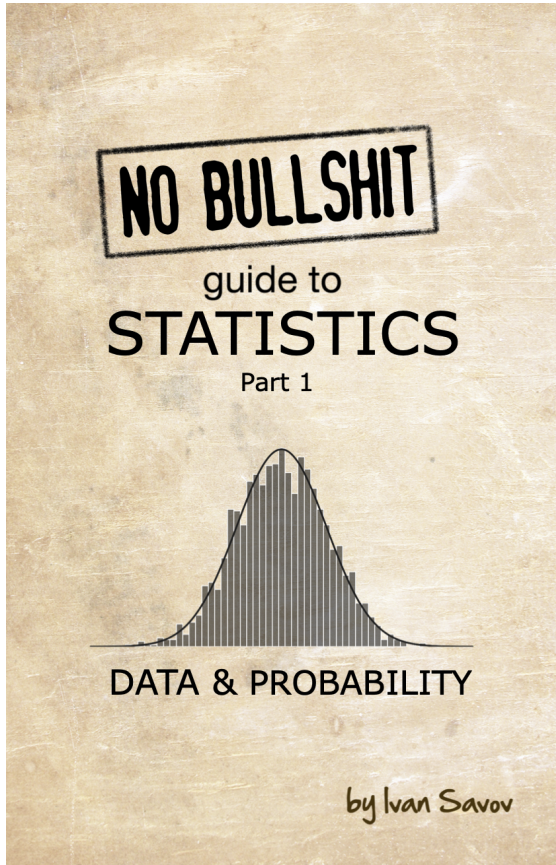


No Bullshit Guide to Statistics

Part 1: Data and Probability

Extended book preview



The full book has 433 pages. This preview has been chosen to showcase the main ideas from Part 1 of the book and includes chapter intros, key formulas, figures, and code examples. Get the eBook here: <https://gum.co/noBSstats>.

Contents

Preface	v
Introduction	1
1 Data	5
1.1 Introduction to data	5
1.1.1 Definitions	6
1.1.2 Study design and randomization	13
1.1.3 Discussion	18
1.1.4 Exercises	21
1.2 Data in practice	22
1.2.1 Getting started with JupyterLab	23
1.2.2 Data management with Pandas	26
1.2.3 Data visualizations with Seaborn	37
1.2.4 Real-world datasets	41
1.2.5 Discussion	50
1.2.6 Exercises	53
1.3 Descriptive statistics	55
1.3.1 Numerical variables	55
1.3.2 Relations between numerical variables	69
1.3.3 Comparing two groups of numerical variables	73
1.3.4 Categorical variables	76
1.3.5 Explanations	85
1.3.6 Discussion	89
1.4 Conclusion	95
1.5 Data problems	96
2 Probability	97
2.1 Discrete random variables	103
2.1.1 Definitions	103
2.1.2 Cumulative distribution functions	110
2.1.3 Expected value calculations	113
2.1.4 Computer models for random variables	119
2.1.5 Hard disks example	121
2.1.6 Exercises	125
2.1.7 Discussion	127

2.2	Multiple random variables	132
2.2.1	Definitions	132
2.2.2	Examples of joint probability distributions	134
2.2.3	Conditional probability distributions	139
2.2.4	Probability formulas and rules	145
2.2.5	Multivariable expectations	147
2.2.6	Independent random variables	149
2.2.7	Discussion	153
2.2.8	Exercises	154
2.3	Inventory of discrete distributions	155
2.3.1	Discrete uniform distribution	156
2.3.2	Bernoulli distribution	157
2.3.3	Mathematical interlude: counting combinations	158
2.3.4	Binomial distribution	159
2.3.5	Poisson distribution	160
2.3.6	Geometric distribution	161
2.3.7	Negative binomial distribution	162
2.3.8	Computer models for discrete distributions	163
2.3.9	Explanations	167
2.3.10	Discussion	172
2.3.11	Exercises	173
2.4	Continuous random variables	175
2.4.1	Definitions	175
2.4.2	Calculus prerequisites	177
2.4.3	Computer models for random variables	181
2.4.4	Cumulative distribution functions	186
2.4.5	Calculating expectations	190
2.4.6	Example 3: kombucha volume	194
2.4.7	Discussion	200
2.4.8	Exercises	203
2.5	Multiple continuous random variables	204
2.5.1	Joint probability density function	204
2.5.2	Example: bivariate normal distribution	205
2.5.3	Marginal distribution functions	206
2.5.4	Conditional probability distributions	207
2.5.5	Probability formulas	209
2.5.6	Multivariable expectation	210
2.5.7	Exercises	214
2.6	Inventory of continuous distributions	215
2.6.1	Uniform distribution	217
2.6.2	Exponential distribution	219
2.6.3	Normal distribution	220
2.6.4	Standard normal distribution	221
2.6.5	Mathematical interlude: the gamma function	222
2.6.6	Student's t -distribution	223
2.6.7	Chi-square distribution	224
2.6.8	Fisher-Snedecor F -distribution	225
2.6.9	Gamma distribution	226

2.6.10	Beta distribution	227
2.6.11	Laplace distribution	227
2.6.12	Explanations	228
2.6.13	Discussion	233
2.6.14	Exercises	235
2.7	Simulation and empirical distributions	237
2.7.1	Why simulate?	237
2.7.2	Definitions	238
2.7.3	Generating observations from random variables	238
2.7.4	Empirical distributions	242
2.7.5	Random variable generation from scratch	247
2.7.6	Discussion	255
2.7.7	Exercises	257
2.8	Probability models for random samples	259
2.8.1	Definitions	259
2.8.2	Sample statistics	260
2.8.3	Sampling distribution of the mean	263
2.8.4	Central limit theorem	272
2.8.5	Discussion	277
2.8.6	Exercises	279
2.9	Conclusion	280
2.10	Probability problems	286
End matter		289
	Conclusion	289
	Review of key ideas	289
	Review of practical skills	290
	Further reading	290
A Answers and solutions		291
B Notation		299
	Math notation	299
	Calculus notation	304
	Data notation	305
	Probability notation	306
	Probability distributions reference	307
C Python tutorial		309
D Pandas tutorial		357
E Seaborn tutorial		413
F Calculus tutorial		419
Bibliography		421

Statistics prerequisites concept map

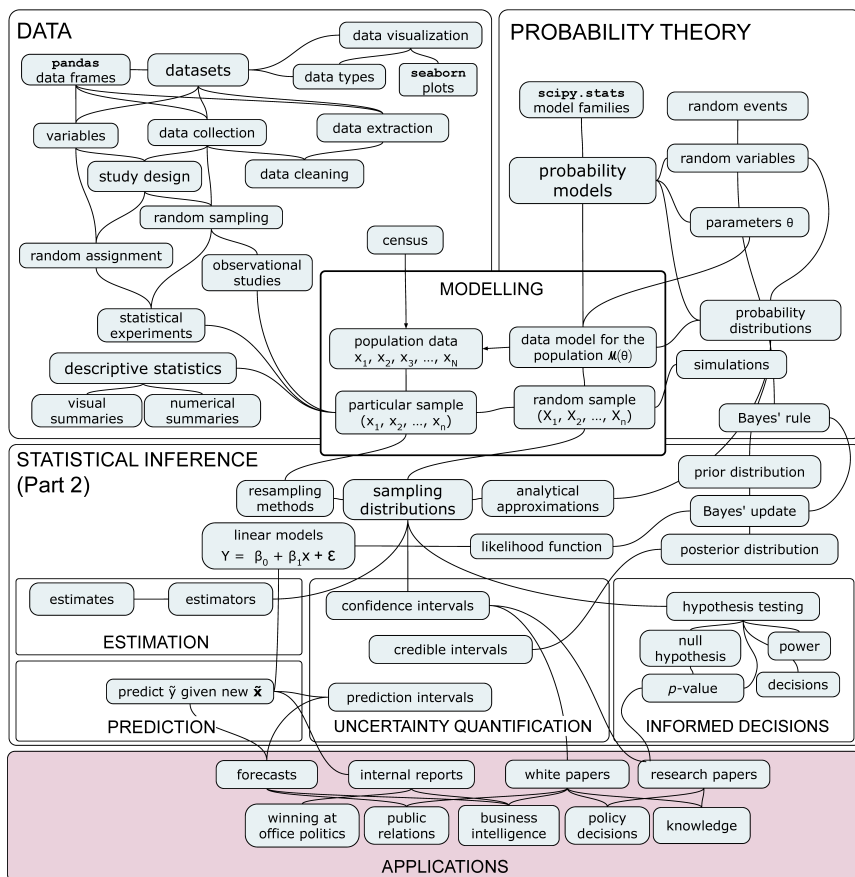


Figure 1: This concept map provides an overview of the topics and concepts that you'll learn in this book. The goal of Part 1 of the book is to establish a solid foundation of knowledge about DATA (Chapter 1) and PROBABILITY (Chapter 2) that will enable you to understand the statistical inference topics in Part 2 of the book.

Preface

This book covers the standard topics for a first-year university-level statistics course, as well as modern statistics topics like resampling methods and Bayesian inference. The text is written in a style that is conversational, jargon-free, and to the point. Imagine asking a friend who is very knowledgeable about data, probability, and statistics to explain the essential ideas of these subjects to you. Your friend would probably break down the subjects into easy-to-understand concepts, and explain what each concept is about concisely, then show you a bunch of practical examples to illustrate what the new concepts allow you to do. The goal of this book is to be like that friend, helping you to understand the core ideas of statistics and learn the practical skills you need to apply statistics in real-world situations.

Is this book for you?

This textbook is written with students and adult learners in mind. Students taking a statistics class will benefit from the in-depth explanations of statistical concepts that this book provides and the numerous practice exercises. Working professionals will benefit from the hands-on examples of real-world data analyses that illustrate how to use statistics in a business context. More generally, everyone can benefit from learning some stats to better make sense of the results reported in news articles and research papers.

Why this book?

Most university-level statistics textbooks use an outdated approach to teaching statistics based on pre-packaged recipes for statistical analysis. Students learn about individual recipes in a disconnected manner, without truly understanding how statistical procedures work in general or the assumptions behind them. Traditional textbooks rarely discuss practical aspects of data management like cleaning up “messy” datasets. This sucks all around: the traditional

approach leaves students with no understanding of statistical theory and no practical data analysis skills.

This book takes a different approach. Instead of presenting individual recipes, the book aims to explain general principles of statistical inference without skipping any steps. Readers learn practical topics like data management and theoretical topics like probability modelling, which together form the foundation needed to understand statistics.

You can think of learning statistics as a bike ride in a mountainous region. The pre-packaged recipes for statistical analysis are like shortcuts that let you climb quickly, but don't develop your muscles sufficiently to continue your journey after the first few hills. This book takes a slow-and-steady approach that builds your muscles over time, which then makes the climb up the statistics mountains much smoother.

A climb in two parts

This book has two parts: Part 1 covers prerequisites, while Part 2 is about statistical inference. The goal of the first part is to develop your practical data management skills and your conceptual understanding of probability theory. Chapter 1 provides an introduction to datasets, sampling procedures, and descriptive statistics. Chapter 2 is an introduction to probability models, random variables, and probability distributions. These are the key prerequisites that form the foundation for the statistics topics covered in Part 2 of the book. Think of Part 1 as the warm-up phase of the climb where you build your muscles before tackling the later, steeper climbs in the mountains.

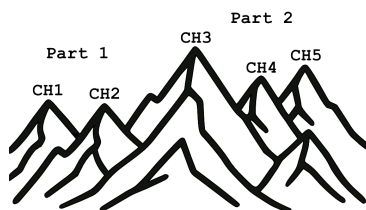


Figure 2: Bike trip in the statistics mountains. Part 1 is the warm-up where we learn about data and probability theory, which are the practical skills required to handle the statistical inference topics in Part 2.

Part 2 of the book will introduce the key ideas of statistical inference, including classical frequentist statistics (Chapter 3), linear models (Chapter 4), and Bayesian statistics (Chapter 5). This selection of topics is broader than the standard undergraduate statis-

tics curriculum, and includes topics from graduate-level statistics courses. The goal of this “enriched” curriculum is to cover topics and procedures that are useful in the real world beyond the classroom.

Consult the concept map in Figure 1 (page iv) and the table of contents for an overview of the topics covered in the book.

A modern approach to statistics

This book uses every available tool in the educator’s toolbox to help readers better understand probability and statistics. The explanations provided include a combination of text, graphics, formulas, code, and computer simulations to make statistics concepts more accessible and intuitive. We’ll use **Python** for the code examples and **JupyterLab** as the computational platform. Taking a computational approach to probability calculations will enable us to cover advanced probability concepts instead of being limited to simple formulas. We provide **computational notebooks** to accompany each section of the text. Playing with these notebooks will allow readers to reproduce all the presented results themselves, and to try variations of the calculations with different parameters. Each chapter includes **exercises and problems** for readers to practice the new material. The exercises are simple and quick to solve, while the problems are more challenging and aim to integrate the new concepts covered in each chapter. The datasets and computational notebooks from the book are available through the book’s website, noBSstats.com.

About the author

I completed my undergraduate studies in electrical engineering at McGill University, then I stayed on for an M.Sc. in physics and a Ph.D. in computer science. After leaving academia, I worked in technical roles including sysadmin, web developer, and software architect, which taught me a lot about using data for business decisions. I have 17 years of teaching experience as a math tutor and textbook author.

During the last seven years, I’ve read hundreds of research papers and textbooks on statistics, watched countless talks and video lectures, and consulted with stats experts to figure out the best way to present the core ideas of statistics. This book contains the distillation of everything I know about probability and statistical inference. I hope it helps you learn statistical thinking and allows you to apply statistics to the problems you’re interested in.

Ivan Savov

October 2025, Montréal

Introduction

Understanding statistics is of strategic importance in the modern world as it allows us to make sense of data: scientific data, business data, survey responses, personal data, health data, activity logs, and so much more. In the 21st century, statistics is no longer a “specialty” topic, important only for academics and researchers, but increasingly relevant for businesspeople, technologists, and the general public.

Statistics has applications in many domains: education, health-care, sports, business (sales, finance, advertising, marketing, quality control, insurance), economics, agriculture, engineering, and science. Every domain of research that uses the scientific method depends on the use of statistics. Statistical procedures allow us to reach conclusions that are justifiable and capable of convincing others, even skeptics. Anyone who wants to use data to make informed decisions needs to know statistics. Understanding the language of statistics also allows us to critically evaluate other people’s statistical arguments and interpret the results that appear in business reports and scientific papers.

What is statistics?

Statistics is a branch of mathematics that is concerned with collecting, analyzing, and learning from data, based on various principles, techniques, and procedures. Statistics applications include summarizing datasets, estimating unknown quantities, making decisions, and predicting future observations.

The word *statistics* is derived from the Italian word *stato* (state). The first uses of statistics were to collect data about the state: births, deaths, marriages, and other numbers that are useful when making political and military decisions. Over time, the term statistics evolved in meaning to apply not just to human populations, but generally to any type of data: biological, agricultural, scientific, business, etc. We can subdivide statistics into two parts.

- *Descriptive statistics* is concerned with summarizing and visu-

alizing data. For example, we can use the class average as a summary of the grades of all students in a class.

- *Inferential statistics* aims to learn the properties of a population based on a sample taken from that population. For example, we can use the grades of a sample of students from different schools to estimate the grades of all students in the city.

Descriptive statistics (see Section 1.3, which is 40 pages long) is much simpler than inferential statistics (see Part 2 of the book, which is 655 pages long). The complexity in inferential statistics comes from the fact that we're trying to *generalize* from the properties of a sample to the properties of the unknown population, which requires special techniques and assumptions.

Plan of attack

Statistics has a reputation for being notoriously confusing and difficult to understand, but that's mostly because people approach the subject without the necessary foundation. The good news is that readers who come to statistics with the appropriate prerequisite knowledge will actually have a good time! Yes, there will still be some complex procedures, math formulas, and various rules of thumb to learn, but all this complexity is manageable once you're familiar with the basic building blocks. The book is organized into five chapters split into two parts. The first two chapters (Part 1) build the foundation for the material covered in the latter three chapters (Part 2).

Overview of Part 1 of the book

We'll now provide an overview of the topics covered in each chapter, so you'll know what kind of trouble you're getting yourself into.

Chapter 1: Data We'll start by introducing data in Section 1.1, and define basic notions like *population*, *sample*, and the *random selection* procedure that we use to obtain representative samples. We'll also discuss the notion of *random assignment*, which is a key ingredient in statistical experiments. In Section 1.2, we'll switch to hands-on work: data manipulation using the Python library Pandas, and data visualization using the Seaborn library. We'll also introduce the datasets used in the statistical analyses throughout the book. In Section 1.3, we'll introduce the key ideas of *descriptive statistics*, which allow us to describe datasets using numerical and visual summaries.

By the end of the data chapter, you'll have the skills required to load datasets and "look at the data" using descriptive statistics (min, max, mean, variance, median) and visualizations (histograms, box plots, scatter plots, etc.), which are essential skills for data analysis.

Chapter 2: Probability theory The second prerequisite for statistics is to know about probability models, which are the workhorses of statistics. In Chapter 2, we'll explore probability distributions using math equations, graphs, and computer simulations. We'll take a deep dive into probability theory to ensure you understand all the math and become familiar with the most important discrete and continuous distributions (sections 2.3 and 2.6).

We take a computational approach to probability calculations in this book. We'll use the powerful Python libraries for numerical computing (NumPy) and scientific computing (SciPy) to build probability models, visualize probability distributions, and do practical calculations. For example, whenever we need to deal with scary-looking math like summations $\sum_{i=1}^n x_i$ and integrals $\int_{x=a}^{x=b} f(x) dx$, we'll compute the Python expressions `sum(xs)` and `quad(f, a, b)`, which provide the same answers.

Solid foundation for statistical inference The goal of the data and probability chapters (Part 1 of the book) is to build a solid foundation for the *statistical inference* topics in Part 2 of the book. You can think of probability theory as the engine that powers statistical inference, and data as the fuel. Your knowledge of probability distributions and your experience with data management tasks will allow you to understand statistics concepts and procedures at the necessary depth.

* * *

I'm not going to lie to you and say learning about data and probability is going to be easy. There will be some uphill, but you can totally handle it. I have done my best to make the uphill as smooth as possible, and you have Python on your side, so the chances are pretty good that you'll survive this journey. You have to trust me that understanding data and probability theory is a worthwhile investment. All the "brain sweat" you invest now will pay dividends when you get to the statistical inference topics in Part 2.

Are you ready for this? Put on your helmet and let's go!

Chapter 1

Data

Data is the fuel for statistics. The successful application of statistical procedures depends on the data collection and processing steps that precede them. Our ability to answer scientific and business questions from data depends on how the data-collection process was planned and executed, which determines whether we have the necessary type of data to answer the questions we're interested in.

Understanding data is an essential prerequisite for applying statistics in real-world scenarios. This chapter will beef up your knowledge of data collection (Section 1.1), data processing and visualization (Section 1.2), and data summarization (Section 1.3).

In this chapter, I'm going to show you how to ...

- **classify** the different types of data (numerical vs. categorical)
- **recognize** the importance of random sampling and random assignment
- **load** datasets stored in comma-separated values (CSV) files
- **compute** numerical data summaries (descriptive statistics)
- **generate** strip plots, histograms, box plots, and other data visualizations

1.1 Introduction to data

In order to do statistical analysis, we need to have data to analyze. This is why we'll start our journey into statistics by introducing the core concepts of data collection. The more you know about data collection strategies, the better you'll be equipped to apply the statistical analysis techniques that we'll learn in later chapters.

We use statistics to answer questions about real-world phenomena we're interested in. We assume it is possible to collect relevant data through observations and measurements of the quantities of

interest. Broadly speaking, the purpose of statistical analysis is to detect and measure the existence of some “pattern” in the data. Statistical analysis can be used to confirm that a predicted pattern exists, to estimate an unknown quantity, or to detect when an unexpected pattern occurs.

In this section, we’ll talk about the central importance of data for all of statistical practice. We’ll start by introducing the basic definitions and terminology used to describe datasets. We’ll then discuss the *random sampling* and *random assignment* techniques used as part of the data collection process.

1.1.1 Definitions

Let’s look at the technical terms we use when talking about data.

Datasets

We refer to the data used in a statistical analysis as a *dataset*. In this book, we’ll focus on *tabular data*, which is the most widely used kind of data in statistics. We use the following terminology to describe tabular datasets.

- *data table* or *data frame*: describes data stored in tabular format, like a spreadsheet with rows and columns. A *dataset* consists of one or more data tables.
- *variable*: a characteristic of the individual, item, or event that is measured in the data. Variables are also sometimes called *features* or *attributes*. For example, in a health study, the height and weight of individuals would be two variables that are measured. Variables usually correspond to the different columns of the data table.
- *observation*: the measurements for a single individual, item, or event. Observations are also sometimes called *cases* or *observational units*. For example, the measurements collected for one individual in a health study (name, age, height, weight, treatment, outcome, etc.) correspond to one observation. Observations usually correspond to the rows of the data table.

Table 1.1 shows an example data table that contains 12 observations of seven variables.

In addition to the data, a dataset contains *metadata* (data about the data), which provides additional “context” information, including *when*, *how*, and *why* the data was collected. Metadata usually includes a *codebook* that describes each of the variables, and specifies

index	username	country	variable				
			age	ezlvl	time	points	finished
0	mary	us	38	0	124.94	418	0
1	jane	ca	21	0	331.64	1149	1
2	emil	fr	52	1	324.61	1321	1
3	ivan	ca	50	1	39.51	226	0
4	hasan	tr	26	1	253.19	815	0
5	jordan	us	45	0	28.49	206	0
6	sanjay	ca	27	1	585.88	2344	1
7	lena	uk	23	0	408.76	1745	1
8	shuo	cn	24	1	194.77	1043	0
9	r0byn	us	59	0	255.55	1102	0
10	anna	pl	18	0	303.66	1209	1
11	joro	bg	22	1	381.97	1491	1

Table 1.1: A data table that contains observations of seven variables for 12 players of a computer game. Each row in this table corresponds to one player. Each column corresponds to one characteristic that was measured for all the players.

the units of measurement. Detailed and complete metadata is essential for correct interpretation of the data.

Example: the players dataset Let's illustrate the new terminology by looking at an example dataset of player profiles from a computer game. The players dataset is shown in Table 1.1. This dataset was collected as part of a statistical experiment whose goal was to test if making the first level of the computer game easier will increase the time players spend in the game. Half the players were presented a special version of the game with an easy first level ($ezlvl=1$), while the other half played the normal version of the game ($ezlvl=0$). We want to know if the easy level made players spend more time in the game.

The players dataset contains observations of seven variables for 12 different players. The leftmost column is called the *index* and is equivalent to the row numbers in a spreadsheet. The first row of the table is called the *header* and contains the variable names.

Each row of the dataset shown in Table 1.1 corresponds to one observation (the data for one player). We have recorded the following characteristics for each player: *username*, *country*, *age*, *ezlvl*, *time*, *points*, and *finished*. For example, the player with username *sanjay*, is a 27-year-old Canadian (*ca*), who spent 585.88 minutes playing the game, earned 2344 points, and finished the game (*finished*=1). The value 1 for the *ezlvl* variable tells us that Sanjay played the game version with the easy first level.

Each column of the data table contains the values of one of the variables that we measured for all players. You can also think of

each variable as a list of 12 values. For example, the variable `age` is a list of 12 values [38, 21, 52, 50, 26, 45, 27, 23, 24, 59, 18, 22], where each value corresponds to the age of one of the players. We can analyze each of the variables on its own (e.g., compute the average age of the players), or look for relations between variables (e.g., how does the `ezlvl` variable influence the `time` variable).

Variable types

We make a distinction between *numerical* and *categorical* variables:

- *numerical variables* correspond to quantitative measurements recorded as numbers. Numerical variables can be integers (e.g. `age`) or decimal numbers (e.g. `time`).
- *categorical variables* are labels that take on one of a discrete set of possible values, like the answers to true or false questions, the presence or absence of some characteristic (1 or 0), blood group types (A, B, AB, O), or a person's country of residence. The variables `username`, `country`, `ezlvl`, and `finished` in the players dataset are all categorical variables.

The statistical operations we can perform on numerical and categorical variables are completely different. Numerical variables can be manipulated using arithmetic operations like sums, differences, and products, while categorical variables can only be used for grouping and counting observations.

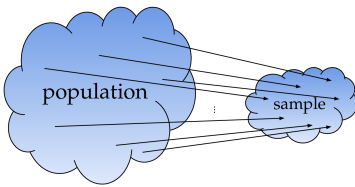
Note that categorical variables are sometimes represented using numbers, such as the values 1 and 0. For example, the variable `finished` in the players dataset (see Table 1.1) contains 1 for players who finished the game, and 0 for players who didn't finish the game.

Populations and samples

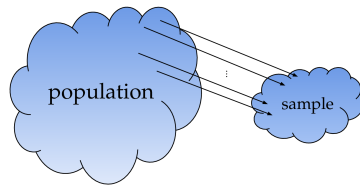
The *population* is the group of interest for the statistical analysis. This term can refer to people (players, students, patients, clients, website visitors, etc.) but also to groups of animals, insects, objects, or events.

- *population*: all the items or individuals in the group of interest. We'll denote the population size (the number of individuals in the population) using uppercase N , which can be in the tens, hundreds, thousands, millions, or billions. Often the population size is unknown.
- *census*: the process of collecting data for the entire population. This kind of exhaustive data collection is usually very costly to perform, so instead, most statistical analyses are performed on a subset of the population called a *sample*.

- *sample*: a subset of the population that has been measured for statistical analysis. We'll denote the sample size as lowercase n . Usually, n is much smaller than the population size N .
- *representative sample*: a sample that has the same characteristics as the population. See Figure 1.1 (a). For example, if the population contains a mix of people in different age groups, the people included in a representative sample must contain a similar mix of different age groups.
- *biased sample*: samples that are not representative of the population are called *biased*. See Figure 1.1 (b). For example, a sample that contains only young people is not representative of the general population.
- *random sample*: a sample selected from the population in such a way that each individual has an equal chance of being included in the sample. Using random sampling is one way to obtain representative samples.



(a) Representative sample selection



(b) Biased sample selection

Figure 1.1: A *sample* is a subset of the *population* selected for performing statistical analysis. If the sample is representative of the population as in (a), the results of the statistical analysis performed on the sample will also apply to the population as a whole. If the sample is biased as in (b), the results of the statistical analysis will not generalize to the population as a whole.

Variable names used in statistical analyses

The purpose of statistical analysis is to find patterns in the data, to extract scientific knowledge, and reach justified conclusions. Typically, we want to answer a question about how the values of one variable depend on the values of another variable.

In the context of the players dataset, an example of a statistical question we might want to answer is whether young players and old players spend different amounts of time in the game. What is the influence of the age variable on the time variable in the players dataset?

Statisticians use the following terms when referring to variables, depending on the role they play in the statistical analysis:

1.2 Data in practice

We're blessed to be living in the XXIst century when computational tools for data analysis are easily accessible. We don't have to memorize complicated formulas or perform tedious calculations using pen-and-paper, since we can use computational tools like Python, Pandas, and Seaborn to do statistical calculations. By learning a thing or two about the Pandas and Seaborn libraries (which is the goal of this section), you'll know about the best-in-class toolset for data management currently used by data scientists, statisticians, business analysts, and machine learning researchers.

