12 Examining and testing naturally occurring number sequences

This chapter covers

- Benford's law and naturally occurring number sequences
- Chi-square goodness of fit test
- Mean absolute deviation
- The distortion factor and the z-statistic
- Mantissa statistics

Numeric data sets that follow a Benford distribution exhibit a much higher frequency of smaller leading digits than larger leading digits. The phenomenon is mostly prevalent in numeric data that spans several orders of magnitude and is therefore best represented on a logarithmic, rather than linear, scale.

Fraudsters often make the mistake of transmuting leading 1s and 2s to 8s and 9s on invoices, expenses, tax returns, and the like to maximize gains against their risks, on the assumption that, regardless of the data set, 8s and 9s are just as probable as 1s and 2s or that, when randomness prevails, larger digits should sometimes be expected to actually occur more frequently than smaller digits. In fact, Benford's law is most often used in fraud detection; serious deviations from a Benford distribution may be an indication of fraudulent activity and therefore the reason for further investigation. But there are other applications as well: data integrity, economic data analysis, scientific research, digital forensics, and population studies, just to name a few, which is to say state that Benford's law is a valuable tool for uncovering potential anomalies in numeric data sets across several domains.

Not unlike regression analysis and other techniques, Benford's law provides a powerful method for detecting patterns in data that might not be immediately obvious. It offers a data-driven approach to decision-making by identifying irregularities that standard statistical methods may overlook. By integrating probability-based methods and statistical validation, Benford's law complements the other quantitative techniques explored in this book, reinforcing the broader theme of using mathematical principles to analyze and interpret real-world data.

After getting you thoroughly grounded in all things Benford's law, we'll discuss where it best applies and where it absolutely does not; we'll draw a perfect Benford distribution and then compare it to uniform and random distributions that are commonly observed in data sets to which Benford's law does not apply; we'll then evaluate Benford's law on a trio of real-world data sets that should follow a Benford, or logarithmic, distribution; and finally, we'll run a series of statistical tests against one of our three data sets to measure how well, or not so well, the data conforms to a Benford distribution. Numeric data sets that should conform to Benford's law but don't *might* be an indication of fraudulent or other suspicious activity. By the end of this chapter, you should have a solid theoretical and practical understanding of Benford's law; you will know a Benford distribution by sight and have the ability to readily compare and contrast it against other probability distributions; and you will learn how to apply several statistical tests against numeric data to precisely determine whether numeric data truly obeys Benford's law. Let's get started with a methodical explanation of Benford's law and a brief discussion of where Benford's law is most prevalent.

12.1 Benford's law explained

According to Benford's law, which is sometimes referred to as the *first-digit law*, many sets of numeric data do not inherently follow a uniform or random distribution, as we might expect. (A uniform distribution is a probability distribution in which all outcomes are equally likely; a random distribution, on the other hand, is a probability distribution where the outcomes are not deterministic.) Instead, says Benford's law, smaller leading digits (1, 2, 3) occur much more frequently than larger leading digits (7, 8, 9). In data sets that follow Benford's law, the numeral 1 is the leading digit in approximately 30% of the observations, and the numeral 9 is the leading digit in fewer than 5% of the observations. (Zeros are never leading digits, so they are excluded from Benford's law; and in any event, zeros don't naturally fit into a logarithmic distribution, which is the very essence of Benford's law.)

In addition to being known as the first-digit law, Benford's law is sometimes referred to as the *Newcomb–Benford law*. That's because a mathematician named Simon Newcomb first discovered the phenomenon in 1881. Frank Benford, a physicist, shed light on it in 1938.

A set of numeric data is said to follow Benford's law when the leading digit, usually denoted as *d*, occurs with a probability in accordance with the following equation:

$$P(d) = \log 10(d + 1) - \log 10(d) = \log 10\left(\frac{d + 1}{d}\right) = \log 10\left(1 + \frac{1}{d}\right)$$

Let's do the math by substituting the digits 1 through 9 for *d* and then plot the distribution in a Matplotlib bar line chart. But first, we import the pandas library and create a data frame called `df1`; for the moment, `df1` contains just a single variable, `d`, which is merely a list of integers between `1` (inclusive) and `10` (exclusive). The `pd.DataFrame()` method creates the data frame, `d` is the name of the lone variable, and the `list()` method generates a list of integers from `1` through `9` (inclusive):

```
>>> import pandas as pd
>>> df1 = pd.DataFrame({'d': list(range(1, 10))})
>>> print(df1)
   d
0  1
1  2
2  3
3  4
4  5
5  6
6  7
7  8
8  9
```

Next we call the `assign()` method to create a derived variable called `benford`, which equals the base-10 logarithm of `1` plus the quotient between `1` and the original `df1` variable `d`, rounded to three decimal places. We're replicating the Benford probability equation with a line of Python code and plugging in the digits `1` through `9` in place of *d*. This sort of mathematical operation first requires that we import the NumPy library. The `assign()` method creates the new variable called `benford` and assigns it to the `df1` data frame. The `round()` method rounds each result to three decimal places (although it could be any whole number we choose to pass):

```
>>> import numpy as np
>>> df1 = df1.assign(benford = round(np.log10(1 + (1 / df1.d)), 3))
>>> print(df1)
   d  benford
0  1    0.301
1  2    0.176
2  3    0.125
3  4    0.097
```

```
4  5    0.079
5  6    0.067
6  7    0.058
7  8    0.051
8  9    0.046
```

Now we have a data source for our bar line chart. Note that we must first import the `matplotlib` library before executing any of the code that follows:

```
>>> import matplotlib.pyplot as plt
>>> plt.bar(df1['d'], df1['benford'], color = 'dodgerblue',
>>>         edgecolor = 'dodgerblue')
>>> plt.plot(df1['d'], df1['benford'],
>>>          'r-o', linewidth = 1.5)
>>> for i, benford_value in enumerate(df1['benford']):
>>>     plt.text(i + 1, benford_value, f'{benford_value * 100:.2f}%',
>>>              ha = 'center', va = 'bottom',
>>>              fontweight = 'bold',
>>>              color = 'black')
>>> plt.title('Perfect Benford distribution',
>>>           fontweight = 'bold')
>>> plt.xlabel('First Digit')
>>> plt.ylabel('Distribution Percentage')
>>> plt.xticks(range(1, 9))
>>> plt.gca().yaxis.set_major_formatter(
>>>     plt.FuncFormatter(lambda x, _: f'{x * 100:.0f}%')
>>> )
>>> plt.show()
```

Imports the matplotlib library

Creates a bar chart with a custom color scheme and assigns the x-axis and y-axis variables

Draws a red solid line 15 times the default width over the bars, with circular markers

Iterates over the benford values in df1, placing a bold black text label atop each bar that shows the value as a percentage with two decimal places

Sets the title

Sets the x-axis label

Sets the y-axis label

Sets the range of the x-axis tick marks from 1 to 9

Formats the y-axis labels as percentages (e.g., 30%)

Displays the plot

Figure 12.1 shows what a perfect Benford distribution looks like. It may go without saying, but the probabilities should sum to exactly 1. We can verify this by passing the `df1` variable `benford` to the `sum()` method, like so:

```
>>> print(sum(df1.benford))
1.0
```

Furthermore, the values in the `benford` variable are proportional to the space between d and $d + 1$ on a logarithmic scale. Just as the numerals 1 and 2 and 2 and 3, for instance, are equally spaced on a linear scale, the numerals 10 and 100 and 100 and 1,000 are equally spaced on a logarithmic scale. Whereas linear scales have equal intervals and a constant difference between values, logarithmic scales have unequal intervals due to a constant ratio or multiplication factor between values. This property is important to understand because it highlights the usefulness of logarithmic scales in representing data that spans several orders of magnitude. In many real-world scenarios, data can sometimes vary widely in magnitude; thus, using a logarithmic scale allows for a more compact representation that better captures and shows this variation. And because Benford's law applies only to data distributed across several orders of magnitude, a logarithmic scale must prevail.

