze each part. In our initial DBSCAN analysis, the algorithm failed to correctly cluster US cities. Most Eastern US cities fell into a single cluster. We could have abandoned our DBSCAN approach. Instead, we clustered the US cities separately, using more appropriate parameters. Analyzing the dataset in 2 separate parts led to better clustering results.