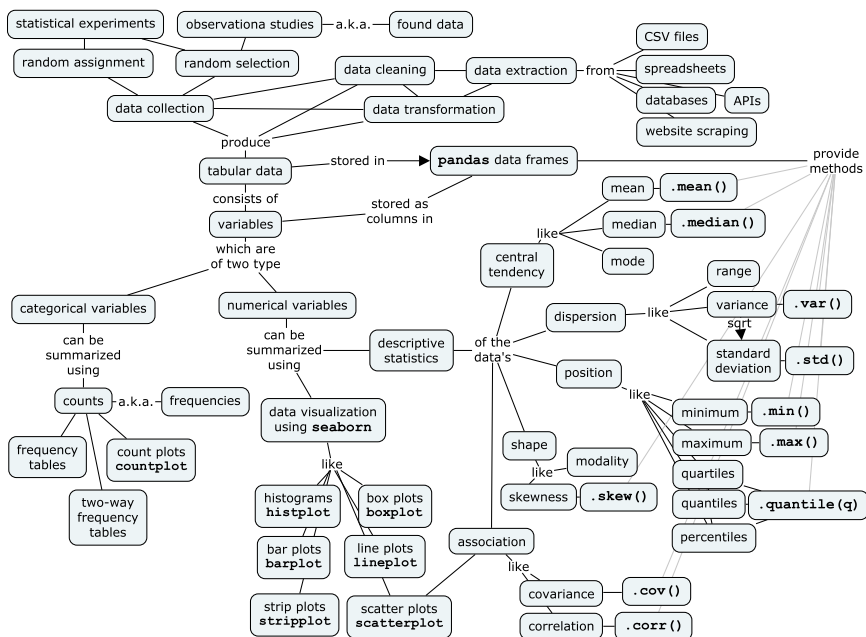



Figure 1.4: Concept map that illustrates the practical considerations you need to think about when collecting data for a statistical analysis. The bottom half of the concept map is a preview of the descriptive statistics concepts we'll learn in the next section (Section 1.3).

A common misconception about statistics is that it's someone else's job to collect data, and offer it to you in a well-organized, clean format ready for statistical analysis. This is far from the truth! In reality, statisticians and other data professionals spend a large proportion of their time collecting, preprocessing, and informally exploring datasets in preparation for doing actual statistical analyses. The topic of practical data management is usually omitted from introductory

statistics courses, because teachers think it would be too complicated for beginners to learn. I don't think so, and I plan to teach you the essential skills you need to work with realistic datasets. Specifically, I'm going to show you how to use the JupyterLab computational environment, the Pandas library for data manipulation, and the Seaborn library for generating statistical visualizations. Figure 1.4 shows an overview of the concepts we'll discuss in this section.

This is going to be a hands-on, try-things-for-yourself section and not a passive reading section. The main goal of this section is to ensure you have a working computational environment on your computer (JupyterLab Desktop), and know how to use the Pandas and Seaborn libraries for basic data analysis tasks. The secondary goal of this section is to introduce the datasets that we'll use in the remainder of the book. The two goals combine synergistically, since we need examples of datasets to showcase the power of the Pandas and Seaborn functionality.

1.2.1 Getting started with JupyterLab

We'll start by setting up a statistical computing environment (JupyterLab Desktop) on your computer, which will allow you to run computational notebooks. You can think of a computational notebook as a fancy calculator—you input Python commands (similar to the buttons on a calculator), then run the commands to see the result (similar to what happens when you press the  button on a calculator). Unlike calculators that have a limited number of operations (buttons), computational notebooks give you access to the entire Python programming language, and numerous powerful Python libraries for data management, data visualization, and statistical analysis.

Download and install JupyterLab

JupyterLab Desktop is a convenient all-in-one application that you can install on your computer to take advantage of everything Python has to offer for data analysis and statistics. Follow the instructions in Appendix C (see page 309) to download and install JupyterLab Desktop. If you're new to Python, I strongly recommend that you go through the entire Python tutorial in Appendix C before continuing with the rest of this section. I'm not expecting you to be a Python expert, but I want you to be comfortable with the basic Python commands used for calculating expressions, manipulating lists, and calling functions.

Download the notebooks and datasets for the book

I have prepared a collection of notebooks and datasets to accompany this book. You can view and download these notebooks and datasets from the book's website <https://noBSstats.com> or from this GitHub page: <https://github.com/minireference/noBSstats>. Instead of downloading notebooks and datasets one by one, I recommend that you download the entire repository as a ZIP archive using the steps illustrated in Figure 1.5. After downloading the ZIP archive, double-click on the file to extract its contents, and move the resulting folder `noBSstats` to a location on your computer where you normally keep your documents.

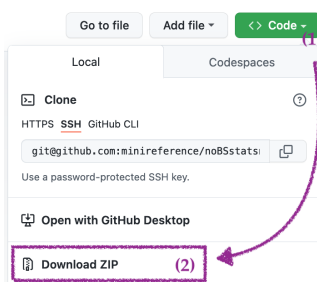


Figure 1.5: Illustration of the steps to download the contents of the entire `noBSstats` repository as a single ZIP archive. Use the **Code** drop-down (1) then select the **Download ZIP** option (2).

The ZIP archive includes all the datasets and computational notebooks for the book. Use the **File browser** pane in the JupyterLab to navigate to the location where you saved the `noBSstats` folder. Inside you should see subfolders called `datasets`, `notebooks`, `exercises`, `tutorials`, etc. Look around to get an idea of the files available in each subfolder.

Datasets used in the book

You can find all the datasets inside the `datasets` subfolder of the `noBSstats` folder. Use the JupyterLab file browser pane to view the contents of the `datasets` subfolder. For example, the `players` dataset is stored in the file `datasets/players.csv`.

Alternatively, you can download individual datasets directly from the book's website, under the `datasets` directory. For example, the data file `players.csv` can be downloaded from the URL <https://noBSstats.com/datasets/players.csv>, and similarly for the other datasets.

Later in this section, we'll provide more information about all the datasets in this folder and discuss the statistical questions we want to answer from each of them. Look ahead to Table 1.3 on page 42 if you're feeling impatient.

Interactive notebooks for each section

Each section of this book has a notebook companion that includes the code examples from the text. I expect you to play with these notebooks in parallel with reading the text, so that you'll get some hands-on experience of doing data calculations and generating data visualizations. The notebooks are located in the `notebooks` subfolder. For example, the notebook companion for this section is `notebooks/12_data_in_practice.ipynb`. I recommend that you open this notebook now in JupyterLab, so that you'll be ready to run the code examples you'll encounter later in this section.

Exercises notebooks

I've also prepared starter notebooks for the exercises in each section. The exercises notebooks contain partially-filled code cells for each exercise question. You can find the exercises notebooks in the `exercises` subfolder. For example, to try the exercises for this section, open the notebook `exercises_12_practical_data.ipynb` in the `exercises` folder and start filling in the missing parts in the code cells.

* * *

From here on, I'll assume you have JupyterLab Desktop installed on your computer and have downloaded all the datasets and notebooks for the book, so that you can follow the code examples interactively. Here are a few quick exercises you can try, to make sure you've got the basic setup working correctly.

E1.3 Create a new notebook called `MyCalculations.ipynb` and use code cell to compute the sum of 3456 and 789.

E1.4 Open the notebook `exercises_12_practical_data.ipynb` located in the `exercises` folder and repeat the calculation $3456+789$ in the code cell labelled E1.4.

1.2.2 Data management with Pandas

Pandas is a versatile toolbox for data management in Python. You can think of Pandas as a Swiss Army knife for working with data, since it includes *a lot* of functions for working with various types of data, performing data manipulations, and doing statistical calculations. Learning a bit of Pandas will allow you to work with real-world datasets of all shapes and sizes, so it is a generally useful skill to have if you plan to do anything data-related in the future.

The good news is that you don't need to learn all this functionality at once. Knowing just a few basic Pandas concepts and commands is enough to get you started. This subsection is a Pandas crash course that will introduce you to the two main data structures that the Pandas library provides: *data frame* objects for storing tabular data, and *series* objects for storing lists of values. We'll focus on the specific data manipulation tasks that you need to know to understand the examples in the book. For a more in-depth coverage of Pandas functionality, I'll refer you to the Pandas tutorial in Appendix D.

Okay, enough talk, let's get started! Open the notebook `12_data_in_practice.ipynb` in JupyterLab Desktop so you can run the commands in parallel and follow the explanations interactively. The first step is to import the `pandas` module, which is usually done in the beginning of the notebook. There is a widespread convention to import the `pandas` module under the short alias `pd`.

```
>>> import pandas as pd
```

code
1.2.1

This import statement makes all the Pandas functionality available under the name `pd`.

Loading datasets

The first step to any data analysis is to load the data we want to work on into a Pandas *data frame*. We'll illustrate the process by loading the data file `players.csv` located in the `datasets` directory, which is a sibling of the `notebooks` directory.

The file extension `.csv` tells us the file contains text data formatted as Comma-Separated Values (CSV). We can view the contents of the file `players.csv` using a text editor like Notepad.exe on Windows or TextEdit on macOS. The file contents are shown below.

```
username, country, age, ezlvl, time, points, finished
mary, us, 38, 0, 124.94, 418, 0
jane, ca, 21, 0, 331.64, 1149, 1
emil, fr, 52, 1, 324.61, 1321, 1
ivan, ca, 50, 1, 39.51, 226, 0
hasan, tr, 26, 1, 253.19, 815, 0
```

code
1.2.2

```
jordan,us,45,0,28.49,206,0
sanjay,ca,27,1,350.0,1401,1
lena,uk,23,0,408.76,1745,1
shuo,cn,24,1,194.77,1043,0
r0byn,us,59,0,255.55,1102,0
anna,pl,18,0,303.66,1209,1
joro,bg,22,1,381.97,1491,1
```

We see the data file `players.csv` consists of 13 lines of text, and each line contains—as promised by the `.csv` file extension—values separated by commas. The first line in the data file is called the “header” and contains the names of the variables.

The Pandas function for loading CSV files is `pd.read_csv(<path>)`, where `<path>` describes the location of the data file. The current notebook is located in the `notebooks` directory, which is a sibling to the `datasets` directory, so the *relative path* to the `players` dataset is `"../datasets/players.csv"`. In words, this path means: go up to the parent directory (the two dots), then go inside the `datasets` directory, and open the file named `players.csv`.

The code below shows how to use the function `pd.read_csv` to load the `players.csv` data into a Pandas data frame object called `players`. We choose a name for the data frame that matches the dataset name to remind us where the data came from. We then display the contents of the variable `players` by entering its name on a second line, relying on the default behaviour of the notebook environment to show the value of the last expression in a code cell.

```
code >>> players = pd.read_csv("../datasets/players.csv")
1.2.3 >>> players
      username country  age  ezlvl   time  points  finished
0         mary     us   38     0  124.94    418         0
1         jane     ca   21     0  331.64   1149         1
2         emil     fr   52     1  324.61  1321         1
3         ivan     ca   50     1   39.51    226         0
4        hasan     tr   26     1  253.19    815         0
5        jordan     us   45     0   28.49    206         0
6        sanjay     ca   27     1  585.88  2344         1
7         lena     uk   23     0  408.76  1745         1
8         shuo     cn   24     1  194.77  1043         0
9         r0byn     us   59     0  255.55  1102         0
10        anna     pl   18     0  303.66  1209         1
11        joro     bg   22     1  381.97  1491         1
```

Recall we’ve already seen the `players` dataset in the previous section (see Table 1.1 on page 7). The `players` dataset consists of $n = 12$ observations of players’ activity in a computer game. The variable `username` is a unique identifier for each player. The variables `country` and `age` provide some basic player demographics. The variable `ezlvl` indicates whether the player was part of the “easy level” experiment. The `time`, `points`, and `finished` variables describe the

Pandas exercises

I highly recommend you try these exercises, because the hands-on approach is the best way to learn to use Pandas.

E1.5 Open the file `players.csv` using LibreOffice or another spreadsheet program. Compute the mean and the standard deviation of the age variable using spreadsheet functions.

Hint: Create new cells containing formulas based on the spreadsheet functions `AVERAGE(...)` and `STDEV(...)`.

E1.6 Compute the mean of the `points` variable in the `players` dataset.

E1.7 Try loading a few of the other datasets into a data frame using the function `pd.read_csv()`, then use the `.head()` method to print the first few rows of each dataset, and `.shape` to display the number of rows and columns.

E1.8 Load dataset `students.csv` and compute the mean of the variable `score`.

E1.9 Create a new notebook cell and use the command `?ages.sum` to display the Pandas help menu for the `.sum` method, which describes all the options you can use when calling the method, and often includes usage examples. Using the question mark prefix `?ages.sum` is a shortcut for calling `help(ages.sum)`. Try using the `?`-prefix to view the help menus of the other methods we used in this section: `ages.count`, `ages.mean`, `ages.std`, etc.

1.2.3 Data visualizations with Seaborn

Seaborn is a popular Python library for statistical data visualization. Seaborn provides functions for generating *strip plots*, *scatter plots*, *box plots*, *histograms*, and other statistical plots for data stored in Pandas series and data frames. In this section, we'll look at some examples of statistical visualizations of the `players` dataset to give you a taste of the type of plots we can generate using the Seaborn library.

To use Seaborn, the first step is to import the `seaborn` module in the current notebook.

```
code >>> import seaborn as sns
1.2.25
```

Importing `seaborn` under the alias `sns` is a widespread convention, similar to the convention of importing `pandas` under the alias `pd`.

Strip plot of the time variable

A *strip plot* is a statistical visualization for numerical variables where each observation is represented as a point. The Seaborn function

for drawing strip plots is `stripplot`. We'll now use this function to generate a strip plot of the time variable in the players data frame.

To generate a strip plot, we pass the data frame `players` as the data argument to the Seaborn function `sns.stripplot`, and specify the column name "time" (in quotes) as the `x` argument.

```
>>> sns.stripplot(data=players, x="time")
```

The result is shown in Figure 1.7.

code
1.2.26

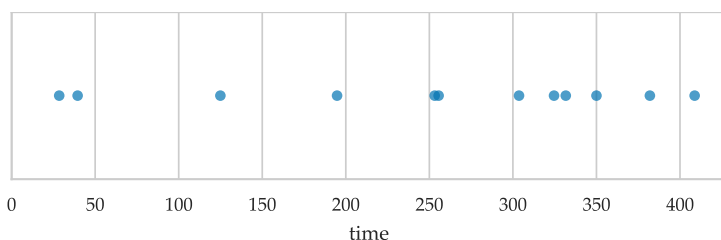


Figure 1.7: Strip plot of the time variable from the players dataset. A strip plot is a one-dimensional plot where each observation is mapped to a point at the location that corresponds to its value.

The first argument, `data=players`, tells Seaborn to take the data from the `players` data frame. The second argument, `x="time"`, indicates we want to represent the time variable on the x -axis. This is the general pattern for calling all Seaborn plot functions: we describe where the data lives and the properties of the plot we want to see, and the Seaborn plot function takes care of all the rest. In this case, the function `stripplot` extracted the data from the "time" column of the `players` data frame, automatically chose the limits of the x -axis so the data will fit, and set the x -axis title based on the variable name.

Seaborn makes it easy to map multiple variables to different visual properties (aesthetics) of the plot. For example, we can enhance the strip plot by mapping the `ezlvl` variable to the colour (hue) of the points in the plot.

```
>>> sns.stripplot(data=players, x="time", hue="ezlvl")
```

Result is shown in Figure 1.8.

code
1.2.27

The addition of the argument `hue="ezlvl"` tells Seaborn to choose the colour of the points based on the `ezlvl` categorical variable.

Studying the effect of `ezlvl` on time

Recall the `players` dataset was collected as part of an experiment designed to answer the question "Does the easy first level lead to improved user retention?" We want to compare the time variable (total time players spent in the game) of players who were shown

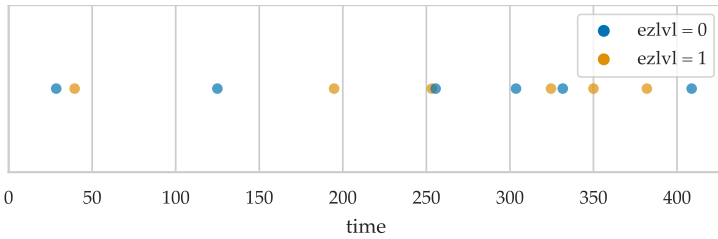


Figure 1.8: Strip plot of the time variable, where the colour of each point is determined by the `ezlvl` variable in the players dataset.

the “easy level” version of the game (`ezlvl=1`) to the control group of players who played the regular version of the game (`ezlvl=0`).

Figure 1.9 shows a strip plot that can help us visualize the time variable for the two groups of players. The code we used to generate this figure is as follows.

```
code >>> sns.stripplot(data=players, x="time", y="ezlvl",
1.2.28                      hue="ezlvl", orient="h", legend=None)
Result is shown in Figure 1.9.
```

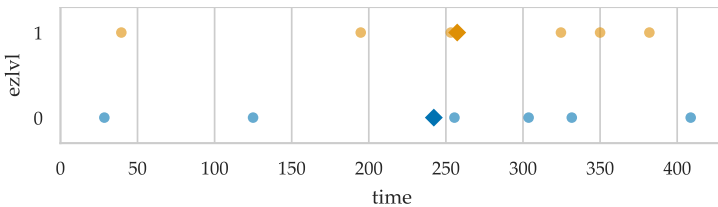


Figure 1.9: Comparison of the time variable in the players data grouped by `ezlvl`. The diamond shapes represent the means of the two groups.

Note we were able to customize the plot by passing different arguments and options to the function `sns.stripplot`. The arguments `data=players` and `x="time"` are the same as what we saw earlier. Next we tell Seaborn to use the `ezlvl` variable for the y-position and the hue of the points in the plot. The options `orient="h"` (horizontal layout) and `legend=None` (remove legend) perform additional visual customizations of the plot.



The strip plot in Figure 1.9 includes additional diamond annotations that correspond to the means of the two groups, which we computed using the following Pandas expressions:

```
code >>> players[players["ezlvl"]==0]["time"].mean()
1.2.29 242.17333333333332
>>> players[players["ezlvl"]==1]["time"].mean()
257.34166666666664
```

1.3 Descriptive statistics

The goal of descriptive statistics is to characterize the essential properties of a dataset. We use numerical and graphical summaries to describe important aspects of datasets. Computing descriptive statistics is an essential first step in any data analysis, and a fundamental skill that you'll need throughout the book.

We can obtain a condensed summary of the data by calculating certain representative values called *summary statistics*. A summary statistic is a numerical value computed from the data that succinctly describes a particular characteristic of the data, like the minimum, maximum, or the average. We'll use the methods of the Pandas library to compute summary statistics for data stored in Pandas series and data frames.

We can also get an overall impression of any dataset by making a *visual summary*: a plot that shows the characteristics of the data. We briefly introduced strip plots and scatter plots in the previous section. In this section we'll revisit these types of plots, and show other statistical visualizations that we can create using the Seaborn library, like bar plots  and box plots . By mapping characteristics of data onto visual elements of a graph, we can get a quick overview of the dataset. The human visual cortex is surprisingly efficient at spotting patterns and trends in data presented graphically, so it's worth learning how to create visual summaries to take advantage of your innate pattern-spotting abilities.

In this section, we'll introduce the fundamentals of descriptive statistics, including definitions and general principles, and provide illustrative examples based on the Pandas and Seaborn libraries. The descriptive statistics and data visualizations for numerical and categorical variables are very different, so we'll discuss them separately, starting with numerical variables first.

1.3.1 Numerical variables

Numerical variables describe quantities like weight, length, temperature, and time. The values of numerical variables can be compared, sorted, added and subtracted, and used in other math operations.

Definitions and formulas

Let's start by defining the new terms and showing the formulas for calculating summary statistics. We'll state the definitions in terms of a generic sample of size n , denoted $\mathbf{x} = [x_1, x_2, x_3, \dots, x_n]$. You can think of \mathbf{x} as the measurements of the variable x collected from

n individuals. Note the convention to use the boldface symbol \mathbf{x} to denote the sample as a whole.

Measures of central tendency We often want to summarize the sample $\mathbf{x} = [x_1, x_2, x_3, \dots, x_n]$ using a single number that is representative of the values in the sample. One way to choose such a representative number is to find the “middle” of the distribution of the values in the sample. There are several different ways to describe the middle of a list of numerical values, which we’ll now discuss.

- **mean:** the *arithmetic mean* or *average* of the sample is computed by taking the sum of all the values divided by the sample size:

$$\bar{x} = \mathbf{mean}(\mathbf{x}) = \frac{x_1 + x_2 + x_3 + \dots + x_n}{n} = \frac{1}{n} \sum_{i=1}^n x_i.$$

Note the shorthand notation for the mean uses a bar on top of the variable name. The symbol \sum (capital Greek letter *sigma*) stands for summation, and the math expression $\sum_{i=1}^n x_i$ means “sum of all the values x_i from x_1 until x_n .”

- **med:** the *median* is defined as the middle of the sample, when the values appear in sorted order. Half the values in the sample are smaller than the median $\mathbf{med}(\mathbf{x})$, and half the values are larger than $\mathbf{med}(\mathbf{x})$, as shown in Figure 1.15.

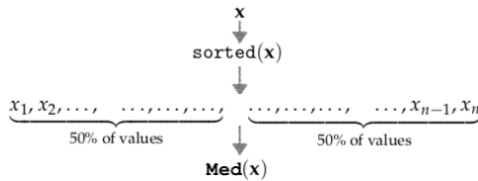


Figure 1.15: Illustration of the median value, $\mathbf{med}(\mathbf{x})$, which splits the sample into two equal parts. Half the values in the sample \mathbf{x} are smaller than the median, and the other half are larger than the median.

If the sample \mathbf{x} contains an odd number of values, the median is the middle value in the sorted list. If the sample \mathbf{x} contains an even number of values, the median is defined as the average of the two middle values: $\mathbf{med}(\mathbf{x}) = \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$.

- **mode:** the *mode* is the most frequently observed value in the data. A variable can have no mode when no single value appears more often than any other, or it can have more than one mode when there is a “tie” for the most common value.

Consider, for example, the sample $\mathbf{x} = [1, 1, 2, 3, 93]$ of size $n = 5$. The mean of \mathbf{x} is $\bar{\mathbf{x}} = \mathbf{mean}(\mathbf{x}) = \frac{1}{5}(1 + 1 + 2 + 3 + 93) = \frac{100}{5} = 20$, the median is $\mathbf{med}(\mathbf{x}) = 2$, and the mode is $\mathbf{mode}(\mathbf{x}) = 1$.

Measures of position We often want to describe the position of particular values within the sample \mathbf{x} , when it appears in sorted order. The median $\mathbf{med}(\mathbf{x})$ describes the middle of the sample. In addition to the median, there are several other useful statistics for describing values at specific positions within the sample.

- **min**: the *minimum* is the smallest value in the data.
- **max**: the *maximum* is the largest value in the data.
- **Q_1, Q_2, Q_3** : the three *quartiles* divide the data into four equal parts, which is similar to how the median $\mathbf{med}(\mathbf{x})$ divides the data into two equal parts. You can think of $Q_1(\mathbf{x})$, $Q_2(\mathbf{x})$, and $Q_3(\mathbf{x})$ as “fence posts” that divide the data into four quarters, as illustrated in Figure 1.16. Note $Q_2(\mathbf{x})$ is the same as $\mathbf{med}(\mathbf{x})$.

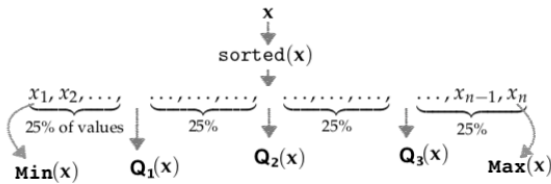


Figure 1.16: Illustration of the four quartiles $Q_1(\mathbf{x})$, $Q_2(\mathbf{x})$, and $Q_3(\mathbf{x})$ that split the sorted data into four equal parts. Note $Q_2(\mathbf{x}) = \mathbf{med}(\mathbf{x})$.

- **Percentiles**: the percentiles are similar to the quartiles, but divide the data into 100 equal parts instead of four parts. For example, the 95th percentile is denoted $P_{95}(\mathbf{x})$ and describes a value that is greater than 95% of the values in \mathbf{x} .
- **Quantiles**: the q^{th} quantile splits the data into two parts: a fraction q of the data is smaller, and the remaining fraction $1 - q$ of the data is larger. Quantiles are similar to percentiles, but are defined in terms of a fraction q between 0 and 1, while percentiles use a percentage value between 0 and 100.

The three measures of position, quartiles, percentiles, and quantiles, all provide the same information but use different units. Quartiles describe the data split into four chunks, percentiles use 100 chunks, while quantiles use a continuous quantity between 0 and 1. For example, the first quartile $Q_1(\mathbf{x})$, is the same as the 25th percentile, which is the same as the $q = 0.25$ quantile. The second quartile $Q_2(\mathbf{x}) = \mathbf{med}(\mathbf{x})$ is equivalent to the 50th percentile and the $q = 0.5$

quantile. The third quartile $Q_3(\mathbf{x})$ is equal to the 75th percentile and the 0.75th quantile.

Taken together, the five numbers $\min(\mathbf{x})$, $Q_1(\mathbf{x})$, $Q_2(\mathbf{x})$, $Q_3(\mathbf{x})$, and $\max(\mathbf{x})$ are called the *five-number summary* of the data, which tells us the boundaries of four regions that each contain 25% of the data when it appears in sorted order.

Measures of dispersion Another important characteristic of any sample is how “spread out” it is, which we call the *dispersion* of the data. There are several common measures for quantifying the dispersion of the sample \mathbf{x} .

- **range:** the *range* is the difference between the maximum and minimum values, $\text{range}(\mathbf{x}) = \max(\mathbf{x}) - \min(\mathbf{x})$.
- **IQR:** the *interquartile range* is defined as the distance between the first and third quartiles, $\text{IQR}(\mathbf{x}) = Q_3(\mathbf{x}) - Q_1(\mathbf{x})$, and tells us the width of the middle fifty percent of the data.
- **var:** the sample *variance* is computed from the sum of the squared differences from the mean divided by $n - 1$:

$$\text{var}(\mathbf{x}) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2.$$

We use the shorthand notation s_x^2 to describe the variance. Note the formula contains a division by $(n - 1)$ instead of n , which is called *Bessel’s correction*. We’ll explain the reason for using Bessel’s correction in Section 3.1 where we’ll learn how to use the sample variance to estimate the variance of the wider population from which the sample was taken.

- **std:** the *standard deviation* is the square root of the variance:

$$\text{std}(\mathbf{x}) = \sqrt{\text{var}(\mathbf{x})} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}.$$

We use the shorthand notation s_x for the standard deviation.

The variance and the standard deviation are used often in statistics formulas and procedures, this is why statisticians use the shorthand notation s_x^2 and s_x for these quantities. The variance is the square of the standard deviation, so they essentially measure the same thing. We usually show standard deviation when reporting results because standard deviation is measured in the same units as the data, unlike the variance, which is measured in squared units.