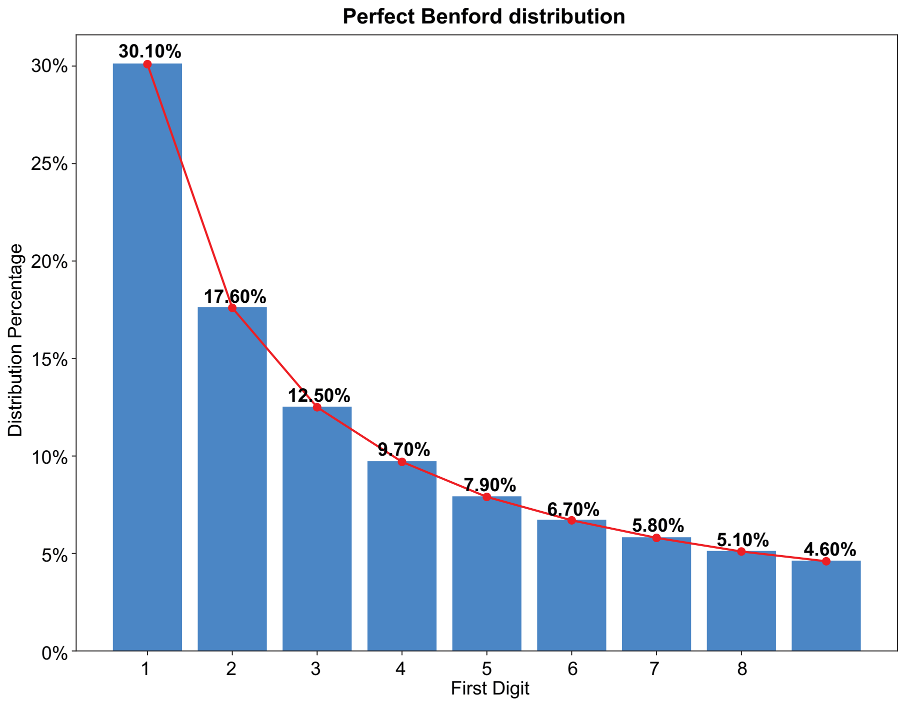

Figure 12.1 A numeric data series perfectly follows a Benford distribution when the leading digits, plotted along the *x* axis, each have a likelihood of occurrence equal to the probability plotted along the *y* axis. So, for instance, the numeral 1 should be the leading, or first, digit in approximately 30% of the observations, and the numeral 9 should be the leading digit in fewer than 5% of the observations.

Consider once more that the probability of a leading digit d. $P(d)$ is expressed by the following formula:

$$P(d) = \log 10\left(1 + \frac{1}{d}\right)$$

And let's further consider a pair of leading digits, 1 and 9, which means

$$P(1) = \log 10\left(1 + \frac{1}{1}\right) = \log 10(2)$$

and

$$P(9) = \log 10\left(1 + \frac{1}{9}\right) = \log 10(1.11)$$

Thus, the probability of the numeral 1 being the leading digit equals 0.301, whereas the probability of the numeral 9 being the leading digit equals 0.046. Here's how to perform these arithmetic operations in Python. The `np.log()` method calculates the base-10 logarithm of whatever whole or fractional number is passed to it:

```
>>> print(round(np.log10(2), 3))
0.301

>>> print(round(np.log10(1.1111111), 3))
0.046
```

We already know these results, of course. But there's another point to be made here: these results (0.301 and 0.046, respectively) represent not only the probabilities of 1 and 9 being leading digits in a numeric data set that follows Benford's law but also the space, or interval, from 1 and 9 to the next values on a logarithmic scale. Thus, the probabilities are exactly proportional to the interval width. We can infer, then, that the probabilities of leading digits under Benford's law are directly proportional to the intervals between successive powers of 10 on a logarithmic scale.

Perhaps even more interesting, and equally noncoincidental, is that the fractional part of logarithmic data, derived from the raw data and not the leading digits, is uniformly distributed between 0 and 1 when Benford's law is in effect. This property, known as the *uniform distribution of mantissas*, provides an additional way to assess whether a data set conforms to Benford's law. Rather than relying solely on the frequency of leading digits, we can evaluate how closely the mantissa values align with a uniform distribution, offering a statistical check on the data's adherence to a logarithmic pattern. Although measures like the mean and variance of the mantissa values can offer insights, it is their overall distribution that serves as a key indicator of Benford conformity. This approach allows for a more refined test of whether a data set truly follows Benford's law, beyond just comparing leading-digit frequencies. We'll explore this later when we examine the mantissa statistics.

12.2 Naturally occurring number sequences

It's frequently stated that Benford's law applies only to naturally occurring number sequences, which is somewhat nebulous. More precisely, Benford's law most accurately applies to numeric data when the following four conditions are met:

1. The data is distributed across multiple orders of magnitude. An *order of magnitude* is a measure of the scale, or size, of a value, most often expressed as a power of 10. So, for instance, when a value is 1 order of magnitude greater than another value from the same data series, it is approximately 10 times greater.
2. The data does not have any preestablished minimum or maximum. Although every finite data set technically has a minimum and a maximum, Benford's law is more likely to apply when these boundaries are not arbitrarily imposed. If the limits are naturally occurring rather than artificially constrained—such as physical measurement limits or policy-imposed caps—the data is more likely to exhibit Benford-like behavior.

3. The data does not consist of numerals used as identifiers. This includes Social Security numbers, bank account numbers, invoice numbers, telephone numbers, and employee numbers, just to name a few. These values are typically assigned systematically or sequentially rather than being the result of natural random processes. Because they do not arise from organic distributions or naturally varying magnitudes, they do not exhibit the logarithmic patterns required for Benford's law to apply.
4. The data has a mean that is greater—sometimes significantly greater—than the median. This means the data is right-skewed, or positively skewed; when plotted, most of the data is therefore on the left with a long thin tail extending to the right. This type of distribution often exhibits high kurtosis, indicating that the data has more extreme values or outliers than a normal distribution, further reinforcing the applicability of Benford's law.

Thus, Benford's law applies, for instance, to street addresses, lengths of rivers and other waterways, population counts, baseball statistics, sales figures, utility bills, electronic file sizes, and stock prices. It doesn't apply, just to give a few additional examples, to calendar dates, zip codes, or IQ scores.

12.3 Uniform and random distributions

Our purpose here is to plot uniform and random distributions for comparison purposes against a perfect Benford distribution. We sometimes assume, perhaps naively, that numeric data follows one of these two distributions when, in fact, Benford's law actually prevails if certain conditions are held to be true. We'll demonstrate how to generate uniform and random distributions by calling a pair of NumPy functions, create a pair of Matplotlib bar charts, and then print both plots as a single figure.

Comparing a Benford distribution to uniform and random distributions helps highlight the distinctive patterns and characteristics between each distribution type. More specifically, it highlights how the frequency of leading digits varies across data sets that obey Benford's law (where smaller digits are more frequent than larger digits) versus uniform distributions (where all digits occur with roughly equal probability) and random distributions (where digits occur with equal probability but without any specific pattern).

12.3.1 Uniform distribution

We need a data frame with two vectors. Our data frame, `df2`, first contains a variable called `row_number`, which is a list of integers between 1 (inclusive) and 1,001 (exclusive):

```
>>> df2 = pd.DataFrame({'row_number': list(range(1, 1001))})
```

Then we make a call to the `assign()` method to create a second `df2` vector, `uniform_distribution`. The `np.random.randint()` method generates 1,000 random integers between 1 (inclusive) and 10 (exclusive) and stores the results in `uniform_distribution`:

```
>>> df2 = df2.assign(uniform_distribution =
>>>                  np.random.randint(1, 10, size = 1000))
```

The `head()` and `tail()` methods return the first three and last three `df2` observations:

```
>>> print(df2.head(n = 3))
   row_number  uniform_distribution
0           1                     4
1           2                     6
2           3                     5
>>> print(df2.tail(n = 3))
     row_number  uniform_distribution
997         998                     6
998         999                     1
999        1000                     1
```

You won't get these same results. That's because `np.random.randint()` returns an array of *random* integers. Although every value assigned to `uniform_distribution` will always equal some integer between 1 (inclusive) and 10 (exclusive), results will vary with every run; otherwise, it wouldn't be random.

Next we group `df2` by the values in `uniform_distribution`, count the number of occurrences for each unique value, assign the results to a variable called `count`, and create a new data frame called `results1`. The `groupby()` method groups `df2` by the unique values in the variable `uniform_distribution`; `size()` counts the number of occurrences for each unique value; and `reset_index()` stores the outputs from `size()` in a new column, or variable, called `count`. After all that, we get a new data frame called `results1` that contains nine rows and a pair of columns called `uniform_distribution` and `count`:

```
>>> results1 = df2.groupby('uniform_distribution') \
>>>     .size() \
>>>     .reset_index(name = 'count')
>>> print(results1)
   uniform_distribution  count
0                     1    109
1                     2    116
2                     3    110
3                     4    110
4                     5    101
5                     6    112
6                     7    130
7                     8    104
8                     9    108
```

We'll temporarily hold these results in reserve until we've also generated a random distribution.

