**Understanding Descriptive Statistics**

Types of Measures

Population and Samples

Outliers

Choosing Python Statistics Libraries

**Python Statistics Libraries**

Calculating Descriptive Statistics

Measures of Central Tendenc

Measures of Variability

Summary of Descriptive Statistics

Measures of Correlation Between Pairs of Data

Working With 2D Data

Axes

DataFrames

**Visualizing Data**

Box Plots

Histograms

Pie Charts

Bar Charts

X-Y Plots

Heatmaps

**Understanding Descriptive Statistics**

**Descriptive statistics** is about describing and summarizing data. It uses two main approaches:

1. **The quantitative approach** describes and summarizes data numerically.
2. **The visual approach** illustrates data with charts, plots, histograms, and other graphs.

You can apply descriptive statistics to one or many datasets or variables. When you describe and summarize a single variable, you’re performing **univariate analysis**. When you search for statistical relationships among a pair of variables, you’re doing a **bivariate analysis**. Similarly, a **multivariate analysis** is concerned with multiple variables at once.

Types of Measures

In this tutorial, you’ll learn about the following types of measures in descriptive statistics:

* **Central tendency** tells you about the centers of the data. Useful measures include the mean, median, and mode.
* **Variability** tells you about the spread of the data. Useful measures include variance and standard deviation.
* **Correlation or joint variability** tells you about the relation between a pair of variables in a dataset. Useful measures include covariance and the correlation coefficient.

You’ll learn how to understand and calculate these measures with Python.

Population and Samples

In statistics, the **population** is a set of all elements or items that you’re interested in. Populations are often vast, which makes them inappropriate for collecting and analyzing data. That’s why statisticians usually try to make some conclusions about a population by choosing and examining a representative subset of that population.

This subset of a population is called a **sample**. Ideally, the sample should preserve the essential statistical features of the population to a satisfactory extent. That way, you’ll be able to use the sample to glean conclusions about the population.

Outliers

An **outlier** is a data point that differs significantly from the majority of the data taken from a sample or population. There are many possible causes of outliers, but here are a few to start you off:

* **Natural variation** in data
* **Change** in the behavior of the observed system
* **Errors** in data collection

Data collection errors are a particularly prominent cause of outliers. For example, the limitations of measurement instruments or procedures can mean that the correct data is simply not obtainable. Other errors can be caused by miscalculations, data contamination, human error, and more.

There isn’t a precise mathematical definition of outliers. You have to rely on experience, knowledge about the subject of interest, and common sense to determine if a data point is an outlier and how to handle it.

**Choosing Python Statistics Libraries**

There are many Python statistics libraries out there for you to work with, but in this tutorial, you’ll be learning about some of the most popular and widely used ones:

* **Python’s**[**statistics**](https://docs.python.org/3/library/statistics.html) is a built-in Python library for descriptive statistics. You can use it if your datasets are not too large or if you can’t rely on importing other libraries.
* [**NumPy**](https://docs.scipy.org/doc/numpy/user/index.html) is a third-party library for numerical computing, optimized for working with single- and multi-dimensional arrays. Its primary type is the array type called [ndarray](https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html). This library contains many [routines](https://docs.scipy.org/doc/numpy/reference/routines.statistics.html) for statistical analysis.
* [**SciPy**](https://www.scipy.org/getting-started.html) is a third-party library for scientific computing based on NumPy. It offers additional functionality compared to NumPy, including [scipy.stats](https://docs.scipy.org/doc/scipy/reference/stats.html) for statistical analysis.
* [**Pandas**](https://pandas.pydata.org/pandas-docs/stable/) is a third-party library for numerical computing based on NumPy. It excels in handling labeled one-dimensional (1D) data with [Series](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html) objects and two-dimensional (2D) data with [DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html) objects.
* [**Matplotlib**](https://matplotlib.org/) is a third-party library for data visualization. It works well in combination with NumPy, SciPy, and Pandas.

Note that, in many cases, Series and [DataFrame](https://realpython.com/pandas-dataframe/) objects can be used in place of NumPy arrays. Often, you might just pass them to a NumPy or SciPy statistical function. In addition, you can get the unlabeled data from a Series or DataFrame as a np.ndarray object by calling [.values](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.values.html) or [.to\_numpy()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_numpy.html).

## Calculating Descriptive Statistics

>>> import math

>>> import statistics

>>> import numpy as np

>>> import scipy.stats

>>> import pandas as pd

These are all the packages you’ll need for Python statistics calculations. Usually, you won’t use Python’s built-in math package, but it’ll be useful in this tutorial. Later, you’ll import matplotlib.pyplot for data visualization.

Let’s create some data to work with. You’ll start with Python lists that contain some arbitrary numeric data:

>>> x = [8.0, 1, 2.5, 4, 28.0]

>>> x\_with\_nan = [8.0, 1, 2.5, math.nan, 4, 28.0]

>>> x

[8.0, 1, 2.5, 4, 28.0]

>>> x\_with\_nan

[8.0, 1, 2.5, nan, 4, 28.0]

### Measures of Central Tendency

The **measures of central tendency** show the central or middle values of datasets. There are several definitions of what’s considered to be the center of a dataset. In this tutorial, you’ll learn how to identify and calculate these measures of central tendency:

* Mean
* Weighted mean
* Geometric mean
* Harmonic mean
* Median
* Mode

#### **Mean**

The **sample mean**, also called the **sample arithmetic mean** or simply the **average**, is the arithmetic average of all the items in a dataset. The mean of a dataset 𝑥 is mathematically expressed as Σᵢ𝑥ᵢ/𝑛, where 𝑖 = 1, 2, …, 𝑛. In other words, it’s the sum of all the elements 𝑥ᵢ divided by the number of items in the dataset 𝑥.

The green dots represent the data points 1, 2.5, 4, 8, and 28. The red dashed line is their mean, or (1 + 2.5 + 4 + 8 + 28) / 5 = 8.7.

You can calculate the mean with pure Python using [sum()](https://docs.python.org/3/library/functions.html#sum) and [len()](https://docs.python.org/3/library/functions.html" \l "len), without importing libraries:

>>> mean\_ = sum(x) / len(x)

>>> mean\_

8.7

Although this is clean and elegant, you can also apply built-in Python statistics functions:

>>> mean\_ = statistics.mean(x)

>>> mean\_

8.7

>>> mean\_ = statistics.fmean(x)

>>> mean\_

8.7

You’ve called the functions [mean()](https://docs.python.org/3/library/statistics.html#statistics.mean) and [fmean()](https://docs.python.org/3/library/statistics.html" \l "statistics.fmean) from the built-in Python statistics library and got the same result as you did with pure Python. fmean() is introduced in [Python 3.8](https://docs.python.org/3/whatsnew/3.8.html#statistics) as a faster alternative to mean(). It always returns a floating-point number.

However, if there are nan values among your data, then statistics.mean() and statistics.fmean() will return nan as the output:

>> mean\_ = statistics.mean(x\_with\_nan)

>>> mean\_

nan

>>> mean\_ = statistics.fmean(x\_with\_nan)

>>> mean\_

nan

This result is consistent with the behavior of sum(), because sum(x\_with\_nan) also returns nan.

If you use NumPy, then you can get the mean with [np.mean()](https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html):

>>> mean\_ = np.mean(y)

>>> mean\_

8.7

In the example above, mean() is a function, but you can use the corresponding method [.mean()](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.mean.html) as well:

>> mean\_ = y.mean()

>>> mean\_

8.7