Knowing the mean \bar{x} and the standard deviation s_x of the sample x is a very good way to summarize its distribution. The mean tells us where the centre of the distribution is, while the standard deviation tells us how tightly or loosely dispersed the data is around the mean. Many values will fall within the interval $[\bar{x} - s_x, \bar{x} + s_x]$, which describes one standard deviation around the mean. We sometimes write this interval as $\bar{x} \pm s_x$. See Figure ?? for an illustration.

Pandas methods When the sample x is stored in a Pandas series object or a column in a Pandas data frame, we can compute all descriptive statistics by calling the appropriate method on the Pandas object. Table 1.4 shows the Pandas methods for computing all the descriptive statistics for numerical variables that we defined in this section. For example, if the sample x is stored as a Pandas series xs , we can compute its mean by calling `xs.mean()`.

Statistic	Name	Pandas method
n	sample size	<code>.count()</code>
$\bar{x} = \text{mean}(x)$	mean	<code>.mean()</code>
$\text{med}(x)$	median	<code>.median()</code>
$s_x^2 = \text{var}(x)$	variance	<code>.var()</code>
$s_x = \text{std}(x)$	standard deviation	<code>.std()</code>
$\text{min}(x)$	minimum	<code>.min()</code>
$Q_1(x)$	first quartile	<code>.quantile(0.25)</code>
$Q_2(x) = \text{med}(x)$	second quartile	<code>.quantile(0.50)</code>
$Q_3(x)$	third quartile	<code>.quantile(0.75)</code>
$P_{90}(x)$	90 th percentile	<code>.quantile(0.90)</code>
$\text{max}(x)$	maximum	<code>.max()</code>

Table 1.4: Summary of descriptive statistics for numerical variables and the Pandas methods for computing them. The same Pandas methods are available on both series and data frame objects.

The Pandas method `.quantile(q)` computes the q^{th} quantile, where q takes on values between 0 and 1. We use the `quantile` method to compute quartiles and percentiles, as shown in Table 1.4. In fact, the minimum and maximum values can also be computed using the `quantile` method: the minimum corresponds to `.quantile(q=0)`, while the maximum is `.quantile(q=1)`.

Descriptive statistics of the students dataset

Enough with the definitions and formulas! Let's look at a hands-on example that illustrates how to compute summary statistics and

bin	values	frequency
[50, 60)	57.0, 57.6	2
[60, 70)	67.0, 69.0, 62.9	3
[70, 80)	75.0, 75.0, 70.3, 76.1, 79.8, 72.7, 75.4, 70.4	8
[80, 90)	84.3	1
[90, 100]	96.2	1

Table 1.6: One-way table of the scores data grouped into bins of width 10. We label the bins using *interval notation*: the interval $[a, b)$ describes the range of numbers starting from a (included) until b (not included).

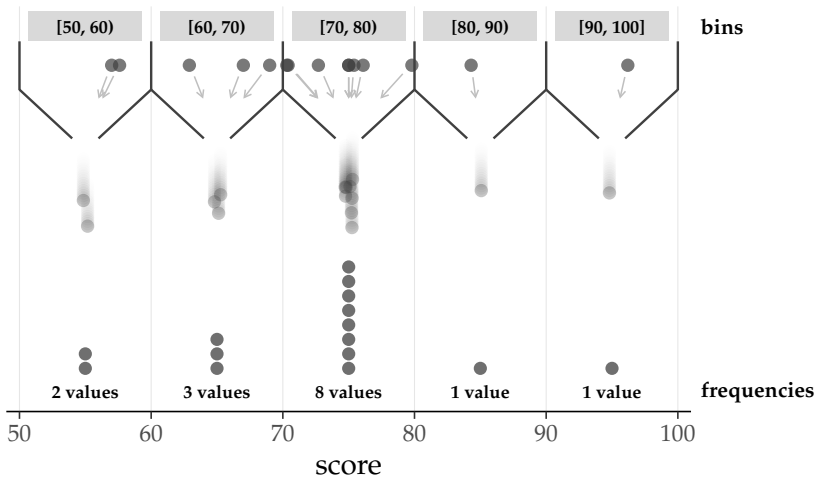


Figure 1.20: Visual representation of the histogram binning process.

The Seaborn function for producing the histogram in Figure 1.21 is `histplot`. The following code shows how to use `sns.histplot`.

```
>>> bins = [50, 60, 70, 80, 90, 100]
>>> sns.histplot(data=students, x="score", bins=bins)
See Figure 1.21.
```

code
1.3.10

In the above code, we call the function `histplot` specifying the data to use for the histogram is in the `students` data frame, and the argument `x="score"` indicates the name of the variable that we're interested in. We manually created a list of values that we want to use as the bin's boundaries in the histogram, then passed this list as the `bins` option when calling the `histplot` function.

The histogram in Figure 1.21 gives us a convenient summary of the students' scores data. We can see how many data points fall within each bin. The bin with the highest frequency is called the

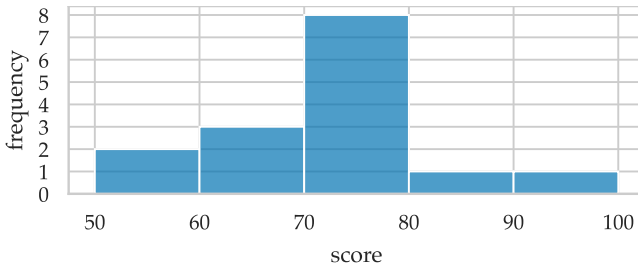


Figure 1.21: Histogram of the score variable from the students data frame. The width of each bar covers an interval of values called a *bin*. The heights of the bars are proportional to the number of observations within each bin. Bins are usually (but not always) of equal width, with no gaps between them.

mode of the histogram.

Quartiles

To draw a histogram, we divide the data into a fixed number of bins. Each bin has the same width and could contain any number of observations. We'll now learn about another type of summary plot that divides the data into intervals of varying width, each containing the same number of observations.

Recall the quartiles $Q_1(s)$, $Q_2(s)$, and $Q_3(s)$ are three “fence posts” that separate the data into four intervals with an equal number of observations in each, as illustrated in Figure 1.22.

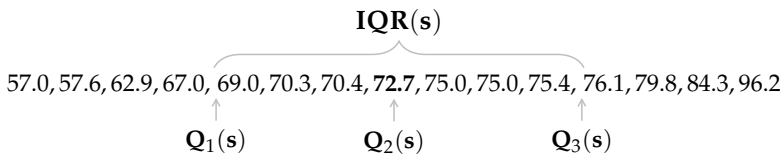


Figure 1.22: Positions of the three quartiles and the interquartile range.

We compute the quartiles using the method `.quantile(q)`, for appropriate choice of the argument `q`.

```
code >>> Q1 = scores.quantile(q=0.25)
1.3.11 >>> Q1
68.0
>>> Q2 = scores.quantile(q=0.5)
>>> Q2
72.7
>>> Q3 = scores.quantile(q=0.75)
>>> Q3
75.75
```

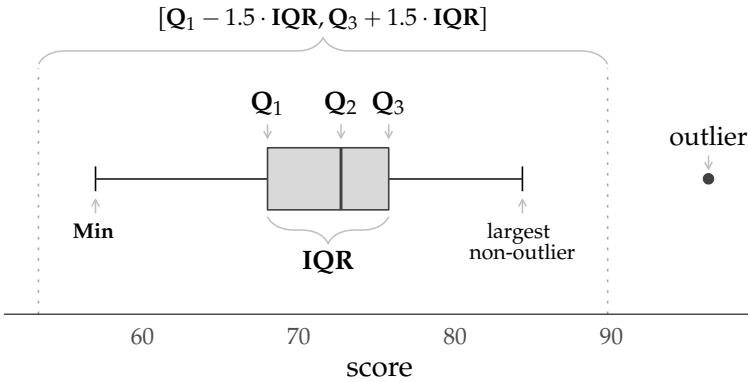


Figure 1.23: Box plot for the `scores` data with additional labels for quantities represented in the plot. The left and right boundaries of the box represent the first and third quartiles. The vertical line in the middle of the box indicates the median. The point on the far-right is called an *outlier*. The lines extending from the box are called *whiskers* and represent the range of the data excluding outliers. The whiskers reach from the smallest and largest values within the interval $[Q_1 - 1.5 \cdot IQR, Q_3 + 1.5 \cdot IQR]$. Any observations that fall outside the whiskers are considered outliers and are represented with a dot.

than $Q_3 + 1.5 \cdot IQR$. The purpose of the whiskers is to provide an “honest” representation of the range of the data. By splitting off the outliers as independent points, the whiskers show us the non-outlier range of the data.

We can use the Seaborn function `boxplot` to produce a box plot.

```
code >>> sns.boxplot(data=students, x="score")
1.3.13 The result is shown in Figure 1.23
```

Box plots are often used to visualize numerical data, since the quartiles Q_1 , Q_2 , and Q_3 provide an excellent summary of the data, and the whiskers tell us the range of all the non-outlier values.

Summary

Let’s review all the descriptive statistics we calculated from the `score` variable in the `students` dataset. Table 1.7 lists all the numerical statistics we computed and the relevant Pandas methods we used to compute each statistic from the `scores` series.

We can compute the most important summary statistics in a single step by calling the `.describe()` method on the `scores` series.

```
code >>> scores.describe()
1.3.14 count      15
      mean      72.58
      std       9.98
```

1.3.5 Explanations

We'll now provide some additional details and explanations about descriptive statistics, which we skipped in the previous pages.

Measures of shape

The concepts of *skewness* and *modality* are used to describe the “shape” of a data distribution, when looking at its histogram.

Skewness Many data distributions have the bulk of their values concentrated in one central region, and the frequency of values gets smaller and smaller as we move away from the central region. The values extending to the left and the right of the main region are called the *tails* of the distribution. When the distribution has a long left tail, we say it is *left-skewed*, and conversely, when the distribution has a long right tail, we say it is *right-skewed*. Distributions that have similar left and right tails are called *symmetrical*. Figure 1.31 shows examples of histograms with different skews.

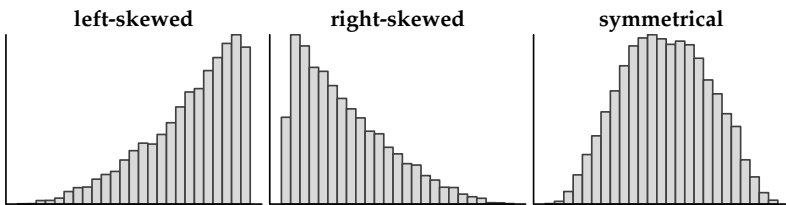


Figure 1.31: Histograms displaying distributions with three different *skews*. When the tail of the distribution extends further to the left, the data is *left-skewed*. When the values extend further to the right, the data is *right-skewed*.

The terms *left-skewed* and *right-skewed* are more qualitative than quantitative, but there are also numerical measures of skewness we can use (more on that in Section 2.6).

Modality The *modality* of a distribution describes how many “peaks” it has. Figure 1.32 shows examples of three distributions with different modalities.

The number of modes in a histogram of the data is an important characteristic describing any dataset. You must be aware if you’re dealing with multimodal distribution in order to choose appropriate statistical analysis procedures in later chapters.

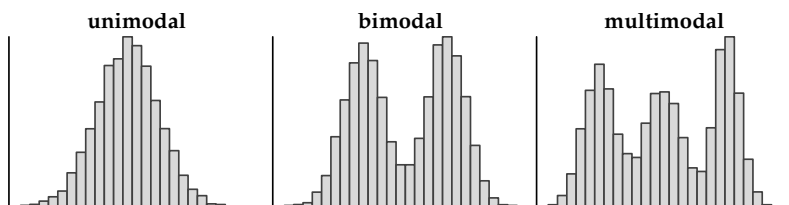


Figure 1.32: Histograms displaying samples with three different *modalities*. Distributions with only one peak in are called *unimodal*. If we see two peaks in the histogram, we say the distribution is *bimodal*. Distributions with more than two peaks are called *multimodal*.

Intuitive interpretations of the mean

There is a useful physical analogy for understanding the mean \bar{x} that I want you to know about. Imagine that each data point has some weight to it, let's say one gram per data point. The location of the mean corresponds to the *centre of mass* of this distribution of weights. If all the weights were placed on a long ruler and lifted into the air, then you would be able to balance the ruler using a single finger by supporting the ruler at the location of the arithmetic mean (the centre of mass), as shown in Figure 1.33. Values smaller than the mean tend to tilt the ruler to the left of your finger, but values larger than the mean counterbalance them, so the ruler will stay balanced.

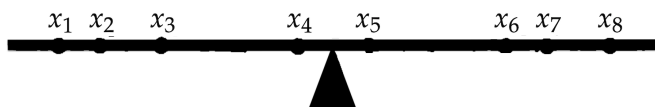


Figure 1.33: The mean \bar{x} is the centre of mass of a distribution of weights.

Another way to think of the mean is as a representative value for the sample $\mathbf{x} = [x_1, x_2, \dots, x_n]$. Suppose we had to replace the values x_i with a single, common value repeated n times, while keeping the total sum of the values the same. We can accomplish this by repeating the mean n times $[\bar{x}, \bar{x}, \dots, \bar{x}]$. Figure 1.34 illustrates this process using the score variable $\mathbf{s} = [s_1, s_2, \dots, s_{15}]$. Part (a) of the figure shows the observed 15 student scores s_i , represented as vertical bars. Part (b) shows how we can replace the data with 15 copies of a single representative value $[\bar{s}, \bar{s}, \dots, \bar{s}]$.

Histogram binning

When we created the histogram of the score variable in Figure 1.21 (see page 65), we divided the data into bins of width 10. Choosing

Chapter 2

Probability

Probability theory is a language for describing randomness, variability, and uncertainty. Understanding probability theory is essential for doing statistics, because probability models allow us to describe the variability in populations and samples. To apply statistical procedures correctly, you need to learn about the various probability models that are used as building blocks in statistical calculations. This is what this chapter is all about.

What are probability models? Probability models are mathematical constructs for describing the different types of variability of real-world quantities. Suppose we are interested in the quantity X , which could be the outcome of a coin toss, a die roll, or a measurement in an industrial process. We refer to these kinds of quantities as *random variables*, and use capital letters to denote them. Unlike a regular variable x that is a placeholder for a single value, the random variable X can have different *outcomes* every time it is observed. The possible outcomes of a coin toss are heads and tails. The possible outcomes of rolling a die are the numbers in the set $\{1, 2, 3, 4, 5, 6\}$. The outcomes of the industrial process are obtained by measuring the quantity of interest, like the volume of liquid in a bottle.

A *probability model* is a way to characterize and quantify the variability of a random variable. We'll use the math notation $X \sim \mathcal{M}(\theta)$ to describe a random variable X that is distributed according to the probability model $\mathcal{M}(\theta)$. The symbol " \sim " is a shorthand for the phrase "distributed according to." Whenever we talk about probability models in general, we'll use the calligraphic letter \mathcal{M} to denote the *model family* and the Greek letter θ (*theta*) to denote the *model parameters*.

Examples of probability model families include the discrete

uniform family \mathcal{U}_d , the continuous uniform family \mathcal{U} , the normal family \mathcal{N} , etc. Each model family describes random variables whose distribution has a particular “shape.” Figure 2.1 shows six examples of random variables, generated from six different probability models that we’ll learn about later in this section.

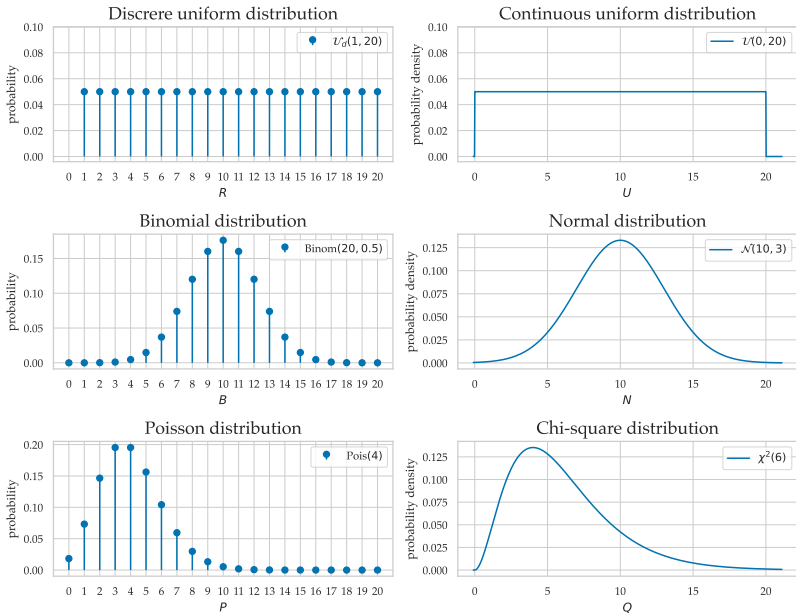


Figure 2.1: Six examples of probability models that describe the variability of six different random variables: R , U , B , N , P , and Q . The three examples on the left side are *discrete* random variables, which can take on integer values. The three examples on the right are *continuous* random variables, which describe smoothly varying quantities.

Without going into too much detail, here is a short description of the six random variables shown in Figure 2.1:

- $R \sim \mathcal{U}_d(1, 20)$ is a *discrete uniform* random variable whose outcomes are random integers in the set $\{1, 2, 3, \dots, 20\}$. All outcomes have an equal probability of occurrence. You can think of R as the result of rolling a 20-sided die.
- $U \sim \mathcal{U}(0, 20)$ is a *continuous uniform* random variable that assigns equal probability over the interval $[0, 20]$. The flatness of the curve tells us that all values between 0 and 20 are equally likely to occur. For example, U could represent how long you’ll have to wait for a bus that runs every 20 minutes.
- $B \sim \text{Binom}(n = 20, p = 0.5)$ is a *binomial* random variable that describes the number of heads observed in $n = 20$ coin tosses

of a coin with probability of heads $p = 0.5$. The most likely outcome is to observe 10 heads, but the outcomes 8, 9, 11, and 12 are also likely to occur.

- $N \sim \mathcal{N}(\mu = 10, \sigma = 3)$ is a *normal* random variable with mean parameter $\mu = 10$ (the Greek letter *mu*) and standard deviation $\sigma = 3$ (the Greek letter *sigma*). The normal model is one of the most useful probability models that describes many real-world quantities.
- $P \sim \text{Pois}(\lambda = 4)$ is a *Poisson* random variable that describes the number of occurrences of some event over a fixed time period. For example, P could represent the number of phone calls you'll receive in one day. The parameter $\lambda = 4$ (the Greek letter *lambda*) tells us that we can expect to receive four calls per day on average.
- $Q \sim \chi^2(\nu = 5)$ is a *chi-square* random variable, which is used to model sums of squared deviations from some expected value. The parameter ν (*nu*) describes the *degrees of freedom* of the distribution, which is equal to the number of squared-deviation terms summed together.

Later in this chapter, we'll describe each of these random variables in more detail and explore their properties and applications to modelling real-world quantities. You can think of probability models as LEGOs that you can play with. We'll start with a simple example based on the normal distribution.

Example: kombucha bottling process Imagine that you work at a kombucha brewery, and you're in charge of the bottling process. Your goal is to put exactly 1 litre (1000 ml) of kombucha in each bottle, but because of the bubbles in the kombucha, there is a natural variability in the total volume that goes into each bottle. Figure 2.2 shows a strip plot of the kombucha volume measurements from the last 500 bottles produced at the brewery.

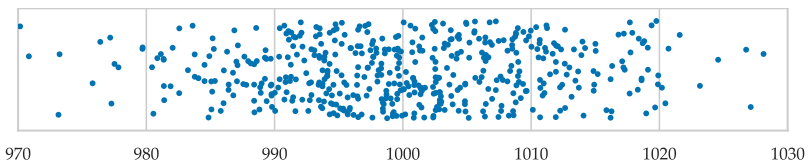


Figure 2.2: Observed values of the volume of kombucha from 500 bottles. The average volume is 1000 ml with a standard deviation of 10 ml.

You can model the volume in each bottle as a normally distributed random variable $K \sim \mathcal{N}(\mu_K = 1000, \sigma_K = 10)$. The mean parameter

$\mu_K = 1000$ determines the centre of the distribution. The standard deviation parameter $\sigma_K = 10$ describes the variability of the values around the centre. Figure 2.3 shows the model (a mathematical construct) for the random variable K , which describes the variability of kombucha volume in different bottles.

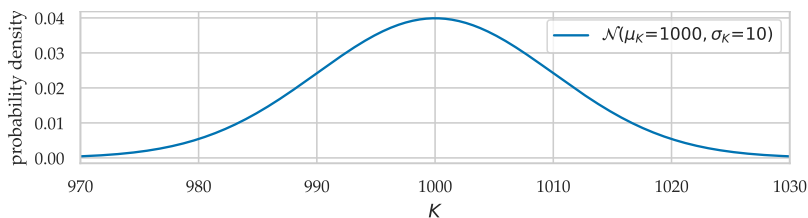


Figure 2.3: The random variable K is distributed according to $\mathcal{N}(1000, 10)$. The probability of observing a value of K is proportional to the probability density that the model assigns to each value of K .

By choosing the right parameters $\mu_K = 1000$ and $\sigma_K = 10$ for the normal model, you obtain a random variable K that approximates the variability observed in the kombucha volumes. The model assigns about 95% of the probability to values in the interval $[980, 1020]$, and very low probability to values of K below 980 and above 1020. Compare the density curve in Figure 2.3 with the strip plot of kombucha volumes in Figure 2.2. The normal model seems to be a good fit: the distribution curve is “thick” (high density) in places where the data points are most concentrated, and “thin” (low density) where the points are rare.

It’s important to understand the connection between the mathematical model shown in Figure 2.3 and the real-world kombucha volume measurements shown in Figure 2.2. The diagram in Figure 2.4 illustrates this “is a model of” relationship that exists between the real-world quantity (the observed volume measurements) and the math-world random variable $K \sim \mathcal{N}(1000, 10)$.

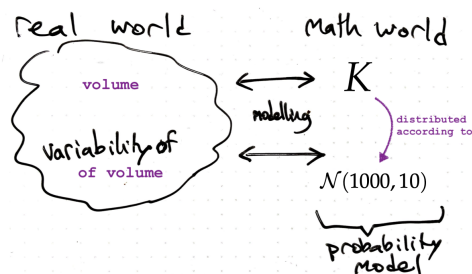


Figure 2.4: Modelling the variability of kombucha volume in different bottles using the random variable $K \sim \mathcal{N}(1000, 10)$.

Using the math model $K \sim \mathcal{N}(1000, 10)$, you can do probability calculations and make predictions related to the kombucha bottling process. The options available to you when using the random variable K include: doing calculations based on math formulas, generating graphs and visualizations, and running computer simulations. For example, you could estimate the probability of observing kombucha volumes $K < 980$, which is an outcome of business importance, since distributors consider bottles containing less than 980 ml to be defective and refuse to pay for them.

The goal of this example was to illustrate that probability models are not just abstract math constructs, but also very practical, since they help us model real-world quantities. We'll continue the discussion on this kombucha scenario in Section 2.4 (see page 194).

* * *

The goal of this chapter is to help you understand probability theory concepts. We'll learn about probability models from multiple perspectives:

- **Math models.** We'll define probability concepts precisely using math equations and formulas. Don't worry if your math skills are a little rusty: each piece of math notation will be translated into words and each formula will be explained.
- **Computer models.** We can encode the probability models as Python code, which allows us to leverage the power of computers for probability calculations. The Python module `scipy.stats` contains predefined probability models for all the random variables we'll use in this chapter.
- **Visualizations.** We'll use graphical representations of probabilities to visualize probability distributions. Behind every math formula, there is a "picture" you can draw to visualize what it is saying, which is often easier to grasp than the math.
- **Simulating random samples.** We can understand the behaviour of random variables by generating thousands of random observations from them. We can use the distribution of random observations to understand the properties of random variables and empirically verify probability laws and formulas.

Each perspective offers a different way to think about probabilities, helps us better understand random variables, and gives us tools for doing calculations.

By the end of this chapter, you'll have gained valuable experience and intuition about probability theory, which will make your study

of statistics in Part 2 of the book much easier and more interesting. Basically, in the next 200 pages, you'll get to know the properties of probability models—the LEGOs for the 21st century, so that we can play with them in the remainder of the book.

Are you ready for this? Let's go!

In this chapter, I'm going to show you how to . . .

- **describe** the random variable X in terms of its PMF/PDF f_X and its CDF F_X
- **build** a computer model for the random variable X using `scipy.stats`
- **plot** the probability function (PMF or PDF) f_X of the random variable X
- **plot** the cumulative distribution function (CDF) F_X of the random variable X
- **calculate** the probability of events using math formulas
- **calculate** the probability of events using computer models
- **calculate** the mean μ_X and the variance σ_X^2 of the random variable X
- **calculate** the expected value $\mathbb{E}_X[g(X)]$ of the function g that depends on the random variable X
- **recognize** when two random variables are independent or dependent
- **model** real-world quantities using random variables
- **generate** a random sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$ from the random variable X

2.1 Discrete random variables

Probability theory started when a bunch of mathematicians went to the casino and tried to use their math skills to compute what they can expect to win from different games of chance. Over time, probability theory was applied to many other situations where uncertainty plays a role, including biological processes, engineering, manufacturing, business, machine learning, and many other domains. Probability theory is the foundation of statistics.

In this section, we'll start our discussion about probability theory with the analysis of simple random variables like the outcomes of a coin toss and a die roll. We'll define the new concepts and terminology for describing random variables and give lots of examples of probability calculations.

2.1.1 Definitions

Probabilities are real numbers between 0 and 1 that we assign to different events to describe how likely they are to occur. We denote the probability of event A as $\Pr(A)$. If an event has zero probability, $\Pr(A) = 0$, this means the event A never happens. At the other extreme, if $\Pr(B) = 1$, this means the event B is certain to occur. Probability values between 0 and 1 describe events that occur some of the time.

Random variables

We use random variables to describe unknown or uncertain quantities and assign probabilities to their different possible outcomes.

- *random variable* X : a quantity that can randomly take on any value in a set of possible values. We denote random variables by uppercase letters like X , Y , and Z .
- *sample space* \mathcal{X} : the calligraphic- X describes the set of all possible values of the random variable X . Each value in the sample space may have a different probability of occurring.
- *outcome* x : the result of observing a random variable. We denote outcomes using lowercase letters. The outcome x describes one particular element of the sample space.
- An *event* is a subset of the sample space. For example, the set of outcomes between a and b , which we denote $\{a \leq X \leq b\}$.
- f_X : the *probability distribution function* is a function that assigns probabilities to the different outcomes in the sample space of the random variable X .

There are two kinds of random variables: discrete and continuous:

- A *discrete* random variable has a sample space \mathcal{X} that consists of discrete values, like the integers $\{0, 1, 2, 3, \dots\}$. See the left side of Figure 2.1 (page 98) for examples of discrete random variables.
- A *continuous* random variable has a continuous sample space \mathcal{X} , like the real numbers $\{0.1, 1.2, 3.14, -2.5, \dots\}$. Review the right side of Figure 2.1 for examples of continuous random variables.

We use different kinds of “math machinery” to compute probabilities for these two types of random variables. In this section, we’ll focus on the definitions and formulas for discrete random variables, and defer the discussion on continuous random variables until Section 2.4.