12.3.2 Random distribution

Once more we'll create a data frame, df3 this time, comprising two vectors. We'll recycle the row_number variable from the df2 data frame and assign it as the first df3 variable: df2[['row_number']] takes row_number from df2, and the copy() method makes a deep copy of it. By making a deep copy, we can make future modifications to df3 as necessary without affecting df2:

```
>>> df3 = df2[['row_number']].copy()
```

Next we'll create a second variable, random_distribution, and assign it to df3. The assign() method assigns random_distribution to df3, np.arrange() generates an array of integers between 1 (inclusive) and 10 (exclusive), and np.random_choice() randomly selects 1,000 values from this array, with replacement:

```
>>> df3 = df3.assign(random_distribution =
>>>                  np.random.choice(np.arange(1, 10), size = 1000))
```

Sequential calls to the head() and tail() methods return the first three and last three df3 observations. Your results will differ:

```
>>> print(df3.head(n = 3))
   row_number  random_distribution
0           1                    3
1           2                    5
2           3                    1
>>> print(df3.tail(n = 3))
     row_number  random_distribution
997         998                    3
998         999                    3
999        1000                    7
```

Finally, we group df3 by the unique values in random_distribution, count the number of occurrences for each, store the results in a variable called count, and assign the results to a new data frame called results2. This is essentially the same snippet of code we first used to create results1; we've just swapped out parameters:

```
>>> results2 = (df3.groupby('random_distribution') \
>>>             .size() \
>>>             .reset_index(name = 'count'))
>>> print(results2)
   random_distribution  count
0                    1    105
1                    2    136
2                    3    118
3                    4    118
4                    5    111
5                    6    105
6                    7    106
7                    8     94
8                    9    107
```

Next, we'll pass `results1` and `results2` into separate snippets of Matplotlib code and create a pair of bar charts combined into a single figure.

12.3.3 Plotted distributions

We start by creating a single figure with two subplots, the first (`ax1`) on top of the second (`ax2`). So, our figure will have dimensions equal to two rows and one column:

```
>>> fig, (ax1, ax2) = plt.subplots(nrows = 2, ncols = 1)
```

Here are the snippets of code for our two Matplotlib bar charts. Most of this should be familiar by now:

Iterates over the count values in results1, placing a bold black text label atop each bar showing each value at the corresponding position on the x axis

Creates the first bar chart with a custom color scheme, and assigns the x-axis and y-axis variables

```
>>> ax1.bar(results1['uniform_distribution'], results1['count'],
>>>         color = 'steelblue',
>>>         edgecolor = 'steelblue')
>>> for i, n in enumerate(results1['count']):
>>>     ax1.text(results1['uniform_distribution'][i], n, str(n),
>>>             ha = 'center', va = 'bottom',
>>>             fontweight = 'bold',
>>>             color = 'black')
>>> ax1.set_title('Uniform distribution\nn = 1,000',
>>>.              fontweight = 'bold')
>>> ax1.set_xlabel('First Digit')
>>> ax1.set_ylabel('Count')
>>> ax1.set_xticks(range(1, 10))
>>> ax2.bar(results2['random_distribution'], results2['count'],
>>>         color = 'steelblue',
>>>         edgecolor = 'steelblue')
>>> for i, n in enumerate(results2['count']):
>>>     ax2.text(results2['random_distribution'][i], n, str(n),
>>>             ha = 'center', va = 'bottom',
>>>             fontweight = 'bold',
>>>             color = 'black')
>>> ax2.set_title('Random distribution\nn = 1,000',
>>>               fontweight = 'bold')
>>> ax2.set_xlabel('First Digit')
>>> ax2.set_ylabel('Count')
>>> ax2.set_xticks(range(1, 10))
>>> plt.tight_layout()
>>> plt.show()
```

Sets the title. Also, \n represents the newline character and therefore triggers a carriage return.

Sets the x-axis label

Sets the y-axis label

Sets the range of the x-axis tick marks from 1 to 9

Creates the second bar chart with a custom color scheme, and assigns the x-axis and y-axis variables

Iterates over the count values in results1, placing a bold black text label atop each bar showing each value at the corresponding position on the x axis

Sets the title

Sets the x-axis label

Sets the y-axis label

Sets the range of the x-axis tick marks from 1 to 9

Creates separation between the plots to prevent overlapping titles and labels

Displays both plots as a single figure

Finally, figure 12.2 shows the results. When a numeric data series is uniformly distributed, or supposed to be, each unique value has an equal probability of occurrence. With 9 unique values and 1,000 records, we might expect `np.random.randint()` to return 111 occurrences for each value, which, of course, it failed to do. But probabilities don't always translate to results, especially at lower record counts. Flipping a coin 10 times might return heads 8 times and tails just 2 times. That's not because the coin isn't fair; it's because we can sometimes get anomalous or less-than-perfect results when the record counts are low or the number of events is low. Had we instead passed, let's say, 10,000 records to the `np.random.randint()` method instead of just 1,000, no doubt these figures would have converged toward a perfect uniform distribution. But the larger point to be made is that a uniform distribution is very different from a Benford distribution. Even a distribution that is less than perfectly flat would not resemble a Benford distribution that is right-skewed or positively skewed.

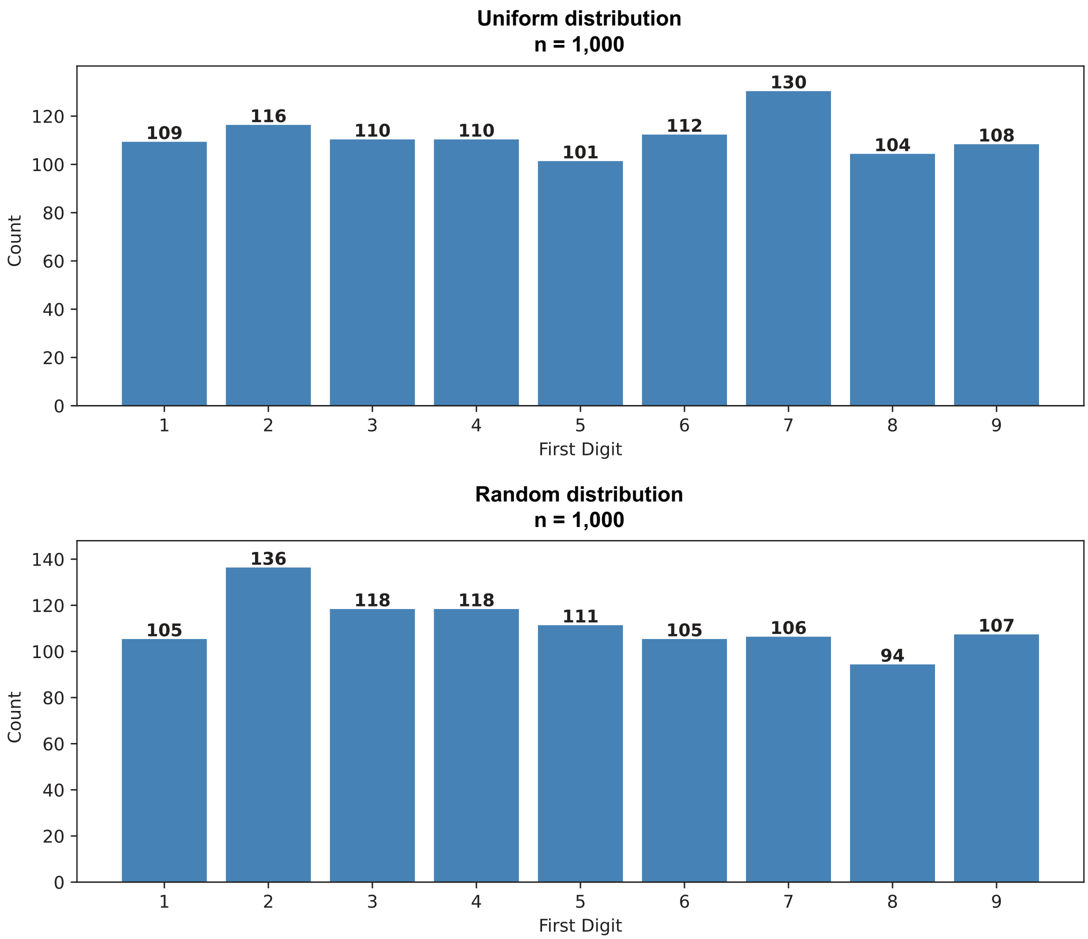

Figure 12.2 A uniform distribution of 1,000 records on top and a random distribution of 1,000 records on the bottom. Many numeric data sets take on one of these two probability distributions. In a Benford distribution, 1s, 2s, and 3s represent more than 60% of the occurrences, but in both these distributions, those digits represent barely 30% of the occurrences.

The same can be said for a random distribution. A Benford distribution, of course, has an obvious pattern; any random distribution, on the other hand, even with 1,000 records or much fewer, will almost always appear indiscriminate, which suggests the impossibility of mistaking a Benford distribution for any uniform or random probability distribution. Let's now examine three real-world data sets that should obey Benford's law to see how well they actually conform to a logarithmic distribution.