Discrete random variables

Examples of discrete sample spaces include the outcomes of a coin toss $\{\text{heads}, \text{tails}\}$, the roll of a die $\{1, 2, 3, 4, 5, 6\}$, or a countably infinite set like the natural numbers $\mathcal{N} \stackrel{\text{def}}{=} \{0, 1, 2, 3, \dots\}$.

A discrete random variable X is described by a *probability mass function* f_X , which tells us how much probability “weight” is assigned to each of the possible outcomes:

$$\Pr(\{X = x\}) = f_X(x), \quad \text{for all } x \text{ in } \mathcal{X}.$$

The abbreviation *PMF* is often used to refer to the probability mass function.

An example of a probability mass function is shown in Figure 2.5.

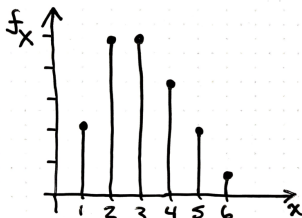


Figure 2.5: Illustration of the probability mass function f_X for the random variable X defined on the sample space $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$. The height of each line in the step plot tells us the probability of this outcome occurring.

Calculating probabilities We often need to compute the probabilities of events that consist of several outcomes, which we’ll describe

as sets $\{a \leq X \leq b\} = \{a, a+1, a+2, \dots, b\}$. The probability of the event $\{a \leq X \leq b\}$ is obtained by computing the sum of the probability mass function for all values in the set:

$$\begin{aligned} \Pr(\{a \leq X \leq b\}) &= f_X(a) + f_X(a+1) + f_X(a+2) + \dots + f_X(b) \\ &= \sum_{x=a}^{x=b} f_X(x). \end{aligned}$$

The symbol \sum (the capital Greek letter *sigma*) is the math notation for describing summations. The subscript of the summation $x = a$ tells us where to start the summation, and superscript $x = b$ tells when to end it. In words, we calculate the probability of an event by adding up the total probability of the outcomes it includes.

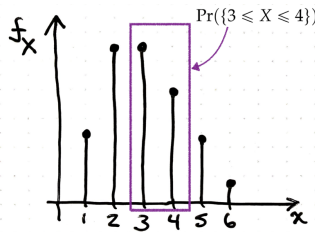


Figure 2.6: Illustration of the probability of the event $\{3 \leq X \leq 4\}$. The combined length of the two lines in the highlighted region correspond to the probability $\Pr(\{3 \leq X \leq 4\}) = \sum_{x=3}^{x=4} f_X(x)$.

The Figure 2.6 illustrates the calculation of the probability that the discrete random variable X will have outcome 3 or 4, which is the sum of the probability mass function values for these two outcomes: $\Pr(\{3 \leq X \leq 4\}) = f_X(3) + f_X(4) = \sum_{x=3}^{x=4} f_X(x)$.

Properties of probability mass functions Every probability mass function f_X satisfies the following math axioms:

- Nonnegativity: $f_X(x) \geq 0$ for all x in \mathcal{X} .
- Unit total: $\sum_{x \in \mathcal{X}} f_X(x) = 1$.

These two conditions are known as *Kolmogorov's axioms of probability*, in honour of the mathematician Andrey Kolmogorov who first introduced them. The first axiom states that probability functions cannot take on negative values. The second axiom states that the total amount of probability over the whole sample space is 1. The symbol " \in " is math shorthand for "element of," so the expression $\sum_{x \in \mathcal{X}} f_X(x)$ describes the summation of the function $f_X(x)$ over all values of x in the set \mathcal{X} .

variable X . Recall that $\mathbb{E}_X[w(X)] = \sum_{x \in \mathcal{X}} w(x) \cdot f_X(x)$, so the method `rvX.expect(w)` is equivalent to calculating

```
sum([w(x)*rvX.pmf(x) for x in range(xmin,xmax+1)]),
```

where `xmin` is the smallest value in the sample space \mathcal{X} and `xmax` is the largest value in \mathcal{X} .

To summarize, anything you might want to know about the random variable X is available at the tip of your fingers once you create the computer model `rvX`. The module `scipy.stats` has predefined computer models for all the important distributions in statistics: `poisson`, `randint`, `bernoulli`, `binom`, `uniform`, `norm`, `t`, `beta`, etc. You'll learn all about these distributions later in this chapter (Section 2.3 for discrete distributions and Section 2.6 for continuous distributions). You can think of these probability models as different kinds of LEGOs available for you to play with.

In the next section, we'll show how to create the computer model `rvH` for the random variable H that describes the number of hard-disks failures.

2.1.5 Hard disks example

Suppose you're the operator of a data centre, and you want to estimate the number of hard disk failures that will occur this month. Based on the specifics of your data centre, you know that 20 hard disk failures occur on average each month, but knowing the average is not sufficient for the types of questions you need to answer. What is the probability of observing 21 hard disk failures? What is the probability of having 25 hard disk failures or less? Can you give a range of outcomes that will occur 95% of the time? These are the types of questions your colleagues are interested in. The software engineering department needs to know the probabilities of different outcomes to design the redundant storage system. The legal department wants to know "worst case" scenarios to write the *Service Level Agreement* documents. The finance people are interested in estimating costs of replacement disks.

All these requests for estimates are piling up in your inbox, but despite your interest in data science topics, you never seem to find the time to do the probabilistic modelling exercise needed to answer these questions. One day during a high-level meeting, your colleagues decide to gang up on you and complain loudly about the lack of estimates, and put you on the spot in front of everyone.

You decide to get this done right then and there, and tell people:

"Relax everyone, we can do these estimates right now using Python."

Everyone seems immediately reassured.

You know the Poisson family of probability models is well suited for describing the random number of hard disk failures in general. To obtain a random variable rvH that has the desired distribution, you need a Poisson model with parameter $\lambda = 20$. You proceed to share your screen so everyone can see, open a Python shell, and start typing:

```
>>> from scipy.stats import poisson
>>> rvH = poisson(20)
```

code
2.1.16

The code above imports the `poisson` model from the SciPy module `scipy.stats` and creates an instance of it called `rvH` initialized with parameter $\lambda = 20$. Now that you have the computer model `rvH`, you can use all the methods like `rvH.pmf(h)`, `rvH.cdf(b)`, `rvH.ppf(q)`, `rvH.rvs(n)`, etc. to do calculations with the random variable H . Feeling reassured by the plethora of methods available to you, you explain what you want from your colleagues during the meeting: “I want you to give me any probability question related to hard disk failure rates now, and I’ll use the probability model to answer your question to the best of my ability. Live.”

There is a moment of silence in the room as people are processing your directives. You decide to use the time to compute the probability of some outcomes:

```
>>> rvH.pmf(20)
0.0888353173920848
>>> rvH.pmf(21)
0.08460506418293791
>>> rvH.pmf(22)
0.07691369471176195
```

code
2.1.17

You explain to your colleagues this means the probability of observing 20 failures next month is 8.88%, the probability of observing 21 failures is 8.46%, and the probability of 22 failures is 7.69%.

Alice from accounting interrupts with a question. “Wait, I thought you said the expected value is 20. Now you’re telling us there is just 8% chance of that happening?”

“Yes, the average is $\mu_H = 20$, but we could have 21, 22, 23, or any other number of failures next month.”

“So we can’t know for sure how many failures will occur?”

“No, we can’t know for sure since failures are random, but we can think about the different possible outcomes and plan accordingly. For example, we could run simulations to—.” You stop yourself mid-sentence because you sense this meeting can go on forever if you start explaining each concept in detail. Better show than tell.

In order to better describe the range of values for the random variable H , you compute the two important statistics of the probability distribution:

```
>>> rvH.mean()
20.0
>>> rvH.std()
4.47213595499958
```

You interpret these numbers for your colleagues by saying: “This means that we can expect roughly 20 plus or minus 5 failures on average.”

“What do you mean ‘plus or minus 5’?” asks Bob from sales.

“I mean that the number of failures will likely be between $20 - 5 = 15$ and $20 + 5 = 25$.” You then proceed to compute the exact probability by summing the probabilities of the individual outcomes in that range.

```
code >>> sum([rvH.pmf(h) for h in range(15,25+1)])
2.1.19 0.782950746174042 # = Pr({ 15 <= X <= 25 })
```

In other words, 78.2% of data centres like ours will experience between 15 and 25 failures.

You then say “Here is a plot that shows the probabilities of all the outcomes,” while typing in the commands:

```
code >>> import matplotlib.pyplot as plt
2.1.20 >>> import numpy as np
>>> hs = range(0, 40)
>>> fHs = rvH.pmf(hs)
>>> plt.stem(hs, fHs)
Result is shown in Figure 2.12.
```

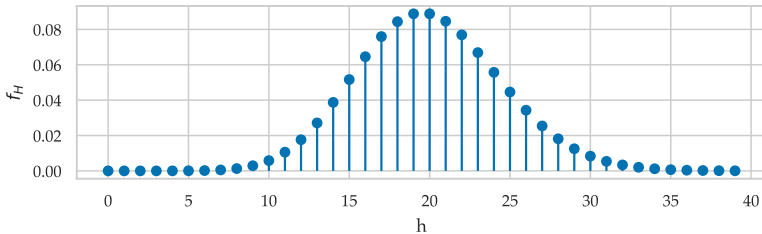


Figure 2.12: Plot of the probability mass function of the Poisson model with parameter $\lambda = 20$. The possible outcomes are clustered around $h = 20$ with most of the probability mass falling in the interval $[15, 25]$.

Desiring to keep the conversation going, you ask “Other questions?”

“I have one.” says Charlotte from software engineering. “I want to know, what is the maximum number of failures that I should plan for.”

“I can’t answer that question because, theoretically speaking, any number of failures can occur. What I can do is give you a 95% confidence interval,” you explain as you’re typing this in:

```
code >>> rvH.ppf(0.95)
2.1.21 28
```

To explain what this number means, you say “95% of the data centres like ours will not observe more than 28 failures.” If you plan for 28 failures as the worst-case scenario, you know there is only a 5% chance that what happens next month will not be covered. Charlotte seems satisfied with the estimate.

David from the marketing department has a question. “Is this thing that you just did AI?,” he asks, thinking about how he can use this in the webcopy for the new company website.

“I suppose you *could* say that, since we’re using probability distributions and probability distributions are also used in AI,” you explain, stretching the definitions.

“What about blockchain? Are we using a blockchain for this?”

“No blockchain,” you interject briskly. “Listen David, let’s proceed one buzzword at a time. You can have ‘AI’ for now. Show me you can sign \$1M worth of new clients using the ‘AI’ buzzword, then come back to me, and I’ll find another buzzword for you.”

“Okay, deal! I can work with that. AI is hot these days.”

Looking around the room, you sense the meeting is coming to a close. Everyone is feeling good about their first data science experience. You decide to wrap things up with some random number generation. “To close the meeting, let me show you some examples of the possible numbers of failures we can expect to see during the next year,” you say while running the command needed to generate 12 random samples from the random variable rvH :

```
>>> rvH.rvs(12)                                     code
[23, 15, 21, 25, 15, 17, 19, 21, 17, 19, 16, 21]      2.1.22
```

You hear several low-level “wows” in the room as the concept of random variable finally sinks in. Simulations of real-world data always work! Finally people get it—the average is 20, but the number of failures can vary a lot around that average.

Later that day, you receive a followup email from Emily from the purchasing department. She wants an estimate of the total cost she should budget for replacement hard disks. The base price is \$200/disk, but it is reduced to \$150/disk if you buy 20 or more disks. In other words, Emily is asking you to compute $\mathbb{E}_H[\text{cost}(H)]$, where the function $\text{cost}(h)$ describes the cost of purchasing h replacement disks. To compute the answer, you first define a Python function for the cost:

```
>>> def cost(h):                                     code
    if h >= 20:                                       2.1.23
        return 150*h
    else:
        return 200*h
```

You can then find the expected value of the function cost by computing the sum over all the possible outcomes, weighing the cost in each case by the probability of this outcome to occur.

```
code >>> sum([cost(h)*rvH.pmf(h) for h in range(0,100+1)])
2.1.24 3381.42
```

Note we truncated the summation up to $h = 100$ because the probabilities $f_H(101)$, $f_H(102)$, etc. are negligibly small.

* * *

I hope reading about this real-world scenario convinced you of the general usefulness of the computer models defined in `scipy.stats` for doing probability calculations. The methods available on the random variable object `rvX` provide us with ways to compute all the quantities we introduced in this section using math equations. This means you don't have to worry about memorizing all the math formulas, you just need to learn how to import one of the predefined probability models in `scipy.stats`, initialize the model with the desired parameters, then use its methods for the probability calculations you need. In the above example, we didn't have to manually input the complicated-looking math formula $f_H(h) = \frac{\lambda^h e^{-\lambda}}{h!}$ for the Poisson distribution, because the method `rvH.pmf(h)` already contains this formula!

To create a computer model for a random variable based on one of the models defined in `scipy.stats`, use the code `rvX = <model>(<params>)`, where `<model>` is the name of the model you imported from `scipy.stats`, and `<params>` is a comma-separated list of model parameters. The parameters will depend on the model. You'll learn about the most important discrete distribution in Section 2.3. See Table 2.1 on page 163 for a list of the discrete distributions available in `scipy.stats`.

2.1.6 Exercises

In these exercises, we'll explore the methods of random variable objects created from `scipy.stats` models. Exercises E2.14 and E2.15 will use the `randint(alpha, beta+1)` model, which corresponds to the discrete probability distribution $U_d(\alpha, \beta)$. In exercise E2.16, we'll use the Poisson model `poisson(lam) = Pois(λ)`.

You'll need to use a Jupyter notebook and run appropriate commands to import models from `scipy.stats`, create random variable objects, then call the methods on these objects to answer the questions.

2.2 Multiple random variables

We're often interested in analyzing the relationship between two random variables, like X and Y . We can describe the joint variability of the pair (X, Y) using a *joint probability distribution* f_{XY} . Different types of joint probability distributions f_{XY} can be used to model different types of relationships between the two random variables.

In this section, we'll study the formulas we use for probability calculations with multiple random variables. We'll introduce new math concepts like the *conditional probability distribution* $f_{Y|X}$, read "probability of Y given X ," which describes the uncertainty about Y that remains once we observe the value of X . The joint probability distribution f_{XY} and the conditionals $f_{Y|X}$ and $f_{X|Y}$ are the tools we use to model the relationships between random variables.

The formulas and calculations in this section mirror the formulas and calculations we learned for single random variables in the previous section, but the sample space is two-dimensional. Since we're dealing with multiple variables, the graphical representation of probability distributions will look different, but the general probability concepts and calculations will be very similar.

2.2.1 Definitions

Consider the pair of random variables (X, Y) defined over the sample space $\mathcal{X} \times \mathcal{Y}$, which is the *product* of the sample spaces for the two random variables \mathcal{X} and \mathcal{Y} . We call $\mathcal{X} \times \mathcal{Y}$ the *joint sample space* of the random variables (X, Y) . A particular outcome in the joint sample space looks like a pair of numbers (x, y) , where x is the value observed for the random variable X , and y is the value observed for the random variable Y . Geometrically speaking, the joint sample space is a two-dimensional region.

At this point, it might be worth clarifying again that a "joint sample space" is not some kind of establishment that you visit to try different types of marijuana smokables. Rest assured, learning about multi-variable probability distributions will lead you to some all-natural experiences of *knowledge buzz*, as you learn about some of the most important probability ideas, which form the foundation of statistics.

- X, Y : a pair of random variables
- $\mathcal{X} \times \mathcal{Y}$: the *joint sample space* of the random variables X and Y
- $f_{XY}(x, y)$ the *joint probability mass function* of the random variables X and Y . The function f_{XY} has the form $f_{XY}: \mathcal{X} \times \mathcal{Y} \rightarrow [0, 1]$, and it tells us the probability of each of the possible pairs

of outcomes:

$$f_{XY}(x, y) \stackrel{\text{def}}{=} \Pr(\{X = x, Y = y\}),$$

for all $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

The math machinery for working with multiple random variables is similar to what we saw in the previous section, but we'll be working with functions like $f_{XY}(x, y)$ that depend on two inputs, instead of single-variable functions like $f_X(x)$.

The events in the joint sample space consist of subsets of the sample space. For example, the event where X takes on a value between $x = a$ and $x = b$, and Y takes on a value between $y = c$ and $y = d$ is written as $A = \{a \leq X \leq b, c \leq Y \leq d\}$ in set notation. Geometrically speaking, this event describes a rectangular region with width $b - a$ and height $d - c$. The probability of this outcome A is obtained by calculating the sum of the probability mass function f_{XY} over all the outcomes in the subset A :

$$\Pr(A) = \sum_{(x,y) \in A} f_{XY}(x, y) = \sum_{x=a}^{x=b} \sum_{y=c}^{y=d} f_{XY}(x, y).$$

Up until here, there is nothing new going on. The concept of a joint probability mass function f_{XY} is directly analogous to the probability mass functions we saw in the previous section, but with more dimensions. You have to trust me on this one: double summation formulas might look intimidating, but there is nothing fancy going on. It's still the same idea of calculating the "total" amount of f_{XY} over a set of outcomes.

Marginal distributions

Starting from the joint probability mass function $f_{XY}(x, y)$, we can define the *marginal* probability distributions. The process of marginalization describes the uncertainty in one random variable when we *don't know* the other random variable:

- $f_X(x) \stackrel{\text{def}}{=} \sum_{y \in \mathcal{Y}} f_{XY}(x, y)$ is the *marginal distribution* for the random variable X . The marginal distribution f_X describes the uncertainty in the random variable X when the random variable Y is unknown.
- $f_Y(y) \stackrel{\text{def}}{=} \sum_{x \in \mathcal{X}} f_{XY}(x, y)$ is the *marginal distribution* for the random variable Y .

When a random variable is unknown, we model our ignorance by summing over its possible values.

Example 1: two coin tosses Consider the scenario in which we toss a fair coin twice. We'll model the outcome of each coin toss as a random variable $C \in \{\text{heads}, \text{tails}\}$, with probability mass function f_C with values $f_C(\text{heads}) = \frac{1}{2}$ and $f_C(\text{tails}) = \frac{1}{2}$.

The joint sample space for the two coin tosses contains four possible outcomes:

$$\{\text{heads}, \text{tails}\} \times \{\text{heads}, \text{tails}\} = \{(\text{heads}, \text{heads}), (\text{heads}, \text{tails}), (\text{tails}, \text{heads}), (\text{tails}, \text{tails})\}.$$

The joint probability mass function $f_{C_1 C_2}(c_1, c_2)$ describes the two coin tosses, and it has the same value for all four possible outcomes:

$$f_{C_1 C_2}(c_1, c_2) = f_C(c_1) \cdot f_C(c_2) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} = 0.25.$$

Note the joint probability mass function $f_{C_1 C_2}$ is the product of the probability mass functions for the individual coins f_C . Figure 2.16 shows a stem plot of the joint probability mass function $f_{C_1 C_2}$. The height of each stem corresponds to the probability of this outcome.

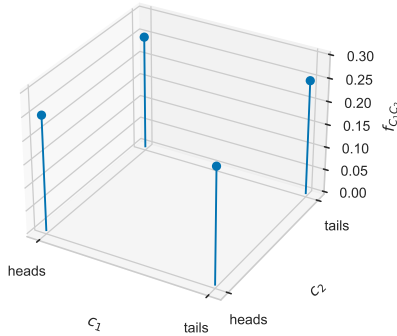


Figure 2.16: The joint probability mass function $f_{C_1 C_2}$ for two coin tosses.

The total probability satisfies Kolmogorov's second axiom:

$$\sum_{c_1 \in \{\text{heads}, \text{tails}\}} \sum_{c_2 \in \{\text{heads}, \text{tails}\}} f_{C_1 C_2}(c_1, c_2) = 1.$$

If we want to compute the probability of "one heads," we need to sum together all the possible outcomes that contain exactly one head, of which there are two:

$$\begin{aligned} \Pr(\{\text{one heads}\}) &= f_{C_1 C_2}(\text{heads}, \text{tails}) + f_{C_1 C_2}(\text{tails}, \text{heads}) \\ &= \frac{1}{4} + \frac{1}{4} = \frac{1}{2}. \end{aligned}$$

Tree diagrams We can use a *tree diagram* to visualize the probabilities in a sequence of random events. Each branch corresponds to one possible outcome, and the probability of this outcome is indicated as a number below the branch. The tree diagram in Figure 2.17 shows a graphical representation of the probabilities of the sequence of coin tosses. Reading the diagram from left to right, we see that the first coin toss C_1 causes the world of possibilities to split into two branches, then the second coin toss C_2 causes another split, so we end up with four possible final outcomes. We compute the probability of each final outcome by multiplying the probabilities along its branch.

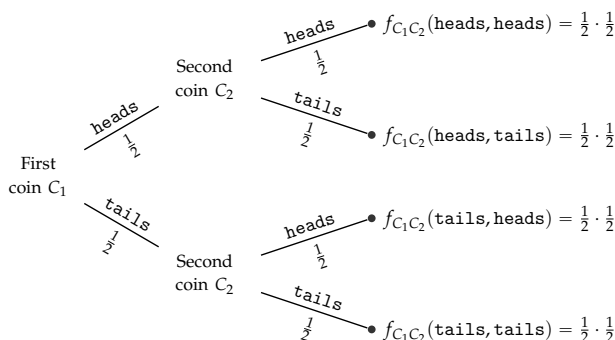


Figure 2.17: Tree diagram of the joint probability mass function $f_{C_1 C_2}$ for two coin tosses.

Example 2: rolling a pair of dice Consider the pair of random variables (D_1, D_2) that represent the outcome of rolling two balanced dice. The random variables D_1 and D_2 are both described by probability mass function $f_D(d) = \frac{1}{6}$, for all d in the sample space $\{1, 2, 3, 4, 5, 6\}$. Their joint probability mass function $f_{D_1 D_2}$ is the product of the two copies of the distribution f_D :

$$f_{D_1 D_2}(d_1, d_2) = f_D(d_1) \cdot f_D(d_2) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}.$$

Suppose we want to calculate the probability of “rolling a seven.” This event is described by the equation $D_1 + D_2 = 7$ and corresponds to the set of outcomes $\{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)\}$. The probability of this event is the sum of $f_{D_1 D_2}$ over this set of outcomes:

$$\begin{aligned}
 \Pr(\{D_1 + D_2 = 7\}) &= \Pr(\{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)\}) \\
 &= f_{D_1 D_2}(1, 6) + f_{D_1 D_2}(2, 5) + f_{D_1 D_2}(3, 4) \\
 &\quad + f_{D_1 D_2}(4, 3) + f_{D_1 D_2}(5, 2) + f_{D_1 D_2}(6, 1) \\
 &= \frac{1}{36} + \frac{1}{36} + \frac{1}{36} + \frac{1}{36} + \frac{1}{36} + \frac{1}{36} = \frac{1}{6} = 0.1\bar{6}.
 \end{aligned}$$

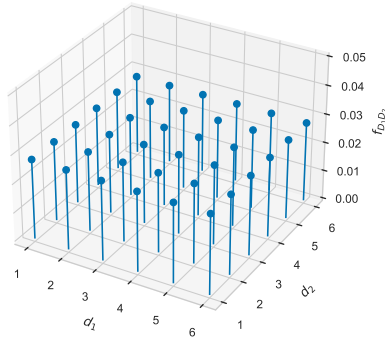


Figure 2.18: The joint probability mass function $f_{D_1 D_2}$ for the two dice rolls.

Geometrically speaking, this summation corresponds to adding up the total length of stems on the diagonal line that goes from $(1, 6)$ to $(6, 1)$, as shown in Figure 2.19.

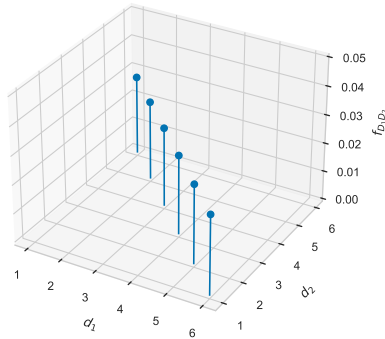


Figure 2.19: Subset of the weights of the joint probability mass function $f_{D_1 D_2}$ that corresponds to the event $D_1 + D_2 = 7$.

Marginal distribution functions

The *marginal probability mass function* f_X is obtained from the joint distribution f_{XY} by summing over all possible outcomes of the variable Y :

$$f_X(x) \stackrel{\text{def}}{=} \sum_{y \in \mathcal{Y}} f_{XY}(x, y).$$

The idea for a marginal distribution f_X corresponds to a description of the random variable X when the random variable Y is unknown. The name *marginal distribution* comes from the procedure we use

2.3 Inventory of discrete distributions

We'll now introduce some probability distributions that are used in statistics. Different types of data are described by different types of probability models, depending on the underlying process that generates the data. Knowing about the different discrete distributions allows us to make informed decisions about which distribution to use to model a given dataset. In this section, we'll learn about different discrete probability model families and their parameters.

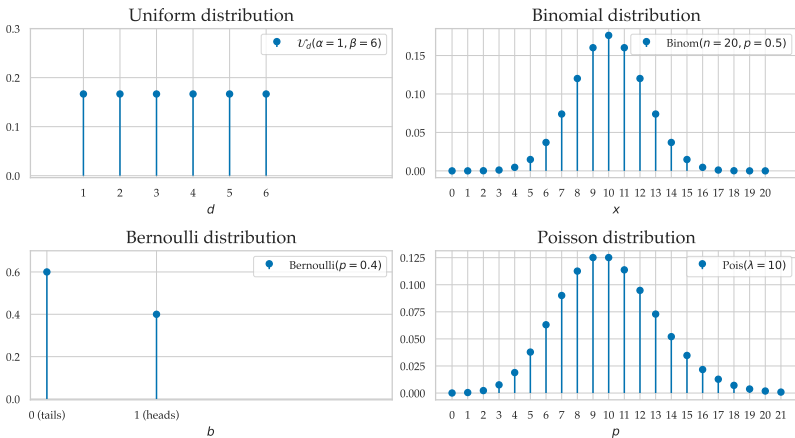


Figure 2.28: Graphs of the probability mass functions of four common discrete probability distributions: uniform, binomial, Bernoulli, and Poisson.

Figure 2.28 shows four discrete probability distributions that are commonly used in statistics:

- The *discrete uniform* distribution $\mathcal{U}_d(\alpha = 1, \beta = 6)$ models a six-sided die, where each of the six outcomes is equally likely.
- The *Bernoulli* distribution $\text{Bernoulli}(p)$ models the outcome of a binary event, like a coin toss. The outcome 1 (heads) has probability p , and the outcome 0 (tails) has probability $(1 - p)$.
- The *binomial* distribution $\text{Binom}(n, p)$ models the number of heads observed in a sequence of n coin tosses.
- The *Poisson* distribution $\text{Pois}(\lambda)$ models the number of times an event occurs over some time interval.

These distributions come up often in statistical analysis scenarios, so it's a good idea to get to know them. You're not expected to memorize any of the math formulas associated with these distributions, but I want you to look at their probability mass function plots to get an intuitive idea about the "shape" of each distribution.

2.3.1 Discrete uniform distribution

The *discrete uniform distribution* $\mathcal{U}_d(\alpha, \beta)$ models a random variable that takes on integer values ranging from α to β with equal probability. This distribution is called “uniform” because all outcomes have the same probability. The probability mass function of the random variable $X \sim \mathcal{U}_d(\alpha, \beta)$ is $f_X(x) = \frac{1}{\beta+1-\alpha}$, for all integers x between α and β . The sample space is $\mathcal{X} = \{\alpha, \alpha+1, \alpha+2, \dots, \beta-1, \beta\}$ and it contains a total of $(\beta+1-\alpha)$ equally likely possible outcomes.

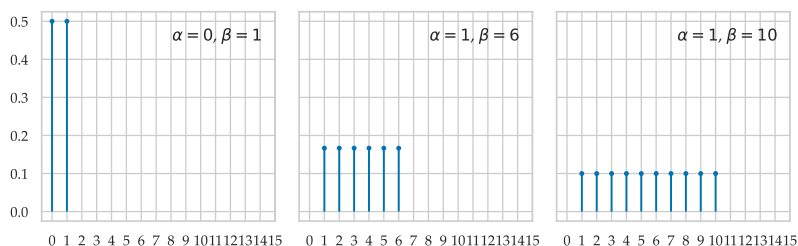


Figure 2.29: Discrete uniform distributions obtained for different choices of the parameters α and β . The uniform distributions $\mathcal{U}_d(0, 1)$ and $\mathcal{U}_d(1, 6)$ correspond to the coin toss C and die roll D random variables we saw in Section 2.1. The distribution $\mathcal{U}_d(1, 10)$ describes a 10-sided die.

The mean of the random variable $X \sim \mathcal{U}_d(\alpha, \beta)$ is $\mu_X = \frac{\alpha+\beta}{2}$, and its variance is $\sigma_X^2 = \frac{1}{12}((\beta+1-\alpha)^2 - 1)$.

Applications The action of picking an integer between α and β at random comes up all over the place. In a sweepstakes contest, we want to award a prize randomly to one of the n participants. We can assign participants numeric identifiers between 1 and n , then generate an observation from the random variable $\mathcal{U}_d(1, n)$ to determine who gets the prize. Given a list of n songs, the “shuffle” play mode uses observations from the distribution $\mathcal{U}_d(1, n)$ to choose which song to play next. We could also use $\mathcal{U}_d(1, n)$ to create a random workout by choosing an exercise at random from a list of n exercises.

Computer model To create a computer model for the random variable $R \sim \mathcal{U}_d(\alpha, \beta)$, use the code `rvR = randint(alpha, beta+1)` after importing the `randint` family from the `scipy.stats` module. We provide a complete code example using `rvR` at the end of this section.

[This Wikipedia article has more info and additional formulas]

https://wikipedia.org/wiki/Discrete_uniform_distribution

2.3.8 Computer models for discrete distributions

The easiest way to work with the probability distributions we described in this section is to use the computer models defined in the `scipy.stats` module. We'll now present the various model families available in `scipy.stats` and illustrate how to use them for probability calculations.

The first step to creating a computer model for the random variable $X \sim \mathcal{M}(\theta_X)$ is to import the model family \mathcal{M} using the code from `scipy.stats` `import <model>`, where `<model>` is the name of a discrete probability distribution family. The second step is to create a random variable object `rvX = <model>(<params>)`, where `<params>` is a comma-separated list of parameters of the model. Table 2.1 shows the names of the discrete probability model families available in `scipy.stats`, as well as the parameters you need to specify for each model.

Math notation	Code notation	Parameters
$\mathcal{U}_d(\alpha, \beta)$	<code>randint(alpha, beta+1)</code>	$\alpha, \beta \in \mathbb{Z}$
$\text{Bernoulli}(p)$	<code>bernoulli(p)</code>	$p \in [0, 1]$
$\text{Binom}(n, p)$	<code>binom(n, p)</code>	$n \in \mathbb{N}_+, p \in [0, 1]$
$\text{Pois}(\lambda)$	<code>poisson(0, 1/lam)</code>	$\lambda \in \mathbb{R}_+$
$\text{Geom}(p)$	<code>geom(p)</code>	$p \in (0, 1)$
$\text{NBinom}(r, p)$	<code>nbinom(r, p)</code>	$r \in \mathbb{N}_+, p \in [0, 1]$

Table 2.1: List of the discrete probability models available in `scipy.stats`.

Once we have created the random variable object `rvX`, we can use its methods to do probability calculations and visualize the probability mass function f_X . You can see a summary of all the `rvX` methods in Table 2.2.

We've already seen these methods in action in Section 2.1, where we created the computer model `rvH` for the hard disk failures random variable $H \sim \text{Pois}(\lambda = 20)$, then used the computer model for various probability calculations (see page 121). We'll now present another example to remind you of the possible calculations and visualizations.

Building a computer model

Consider the random variable $R \sim \mathcal{U}_d(1, 4)$ that describes the outcome of rolling a four-sided die. To create a computer model that corresponds to this random variable, we import the `randint` model family from the `scipy.stats` module, then create the random

method	math formula	description
<code>rvX.pmf(x)</code>	$f_X(x)$	probability mass function
<code>rvX.cdf(b)</code>	$F_X(b)$	cumulative distribution function
<code>rvX.ppf(q)</code>	$F_X^{-1}(q)$	inverse of the CDF function
<code>rvX.mean()</code>	$\mu_X = \mathbf{Mean}(X)$	mean of the distribution
<code>rvX.var()</code>	$\sigma_X^2 = \mathbf{Var}(X)$	variance of the distribution
<code>rvX.std()</code>	$\sigma_X = \mathbf{Std}(X)$	standard deviation
<code>rvX.median()</code>	$F_X^{-1}(\frac{1}{2})$	median of the distribution
<code>rvX.support()</code>	$\min(\mathcal{X}), \max(\mathcal{X})$	bounds of the sample space \mathcal{X}
<code>rvX.expect(w)</code>	$\mathbb{E}_X[w(X)]$	expected value of $w(X)$
<code>rvX.rvs(n)</code>	$[x_1, x_2, \dots, x_n]$	generate n observations from X

Table 2.2: Summary of the methods of random variable objects `rvX = X` available on all discrete probability models defined in `scipy.stats`.

variable object `rvR` using the parameters $\alpha = \text{alpha} = 1$ and $\beta = \text{beta} = 4$.

```
>>> from scipy.stats import randint
>>> alpha = 1 # start at 1
>>> beta = 4 # stop at 4
>>> rvR = randint(alpha, beta+1)
```

code
2.3.3

Note the second argument we provide `randint` is `beta+1` and not `beta`. This is because the `randint` model follows the Python convention for specifying a range of integers as not including the upper limit.

Now that we have the computer model `rvR`, we can call its methods to see the properties of the random variable $R \sim \mathcal{U}_d(1, 4)$.

```
>>> rvR.mean()
2.5
>>> rvR.var()
1.25
```

code
2.3.4

The math formulas for the mean $\mu_R = \frac{\alpha+\beta}{2} = \frac{1+4}{2} = 2.5$ and the variance $\sigma_R^2 = \frac{(\beta+1-\alpha)^2-1}{12} = \frac{16-1}{12} = 1.25$ give the same values. We can find the sample space of R by calling the method `rvR.support()`, which returns the lower and upper bounds of the sample space \mathcal{R} .

```
>>> rvR.support()
(1, 4)
```

code
2.3.5

This result tells us the sample space of R is the set of integers between 1 and 4, which we can write as $\mathcal{R} = \{1, 2, 3, 4\}$.

2.3.10 Discussion

Relations between distributions

There are many useful relations between the probability distributions we introduced in this section. Indeed, certain distributions like the Bernoulli are used as building blocks to construct the other distributions, so it can be very helpful to know about these relations.

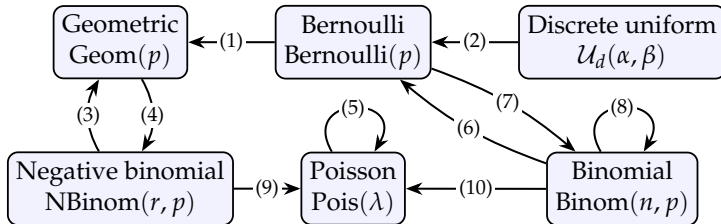


Figure 2.37: Graph of the relations between discrete distributions.

Figure 2.37 shows a graph of the most important relations between discrete random variables. The explanation for each numbered arrow in the graph is provided below.

- (1) The geometric distribution $\text{Geom}(p)$ is the waiting time until the first success in a sequence of repeated $\text{Bernoulli}(p)$ trials.
- (2) The 50-50 Bernoulli distribution $\text{Bernoulli}(p = \frac{1}{2})$ is the same as the binary discrete uniform distribution $\mathcal{U}_d(\alpha = 0, \beta = 1)$.
- (3) The negative binomial with $r = 1$ is the geometric distribution.
- (4) The negative binomial is the sum of r independent geometric random variables: $\text{NBinom}(r, p) = \sum_{i=1}^r \text{Geom}(p)$.
- (5) The sum of two independent Poisson random variables is also a Poisson random variable. Specifically, given $X_1 \sim \text{Pois}(\lambda_1)$ and $X_2 \sim \text{Pois}(\lambda_2)$, then $X_1 + X_2 \sim \text{Pois}(\lambda_1 + \lambda_2)$.
- (6) The Bernoulli distribution is a special case of the binomial distribution with $n = 1$.
- (7) The binomial distribution $\text{Binom}(n, p)$ is obtained as the sum of n independent $\text{Bernoulli}(p)$ random variables.
- (8) The sum of two independent binomial random variables is also a binomial random variable. Specifically, if $X \sim \text{Binom}(n, p)$ and $Y \sim \text{Binom}(m, p)$, then $X + Y \sim \text{Binom}(n + m, p)$.
- (9) The negative binomial with $r \rightarrow \infty$ and $p \rightarrow 1$ becomes the Poisson distribution with $\lambda = r(1 - p)$.
- (10) We can obtain the Poisson distribution from the binomial distribution by setting $\lambda = np$ and letting n grow to infinity.

2.4 Continuous random variables

Continuous random variables describe smoothly varying quantities represented by real numbers. Examples of continuous random variables include the height and weight of individuals, physical measurements like time, length, area, volume, distance, etc.

The probability concepts and formulas we'll learn in this section are analogous to the discrete random variables concepts and formulas that we saw previously in Section 2.1. Indeed, continuous random variables are conceptually similar to discrete random variables, except for the math machinery: we compute probabilities using *integrals* instead of summations.

2.4.1 Definitions

We'll start with a review of the main concepts and notation for random variables, then discuss the new math machinery for computing probabilities.

- *random variable* X : a continuously varying quantity.
- *sample space* \mathcal{X} : all possible outcomes of the random variable X . The sample space of a continuous random variable can be the set of real numbers $\mathbb{R} = (-\infty, \infty)$, or a subset of \mathbb{R} like the non-negative real numbers $\mathbb{R}_+ = [0, \infty)$, or the interval $[0, 1]$.
- *outcome* x : a particular value $\{X = x\}$ that can occur as a result of observing the random variable X .
- *event* E : a subset of the sample space. For example, the event $E = \{a \leq X \leq b\}$ describes all outcomes of the random variable X that fall between a and b .
- f_X : the *probability density function* of the random variable X .

The probability density function, often abbreviated as PDF, is a function of the form $f_X : \mathcal{X} \rightarrow \mathbb{R}_+$ that assigns a nonnegative number to each outcome x in the sample space \mathcal{X} . The shape of the function f_X tells us which regions of the sample space contain the more likely outcomes of the random variable X , and which regions contain the less likely ones. Figure 2.38 (a) shows an example of a probability density function f_X for a continuous random variable defined on the sample space $\mathcal{X} = [0, \infty)$.

Computing probabilities of events We compute the probability of the event $\{a \leq X \leq b\}$ by *integrating* the probability density function

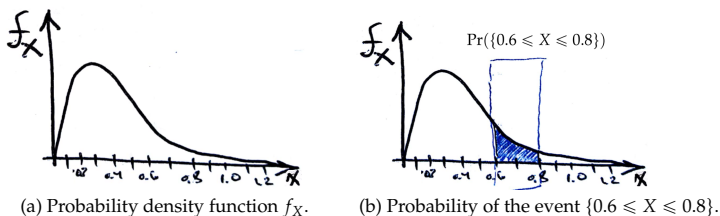


Figure 2.38: The probability density function f_X of some random variable X . The total area under the graph of f_X is 1. The highlighted area corresponds to the probability calculation $\Pr(\{0.6 \leq X \leq 0.8\}) = \int_{x=0.6}^{x=0.8} f_X(x) dx$.

f_X between $x = a$ and $x = b$:

$$\Pr(\{a \leq X \leq b\}) \stackrel{\text{def}}{=} \int_{x=a}^{x=b} f_X(x) dx.$$

Intuitively, the integral of f_X from $x = a$ until $x = b$ computes the total amount of density that lies in this interval of x -values.

The Figure 2.38 (b) illustrates the probability calculation for the event $\{0.6 \leq X \leq 0.8\}$ which includes all outcomes of the random variable X between 0.6 and 0.8. To compute the probability of the event $\{0.6 \leq X \leq 0.8\}$, we calculate the integral $\int_{x=0.6}^{x=0.8} f_X(x) dx$. The value of this integral corresponds to the area under the graph of the function f_X between $x = 0.6$ and $x = 0.8$.

The probability density function f_X of the random variable X satisfies Kolmogorov's axioms of probability:

- Nonnegativity: $f_X(x) \geq 0$ for all $x \in \mathcal{X}$.
- Unit total: $\int_{x \in \mathcal{X}} f_X(x) dx = 1$.

The first axiom states that probability functions cannot take on negative values. The second axiom states that the total amount of probability distributed over the whole sample space is 1.

* * *

If this is the first time you're seeing an integral $\int \cdot dx$, you might be wondering if this is not some sort of complicated new development. No worries! The integral $\int_{x=a}^{x=b} g(x) dx$ is just the continuous version of the summation $\sum_{x=a}^{x=b} h(x)$, and it represents the operation of "adding up" the total of some continuous quantity $g(x)$. In the next subsection, we'll introduce integral notation and explain what you need to know about integrals to do calculations with continuous random variables.

2.4.2 Calculus prerequisites

Integration is the operation of computing the “total amount accumulated” of some quantity over a range of inputs. Learning to compute integrals by hand requires taking an entire university-level course, but the basic idea is very simple, as you’ll see next. We have Python on our side, which means we can use the computer to calculate integrals for us.

Functions

Consider the function f that takes real numbers as inputs and produces real numbers as outputs, $f : \mathbb{R} \rightarrow \mathbb{R}$. Functions are usually defined as math formulas that tell us how to compute the output $f(x)$ for a given input x . We can also define functions in code. For example, the code below defines the Python function g that implements the same calculation as the math function $g(x) = \frac{1}{2}x$.

```
code >>> def g(x):
2.4.1         return 0.5 * x
```

We can evaluate the function g on the input $x = 5$ as follows:

```
code >>> g(5)
2.4.2 2.5
```

The output of g for the input $x = 5$ is $g(5) = \frac{1}{2}(5) = 2.5$.

We can visualize the output values of the function g for a whole range of inputs by plotting the *function graph*, which is a curve that passes through all the coordinates pairs $(x, g(x))$ in the Cartesian plane. The code below shows how to plot the graph of the function g .

```
code >>> import numpy as np
2.4.3 >>> xs = np.linspace(0, 10, 1000)
>>> gs = g(xs)
>>> import seaborn as sns
>>> sns.lineplot(x=xs, y=gs)
The plot is shown in Figure 2.39.
```

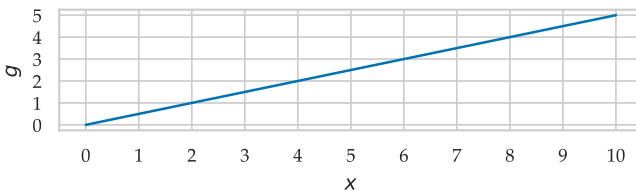


Figure 2.39: Graph of the function $g(x) = \frac{1}{2}x$ between $x = 0$ and $x = 10$.

The first line in code block 2.4.3 imports the module `numpy` under the alias `np`. We then use the function `np.linspace` to create an array

(a list of numbers) `xs` that contains 1000 inputs ranging from $x = 0$ to $x = 10$. Next, we apply the function g to the array of inputs `xs` and store the result in the array `gs`. At this point, the arrays `xs` and `gs` contain 1000 input-output pairs of the form $(x, g(x))$, which is the data we need to plot the graph of the function g . We call the Seaborn function `sns.lineplot` setting the x -coordinates to `xs` and the y -coordinates to `gs` to produce the line plot shown in Figure 2.39.

This sequence of three operations: (1) creating a list of inputs, (2) evaluating the function on the inputs, and (3) calling `sns.lineplot` is very common. We'll use similar operations repeatedly in the rest of the book to visualize functions and probability distributions.

Integrals

The integral of the function $f(x)$ from $x = a$ to $x = b$ is denoted $\int_{x=a}^{x=b} f(x) dx$. You can think of the integration sign \int as a continuous version of the summation symbol \sum . Indeed, the integral $\int_{x=a}^{x=b} f(x) dx$ of the continuous function $f(x)$ is directly analogous to the summation $\sum_{i=a}^{i=b} a_i$ of the sequence of numbers a_i . In both cases, we're computing the total of some quantity accumulated between the starting point a and the stopping point b .

Geometrically speaking, this integral corresponds to computing the *area* enclosed below the graph of $f(x)$ between $x = a$ and $x = b$:

$$A_f(a, b) = \int_{x=a}^{x=b} f(x) dx.$$

We refer to the numbers a and b as the *limits of integration*. If this is the first time you're seeing integrals, it's understandable if you feel intimidated by the fancy-looking math notation, but, trust me, there is nothing to worry about! An integral is just a "continuous sum" of all the values of $f(x)$ between $x = a$ and $x = b$.

For example, the area under $g(x) = \frac{1}{2}x$ between $x = 2$ and $x = 6$ corresponds to the following integral calculation:

$$A_g(2, 6) = \int_{x=2}^{x=6} g(x) dx.$$

Figure 2.40 shows the area under the graph of $g(x)$ between $x = 2$ and $x = 6$. The region $A_g(2, 6)$ has a simple geometrical shape (a trapezoid), which allows us to compute its area $\int_{x=2}^{x=6} g(x) dx = 8$. See exercise EXX for the details.

Taking an integral calculus course would teach you various techniques and procedures for computing integrals using pen and

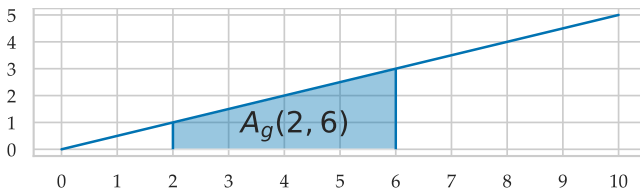


Figure 2.40: Integral of the function $g(x)$ between $x = 2$ and $x = 6$.

paper. Outside of a calculus class, however, we rarely have to compute integrals by hand, and instead rely on computers to calculate integrals for us.

The Python function `quad` from the `scipy.integrate` module allows us to compute the integral of any function. The name `quad` is short for “quadrature,” which is the historical math term used for find-the-area procedures. For example, here is the code to calculate the area under the graph of the function $g(x) = \frac{1}{2}x$ between $x = 2$ and $x = 6$.

```
code >>> from scipy.integrate import quad
2.4.4 >>> quad(g, a=2, b=6)[0]
      8.0
```

We call the function `quad` with the function `g` as the first argument, and the limits of integration $a = 2$ and $b = 6$ as the second and third arguments.

We’ll use the function `quad` often in the remainder of the book to compute various integrals as part of probability and statistics calculations. Make sure you understand what is going on in the above code example. The main takeaway message is that the `quad` function is your friend: all the scary-looking math equations that contain the \int symbol can be computed using one line of Python code.

Integral functions

The *integral function* $F(b)$ corresponds to the area calculation with a variable upper limit of integration $A_f(0, b)$. The variable b that we use to denote the input of the integral function F corresponds to the upper limit of integration in the following calculation:

$$F(b) \stackrel{\text{def}}{=} A_f(0, b) = \int_{x=0}^{x=b} f(x) dx.$$

The choice of $x = 0$ as the lower limit of integration is arbitrary. We can define an alternative integral function by using a different starting point, but the difference will be an additive constant. The math convention is to denote the integral function using the capital

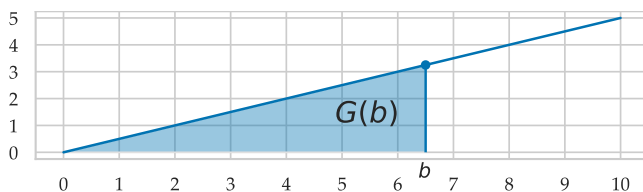


Figure 2.41: The integral function $G(b) = \int_{x=0}^b g(x) dx$ calculates the area under $g(x)$ as a function of the upper limit of integration b .

of the letter used to denote the original function. For example, the integral function of the function $g(x) = \frac{1}{2}x$ is $G(b) = A_g(0, b) = \frac{1}{4}b^2$. Figure 2.41 shows an illustration of the function $G(b)$, which is the area under $g(x)$ between $x = 0$ and $x = b$.

The integral function $F(b)$ contains the “precomputed” information about the area under the graph of $f(x)$ for *all* possible limits of integration. We can obtain the area under $f(x)$ between $x = a$ and $x = b$ by calculating the *change* in the integral function as follows:

$$A_f(a, b) = \int_{x=a}^{x=b} f(x) dx = F(b) - F(a).$$

This formula computes the area $A_f(a, b)$ as the difference between the areas of two regions: the area until $x = b$ minus the area until $x = a$, as illustrated in Figure 2.42.

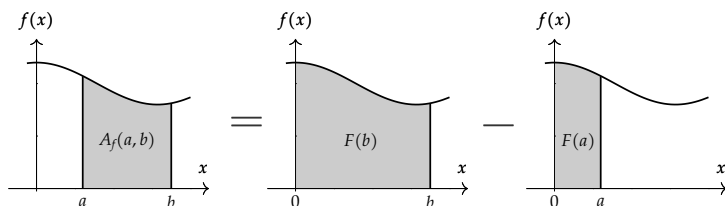


Figure 2.42: The area under the graph of $f(x)$ between $x = a$ and $x = b$ can be computed using the formula $A_f(a, b) = F(b) - F(a)$, which describes the change in the value of the integral function $F(x)$ between $x = a$ and $x = b$.

For example, knowing the integral function $G(b) = \frac{1}{4}b^2$ allows us to find the area under the graph of $g(x)$ between any $x = a$ and $x = b$ by computing the difference $A_g(a, b) = G(b) - G(a) = \frac{1}{4}b^2 - \frac{1}{4}a^2$. Let’s use this formula to reproduce the area calculation $A_g(2, 6)$ that we computed earlier. The formula $A_g(2, 6) = G(6) - G(2)$ gives the same answer $A_g(2, 6) = \frac{1}{4}6^2 - \frac{1}{4}2^2 = \frac{36}{4} - 1 = 8$.

I hope the above calculations and visualizations helped you see that the scary-looking integral sign \int is not that complicated. It’s just

Bulk of the normal distribution

We're often interested in calculating an interval that contains “the bulk” of the distribution f_X . We want to find an interval (a subset of the sample space) where most of the observations of the random variable X will fall. One way to construct such a high-density interval is to define it in terms of the mean μ_X and the standard deviation σ_X of the random variable. For example, we can define the interval I_n that contains all values that are within n standard deviations of the mean of the random variable X as follows:

$$I_n = [\mu_X - n\sigma_X, \mu_X + n\sigma_X].$$

We can then compute the probability $\Pr(\{X \in I_n\})$, which tells us how likely is X to be within n standard deviations of its mean.

For a normally distributed random variable like the random variable $K \sim \mathcal{N}(\mu_K = 1000, \sigma_K = 10)$ that we saw in Example 2, we can calculate the value of the probability $\Pr(\{K \in I_n\})$ numerically:

$$\Pr(\{K \in I_n\}) = \Pr(\{\mu_K - n\sigma_K \leq K \leq \mu_K + n\sigma_K\}) = p_n,$$

where $p_n = 0.682$ for $n = 1$, $p_n = 0.954$ for $n = 2$, and $p_n = 0.997$ for $n = 3$. See Figure 2.52 for an illustration. These total-probability-within- n -standard-deviations-of-the-mean values are the same for all normally distributed variables, and is sometimes called the “68-95-99.7 rule.” The interval I_1 is not particularly interesting, since it covers only 68.2% of the probability mass of the random variable X . The interval I_2 contains 95.4% of the total weight of the distribution, which is “most” of it. Intuitively, if we predict the event $\{K \in I_2\}$ will occur, we'll be correct 95.4% of the time.

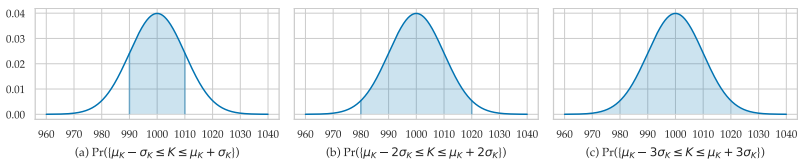


Figure 2.52: The probability of observing the random variable K within n standard deviations of the mean. In (a) we show the calculation $\Pr(\{990 \leq K \leq 1010\}) = 0.682$, which is roughly two thirds of the total mass of the distribution. The shaded region in (b) show the calculation $\Pr(\{980 \leq K \leq 1020\}) = 0.954$, which means 95.4% of the probability mass of f_K is situated within two standard deviations of the mean. The shaded region in (c) shows the three-sigma confidence interval that contains 99.7% of the probability.

The technical term for the interval I_n is *confidence interval*, meaning we're confident, to a certain degree of probability p_n , that future

outcomes of the random variable K will fall in this interval. We say $I_2 = [\mu_K - 2\sigma_K, \mu_K + 2\sigma_K] = [980, 1020]$ is a 95.4% confidence interval for the random variable K . This means, if we generate thousands of observations from the random variable K , in the long run, 95.4% of these observations will be contained in the interval $[\mu_K - 2\sigma_K, \mu_K + 2\sigma_K]$. For an even higher degree of “confidence,” we can choose the three-sigma interval $I_3 = [\mu_K - 3\sigma_K, \mu_K + 3\sigma_K] = [970, 1030]$, which contains 99.7% of all observations.

Confidence intervals play an important role in statistics, where they are used to report the uncertainty in estimates.

Tails of the normal distribution

The “tails” of the distribution contain the unlikely outcomes for the random variable. Figure 2.53 shows the tails of the distribution f_K , which are defined as the observations that are more than n standard deviations away from the mean.

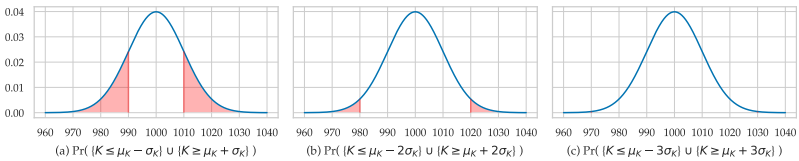


Figure 2.53: Illustration of the tails of the distribution f_K for the random variable $K \sim \mathcal{N}(1000, 10)$. The highlighted regions correspond to the complements of the regions highlighted in Figure 2.52. Subplot (b) shows that the probability of observing values of K that are more than two standard deviations away from the mean μ_K is 4.55%, which we can qualify as “very unlikely.” Subplot (c) shows the probability of observing K more than three standard deviations away from its mean is 0.27%, which we can qualify as “extremely unlikely.”