12.4 Examples

Our plan is to import three data sets, compute and plot the distribution of leading digits, and compare these distributions to a perfect Benford, or logarithmic, distribution. These three examples will dispel any lingering doubts that smaller leading digits are far more prevalent in some numeric data sets than larger leading digits.

12.4.1 Street addresses

The first of the three data sets we'll examine is a list of street addresses from East Baton Rouge Parish in Louisiana. The data was downloaded from the Baton Rouge website. According to Benford's law, approximately 30% of the street addresses should begin with the numeral 1, and fewer than 5% of the same addresses should begin with the numeral 9. Let's put that hypothesis to the test.

To import the data, saved as a .csv file in our working directory, we pass the full filename, bounded by opening and closing quotation marks, to the `pd.read_csv()` method. The `usecols = [0]` parameter instructs `pd.read_csv()` to import only the first column of the file. From this short snippet of code, we get a data frame called `street_addresses`:

```
>>> street_addresses = pd.read_csv('street_address_listing.csv',
>>>             usecols = [0])
```

Then we call the `info()` method to get a concise summary of the `street_addresses` data frame:

```
>>> print(street_addresses.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 194836 entries, 0 to 194835
Data columns (total 1 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   ADDRESS_NO  194836 non-null  object
dtypes: object(1)
memory usage: 1.5+ MB
None
```

We subsequently learn the following about our data:

- The number of rows, or entries, equals 194,836. The `info()` method excludes the header from the row count, so `street_addresses` contains 194,836 address numbers.

- There are no null values to be concerned with; we know this because the number of non-null values matches the row count.
- The one column we imported is ADDRESS_NO. Although the data contains other variables—street names, zip codes, and district numbers, just to name a few—our analysis requires address numbers only.

But to evaluate whether East Baton Rouge Parish street addresses conform to a Benford distribution, we need a data frame that groups the data by leading digits between 1 and 9 and contains the percentage of occurrences for each group relative to the total record count—in other words, a data frame much like df1, which means we need to perform a short series of data wrangling operations. We begin by adding a new variable to the street_addresses data frame called first_digit; it's created by calling the apply() method, which converts the values in ADDRESS_NO to strings, extracts the first character [0], and then stores it in our new variable. Consequently, street_addresses has a second variable, first_digit, in which each value represents the first, or leading, digit in the corresponding address number:

```
>>> street_addresses['first_digit'] = (street_addresses['ADDRESS_NO'] \
                                       .apply(lambda x: str(x)[0]))
```

A subsequent call to the head() method returns the first 10 records for our review:

```
>>> print(street_addresses.head(10))
       ADDRESS_NO first_digit
0  7353 STE B 282           7
1      9007 STE 9           9
2     5830 STE A6           5
3    4520 STE 103           4
4      8334 STE D           8
5    4250 UNIT 14           4
6            8316           8
7      1707 STE E           1
8           14261           1
9            8926           8
```

Next we apply the set() method to the first_digit variable. set() returns a set of unique values found in first_digit called unique_values:

```
>>> unique_values = set(street_addresses.first_digit)
>>> print(unique_values)
{'9', 'B', '8', 'T', '5', '3', '6', '1', '7', '4', '2', 'A'}
```

We have an issue that requires our immediate attention: street_addresses contains some number of records—we don't know how many—in which first_digit equals A, B, or T and therefore is not a numeral between 1 and 9. The following snippet of code subsets street_addresses in those records where first_digit equals a numeral between 1 and 9. In other words, it discards any and all records where first_digit equals a value other than a numeral between 1 and 9:

```
>>> street_addresses = street_addresses[street_addresses['first_digit'] \
>>>     .isin(['1', '2', '3', '4', '5', '6', '7', '8', '9'])]
```

Then we pass `first_digit` to the `astype()` method to convert it to an integer. Our next snippet of code requires `first_digit` to be either an integer (`int`) or a float (contains a fractional component); otherwise, it will throw an error and won't run. This is our final data wrangling operation:

```
>>> street_addresses['first_digit'] = (street_addresses['first_digit'] \
>>>                                    .astype(int))
```

Now that we have a variable of leading digits that's been cleansed and converted, we can perform our analysis. We import the `benford` library and then make a call to the `bf.first_digits()` method, which requires a minimum of two parameters:

- `street_addresses.first_digit` instructs `bf.first_digits()` to access the variable `first_digit` from the `street_addresses` data frame.
- `digs = 1` tells `bf.first_digits()` to evaluate just the leading digit (it doesn't matter that the values in `first.digit` are just one byte).

Here's the code:

```
>>> import benford as bf
>>> bf_street_addresses = bf.first_digits(street_addresses.first_digit,
>>>                                       digs = 1)
```

This returns two objects:

- A 9 × 4 data frame called `bf_street_addresses` with the following columns:
  - `First_1_Dig`, which contains the numerals `1` through `9`.
  - `Counts`, which equals the number of occurrences grouped by leading digit.
  - `Found`, which equals the percentage of each group relative to the total record count, rounded to six decimal places.
  - `Expected`, which is similar to the `df1` variable `benford`, but rounded to six decimal places. It therefore represents a perfect Benford distribution.
- A preconfigured bar line chart that compares the observed distribution of leading digits to a perfect Benford distribution. The plot is displayed automatically. It contains the following elements:
  - An *x* axis called `Digits` that points to `First_1_Dig`
  - A *y* axis called `Distribution (%)` that points to `Found` and `Expected`
  - Bars that represent the observed distribution (`Found`)
  - A solid line that represents a perfect Benford distribution (`Expected`)
  - A title called `Expected vs. Found Distributions`
  - A legend in the upper-right corner

We intend to print the data frame later (and assign a derived variable to it for testing purposes), but figure 12.3 shows the plot in the meantime. Although it is not a perfect Benford distribution—the real world is rarely perfect—leading digits in street addresses from East Baton Rouge Parish nonetheless assume an obvious Benford distribution, where smaller leading digits are significantly more prevalent than larger leading digits. This pattern aligns with Benford's law because street addresses—unlike ZIP codes or assigned identifiers—arise from organic, right-skewed numeric growth and span multiple orders of magnitude. Although 1s are actually more numerous than we might expect, 1s plus 2s plus 3s equal 60.4% of the total record count, compared to a perfect Benford distribution where those same digits represent 60.2% of all observations. Let's now examine world population figures.

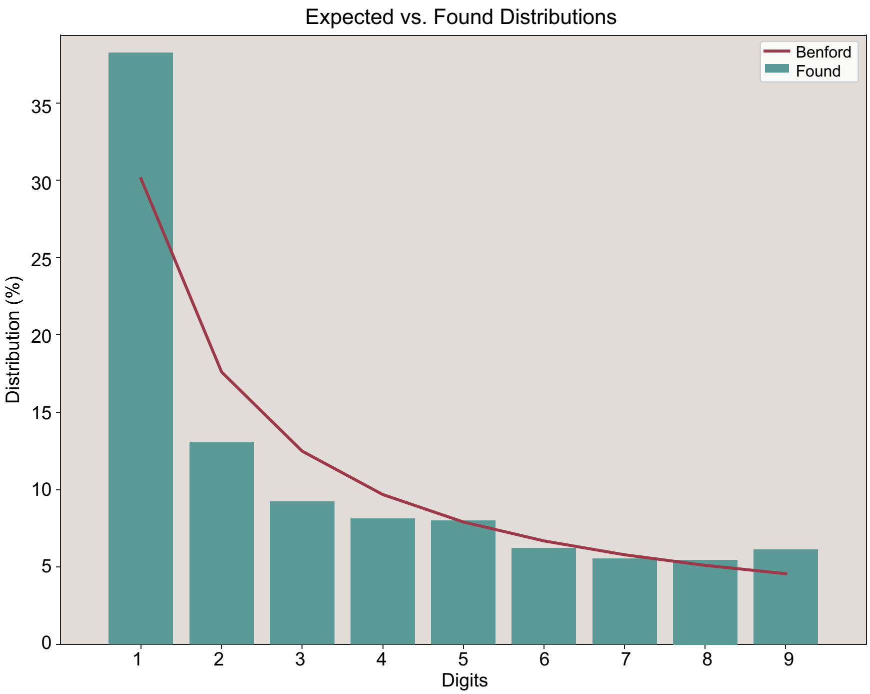

Figure 12.3 The distribution of leading digits in street addresses from East Baton Rouge Parish in Louisiana. The distribution closely resembles a perfect Benford distribution, where smaller leading digits are much more prevalent than larger leading digits. In fact, 1s, 2s, and 3s are the leading digits in 60.4% of the street addresses; in a perfect Benford distribution, those digits represent 60.2% of the observations.

12.4.2 World population figures

Our second of three data sets to examine is a .csv file downloaded from Kaggle that contains 2020 population counts for 235 countries and territories. We begin by calling the `pd.read_csv()` method to import a file that we since saved in our working directory, called population_by_country_2020.csv. Once again, we instruct `pd.read_csv()` to only read the data we absolutely require for our analysis; so, we pass a second parameter to the `pd.read_csv()` method, `usecols = [1]`, which tells `pd.read_csv()`

to only import the second column from the left. As a result, we get a data frame called `populations`, which contains the per-country and per-territory population counts.

Next we call the `info()` method, which prints a concise summary of the `populations` data frame:

```
>>> populations = pd.read_csv(population_by_country_2020.csv',
>>>               usecols = [1])
>>> print(populations.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 235 entries, 0 to 234
Data columns (total 1 column):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Population  235 non-null    int64
dtypes: int64(1)
memory usage: 2.0 KB
None
```

This returns, most significantly,

- The number of rows, or entries
- An itemized list of columns from left to right, or starting from 0 (just `Population`)
- The number of non-null values

As before, we need a second variable of leading digits, converted to type integer, that we can plot and analyze.

We start by creating that variable, called `first_digit`, which now equals the leading digit from the original variable `Population`. The `apply()` method converts the values in `Population` to strings, extracts the first character `[0]`, and stores it in `first_digit`. As a result, the `populations` data frame contains a second variable, `first_digit`, in which each value represents the first, or leading, digit in the corresponding population count.

A subsequent call to the `head()` method returns the first 10 records for our review:

```
>>> populations['first_digit'] = (populations['Population'] \
>>>                                .apply(lambda x: str(x)[0]))
>>> print(populations.head(10))
   Population first_digit
0  1440297825           1
1  1382345085           1
2   331341050           3
3   274021604           2
4   221612785           2
5   212821986           2
6   206984347           2
7   164972348           1
8   145945524           1
9   129166028           1
```

Now we convert first_digit to an integer:

```
>>> populations['first_digit'] = populations['first_digit'].astype(int)
```

And then we pass first_digit to the bf.first_digits() method to get a data frame called bf_populations and another bar line chart that displays the observed versus expected distributions of leading digits:

```
>>> bf_populations = bf.first_digits(populations.first_digit, digs = 1)
```

Figure 12.4 shows the plot. Again, we get a distribution that is undoubtedly consistent with Benford's law. The smallest leading digits (1s, 2s, and 3s) represent 58.3% of the record count, and the largest leading digits (7s, 8s, and 9s) represent 14.1% of all records. By comparison, these same figures are 60.2% and 14.5% in a perfect Benford distribution. Pretty close.

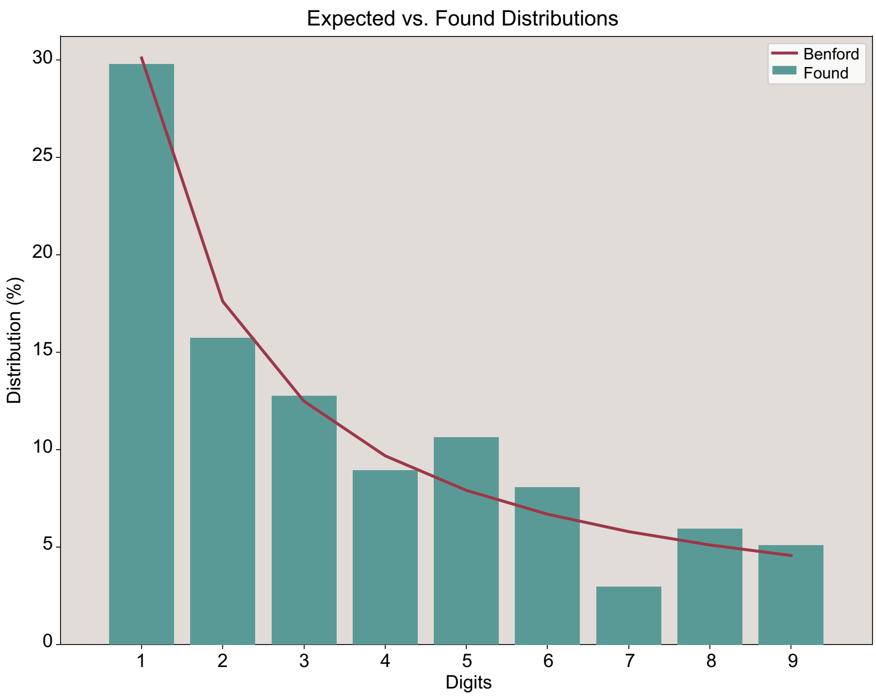

Figure 12.4 The distribution of leading digits in world population counts from 2020. Once more, we see a distribution in accordance with Benford's law. In this case, 1s, 2s, and 3s constitute 58.3% of the record count (versus a perfect Benford distribution of 60.2%), whereas 7s, 8s, and 9s represent 14.1% of all records (compared to 14.5% from a perfect Benford distribution).

Our third and final data set contains 2010 payment figures from a utility company operating on the West Coast of the United States. We'll travel a familiar path to get from the raw data to a summarized data frame for plotting and analysis.

12.4.3 Payment amounts

Our third and last data set to examine is corporate_payments.csv, also stored in our working directory. The data was acquired by attaching it to an R script and writing it to a .csv file. We make a call to the pandas `pd.read_csv()` method to import just the data contained in the fourth column and assign it to a data frame called `payments`; we then immediately call the `info()` method to get basic summary information returned. This snippet of code and most of the others that follow are similar to what you've previously been exposed to:

```
>>> payments = pd.read_csv('corporate_payments.csv',
>>>              usecols = [3])
>>> print(payments.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 189470 entries, 0 to 189469
Data columns (total 1 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Amount  189470 non-null  float64
dtypes: float64(1)
memory usage: 1.4 MB
None
```

Most significantly, we see that `payments` contains 189,470 records and no null values. The name of the one column we imported is `Amount`, of type `float`.

Next, we convert the values in `Amount` to strings, extract the first character `[0]`, and store the same in a new variable called `first_digit`. A call to the `head()` method then returns the top 10 records:

```
>>> payments['first_digit'] = payments['Amount'].apply(lambda x: str(x)[0])
>>> print(payments.head(10))
   Amount first_digit
0   36.08           3
1   77.80           7
2   34.97           3
3   59.00           5
4   59.56           5
5   50.38           5
6   26.57           2
7  102.17           1
8   25.19           2
9   37.31           3
```

It's not uncommon for payments (or invoice amounts) to equal a negative number or 0, due to credits, adjustments, refunds or returns, free services, write-offs—whatever. Where `Amount` equals a negative number, we extracted the minus (`-`) operator rather than the leading digit; where `Amount` equals 0, we extracted a leading digit that shouldn't factor into our analysis. We therefore subset the `payments` data frame where `first_digit` is greater than or equal to `1` and less than or equal to `9`:

```
>>> payments = payments[(payments.first_digit >= '1') &
                        (payments.first_digit <= '9')]
```

Then we convert the values in `first_digit` to integers:

```
>>> payments['first_digit'] = payments['first_digit'].astype(int)
```

And finally, we pass `first_digit` to the `bf.first_digits()` method, which returns a data frame called `bf_payments` as well as the bar line chart shown in figure 12.5.

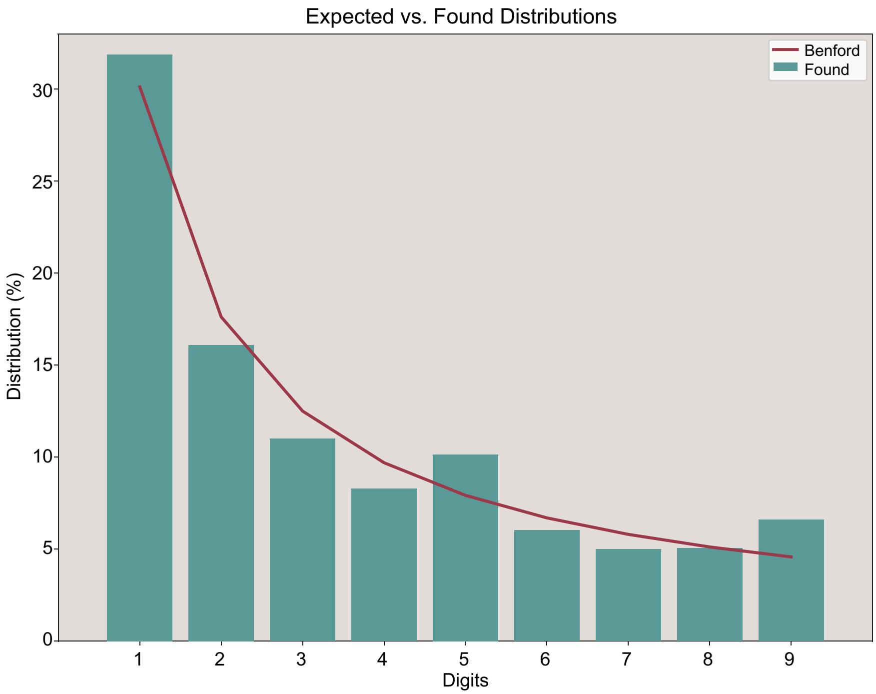

Figure 12.5 The distribution of leading digits in 2010 payment amounts from a utility company on the US West Coast. Again, we see a distribution that obviously obeys Benford's law. The smallest leading digits (1s, 2s, and 3s) represent exactly 59% of the total record count, and the largest leading digits (7s, 8s, and 9s) represent just 16.6% of the records. Once more, in a perfect Benford distribution, these figures are 60.2% and 14.5%, respectively.