The observations in the tails of the distribution are deemed “surprising” or “unexpected.” The notion of “unexpected events” plays a central role in the statistical concept of hypothesis testing in Part 2 of the book.

If we observe an outcome that is part of the tails of some distribution, we’ll label it “unexpected” and interpret this result as interesting (statistically significant), and worthy of reporting and publication in scientific journals. The whole hypothesis testing procedure can be summarized as a “is it in the tail of the distribution” check, where the distribution in question is specially designed to model the probability of different outcomes under the current theory. If the observed outcome is “unexpected” under the current theory, then this lends support to the need to seek alternative theories.

2.5 Multiple continuous random variables

Let's now talk about the pair of continuous random variables (X, Y) defined over the *joint sample space* $\mathcal{X} \times \mathcal{Y}$. The *outcomes* in the joint sample space consist of a pair of real numbers (x, y) . Geometrically speaking, the joint sample space is a two-dimensional region, which is some subset of the Cartesian plane $\mathbb{R} \times \mathbb{R}$. This section will be a “rerun” of what we learned about multiple random variables in Section 2.2, but this time we'll show the formulas for continuous random variables.

2.5.1 Joint probability density function

The *joint probability density function* f_{XY} describes the variability of the random variables X and Y . By choosing the appropriate function f_{XY} , we can describe and model various relationships between the two random variables. Since the probability density function $f_{XY}(x, y)$ has two inputs, we plot its graph as a three-dimensional surface, as shown in Figure 2.54 (a). The height of the surface is given by $f_{XY}(x, y)$, for each coordinate (x, y) in the two-dimensional sample space.

We can also represent the probability density function f_{XY} using a *contour plot*, in which dark-shaded regions indicate higher probability density, as shown in Figure 2.54 (b). The contour plot allows us to see more clearly the joint variability of the random variables X and Y : larger X values are associated with larger Y values, so it suggests a linear relationship between these two variables.

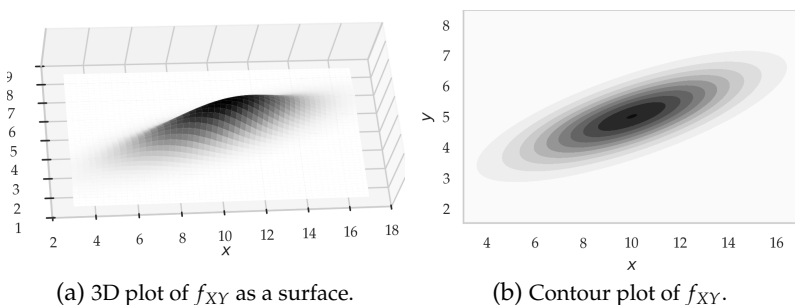


Figure 2.54: Graphical representations of a joint probability density function f_{XY} . In (a) we plot the graph of f_{XY} as a surface in a three-dimensional space. In (b) we see a plot of f_{XY} in two dimensions (as if looking from above), where the third dimension is represented using shading. The dark-shaded regions correspond to locations with higher probability density.

Events are represented as subsets of the joint sample space. For

The idea for a marginal distribution f_X is to get rid of the Y randomness. The marginal f_X corresponds to a description of the random variable X when the random variable Y is unknown. The name *marginal distribution* comes from the procedure we use to compute it, by “summing” over all y values for a given x and writing the total in the margin. See Figure 2.20 for an illustration. Recall also Table 1.9 on page 79, where we used a similar procedure to compute the marginal frequencies in a two-way table.

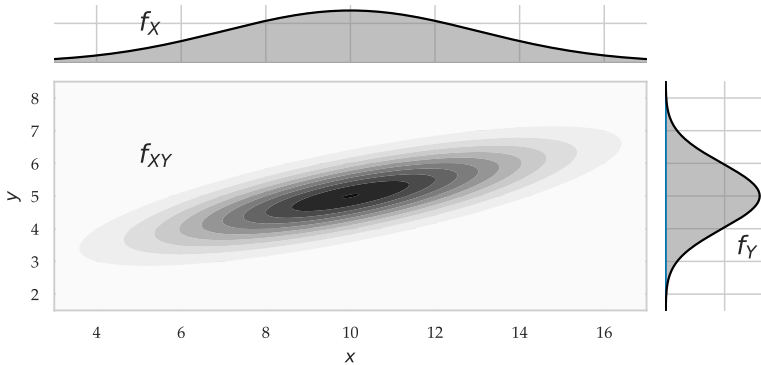


Figure 2.55: The top panel shows the marginal distribution f_X , which is obtained by integrating f_{XY} over all values of Y . The right panel shows the marginal distribution f_Y obtained by integrating f_{XY} over all values of X .

Similarly, the marginal distribution f_Y is obtained from the joint distribution f_{XY} by integrating over all possible values of X : $f_Y(y) = \int_{x=-\infty}^{x=\infty} f_{XY}(x, y) dx$. The marginal distribution f_Y describes the uncertainty about Y when we don’t know the value of X .

For example, the marginal distribution f_X computed from the bivariate normal distribution f_{XY} that we defined in the previous section is

$$f_X(x) = \frac{1}{\sigma_X \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{x - \mu_X}{\sigma_X} \right)^2},$$

which is the normal distribution $\mathcal{N}(\mu_X, \sigma_X)$. The Y -marginal is also a normal distribution $Y \sim \mathcal{N}(\mu_Y, \sigma_Y)$. In other words, when considering the random variables X and Y individually, they are described by normal distributions.

2.5.4 Conditional probability distributions

The *conditional probability density* functions $f_{X|Y}$ and $f_{Y|X}$ describe the uncertainty that remains in one random variable after we have observed the value of the other random variable. The conditional

distributions are defined as follows:

$$f_{X|Y=y}(x|y) = \frac{f_{XY}(x,y)}{f_Y(y)} \quad \text{and} \quad f_{Y|X=x}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)}.$$

The vertical bar is pronounced “given” and describes situations where the value of the random variable after the vertical bar is known. For example, the conditional distribution $f_{Y|X}(y|x_a)$ describes the probabilities of the random variable Y , given we know the value of the random variable X is x_a .

Figure 2.56 illustrates the two-step process we use to plot the conditional distribution $f_{Y|X=x_a}(y|x_a)$, at several different values of x_a . Given the observed value $X = x_a$ of the random variable X , the remaining uncertainty about the variable Y corresponds to a “slice” through the joint probability density function $f_{XY}(x_a, y)$. The slices corresponding to different values of x_a are shown in Figure 2.56 (a). These slices are not properly normalized: the area of the slice depends on whether we’re slicing through a high-density region or a low-density region. To normalize the slices, we divide by the marginal distribution $f_X(x_a)$, which produces the conditional distributions $f_{Y|X=x_a}(y|x_a) = \frac{f_{XY}(x_a, y)}{f_X(x_a)}$ illustrated in Figure 2.56 (b).

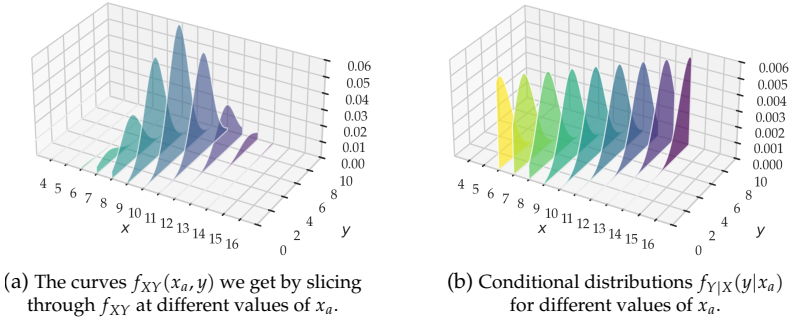


Figure 2.56: Conditional probability distributions $f_{Y|X=x_a}$ are obtained by slicing the joint probability distribution at different values of x_a . The left graph shows the shape of the slices before normalization. The right side shows the normalized slices, which are proper probability distributions.

We obtain a different conditional distribution $f_{Y|X=x_a}$ for each of the possible values of $x_a \in \mathcal{X}$. When the joint distribution f_{XY} is the bivariate normal $\mathcal{N}(\mu_X, \mu_Y, \sigma_X, \sigma_Y, \rho)$, we can use algebra to obtain the exact formula for the conditional distributions $f_{Y|X=x_a}(y|x_a)$. For example, when starting from the joint distribution $\mathcal{N}(\mu_X = 10, \mu_Y = 5, \sigma_X = 3, \sigma_Y = 1, \rho = 0.75)$, the conditional distributions $f_{Y|X=x_a}(y|x_a)$

2.6 Inventory of continuous distributions

We'll now continue the work we started in Section 2.3 to describe the most important probability distributions used in statistics. In the previous section, we saw examples involving the uniform distribution $\mathcal{U}(\alpha, \beta)$ and the normal distribution $\mathcal{N}(\mu, \sigma)$, but there are several other continuous distributions that you need to know about.

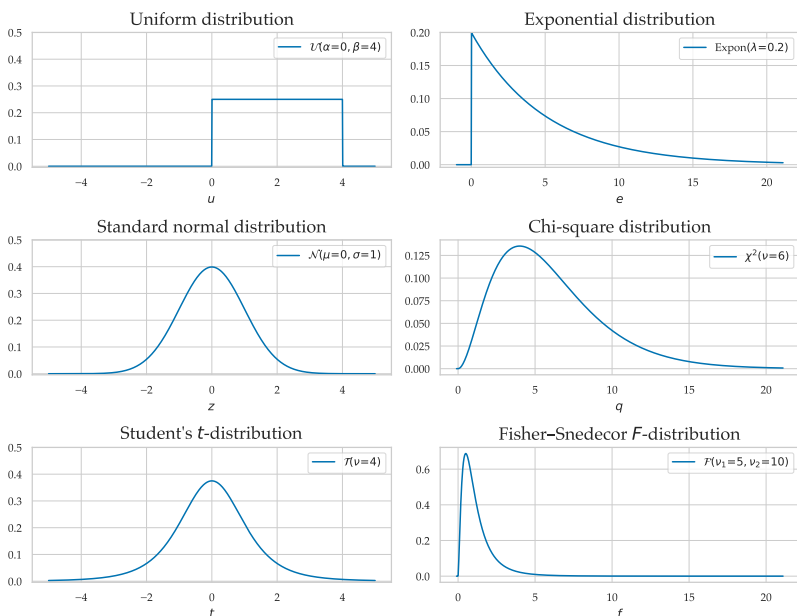


Figure 2.57: Probability density plots of six common continuous distributions. Note the variety of the shapes of the probability densities.

Figure 2.57 shows six examples of continuous distributions:

- The uniform distribution $\mathcal{U}(\alpha, \beta)$ assigns constant probability density over the interval $[\alpha, \beta]$.
- The exponential $\text{Expon}(\lambda)$ is a steadily decaying probability curve defined for nonnegative real numbers.
- Standard normal $\mathcal{N}(\mu = 0, \sigma = 1)$ distribution is a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$.
- Student's t -distribution $\mathcal{T}(\nu)$ is similar to the standard normal but has heavier tails.
- The chi-square χ^2 -distribution describes the sum of squared deviations from an expected value.
- The Fisher–Snedecor F -distribution $F(\nu_1, \nu_2)$ describes ratios of variances.

We'll also discuss the beta distribution $\text{Beta}(\alpha, \beta)$, the gamma distribution $\text{Gamma}(\alpha, \lambda)$, and the Laplace distribution $\text{Laplace}(\mu, b)$, which are used as building blocks in certain statistical models.

Computer models for continuous random variables

The module `scipy.stats` allows us to build computer models for all the continuous probability distributions we'll discuss in this section. Table 2.4 shows a list of the most common probability distributions and the name of the corresponding `scipy.stats` model family.

Math notation	Code notation	Parameters	Notes
$\mathcal{U}(\alpha, \beta)$	<code>uniform(a, b-a)</code>	$\alpha, \beta \in \mathbb{R}$	
$\text{Expon}(\lambda)$	<code>expon(0, 1/lam)</code>	$\lambda \in \mathbb{R}_+$	
$\mathcal{N}(\mu, \sigma)$	<code>norm(mu, sigma)</code>	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}_+$	
$Z \sim \mathcal{N}(0, 1)$	<code>norm(0, 1)</code>	$\mu = 0, \sigma = 1$	
$T_v \sim \mathcal{T}(v)$	<code>tdist(df)</code>	$v \in \mathbb{R}_+$	alias for <code>t</code>
$Q_v \sim \chi^2(v)$	<code>chi2(df)</code>	$v \in \mathbb{N}_+$	
$F \sim \mathcal{F}(v_1, v_2)$	<code>fdist(df1, df2)</code>	$v_1, v_2 \in \mathbb{R}_+$	alias for <code>f</code>
$\text{Gamma}(\alpha, \lambda)$	<code>gammadist(a, 0, 1/lam)</code>	$\alpha, \lambda \in \mathbb{R}_+$	alias for <code>gamma</code>
$\text{Beta}(\alpha, \beta)$	<code>betadist(a, b)</code>	$\alpha, \beta \in \mathbb{R}_+$	alias for <code>beta</code>
$\text{Laplace}(\mu, b)$	<code>laplace(mu, b)</code>	$\mu \in \mathbb{R}, b \in \mathbb{R}_+$	

Table 2.4: Computer models for continuous distributions in `scipy.stats`.

Once you have created a random variable object for one of these distributions, you can use its methods to do probability calculations. See Table 2.3 on page 182 for a reminder of the methods available on `scipy.stats` computer models.

* * *

We now switch to “reference mode” and present the essential characteristics of each probability distribution in point form. These distributions are essential building blocks that we'll use for the data modelling and statistical inference tasks in Part 2 of the book, so it's a good idea that you get to know them. Remember that you're not supposed to memorize the formulas and definitions: you just need to know that these distributions exist, and have an idea of the shapes of their probability density functions.

2.6.1 Uniform distribution

The continuous uniform distribution $\mathcal{U}(\alpha, \beta)$ assigns equal probability to all values on the interval $[\alpha, \beta]$. The probability density function of the random variable $X \sim \mathcal{U}(\alpha, \beta)$ is

$$f_X(x) \stackrel{\text{def}}{=} \begin{cases} \frac{1}{\beta - \alpha} & \text{when } \alpha \leq x \leq \beta, \\ 0 & \text{when } x < \alpha \text{ or } x > \beta. \end{cases}$$

Each x between α and β is equally likely to occur, while values of x outside this range have zero probability of occurring.

Figure 2.58 shows various uniform distributions defined over different intervals $[\alpha, \beta]$. The longer the length of the interval $(\beta - \alpha)$, the lower the probability density, so that the total area remains 1.

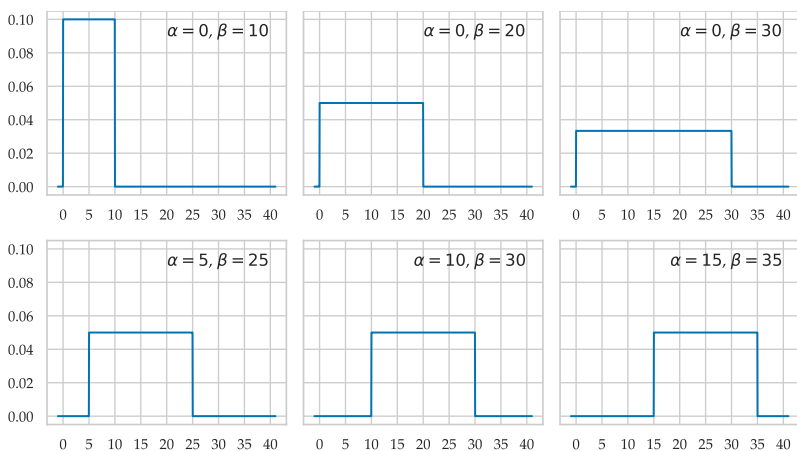


Figure 2.58: Uniform distributions with different parameters α and β .

The mean of the uniform distribution is $\mu_X = \frac{\alpha + \beta}{2}$ and its variance is $\sigma_X^2 = \frac{(\beta - \alpha)^2}{12}$. I'll ask you to verify these formulas in exercise E2.28 and problem P2.16.

Example Figure 2.59 shows side-by-side plots of the probability density function (PDF) f_U and the cumulative distribution function (CDF) $F_U(b) \stackrel{\text{def}}{=} \Pr(\{U \leq b\}) = \int_{-\infty}^b f_U(u) du$ of the uniform random variable $U \sim \mathcal{U}(\alpha = 2, \beta = 7)$. The cumulative distribution function F_U has a very simple shape: a straight line that increases from 0 to 1 as b varies over the sample space $[\alpha, \beta]$.

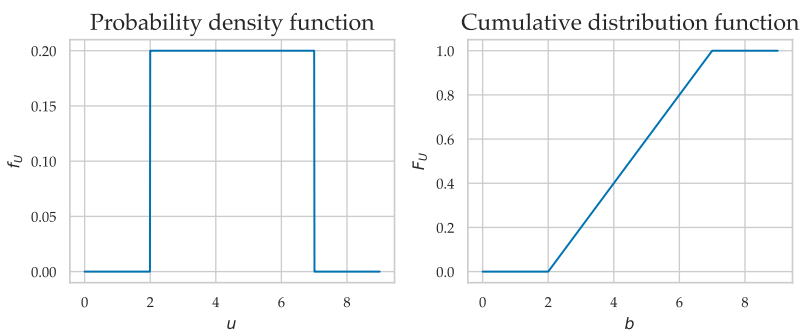


Figure 2.59: Graphs of the probability density function f_U and the cumulative distribution function F_U of the random variable $U \sim \mathcal{U}(\alpha=2, \beta=7)$.

Computer model We can use the uniform model family from `scipy.stats` to create a computer model for a uniformly distributed random variable. For example, to create a computer model for the random variable $U \sim \mathcal{U}(\alpha=2, \beta=7)$, use the following code:

```
>>> from scipy.stats import uniform
>>> alpha = 2
>>> beta = 7
>>> rvU = uniform(alpha, beta-alpha)
```

code
2.6.1

Note we specify the distance $(\beta - \alpha)$ as the second argument to `uniform`, which is different from the math definition $U \sim \mathcal{U}(\alpha, \beta)$.

The standard uniform The *standard uniform distribution* is a special case of the uniform distribution with parameters $\alpha=0$ and $\beta=1$.

Applications The uniform distribution models the general process of choosing at random from a continuous sample space. For example, the random variable $U \sim \mathcal{U}(0, 10)$ models how long you'll have to wait for a bus that runs every 10 minutes. The uniform distribution is also used as a building block to generate random observations from other distributions. We'll discuss the generation of observations from random variables in Section 2.7.

[Additional info about the uniform distribution from Wikipedia]
https://wikipedia.org/wiki/Continuous_uniform_distribution

[More explanations about the uniform distribution by jbststatistics]
<https://www.youtube.com/watch?v=-qt8CPIadWQ>

2.6.6 Student's t -distribution

The shape of the Student's t -distribution $\mathcal{T}(\nu)$ is similar to the shape of the standard normal distribution Z , but with “heavier” tails. Intuitively, when a distribution has heavy tails, it means it assigns more probability to outcomes that are far from the mean.

There are different t -distributions defined by the *degrees of freedom* parameter ν (the Greek letter *nu*), which determines the heaviness of the tails. The probability density function of the random variable $T_\nu \sim \mathcal{T}(\nu)$ is described by the following equation:

$$f_{T_\nu}(x) \stackrel{\text{def}}{=} \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi} \Gamma(\frac{\nu}{2})} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}.$$

Don't worry about this scary-looking math expression—you won't ever need to compute its values by hand.

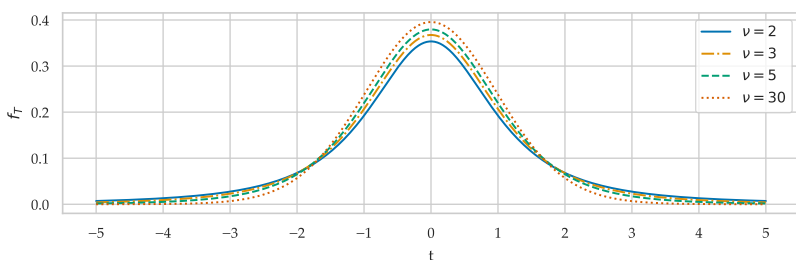


Figure 2.63: Plot of the probability densities of Student's t -distribution for four choices of the degrees of freedom ν . When ν is large the t -distribution T_ν is almost indistinguishable from the standard normal $Z \sim \mathcal{N}(0, 1)$.

The mean of Student's t -distribution is $\mu_{T_\nu} = 0$, and its variance is $\sigma_{T_\nu}^2 = \frac{\nu}{\nu-2}$ when $\nu > 2$. The variance is not defined when $\nu \leq 2$.

Historical background Student's t -distribution was invented by William Gosset while working on quality control at the Guinness brewing company. He published a paper[Stu08] under the pseudonym “Student,” so he was never personally credited for this discovery. Presumably, he used a pseudonym so readers would not be able to tie the analysis to his employer, otherwise people would storm the Guinness brewery asking for the “bad” batches!

Applications We use Student's t -distribution in statistical inference procedures that rely on the sample variance s_X^2 as a substitute for the population variance σ_X^2 . The “heavy tails” of the t -distribution compensate for the extra uncertainty associated with using the estimated sample variance s_X^2 instead of the true population variance σ_X^2 .

[Wikipedia page about Student's t -distribution]

https://wikipedia.org/wiki/Student's_t-distribution

2.6.7 Chi-square distribution

The chi-square distribution describes the variability of the sum of squared deviations from the mean. We use the notation $Q_\nu \sim \chi^2(\nu)$ to denote a chi-square random variable with ν *degrees of freedom*, where χ is the Greek letter *chi* (rhymes with “bye”). The random variable Q_ν describes the sum ν squared standard normal random variables: $Q_\nu \stackrel{\text{def}}{=} (Z_1^2 + Z_2^2 + \cdots + Z_\nu^2) \sim \chi^2(\nu)$, where $Z_i \sim \mathcal{N}(0, 1)$. The probability density function of the random variable Q_ν is

$$f_{Q_\nu}(q) \stackrel{\text{def}}{=} \frac{1}{2^{\frac{\nu}{2}} \Gamma(\nu/2)} q^{\frac{\nu}{2}-1} e^{-\frac{q}{2}}.$$

The parameter ν (Greek *nu*) determines the shape of the distribution. The mean of $Q_\nu \sim \chi^2(\nu)$ is $\mu_{Q_\nu} = \nu$ and its variance is $\sigma_{Q_\nu}^2 = 2\nu$.

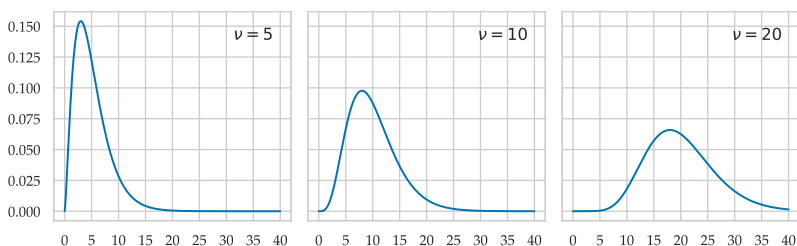


Figure 2.64: Plot of the chi-square distribution for different choices of the parameter ν . As ν gets larger, the peak of the distribution moves to the right.

Don’t be intimidated by the Greek letter χ (*chi*) and the fancy-looking formula for the PDF f_{Q_ν} . The χ^2 -distribution is not a new thing, but a convenient label for the sum of squares of independent standard normal variables. For example, consider the normal random variables $Z_a \sim \mathcal{N}(0, 1)$ and $Z_b \sim \mathcal{N}(0, 1)$. The sum of the squares of these random variable $Z_a^2 + Z_b^2$ is distributed according to $\chi^2(\nu=2)$.

Applications We use the χ^2 -distribution to model the variability of the sum of squared deviations from some expected value. For example, the sample variance s_x^2 is computed from the squared deviations of the individual observations x_i from the sample mean \bar{x} . The sample variance estimates we might obtain from different samples from a normal population are distributed according to a scaled chi-square distribution with $\nu = n - 1$ degrees of freedom. We can use the χ^2 -distribution even when the deviations are only approximately normal. For example, we can use the χ^2 -distribution to model the variability of the sum of squared deviations between the “observed” counts and the “expected” counts under some model.

[Read more about the χ^2 -distribution on Wikipedia]

https://wikipedia.org/wiki/Chi-square_distribution

2.6.8 Fisher–Snedecor F -distribution

The Fisher–Snedecor distribution $F(\nu_1, \nu_2)$ models the variability of ratios of variances. The F -distribution has two degrees of freedom parameters ν_1 and ν_2 , which are sometimes called the numerator and denominator degrees of freedom, respectively. The probability density function of the random variable $F \sim F(\nu_1, \nu_2)$ is

$$f_F(x) \stackrel{\text{def}}{=} \frac{\Gamma(\frac{\nu_1+\nu_2}{2})}{\Gamma(\frac{\nu_1}{2})\Gamma(\frac{\nu_2}{2})} \left(\frac{\nu_1}{\nu_2}\right)^{\nu_1/2} \frac{x^{(\nu_1-2)/2}}{(1+\frac{\nu_1}{\nu_2}x)^{(\nu_1+\nu_2)/2}}.$$

The sample space of the distribution is $[0, \infty)$, its mean is $\mu_F = \frac{\nu_2}{\nu_2-2}$, for $\nu_2 > 2$, and its variance is $\sigma_F^2 = \frac{2\nu_2^2(\nu_1+\nu_2-2)}{\nu_1(\nu_2-2)^2(\nu_2-4)}$, for $\nu_2 > 4$.

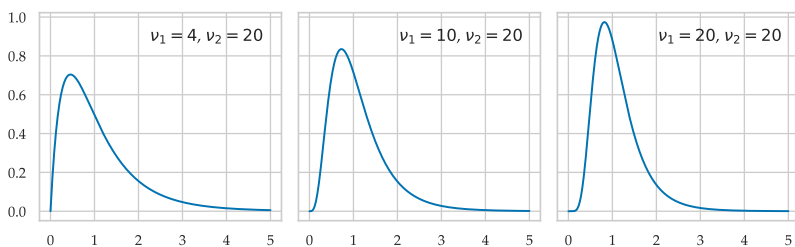


Figure 2.65: Plot of the probability densities of the F -distribution for different choices of the parameters ν_1 and ν_2 .

Most people, including the author, when faced with the complicated math in the definition of the PDF f_F will wag a finger and say “Uh-huh, not for me!” The F -distribution was a mystery for me up until recently. We can get over this fear if we treat the F -distribution as a convenient label for a particular combination of independent standard normal variables. Specifically, the F -distribution describes the average of ν_1 independent squared normals divided by the average of ν_2 independent squared normals. For example, the distribution $F(\nu_1 = 2, \nu_2 = 3)$ describes the variability of the average of two squared standard normal variables, divided by the average of three other squared standard normals: $\frac{\frac{1}{2}(Z_1^2 + Z_2^2)}{\frac{1}{3}(Z_3^2 + Z_4^2 + Z_5^2)} \sim F(\nu_1 = 2, \nu_2 = 3)$.