We've demonstrated that street addresses, world population figures, and now payment amounts—three data sets that should follow Benford's law—do, in fact, follow Benford's law, at least on visual inspection. Computing the distribution of leading digits, plotting it, and estimating their likeness to a perfect Benford distribution is a great start. But there are statistical methods by which to get exact measurements; we'll demonstrate those next.

12.5 Validating Benford's law

Numeric data that, on visual inspection, assumes a Benford distribution may still contain enough anomalies to raise suspicion. We may, or may not, detect these by drawing bar line charts and comparing real-world distributions to a perfect Benford distribution, which is why it's important to also run statistical tests and compare the results to expected thresholds.

Our plan is to pass the `populations` data frame to several statistical and arithmetic functions to determine how well that data really conforms to an expected Benford distribution. Furthermore, we intend to pop the hood on these same operations to provide quantitative insights into what the numbers mean and how they're derived.

> **Law of anomalous numbers**
>
> In addition to being known as the first-digit law and the Newcomb–Benford law, Benford's law is also referred to, sometimes, as the law of anomalous numbers. That's because Benford's law goes beyond just the leading digit in naturally occurring number sequences; the law of anomalous numbers therefore generalizes the phenomenon when the analysis includes, for instance, the first two digits and not just the leading digit. Of course, our scope throughout has been the leading digit only, which is most common.

Although visual inspection of bar charts obviously provides initial insights and visualizations of data distribution, statistical tests and the like offer a more rigorous and systematic approach to evaluating conformity to Benford's law, thereby enhancing the reliability and credibility of conclusions and establishing a strong footing for decision and action. We'll perform four tests in all, beginning with a chi-square goodness of fit test.

12.5.1 Chi-square test

Our first test, a chi-square goodness of fit test, requires a single data wrangling operation as a prerequisite. We need to create a derived variable and assign it to the `bf_populations` data frame we generated earlier. So, we call the `assign()` method to create a new variable called `Expected_Counts`, which is derived by multiplying the values in the `Expected` column by the sum of the values in the `Counts` column. A subsequent call to the `print()` function returns the new and improved `bf_populations` data frame. Although this data set encompasses an entire population rather than a sample, the chi-square test still relies on having a sufficient record count to yield meaningful statistical conclusions:

```
>>> bf_populations = \
>>>     bf_populations.assign(Expected_Counts = \
>>>                           bf_populations.Expected * \
>>>                           sum(bf_populations.Counts))
>>> print(bf_populations)
             Counts     Found  Expected  Expected_Counts
First_1_Dig
1                70  0.297872  0.301030        70.742049
2                37  0.157447  0.176091        41.381446
3                30  0.127660  0.124939        29.360603
```

```
4                  21  0.089362  0.096910        22.773853
5                  25  0.106383  0.079181        18.607593
6                  19  0.080851  0.066947        15.732496
7                   7  0.029787  0.057992        13.628108
8                  14  0.059574  0.051153        12.020843
9                  12  0.051064  0.045757        10.753010
```

Thus, `Expected_Counts` represents the number of occurrences for each leading digit if world populations actually followed a perfect Benford distribution.

With that sorted out, we next import the `stats` module from the `scipy` library, which contains several methods for scientific and technical computing, and then make a call to the `stats.chisquare()` method. A chi-square test is a statistical test used to determine whether there is a significant association between categorical variables; it compares the observed distribution against the expected distribution and then evaluates if any variances are statistically significant. Our null hypothesis is that the data conforms to a Benford distribution. If our chi-square test returns a p-value less than 5%, we'll reject the null hypothesis and conclude the data either best conforms to some other probability distribution or, perhaps, contains more than a few suspicious anomalies. Alternatively, if our chi-square test returns a p-value greater than 5%, we'll fail to reject the null hypothesis and conclude the data does, in fact, obey Benford's law:

```
>>> import scipy.stats as stats
>>> x2 = stats.chisquare(bf_populations.Counts,
>>>                      bf_populations.Expected_Counts)
>>> print(x2)
Power_divergenceResult(statistic=7.192526063677557,
     pvalue=0.5160103482305225)
```

The test statistic equals 7.193, and the p-value equals 51%. Let's examine these results and demonstrate how they're derived, one by one.

CHI-SQUARE STATISTIC

The chi-square statistic is derived by applying the following equation to each leading digit and then summing the results:

$$x^2 = \sum \text{all cells} \frac{(O_i - E_i)^2}{E_i}$$

For every leading digit, we're squaring the difference between observed (O_i) and expected (E_i) counts and dividing that by the expected count (E_i); then we add the quotients, or the chi-square statistic for each leading digit, to get the chi-square statistic for the test. In other words, we're squaring the differences between `Counts` and `Expected_Counts` and dividing those differences by `Expected_Counts`. The best way of further demonstrating this is to arrange the figures in the form of a table and review the results: see table 12.1.

Table 12.1 Chi-square statistics for leading digits in world population counts

Leading digit	Observed (O_i)	Expected (E_i)	$\|O_i \bullet E_i\|$	$(O_i \bullet E_i)^2$	$(O_i \bullet E_i)^2 / E_i$
1	70	70.742	0.742	0.551	0.008
2	37	41.381	4.381	19.193	0.464
3	30	29.361	0.639	0.408	0.014
4	21	22.774	1.774	3.147	0.138
5	25	18.608	6.392	41.858	2.196
6	19	15.732	3.268	10.680	0.679
7	7	13.628	6.628	43.930	3.224
8	14	12.021	1.979	3.916	0.326
9	12	10.753	1.247	1.555	0.145
	235	*235*			*7.192*

When computing the chi-square statistic by hand, we get 7.192, which essentially matches what our test returned (7.193). The variances in leading digits 5 and 7 are mostly accountable for the final test statistic.

Going from left to right, we have the following columns:

- *Leading digit* equals `1` through `9`, or contains every unique leading digit in the variable `First_1_Dig` from the `bf_populations` data frame.
- *Observed* (O_i) equals the actual, or observed, count for every leading digit. Sums to 235, which of course ties back to the number of countries and territories in population_by_country_2020.csv.
- *Expected* (E_i) equals the expected count for every leading digit in accordance with a perfect Benford distribution. For instance, when the leading digit equals `2`, we get the expected count by multiplying 235 times 0.176.
- $|O_i - E_i|$ equals the absolute difference between observed and expected counts.
- $(O_i - E_i)^2$ equals the square of the difference between observed and expected counts.
- $(O_i - E_i)^2 / E_i$ equals the square of the difference between observed and expected counts divided by the expected count. This is therefore the chi-square statistic per leading digit. We get the chi-square statistic for the test by summing the per-digit statistics.

The chi-square statistic is calculated as 7.192, closely matching the result obtained from the `stats.chisquare()` method.

DEGREES OF FREEDOM

The degrees of freedom (df) refer to the number of values in the final calculation of a statistical test that are free to vary. From a chi-square goodness of fit test, the df are derived by applying the following equation, where r equals the number of rows and c equals the number of columns, or categories, in a contingency table:

$$df = (r - 1)(c - 1)$$

Because the row count equals 9 and the number of categories equals 2, the df therefore equal (9 – 1)(2 – 1), or 8. The `stats.chisquare()` method doesn't return this figure; however, it's important to derive it, if only to help us understand how we get the p-value, which of course determines whether the results are statistically significant.

P-VALUE

By cross-referencing the chi-square statistic (7.193) and the df (8) on a chi-square probabilities table, we can get an *estimated* p-value. Figure 12.6 shows a snippet from a chi-square probabilities table. To get an estimated p-value for our chi-square test, find the df on the far left and then locate the two consecutive critical values from the same row that bound the chi-square statistic. The p-value is between the two numbers from the same columns in the header. Thus, the p-value is equal to or greater than 0.10 and less than or equal to 0.90. Thankfully, the `stats.chisquare()` method returns an *exact* p-value with much less effort.

df	0.995	0.99	0.975	0.95	0.90	0.10	0.05	0.025	0.01	0.005
1	---	---	0.001	0.004	0.016	2.706	3.841	5.024	6.635	7.879
2	0.010	0.020	0.051	0.103	0.211	4.605	5.991	7.378	9.210	10.597
3	0.072	0.115	0.216	0.352	0.584	6.251	7.815	9.348	11.345	12.838
4	0.207	0.297	0.484	0.711	1.064	7.779	9.488	11.143	13.277	14.860
5	0.412	0.554	0.831	1.145	1.610	9.236	11.070	12.833	15.086	16.750
6	0.676	0.872	1.237	1.635	2.204	10.645	12.592	14.449	16.812	18.548
7	0.989	1.239	1.690	2.167	2.833	12.017	14.067	16.013	18.475	20.278
8	1.344	1.646	2.180	2.733	3.490	13.362	15.507	17.535	20.090	21.955
9	1.735	2.088	2.700	3.325	4.168	14.684	16.919	19.023	21.666	23.589
10	2.156	2.558	3.247	3.940	4.865	15.987	18.307	20.483	23.209	25.188

Figure 12.6 A snippet from a typical chi-square probabilities table from which we can get an estimated p-value by cross-referencing the degrees of freedom and the chi-square statistic. The degrees of freedom equal 8; the chi-square statistic equals 7.1925, which is between 3.490 and 13.362; the p-value therefore equals some number between 0.10 and 0.90. Although that's a substantial range, it falls entirely above the 5% threshold. Thus, without running any code, we can determine that the results are not statistically significant, and therefore we can, and should, fail to reject the null hypothesis.

Because the p-value is above the 5% significance threshold, we fail to reject the null hypothesis that the data conforms to a Benford distribution. Therefore, we conclude that the leading digits in world population counts do, in fact, obey Benford's law.

12.5.2 Mean absolute deviation

Whereas a chi-square goodness of fit test considers the size of the data as well as the variances between observed and expected counts, a mean absolute deviation (MAD) test ignores the number of rows in the data and merely takes into account the absolute differences between actual and expected proportions. It is expressed by the following equation:

$$\text{MAD} = \frac{\sum |P_i - P0_i|}{K}$$

Where P_i is the observed proportion, PO_i is the expected proportion in accordance with a perfect Benford distribution, and K is the number of leading digits.

In Python, we get the mean absolute deviation by passing `first_digit` from the `populations` data frame to the `bf.mad()` method. Because we're testing just the first, or leading, digit, rather than two or more digits, we also pass `test = 1` to `bf.mad`. And because we're not working with fractional numbers, we additionally pass `decimals = 0`:

```
>>> MAD = bf.mad(populations.first_digit, test = 1, decimals = 0)
>>> print(MAD)
0.012790028799782804
```

The mean absolute deviation is the sum of the per-digit absolute deviations divided by the number of digits (see table 12.2).

Table 12.2 Absolute deviations and the mean absolute deviation for leading digits in world population counts

Leading digit	Observed (P_i)	Expected (PO_i)	$\|P_i$ • $PO_i\|$ / K
1	0.298	0.301	0.003
2	0.157	0.176	0.019
3	0.128	0.125	0.003
4	0.089	0.097	0.008
5	0.106	0.079	0.027
6	0.081	0.067	0.014
7	0.030	0.058	0.028
8	0.060	0.051	0.009
9	0.051	0.046	0.005
	1.000	*1.000*	*0.013*

The data can be itemized like so:

- *Leading digit* equals `1` through `9`, or contains every unique leading digit in the variable `First_1_Dig` from the `bf_populations` data frame.
- *Observed* (P_i) equals the actual, or observed, proportion for every leading digit. Sums to 1.
- *Expected* (PO_i) equals the expected proportion for every leading digit in accordance with a perfect Benford distribution. Sums to 1.

- $|P_i - PO_i| / K$ for the leading digits equals the absolute deviation between the observed and expected proportions. The bottom figure (0.013) is the mean absolute deviation, which is derived by summing the absolute deviations and dividing the total by the number of leading digits.

Mean absolute deviations equal to or very close to 0.000 suggest close conformity to Benford's law; on the other end, mean absolute deviations greater than 0.015 imply nonconformity. When the mean absolute deviation equals 0.013, as it does here, our best conclusion is that the data "marginally conforms" to an expected Benford distribution.

12.5.3 Distortion factor and z-statistic

The distortion factor (DF) is a statistic that represents the percentage of deviation between the actual mean of leading digits that presumably follow Benford's law and the expected mean of the same. It's a meaningful metric by itself, but when the distortion factor is then divided by the `first_digit` standard deviation, we get the z-statistic, which tells us whether we should reject or fail to reject a null hypothesis.

We get the distortion factor by plugging the actual mean (AM) and expected mean (EM) of the leading digits into the following equation:

$$DF = \frac{100(\mathrm{A}M - \mathrm{E}M)}{\mathrm{E}M}$$

We can easily get the actual mean by passing `first_digit` to the `mean()` method:

```
>>> AM = populations['first_digit'].mean()
>>> print(AM)
3.5148936170212766
```

Getting the expected mean requires more effort. A perfect Benford distribution is, of course, constant, but the mean varies due to inconsistencies in data sizes. One way of computing an *approximate* mean of a perfect Benford distribution with 235 observations is to do the following:

1. Multiply each leading digit by the expected count: 1 × 70.742, 2 × 41.381, and so forth.
2. Add the nine products.
3. Divide the sum by 235.

We'll use Python as a calculator to get an approximate expected mean and again to compute the distortion factor:

```
>>> EM = ((1 * 70.742) + (2 * 41.381) + (3 * 29.361) +
>>>       (4 * 22.774) + (5 * 18.608) + (6 * 15.732) +
>>>       (7 * 13.628) + (8 * 12.021) + (9 * 10.753)) / 235
>>> print(EM)
>>> 3.4402382978723405
```

Now that we have the actual mean (3.515) and an approximate expected mean (3.440), we can compute the distortion factor:

```
>>> DF = (100 * (AM - EM)) / EM
>>> print(DF)
2.1700624400091013
```

So, the amount of deviation between the actual and expected means equals 2.17%. The closer the distortion factor is to 0%, the better the data conforms to a Benford distribution.

Let's now get the standard deviation of `first_digit` by making a call to the `np.std()` method. Standard deviation is a measure of the dispersion or variability of numeric data from its mean, thereby indicating how much those values deviate from the average:

```
>>> SD = np.std(populations['first_digit'])
>>> print(SD)
2.4760107519964065
```

Now that we have the distortion factor (2.17) and the `first_digit` standard deviation (2.48), we can compute the z-statistic; and from the z-statistic, we can make another conclusion about whether leading digits from 2020 world population counts conform well enough to an expected Benford distribution. We get the z-statistic by dividing the distortion factor by the standard deviation.

$$\text{z-statistic} = \frac{DF}{\text{Standard Deviation}}$$

Again, using Python as a calculator,

```
>>> z = DF / SD
>>> print(Z)
0.8764349824649916
```

Once more, our null hypothesis is that the data conforms to a Benford distribution. Calculated z-statistics between the critical values −1.96 and 1.96 are insignificant at the 5% threshold; thus, we should again fail to reject the null hypothesis that leading digits in world population counts obey Benford's law.

12.5.4 Mantissa statistics

The logarithm of any number is split into two parts: the numeral to the left of the decimal point is the *characteristic*, or integer, and the fractional part to the right of the decimal point is the *mantissa*. Take the number 287,437, for instance, which was the population of Barbados back in 2020, at least according to our data set; when we compute the logarithm of 287,437, we get 5.458543. The characteristic is 5, and the mantissa is 0.458543.

For our purposes, the key mantissa statistics are the mean, variance, excess kurtosis, and skewness. If the leading digits obey Benford's law, the mantissas should assume a uniform distribution, which can be established by computing the aforementioned statistics:

- The *mean*, often referred to as the *average*, is a measure of central tendency in a numeric data series. It represents the sum of all values divided by the number of records.
- The *variance* measures the amount of dispersion, or variability, in numeric data by quantifying the average squared deviation of each data point from the population mean.
- *Excess kurtosis* measures the tails, or the numeric data points farthest from the mean. It quantifies the "sharpness" or "flatness" of the tail, left or right, compared to a normal, or Gaussian, distribution. When it's equal to 0, the data has the same tail behavior as a normal distribution; when it's positive, the data contains more outliers than a typical normal distribution; and when it's negative, the data instead contains fewer outliers than a normal distribution.
- *Skewness* measures the asymmetry in the distribution of numeric data. When data is perfectly symmetrical, the skewness is 0; when right-skewed, the skewness is positive; and when left-skewed, the skewness is negative.