Applications The F -distribution models ratios of variance-like quantities, and is used for the analysis of variance (ANOVA) statistical tests, which we’ll learn about in Part 2 of the book.

[Learn more about the F -distribution from the Wikipedia page]
<https://wikipedia.org/wiki/F-distribution>

is to look for z-scores $|z_i| \geq 3$, which correspond to observations x_i that are more than three sample standard deviations s_x away from the sample mean \bar{x} .

Another use case for z-scores is to compare measurements performed on different scales. For example, ACT scores range between 1 and 36, while SAT scores range between 400 and 1600. It doesn't make sense to compare an ACT score to a SAT score directly, but if we convert both scores to z-scores, we can compare them since z-scores represent "standardized" performance scores measured in units of one standard deviation relative to the mean score on each exam.

2.6.13 Discussion

Relations between distributions

Figure 2.69 describes the relations and equivalences between the continuous distributions we introduced in this section.

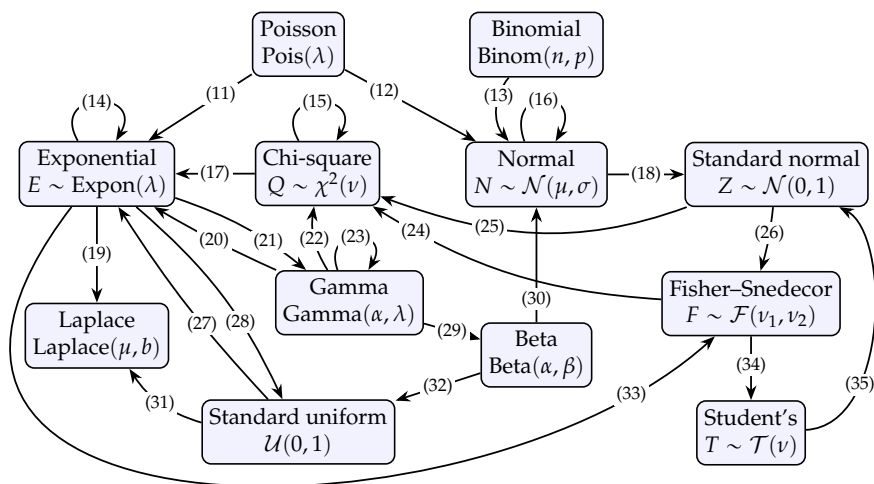


Figure 2.69: Graph of the relations between continuous distributions.

The list below provides the explanations for the numbered arrows.

- (11) The Poisson distribution $\text{Pois}(\lambda)$ counts the number of events that occur during some time interval. The exponential distribution $\text{Expon}(\lambda)$ describes the time between these events.
- (12) We can approximate the Poisson distribution by a normal distribution for large values of λ : $\text{Pois}(\lambda) \approx \mathcal{N}(\mu = \lambda, \sigma = \sqrt{\lambda})$.
- (13) We can approximate the binomial distribution by a normal for large values of n : $\text{Binom}(n, p) \approx \mathcal{N}(\mu = np, \sigma = \sqrt{np(1-p)})$.

- (14) Given two independent exponential random variables $E_1 \sim \text{Expon}(\lambda_1)$ and $E_2 \sim \text{Expon}(\lambda_2)$, the minimum of the two is also exponentially distributed $\min(E_1, E_2) \sim \text{Expon}(\lambda_1 + \lambda_2)$.
- (15) Given a list of χ^2 random variables $Q_i \sim \chi^2(\nu_i)$, their sum is also χ^2 -distributed: $Q_1 + Q_2 + \cdots + Q_n \sim \chi^2(\nu_1 + \nu_2 + \cdots + \nu_n)$.
- (16) Given n independent normal random variables $N_i \sim \mathcal{N}(\mu, \sigma)$, their sum is also normally distributed: $N_1 + N_2 + \cdots + N_n \sim \mathcal{N}(n\mu, \sqrt{n}\sigma)$. This property is the inspiration to the *central limit theorem*, which we'll learn about in Section 2.8.
- (17) A chi-square distribution with 2 degrees of freedom ($\nu = 2$) is equivalent to an exponential distribution with rate $\lambda = \frac{1}{2}$.
- (18) The standard normal Z is a normal with $\mu = 0$ and $\sigma = 1$. We use Z as the representative of the normal location-scale family.
- (19) Given exponential random variables $E_1, E_2 \sim \text{Expon}(\lambda)$, their difference is Laplace distributed $E_1 - E_2 \sim \text{Laplace}(0, b = \frac{1}{\lambda})$.
- (20) The gamma distribution with $\alpha = 1$ is an exponential distribution: $\text{Gamma}(\alpha = 1, \lambda) = \text{Expon}(\lambda)$. This is analogous to the relation (3) between the negative binomial $\text{NBinom}(r = 1, p)$ and the geometric $\text{Geom}(p)$, which we saw in Figure 2.37.
- (21) The sum of n exponential random variables $E_i \sim \text{Expon}(\lambda)$ is a gamma distribution: $E_1 + E_2 + \cdots + E_n \sim \text{Gamma}(\alpha = n, \lambda)$. This is analogous to the relation (4) in Figure 2.37 (see page 172).
- (22) The gamma distribution $\text{Gamma}(\alpha, \lambda = \frac{1}{2})$ is the same as the χ^2 -distribution with $\nu = 2\alpha$.
- (23) The sum of n gamma variables $X_i \sim \text{Gamma}(\alpha_i, \lambda)$ is also a gamma variable: $X_1 + \cdots + X_n \sim \text{Gamma}(\alpha_1 + \cdots + \alpha_n, \lambda)$.
- (24) Given chi-square distributed random variables $Q_{\nu_1} \sim \chi^2(\nu_1)$ and $Q_{\nu_2} \sim \chi^2(\nu_2)$, the distribution of the ratio $\frac{Q_{\nu_1}/\nu_1}{Q_{\nu_2}/\nu_2}$ is $\mathcal{F}(\nu_1, \nu_2)$.
The χ^2 -distribution is a model for variance-like quantities. The F -distribution is a model for *ratios* of variance-like quantities.
- (25) The square of the standard normal $Z \sim \mathcal{N}(0, 1)$ has the chi-square distribution: $Z^2 \sim \chi^2(\nu = 1)$. Combining this with relation (15), we see that $\chi^2(\nu = n)$ describes the sum of n squared standard normal variables: $Z_1^2 + \cdots + Z_n^2 \sim \chi^2(\nu = n)$.
- (26) The F -distribution describes the variability of the average of ν_1 squared standard normal random variables $Z_i \sim \mathcal{N}(0, 1)$ divided by the average of ν_2 other squared standard normals.
The distribution of the ratio $\frac{\frac{1}{\nu_1}(Z_1^2 + Z_2^2 + \cdots + Z_{\nu_1}^2)}{\frac{1}{\nu_2}(Z_1^2 + Z_2^2 + Z_3^2 + \cdots + Z_{\nu_2}^2)}$ is $F(\nu_1, \nu_2)$.

- (27) If $X \sim \text{Laplace}(0, b)$, then $|X| \sim \text{Expon}(\lambda = b^{-1})$.
- (28) Given the uniform random variable $U \sim \mathcal{U}(0, 1)$, the random variable $-\frac{\log U}{\lambda}$ is exponentially distributed $\text{Expon}(\lambda)$. This is an example of the usefulness of the standard uniform random variable U to be transformed into other random variables. We'll learn more about this in Section 2.7 (see page 247).
- (29) Given two gamma random variables $G_1 \sim \text{Gamma}(\alpha_1, \lambda)$ and $G_2 \sim \text{Gamma}(\alpha_2, \lambda)$, then $\frac{G_1}{G_1 + G_2} \sim \text{Beta}(\alpha_1, \alpha_2)$.
- (30) The beta distribution $\text{Beta}(\alpha, \beta)$ with $\alpha = \beta \rightarrow \infty$ approaches the normal with mean $\mu = \frac{\alpha}{\alpha + \beta}$ and variance $\sigma^2 = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$.
- (31) If $U_1 \sim \mathcal{U}(0, 1)$ and $U_2 \sim \mathcal{U}(0, 1)$, then $\log(\frac{U_1}{U_2}) \sim \text{Laplace}(0, 1)$.
- (32) The beta distribution $\text{Beta}(\alpha = 1, \beta = 1)$ is equivalent to the standard uniform distribution $\mathcal{U}(\alpha = 0, \beta = 1)$.
- (33) Given independent random variables $E_1 \sim \text{Expon}(\lambda)$ and $E_2 \sim \text{Expon}(\lambda)$, the ratio $\frac{E_1}{E_2}$ has distribution $\mathcal{F}(\nu_1 = 2, \nu_2 = 2)$.
- (34) The square of Student's t -distribution $T \sim \mathcal{T}(\nu)$ is an F -distribution: $T^2 \sim \mathcal{F}(1, \nu)$.
- (35) As the degrees of freedom parameter ν goes to infinity, the t -distribution $\mathcal{T}(\nu)$ becomes the standard normal $\mathcal{N}(0, 1)$.

You're not responsible for remembering all these connections between distributions, but I want you to know that the distributions are not isolated islands, but related to each other in special cases.

2.6.14 Exercises

E2.28 Calculate the mean of the uniform distribution $U \sim \mathcal{U}(\alpha, \beta)$.

E2.29 The PDF of the random variable $E \sim \text{Expon}(\lambda)$ is given by $f_E(x) = \lambda e^{-\lambda x}$. Use integration to obtain the formula for the cumulative distribution function $F_E(b) = \Pr(\{E \leq b\})$.

Hint: The integral of the exponential function e^{cx} is another exponential function: $\int_{x=0}^{x=b} e^{cx} dx = \frac{1}{c}(e^{cb} - 1)$.

Links

[Read more about the relations between probability distributions]

https://wikipedia.org/wiki/Relationships_among_probability_distributions

[Complete list of the continuous distributions available in SciPy]

<https://docs.scipy.org/doc/scipy/tutorial/stats/continuous.html>

2.7 Simulation and empirical distributions

Consider the random variable X with probability distribution f_X . So far in this chapter, all the random variable calculations we studied involved sums and integrals of f_X . It turns out there is an alternative way to do probability calculations that involves only simple data manipulations on lists of simulated random observations $[x_1, x_2, x_3, \dots, x_N]$, where each x_j is a random draw from the random variable $X \sim f_X$. Provided the list is long enough (think $N = 1000$ or more), the distribution of the simulated observations $[x_1, x_2, x_3, \dots, x_N]$ approximates the randomness of the probability distribution f_X . Indeed, any probability calculation that we might want to do with the distribution f_X can be approximated by a corresponding calculation on the list of observations $[x_1, x_2, x_3, \dots, x_N]$.

2.7.1 Why simulate?

Using computers to simulate observations from random variables is an essential skill that I want you to have, because it's very useful for learning probability and statistics.

Simplified probability calculations Traditionally, doing probability calculations required advanced math knowledge. Today, computer simulations provide a much easier way to work with random variables. No fancy math formulas required—just simple Python manipulation of the lists of observations $[x_1, x_2, x_3, \dots, x_N]$.

Visualizing probability theory Many results in probability theory are described by fancy math formulas. Using simulated observations $[x_1, x_2, x_3, \dots, x_N]$ allows us to visualize theoretical results and gain hands-on intuition about them. Ultimately, it's the mathematical formulas that you need to remember, but visualizing the results of simulations is very helpful for understanding the formulas.

Verify that statistics procedures work In Part 2 of the book, we'll use probability calculations as part of statistics procedures. For example, we might define a statistical procedure that claims to give the correct result 90% of the time. We can use simulation to generate many datasets, apply the statistical procedure to the simulated datasets, and count the proportion of times when the procedure worked correctly, which allows us to experimentally verify if it really works 90% of the time.

In the remainder of this section, we'll learn how to do probability calculations using the list of observations $[x_1, x_2, \dots, x_N]$, instead of the probability distribution function f_X .

math perspective	empirical approximation
X	\rightarrow data $\mathbf{xs} = [x_1, x_2, \dots, x_N]$
PDF f_X	\rightarrow empirical distribution $f_{\mathbf{xs}}$
CDF F_X	\rightarrow empirical CDF $F_{\mathbf{xs}}$
$\mu_X = \mathbb{E}[X]$	\rightarrow $\text{mean}([x_1, x_2, \dots, x_N])$
$\sigma_X^2 = \mathbb{E}[(X - \mu_X)^2]$	\rightarrow $\text{var}([x_1, x_2, \dots, x_N])$
$\sigma_X = \sqrt{\mathbb{E}[(X - \mu_X)^2]}$	\rightarrow $\text{std}([x_1, x_2, \dots, x_N])$
$\mathbb{E}[h(X)]$	\rightarrow $\text{mean}([h(x_1), h(x_2), \dots, h(x_N)])$
$\Pr(\{a \leq X \leq b\})$	\rightarrow $\frac{\text{len}([x_j \text{ for } x_j \text{ in } \mathbf{xs} \text{ if } a \leq x_j \leq b])}{\text{len}(\mathbf{xs})}$

Table 2.5: Equivalence between math concepts and empirical calculations.

the book to better understand concepts in probability theory and statistics.

* * *

The topics we discussed in this section were fairly complicated, but I hope you were able to follow along. Don't worry too much about the fancy terminology like *empirical*-this and *empirical*-that: I used this precise terminology because I want you to know that there is a solid mathematical theory behind the idea of approximating the random variable X by a list of observations \mathbf{xs} —this is not something I pulled out of a hat.

My reason for introducing you to the advanced topic of simulation, is because it will allow us to gain lots of useful intuition about probability and statistics. Anytime you need to visualize or calculate something related to the variable X , you can write some simple Python code (think one or two `for`-loops) to generate random observations and get what you need. We'll see an example of this usefulness in the next section, where we'll learn about two important probability theorems by looking at lots of simulations.

2.7.7 Exercises

E2.30 The cumulative distribution function of the random variable $E \sim \text{Expon}(\lambda)$ is $F_E(b) = 1 - e^{-\lambda b}$, for $b \geq 0$. Find the math expression for the inverse cumulative distribution function $F_E^{-1}(q)$.

2.8 Probability models for random samples

We'll now discuss random samples of the form $\mathbf{X} = (X_1, X_2, \dots, X_n)$, which consist of n instances of the random variable $X \sim f_X$. In particular, we'll study the statistics that we can compute from the random sample \mathbf{X} , like the sample mean $\bar{\mathbf{X}} = \mathbf{mean}(\mathbf{X}) = \frac{1}{n} \sum_{i=1}^n X_i$.

We'll start with some definitions and examples, then describe the *sampling distribution of the mean*, which is a fundamental concept in statistics. We'll conclude this section by stating the *central limit theorem*, which is a foundational building block for many of the statistics procedures that we'll learn in Part 2 of the book.

2.8.1 Definitions

This section builds on the random variables concepts that you're already familiar with:

- X : a random variable with probability distribution f_X .
- $\mathbf{X} = (X_1, X_2, \dots, X_n)$: n instances of the random variable X . Each X_i represents an independent copy of the random variable X , with the same distribution $X_i \sim f_X$. We'll refer to \mathbf{X} as a *random sample*.
- $\mathbf{x} = (x_1, x_2, \dots, x_n)$: a particular sample, which consists of n observations from the distribution f_X .
- *statistic*: any quantity computed from a sample. Examples of statistics include the sample mean $\bar{\mathbf{x}} = \mathbf{mean}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n x_i$ and the sample variance $s^2 = \mathbf{var}(\mathbf{x}) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{\mathbf{x}})^2$.
- *sampling distribution of a statistic*: the distribution of values we obtain when we compute the statistic from a random sample \mathbf{X} . The *sampling distribution of the mean* is defined as the random variable $\bar{\mathbf{X}} = \mathbf{mean}(\mathbf{X}) = \frac{1}{n} \sum_{i=1}^n X_i$. The *sampling distribution of the variance* is defined as $S^2 = \mathbf{var}(\mathbf{X}) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{\mathbf{X}})^2$.

The expression *independent, identically distributed (i.i.d. for short)* refers to the sequence of random variables (X_1, X_2, \dots, X_n) , where the distribution of each X_i is identical, $X_i \sim f_X$, and the X_i are independent. The joint probability distribution for the i.i.d. random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$ is the product of n copies of the distribution f_X :

$$f_{\mathbf{X}} = f_{X_1 X_2 \dots X_n}(x_1, x_2, \dots, x_n) = f_X(x_1)f_X(x_2) \cdots f_X(x_n).$$

The joint distribution $f_{\mathbf{X}}$ describes the samples we might observe when we generate n observations of the random variable X .

2.8.2 Sample statistics

The term *statistic* refers to any quantity of interest computed from a sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$. Examples of statistics include the mean **mean**, the variance **var**, and all the other descriptive statistics we studied in Section 1.3. The term “statistic” refers both to the *function* that computes the quantity of interest and the output of that function, which is a *number*. For example, the sample mean statistic **mean** is a function that takes samples as inputs, and produces numbers as outputs. However, we also use the term sample mean when referring to the output of the function **mean** computed from a particular sample, $\bar{x} = \mathbf{mean}(\mathbf{x})$. To avoid confusion, we’ll refer to **mean** as the *sample mean function*, and to \bar{x} as the *sample mean value*.

The sample mean function is defined as follows:

$$\mathbf{mean}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} (x_1 + x_2 + \dots + x_n).$$

Figure 2.78 illustrates how we apply the sample mean function **mean** to the sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$ to obtain the sample mean value \bar{x} .

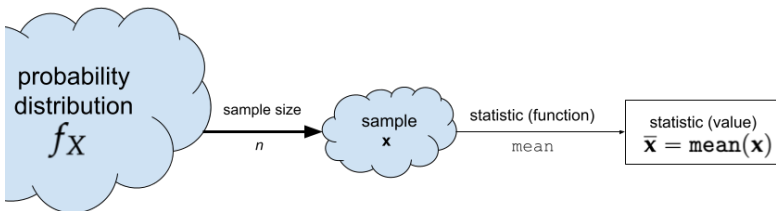


Figure 2.78: Computing the sample mean $\bar{x} = \mathbf{mean}(\mathbf{x})$ of the sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$, which consists of n observations from the distribution f_X .

Let’s write a Python function that computes the sample mean:

```
>>> def mean(sample):
    return sum(sample) / len(sample)
```

code
2.8.1

The function assumes the input `sample` is a list-like object (a Python list, a NumPy array, or a Pandas series). It uses the Python builtin function `sum` to compute the summation, then divides the result by the length of the sample to obtain the sample mean value.

Let’s test the function `mean` by computing the mean of the following sample that contains three observations $\mathbf{x} = (1, 3, 11)$.

```
>>> mean([1, 3, 11])
5.0
```

code
2.8.2

This is correct, since $\bar{x} = \mathbf{mean}(\mathbf{x}) = \frac{1+3+11}{3} = \frac{15}{3} = 5$.

larger and larger samples, it makes sense that the mean computed from the sample of observations will approach the mean of the distribution. The convergence of the sample mean to the distribution mean is a very useful thing, since it gives us a practical procedure for estimating the mean of an unknown distribution f_X . If you want to find μ_X , you can generate a large enough sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$ from f_X , and you know, thanks to the law of large numbers, that the mean computed from the sample $\bar{\mathbf{x}} = \mathbf{mean}(\mathbf{x})$ will approximate the mean of the distribution f_X .

The law of large numbers doesn't tell us anything about the *variability* of the means computed from the random sample \mathbf{X} . In the next section, we'll develop the vocabulary for describing the variability of the statistics values computed from random samples.

2.8.3 Sampling distribution of the mean

The concept of a *sampling distribution* of a statistic is very important for your understanding of statistics. Sampling distributions are considered to be one of the more complicated concepts in probability, but we have Python on our side, which allows us to simulate random samples from different distributions and visualize the sampling distributions computed from these random samples.

The mean of the random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$ is defined using the usual formula:

$$\bar{\mathbf{X}} = \mathbf{mean}(\mathbf{X}) = \frac{1}{n} \sum_{i=1}^n X_i.$$

Note that $\bar{\mathbf{X}}$ is a random variable, since it is an expression computed from the random variables (X_1, X_2, \dots, X_n) . The distribution of the sample mean computed from the random sample \mathbf{X} is called the *sampling distribution of the mean*, and we'll denote it $f_{\bar{\mathbf{X}}}$.

Figure 2.79 shows how we apply the sample mean function **mean** to the random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$ to obtain the random variable $\bar{\mathbf{X}} = \mathbf{mean}(\mathbf{X})$, which describes the distribution of sample means we might observe from **all possible samples of size n** drawn from the distribution f_X .

Note the difference between the lowercase mean $\bar{x} = \mathbf{mean}(\mathbf{x})$, and the uppercase mean $\bar{\mathbf{X}} = \mathbf{mean}(\mathbf{X})$. The lowercase \bar{x} is a number: the value of the mean computed from the particular sample \mathbf{x} , as illustrated earlier in Figure 2.78 on page 260. The uppercase $\bar{\mathbf{X}}$ is a random variable with distribution $f_{\bar{\mathbf{X}}}$, as illustrated in Figure 2.79.

The sampling distribution of the mean $f_{\bar{\mathbf{X}}}$ is an abstract theoretical construct that describes the variability of sample means

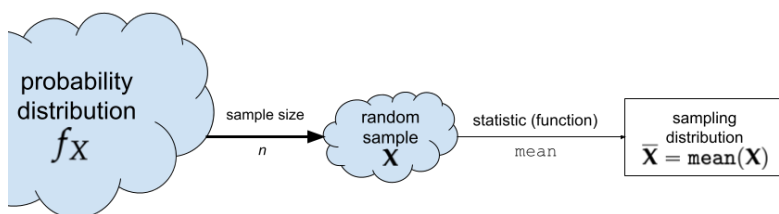


Figure 2.79: The *sampling distribution of the mean* is the distribution of the random variable $\bar{\mathbf{X}} = \text{mean}(\mathbf{X})$, which is the sample mean computed from the random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$.

we might observe from random samples of size n taken from the distribution f_X . It is the outcome of a hypothetical procedure: calculating the mean of a random sample \mathbf{X} . In this book, we'll use two different approaches to study the properties of sampling distributions:

- **Analytical approach.** We can use math equations (analytical formulas) to describe the shape of the sampling distribution $f_{\bar{\mathbf{X}}}$, or at least an approximation to it.
- **Computational approach.** We can use the simulation techniques we learned in Section 2.7 to generate thousands of random samples $[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_N]$, compute the mean from each sample $[\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_3, \dots, \bar{\mathbf{x}}_N]$, then plot a histogram of the sample means to visualize the sampling distribution $f_{\bar{\mathbf{X}}}$.

We'll start by looking at some examples of sampling distributions obtained using the computational approach, then conclude the section by stating the central limit theorem, which gives us an analytical formula for the sampling distribution of the mean that applies for any distribution f_X , provided the sample size n is large enough.

Example 2U: the sampling distribution $\bar{\mathbf{U}}$

Uma just received a reminder that her probability assignment is due in one hour. She quickly goes through the first three questions of the assignment since she remembers most of the material, but then she reaches a question asking her to “compute the mean and the standard deviation of the sampling distribution $f_{\bar{\mathbf{U}}}$ ” and she's not 100% sure what to do. She vaguely remembers hearing about sampling distributions in class, but not the details. She could try to look up the answer online, but given how little time is left to finish the assignment, she feels there is no time to learn all the theory, and instead the right thing to do is to open a Jupyter notebook and try to

solve the problem on her own. The teacher said Python is powerful. Let's see how powerful it is.

The exact wording of the questions is: "Find the mean and the standard deviation of the sampling distribution $f_{\bar{U}}$, where $\mathbf{U} = (U_1, U_2, \dots, U_{30})$ is a random sample that consists of 30 independent observations from the uniform random variable $U \sim \mathcal{U}(0, 1)$."

"Okay Python," she says to herself. "You and me. We've got 56 minutes left to figure this shit out. Let's do this!"

Uma starts by importing the uniform model from the module `scipy.stats`. The question asks about the standard uniform random variable $U \sim \mathcal{U}(0, 1)$, so she creates a computer model `rvU` that represents this random variable.

```
code >>> from scipy.stats import uniform
2.8.6 >>> rvU = uniform(0,1)
```

The question is asking something about the properties of random samples of size $n = 30$ from the random variable `rvU`. Uma is still not sure what $\mathbf{U} = (U_1, U_2, \dots, U_{30})$ means (why the capitals?), but she knows the lowercase version $\mathbf{u} = (u_1, u_2, \dots, u_{30})$ corresponds to generating a sample of size $n = 30$ from the random variable `rvU`. Uma uses the method `rvU.rvs(30)` to generate one such sample.

```
code >>> n = 30
2.8.7 >>> usample = rvU.rvs(n)
>>> usample
array([0.232, 0.042, 0.09, ... 27 more numbers ... ])
```

The assignment question is asking something about the **mean** statistic, so Uma copy-pastes the definition of the Python function `mean` from code block 2.8.1 into her assignment notebook:

```
code >>> def mean(sample):
2.8.8         return sum(sample) / len(sample)
```

Uma then uses the function `mean` to compute the mean of the simulated observations $\mathbf{u} = \text{usample}$ that she just generated.

```
code >>> mean(usample)
2.8.9 0.5887109207913922
```

The question is asking to find the sampling distribution $f_{\bar{U}}$, which is the distribution of the random variable $\bar{U} = \text{mean}(\mathbf{U})$, so Uma feels she's on the right track. She knows how to generate random samples of n observations \mathbf{u} by calling `rvU.rvs(30)`, and she knows how to compute the mean of a particular sample `mean(u)` using the function `mean`. But how do we go from `mean(u)` to `mean(U)`?