When data perfectly obeys Benford's law, the mantissas have the following properties:

$$\text{Mean(Mantissa)} = 0.50$$
$$\text{Variance(Mantissa)} = 0.08$$
$$\text{Excess Kurtosis(Mantissa)} = -1.20$$
$$\text{Skewness(Mantissa)} = 0$$

Before we compute these same statistics against the observed data, and then of course compare actual versus theoretical results, we first need to create a pair of derived variables and assign both to the `populations` data frame.

Our first variable is `log`, which equals the base-10 logarithm of the original variable `Population`. With respect to Barbados, for instance, `Population` equals `287437` and `log` equals `5.458543`:

```
>>> populations['log'] = np.log10(populations['Population'])
```

Our second variable is `mantissa`, which equals the mantissa extracted from the variable `log`. This is derived by subtracting the characteristic, or integer, from `log` and assigning the difference to `mantissa`. To again use Barbados as an example, where `log` equals `5.458543`, `mantissa` equals `0.458543`:

```
>>> populations['mantissa'] = \
        populations['log'] - populations['log'].astype(int)
```

A call to the `head()` method displays the top 10 records for our review:

```
>>> print(populations.head(10))
   Population  first_digit       log  mantissa
0  1440297825            1  9.158452  0.158452
1  1382345085            1  9.140616  0.140616
```

```
2   331341050           3  8.520275  0.520275
3   274021604           2  8.437785  0.437785
4   221612785           2  8.345595  0.345595
5   212821986           2  8.328016  0.328016
6   206984347           2  8.315938  0.315938
7   164972348           1  8.217411  0.217411
8   145945524           1  8.164191  0.164191
9   129166028           1  8.111148  0.111148
```

Now that we have a variable in `populations` that contains the mantissas, we can proceed with calculating the four mantissa statistics.

To get the mean, we pass `mantissa` to the `mean()` method:

```
>>> print(populations['mantissa'].mean())
0.5087762832181758
```

To get the variance, we pass `mantissa` to the `var()` method:

```
>>> print(populations['mantissa'].var())
0.08797164279797032
```

To get the excess kurtosis, we call the `kurtosis()` method from `scipy` and pass a second parameter, `fisher = True`, in addition to the variable name (otherwise, the `kurtosis()` method would return the kurtosis instead of the excess kurtosis):

```
>>> from scipy.stats import kurtosis
>>> print(kurtosis(populations['mantissa'], fisher = True))
-1.177858418956045
```

And to get the skewness, we pass `mantissa` to the `skew()` method, also from `scipy`:

```
>>> from scipy.stats import skew
>>> print(skew(populations['mantissa']))
-0.1128415794074036
```

All of these statistics align very well with those from a perfect Benford distribution.

Finally, we previously mentioned that the mantissas should follow a uniform distribution if the leading digits, in fact, obey Benford's law. Let's complete our analysis by testing this hypothesis; we'll sort and rank the mantissas and then display the results in a Matplotlib plot alongside a perfect uniform distribution.

To sort the variable `mantissa` in ascending order, we make a call to the `sort_values()` method. This will be our *y*-axis variable:

```
>>> populations = populations.sort_values(by = 'mantissa')
```

And to get our *x*-axis variable, we next create a new variable, `rank`, that contains the numerals `1` through `236` (exclusive):

```
>>> populations['rank'] = list(range(1, 236))
```

The `head()` and `tail()` methods return the first three and last three records for our review:

```
>>> print(populations.head(3))
    Population  first_digit       log  mantissa  rank
90    10110233            1  7.004761  0.004761     1
89    10154978            1  7.006679  0.006679     2
88    10191409            1  7.008234  0.008234     3
>>> print(populations.tail(3))
     Population  first_digit       log  mantissa  rank
159      990447            9  5.995831  0.995831   233
92      9910892            9  6.996113  0.996113   234
91      9931333            9  6.997008  0.997008   235
```

The following snippet of Matplotlib code should be familiar by now, with one likely exception: `plt.plot([0, 1], [0, 1]`… draws a red dashed line that represents a perfect uniform distribution starting where *x* and *y* both equal 0 and ending where *x* and *y* equal 1. It is therefore a perfect diagonal line that represents a perfect Benford distribution:

```
>>> plt.plot(populations['rank'], populations['mantissa'],
>>>          color = 'slateblue', linewidth = 1.5)
>>> plt.plot([0, 1], [0, 1], transform = ax.transAxes,
>>>          color = 'red', linestyle = '--')
>>> plt.title('Rank Order of Mantissas', fontweight = 'bold')
>>> plt.xlabel('Rank')
>>> plt.ylabel('Mantissa')
>>> plt.show()
```

Figure 12.7 shows the plot. The sorted mantissas (represented by the solid line) do, in fact, follow a uniform distribution quite well.

By mixing the theoretical with the practical and by combining data wrangling and data visualization techniques, we discovered that real-world numeric data spanning multiple orders of magnitude follows a Benford, or logarithmic, distribution, and not

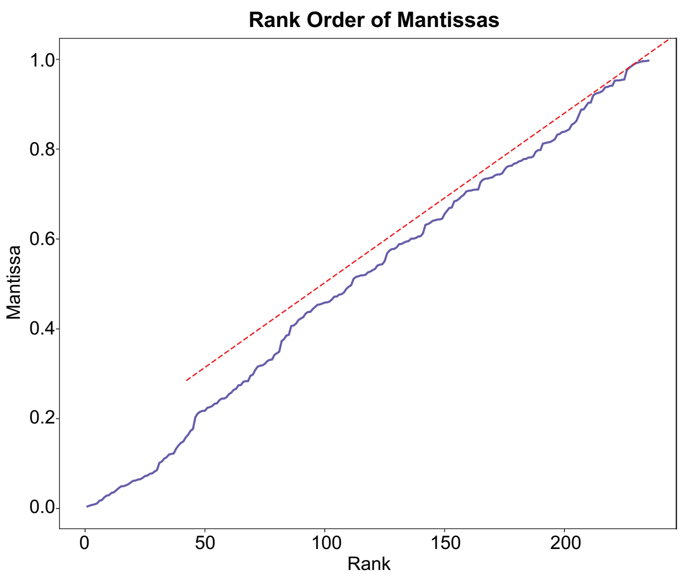

Figure 12.7 The dashed line represents a perfect uniform distribution of mantissas that we would expect to observe when leading digits perfectly obey Benford's law. The solid line represents the mantissas, sorted in ascending order, from the `populations` data frame. The observed mantissas assume a "close-enough" uniform distribution that further suggests world population counts obey Benford's law.

a uniform or random distribution, as many might expect. Furthermore, by subjecting one of our data sets to a battery of statistical tests, we learned how to go above and beyond visual inspection and to apply a series of rigorous and more precise methods to assess and definitively conclude adherence to Benford's law. These learnings empower us to apply similar and equally rigorous methods to other numeric data to test observed distributions against an expected Benford distribution. In the next chapter, we'll demonstrate how to plan, monitor, and control projects.

Summary

- Benford's law, also known as the first-digit law, states that many sets of numeric data follow a logarithmic distribution by which smaller leading digits are more prevalent than larger leading digits.
- Data sets that should obey Benford's law but instead take on a very different distribution might have been manipulated due to fraudulent or other suspicious activity. Significant deviations from Benford's law are just the smoke, not the fire, however. Further investigation might absolutely be warranted when significant deviations from a Benford distribution are observed; at the same time, we should refrain from drawing major conclusions from nothing other than a Benford analysis.
- Understanding Benford's law is especially important in an era where financial fraud, tax evasion, and other forms of data manipulation are increasingly sophisticated. By identifying anomalies in numerical data, analysts and investigators can uncover irregularities that may warrant deeper scrutiny, making Benford's law a valuable tool for fraud detection and forensic accounting.
- Data sets to which Benford's law best applies are those that are distributed across multiple orders of magnitude and don't have preestablished minimum and maximum values; these same sets of data are right-skewed, or positively skewed, by which the mean is greater than the median. Data sets that obey Benford's law are frequently referred to as naturally occurring number sequences.
- We imported three real-world data sets that, going in, were assumed to follow a Benford distribution. By extracting the leading digits from those data sets and plotting the frequency of occurrences for each, we subsequently discovered, at least from visual inspection, that they each obey Benford's law,
- Data comprising numerals used as identifiers may be the best example of numeric data that doesn't qualify as a naturally occurring number sequence. Scores—test scores and IQ scores, for instance—are another good example.
- Don't rely on visual inspection alone when performing a Benford analysis, and by the same token, don't reject or fail to reject a null hypothesis from just one statistical test when options are available. We tested world population data four times and determined from each analysis that 2020 world population figures, based on our rigorous statistical testing, align quite well with the characteristics of a Benford distribution.