To get an idea of the variability of the individual observations U_i , Uma generated 10 samples of size $n = 30$, and creates a combined strip plot of the samples, as shown in Figure 2.80. She also computes the mean for each sample and displays the sample means

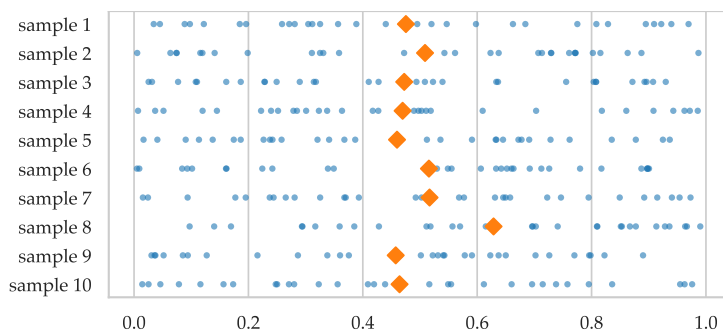


Figure 2.80: Scatter plots of 10 samples of size $n = 30$ from the standard uniform distribution $\mathcal{U}(0,1)$. The sample mean computed from each sample is indicated with a diamond marker.

as diamond markers in each plot. Looking at Figure 2.80, something suddenly clicks for her—the **sampling distribution** $f_{\bar{U}}$ describes the **variability of the diamond markers!**

In order to quantify the variability of the sampling distribution $f_{\bar{U}}$, Uma decides to generate $N = 1000$ random samples $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \dots, \mathbf{u}_N$, and compute the mean $\bar{\mathbf{u}}_j = \text{mean}(\mathbf{u}_j)$ from each sample, to obtain the list of observations $[\bar{\mathbf{u}}_1, \bar{\mathbf{u}}_2, \bar{\mathbf{u}}_3, \dots, \bar{\mathbf{u}}_N]$, which corresponds to 1000 samples from the random variable $\bar{U} = \text{mean}(U)$. She writes the following simulation code that uses a `for`-loop to generate the samples and saves the means $\bar{\mathbf{u}}_j$ to a list named `ubars`.

```
>>> n = 30          # sample size
>>> N = 1000        # number of samples to generate
>>> ubars = []
>>> for j in range(N):
>>>     usample = rvU.rvs(n)
>>>     ubar = mean(usample)
>>>     ubars.append(ubar)
>>> ubars
[0.516, 0.446, 0.526, 0.458, 0.503, ... 995 more means...]
```

code
2.8.10

The list `ubars` contains 1000 observations from the sampling distribution $f_{\bar{U}}$. Uma can visualize the sampling distribution by generating a histogram of the values in the list `ubars`.

```
>>> import seaborn as sns
>>> sns.histplot(ubars, color="r")
>>> sns.scatterplot(x=ubars, y=-5, color="r", marker="D")
The result is shown in Figure 2.81.
```

code
2.8.11

By inspecting Figure 2.81, Uma starts to get an intuitive understanding of the sampling distribution $f_{\bar{U}}$. She notes the distribution is centred at 0.5 and has a standard deviation of roughly 0.05. These are the numbers the assignment question is asking her to calculate!

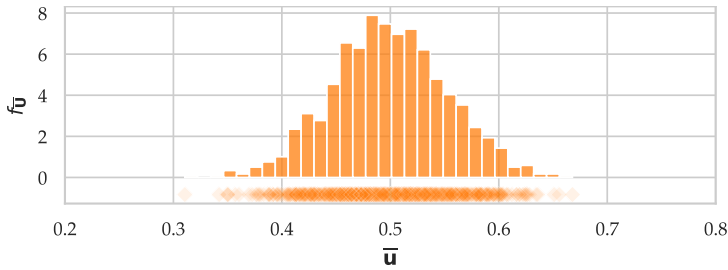


Figure 2.81: The sampling distribution of the mean $f_{\bar{U}}$ computed from 1000 samples of size $n = 30$ from the standard uniform distribution $U \sim \mathcal{U}(0, 1)$.

To obtain precise numerical estimates of the mean and the standard deviation of the sampling distribution, Uma computes the mean and standard deviation of the list `ubars`.

```
code >>> np.mean(ubars),      np.std(ubars)
2.8.12 ( 0.500018076130,      0.05211860142)
```

The mean of the sampling distribution is $\mu_{\bar{U}} = 0.5$ and its standard deviation is $\sigma_{\bar{U}} = 0.052$. She enters these answers to the question and submits the online assignment. The automated grading system responds with a positive feedback—the answers are correct!

Looking at the clock, Uma realized there are still 30 minutes left before the deadline. Maybe the teacher is right after all: using Python for probability simulations really is very useful!

General-purpose generator of sampling distributions

Let's generalize the code that Uma wrote for her assignment so we can generate observations from the sampling distribution of the mean of other random variables. Consider the random variable X with probability distribution f_X . The sampling distribution of the mean for samples of size n is the distribution of the sample mean $\bar{X} = \text{mean}(X)$ computed from the random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$, where each $X_i \sim f_X$.

To obtain the sampling distribution using simulation, we can follow the same steps as Uma did. First, we can generate random samples $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_N$, where each \mathbf{x}_j consists of n independent observations from the random variable X . We then compute the means of these samples, to obtain the list $[\bar{x}_1, \bar{x}_2, \bar{x}_3, \dots, \bar{x}_N]$, which is an empirical approximation to the sampling distribution $f_{\bar{X}}$. We'll now write a Python function `gen_sampling_dist_of_mean` that performs these steps. The function has two required arguments: a `scipy.stats` computer model `rvX` for the random variable X , and

the sample size n . The number of simulated samples we generate is controlled by the option N , which has the default value $N = 1000$.

```
>>> def gen_sampling_dist_of_mean(rvX, n, N=1000):
    xbars = []
    for j in range(N):
        xsample = rvX.rvs(n)
        xbar = np.mean(xsample)
        xbars.append(xbar)
    return xbars
```

code
2.8.13

Let's look at the code line by line. We start by creating the list `xbars` where we'll store the observations from the sampling distribution \bar{X} . We use a `for`-loop to repeatedly generate a random sample `xsample` of size n , compute the mean from that sample `xbar`, and append this sample mean to the list `xbars`.

Let's use the function `gen_sampling_dist_of_mean` to reproduce Uma's calculations of the sampling distribution $f_{\bar{U}}$ from Example 2U. Uma was asked to calculate the sampling distribution of the mean for samples of size $n = 30$ from the standard uniform distribution $U \sim \mathcal{U}(0,1)$. We can reproduce this calculation by passing the computer model `rvU = uniform(0,1)` as the `rvX` argument, and setting the sample size to `n=30`.

```
>>> from scipy.stats import uniform
>>> rvU = uniform(0, 1)
>>> ubars2 = gen_sampling_dist_of_mean(rvU, n=30, N=1000)
>>> ubars2
[0.477, 0.579, 0.46, 0.58, 0.41, ... 995 more means ... ]
>>> np.mean(ubars2), np.std(ubars2)
( 0.5034057868782, 0.052257079327)
```

code
2.8.14

The list of observations from the sampling distribution `ubars2` is different from the list `ubars` that Uma obtained in Example 2U. This is to be expected, since the lists are produced from different random samples. However, the mean and the standard deviation calculated from `ubars2` are very similar to the numbers Uma obtained.

Example 2Z: the sampling distribution \bar{Z}

Uma's classmate Zach got a different question on his assignment, asking him to compute the mean and the standard deviation of the sampling distribution $f_{\bar{Z}}$, where $\mathbf{Z} = (Z_1, Z_2, \dots, Z_{30})$ is a random sample of size 30 from the standard normal $Z \sim \mathcal{N}(0,1)$.

Zach starts by creating the computer model `rvZ = norm(0,1)` for the standard normal random variable $Z \sim \mathcal{N}(0,1)$, then generates a few samples from this distribution.

```
>>> from scipy.stats import norm
>>> rvZ = norm(0,1)
```

code
2.8.15

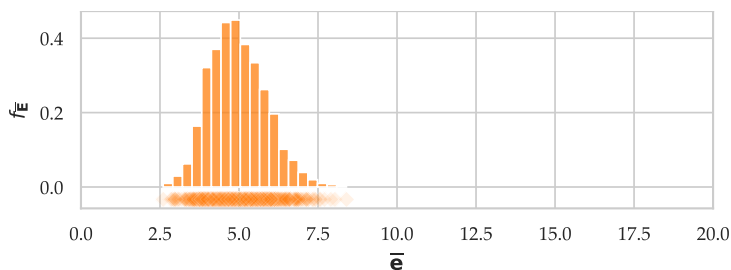


Figure 2.85: Sampling distribution of the mean computed from $N = 1000$ samples of size $n = 30$ from the exponential distribution $E \sim \text{Expon}(\lambda = 0.2)$.

```
>>> np.mean(ebars), np.std(ebars)
(5.02490198427453, 0.9335571543972068)
```

Erica knows the values she calculated are only approximations, so she reports her answers to two decimals: $\mu_{\bar{E}} = 5.02$ and $\sigma_{\bar{E}} = 0.93$.

* * *

The sampling distribution of the mean computed in all the examples above turned out to have a roughly normal shape. We would expect this to happen for samples from the normal distribution f_Z , since it makes sense for the average of n observations from the normal distribution to be normally distributed. However, the uniform f_U and the exponential f_E distributions are not normal, yet the sampling distributions of the mean $f_{\bar{U}}$ and $f_{\bar{E}}$ turn out to be normally distributed. What is up with that? How come the normal distribution pops up out of nowhere? This is not a coincidence: it follows from an important theorem of probability theory, which we'll discuss next.

2.8.4 Central limit theorem

The central limit theorem (CLT) tells us that the sampling distribution of the mean computed from i.i.d. samples of *any* distribution is approximately normally distributed.

Theorem (Central limit theorem). *Consider a random sample of size n denoted $\mathbf{X} = (X_1, X_2, \dots, X_n)$, where each X_i represents an independent draw from the random variable X . Let μ_X denote the mean of the random variable X , and let σ_X denote the standard deviation of X . Then the sampling distribution of the mean $\bar{\mathbf{X}} = \frac{1}{n} \sum_{i=1}^n X_i$ will converge to a normal distribution:*

$$\bar{\mathbf{X}} \sim \mathcal{N}\left(\mu_X, \frac{\sigma_X}{\sqrt{n}}\right),$$

as the sample size n goes to infinity.

We already knew from the law of large numbers that the mean of the sampling distribution \bar{X} will approximately equal the mean of the random variable μ_X , so this is not new.

What is new is that the central limit theorem tells us the “shape” of the sampling distribution of the mean $f_{\bar{X}}$ will be Gaussian. This is an interesting fact by itself: no matter what distribution f_X we start from, the sample means computed from samples taken from this distribution will be normally distributed.

Moreover, the central limit theorem gives us a math formula for the standard deviation of the sampling distribution as a function of the sample size n :

$$\sigma_{\bar{X}} = \frac{\sigma_X}{\sqrt{n}}.$$

This is a very useful formula, since it describes the variability in the means we can expect to observe from samples of different sizes.

What exactly do we mean when we say “as n goes to infinity” in the CLT? I used this informal expression in order to avoid getting into the technical discussions, which are not required for using the central limit theorem in practice. The CLT is *exactly* true only in the limit of infinitely large sample size n . For finite-size samples, $n = 10$, $n = 30$, $n = 100$, etc., the statement of the CLT is only approximately true $f_{\bar{X}} \approx \mathcal{N}\left(\mu_X, \frac{\sigma_X}{\sqrt{n}}\right)$, with the accuracy of this approximation increasing as n becomes larger. There is no universal threshold that we can apply for all situations. For different models, the convergence could be faster (thus allowing small sample sizes like $n = 10$) or slower (requiring larger sample sizes like $n = 100$). Roughly speaking, the closer the distribution f_X is to the normal, the smaller the sample size needs to be for the CLT formula to be acceptably accurate. If the distribution f_X is highly skewed, like the exponential distribution in Example 2E, then larger sample sizes are required for the sampling distribution of the mean to take on a normal shape.

Let’s revisit the U , Z , and E examples and apply the central limit theorem to samples of different sizes to see what happens.

Example 3U: CLT for samples from the uniform

Figure 2.86 shows strip plots for random samples of different sizes generated from the standard uniform $U \sim \mathcal{U}(0, 1)$. Observe that the variability of the sample means (the diamond markers) decreases as the sample size increases.

Figure 2.87 shows a side-by-side comparison of the sampling distributions computed from random samples of different sizes. The histogram in the left panel shows the sampling distribution of the

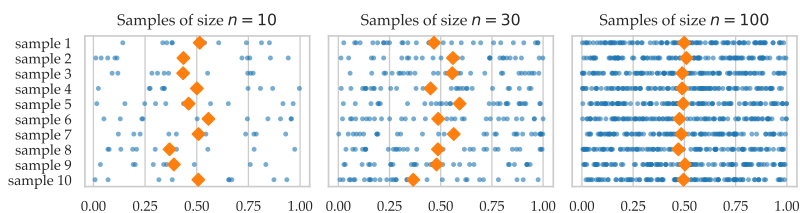


Figure 2.86: Samples of size $n = 10$, $n = 30$, and $n = 100$ from the standard uniform distribution $U \sim \mathcal{U}(0,1)$. The sample mean computed from each sample is indicated by a diamond shape.

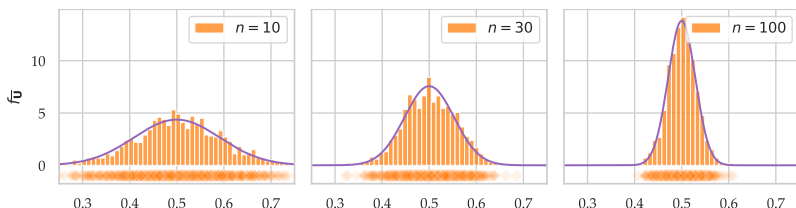


Figure 2.87: Comparison of the sampling distributions of the mean $f_{\bar{U}}$ for samples of size $n = 10$, $n = 30$, and $n = 100$ from the distribution $\mathcal{U}(0,1)$. The purple line indicates the model obtained from the central limit theorem.

mean computed from samples of size $n = 10$. We see there is a reasonable agreement between the histogram and the probability distribution predicted by the central limit theorem $\mathcal{N}(0.5, \sigma_U/\sqrt{10})$. The middle panel shows the sampling distribution computed from sample sizes of size $n = 30$ and the corresponding probability model $\mathcal{N}(0.5, \sigma_U/\sqrt{30})$ predicted by the CLT. The right panel shows the same pilots for samples of size $n = 100$. Note the close agreement between the histograms of the sampling distributions obtained using simulation and the analytical models predicted by the central limit theorem $\mathcal{N}(0.5, \sigma_U/\sqrt{n})$.

The variability of the sampling distribution decreases as the sample size increases. The central limit theorem gives us a precise formula for the standard deviation of the sampling distribution of the mean as a function of the sample size:

$$\sigma_{\bar{U}} = \frac{\sigma_U}{\sqrt{n}}.$$

We can verify the accuracy of this formula by computing the standard deviations of sampling distributions obtained from samples of different sizes and comparing the results to the predictions of the central limit theorem.

We previously generated observations from the sampling distribution of the mean by calling the function

`gen_sampling_dist_of_mean`, but we'll now show a simple Python one-liner that uses the list-comprehension syntax to produce the same result.

```
code >>> N = 10000
2.8.21 >>> ubars10 = [np.mean(rvU.rvs(10)) for j in range(N)]
>>> np.std(ubars10),      rvU.std()/np.sqrt(10)
(0.09097033522620938,    0.09128709291752768)
```

I invite you to look back at the code we used to define the function `gen_sampling_dist_of_mean` in code block 2.8.13 on page 268 to convince yourself that the list-comprehension expression above produces the same result as calling `gen_sampling_dist_of_mean`.

We can also generate observations from the sampling distributions computed from samples of size $n = 30$ and $n = 100$, and compare their standard deviations to the predictions of the CLT.

```
code >>> ubars30 = [np.mean(rvU.rvs(30)) for j in range(N)]
2.8.22 >>> np.std(ubars30),      rvU.std()/np.sqrt(30)
(0.052507004698028686,    0.05270462766947299)
>>> ubars100 = [np.mean(rvU.rvs(100)) for j in range(N)]
>>> np.std(ubars100),      rvU.std()/np.sqrt(100)
(0.028881259542408583,    0.028867513459481287)
```

The simulation results and the theoretical predictions of the CLT show some disagreement for samples of size $n = 10$, but for samples of size $n = 30$ and $n = 100$, the disagreement is very small.

Example 3Z: CLT for samples from the normal

Let's now look at the sampling distribution of the mean for samples from the standard normal distribution $Z \sim \mathcal{N}(0, 1)$. Figure 2.88 shows strip plots of samples of size $n = 10$, $n = 30$, and $n = 100$, as well as the sample means computed from each sample.

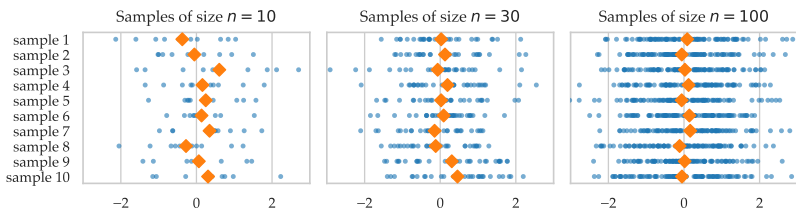


Figure 2.88: Samples from the standard normal distribution $Z \sim \mathcal{N}(0, 1)$.

Figure 2.89 shows histograms of the sampling distribution of the mean \bar{f}_Z computed from $N = 10\,000$ simulated samples of different sizes. The predictions of the central limit theorem are indicated by the purple line in each plot.

According to the central limit theorem, the standard deviation of the sampling distribution $\sigma_{\bar{Z}}$ is approximately equal to $\frac{\sigma_Z}{\sqrt{n}} = \frac{1}{\sqrt{n}}$.

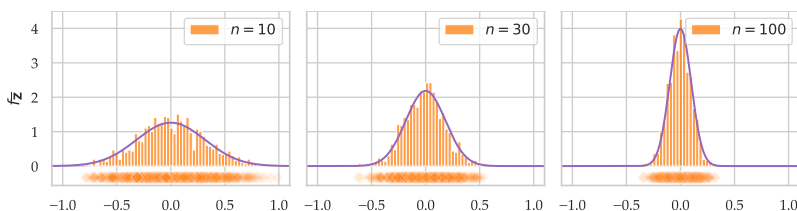


Figure 2.89: Sampling distribution of the mean computed from samples of size $n = 10$, $n = 30$, and $n = 100$ from the standard normal $Z \sim \mathcal{N}(0, 1)$.

We can check the accuracy of this claim by computing the standard deviation of the sampling distributions using simulation.

```
>>> zbars10 = [np.mean(rvZ.rvs(10)) for j in range(N)]      code
>>> np.std(zbars10),      rvZ.std()/np.sqrt(10)           2.8.23
(0.31531583269320146,    0.31622776601683794)
>>> zbars30 = [np.mean(rvZ.rvs(30)) for j in range(N)]
>>> np.std(zbars30),      rvZ.std()/np.sqrt(30)
(0.18357450148601615,    0.18257418583505536)
>>> zbars100 = [np.mean(rvZ.rvs(100)) for j in range(N)]
>>> np.std(zbars100),      rvZ.std()/np.sqrt(100)
(0.10022896783175735,    0.1)
```

We see there is a close agreement between the result computed using simulation and the prediction of the CLT, even for the smallest sample size $n = 10$. Indeed, when the underlying distribution is normal, the results of the central limit theorem are not an approximation, but exact. Recall Zoe's math derivation in Example 2Z.

Example 3E: CLT for samples from the exponential

We'll now repeat the sampling distribution visualizations a third time, this time applying it to using samples from the exponential random variable $E \sim \text{Expon}(\lambda = 0.2)$. The exponential distribution is highly skewed: most of the values fall close to 0, and there is a long tail of values that extends to the right, as can be seen in Figure 2.90. The sample means computed from each sample vary much less than the individual observations. All the sample means seem to be clustered around the mean of the distribution $\mu_E = 5$.

The plots in Figure 2.91 show histograms of the sampling distribution of the mean computed for samples of size $n = 10$, $n = 30$, and $n = 100$. The predictions of the central limit theorem are shown in purple, as in the previous examples. We can clearly see the histogram of the sampling distribution for samples of size $n = 10$ is right skewed, and the CLT model doesn't fit it very well. The sampling distribution for $n = 30$ is less skewed, so the discrepancy with the predictions of the CLT is smaller. For samples of size $n = 100$, the

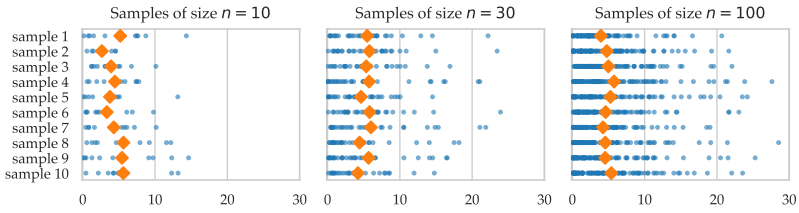


Figure 2.90: Strip plots of ten random samples of size $n = 10$, $n = 30$, and $n = 100$ from the distribution $\text{Expon}(\lambda = 0.2)$.

skewness in the histogram is almost completely gone, and the model predicted by the CLT is a very good fit.

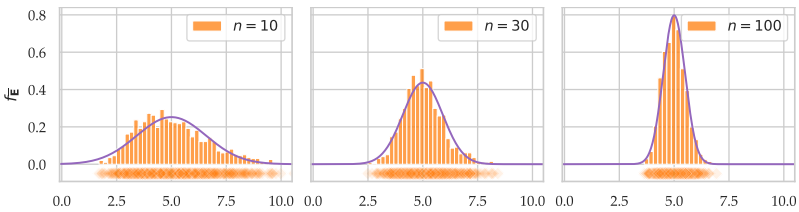


Figure 2.91: Sampling distributions of the mean computed from samples of different sizes from the exponential distribution $E \sim \text{Expon}(\lambda = 0.2)$.

Let's compute the standard deviation of the sampling distributions obtained using simulation to the standard deviations predicted by the CLT formula $\sigma_{\bar{E}} = \frac{\sigma_E}{\sqrt{n}} = \frac{5}{\sqrt{n}}$.

```
code
2.8.24 >>> ebars10 = [np.mean(rvE.rvs(10)) for j in range(N)]
>>> np.std(ebars10), rvE.std()/np.sqrt(10)
(1.599726626168737, 1.5811388300841895)
>>> ebars30 = [np.mean(rvE.rvs(30)) for j in range(N)]
>>> np.std(ebars30), rvE.std()/np.sqrt(30)
(0.9078552687478914, 0.9128709291752769)
>>> ebars100 = [np.mean(rvE.rvs(100)) for j in range(N)]
>>> np.std(ebars100), rvE.std()/np.sqrt(100)
(0.50136020903652, 0.5)
```

The CLT predictions are a little off for samples of size $n = 10$ and $n = 30$, but much better when $n = 100$. The precision of the CLT standard deviation formula $\sigma_{\bar{E}} = \frac{\sigma_E}{\sqrt{n}}$ is very impressive, and more so, given that the distribution f_E is highly skewed.

2.8.5 Discussion

The topics we introduced in this section are fundamental tools that we'll use for statistical inference in Part 2 of the book. It's essential that you understand sampling distributions, and how to generate observations from them.

2.9 Conclusion

I hope the material presented in this chapter helped you become comfortable with random variables and probability distributions, which are the essential prerequisites for understanding the statistical inference topics we'll discuss in Part 2 of the book.

Powerful building blocks

Random variables are an extension of the regular variables we study in mathematics. Instead of a fixed value x , the random variable X has multiple possible values, with the probability of observing different values described by the probability distribution f_X of the random variable.

The probability distributions we saw in sections 2.1 and 2.4 are useful building blocks used in statistics, machine learning, computer science, cryptography, and other fields, as outlined below.

Population data models

We can use probability distributions to model the variability of the data in a population. We refer to probability distributions used in this way as *data models*. For example, suppose we're interested in the distribution of weights of adult men in a small city. The population in this scenario is the set of weight measurements for all $N = 10000$ adult males in the city $\{w_1, w_2, w_3, \dots, w_{10000}\}$. We can model the distribution of weights in this population as a normal random variable $W \sim \mathcal{N}(\mu_W, \sigma_W)$, for some choice of the parameters μ_W and σ_W , as shown in Figure 2.92.

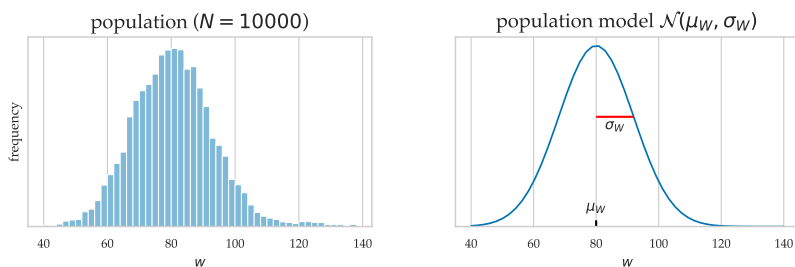


Figure 2.92: The histogram on the left shows the data of the whole population, which we could obtain if we were to measure the weights of all men in the city. The plot on the right shows the probability density function of the population model $W \sim \mathcal{N}(\mu_W, \sigma_W)$, which is an approximation to the real-world distribution of weights.

The normal model (right) is an approximation to the distribution of weights in the population (left). The approximation isn't perfect: the histogram on the left shows the population includes some weights above 120 kg, while the probability model assigns almost zero probability to the event $\{W \geq 120\}$, but overall, the model seems to capture the general shape of the distribution.

This example illustrates how probability distributions allow us to describe the variability in a population using a simple mathematical model, which is one of the key applications of probability theory in statistics. Data models are a succinct representation of the population characteristics in terms of a few parameters like μ_W and σ_W .

Probability distributions used in statistical inference

Probability models are the building blocks for *inferential statistics*. We'll now briefly explain what statistical inference is and highlight the key probability concepts that we'll use in Part 2 of the book.

Overview of statistical inference

Statistical inference is the process of learning about the properties of an unknown population based on a sample from that population. For example, suppose we have collected a sample of weight measurements from $n = 30$ adult men selected at random from the city: $\mathbf{w} = (w_1, w_2, \dots, w_{30})$. We want to use the information from this sample to extrapolate to the properties of the population of all adult men in the city, which we model as the probability distribution $\mathcal{N}(\mu_W, \sigma_W)$, where μ_W and σ_W are unknown parameters (see Figure 2.92). Specifically, we can use the sample mean $\bar{\mathbf{w}}$ as an estimate for the population mean μ_W , as illustrated in Figure 2.93.

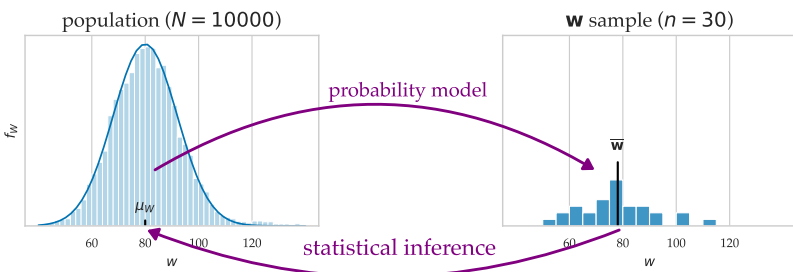


Figure 2.93: The goal of statistical inference is to estimate the parameter μ_W of the population model $W \sim \mathcal{N}(\mu_W, \sigma_W)$ based on the properties of the sample of weights $\mathbf{w} = (w_1, w_2, \dots, w_{30})$.

Statistical inference is the inverse problem of probability theory.

In probability theory, we assume we have a probability model with known parameters and use this model to generate random samples. In statistical inference, we observe a random sample, and we want to know the parameters of the distribution that produced this sample.

Inferential statistics includes tasks like *estimation* (guessing the unknown μ_W), *uncertainty quantification* (finding the precision of the approximation $\bar{\mathbf{w}} \approx \mu_W$), and *hypothesis testing* (making decisions about the unknown μ_W). In the next few pages, I want to highlight some key probability ideas that we'll use in Part 2 of the book.

Sampling distributions

In Section 2.8, we discussed the properties of random samples \mathbf{X} and learned about the *sampling distribution of the mean* $\bar{\mathbf{X}}$ (see page 263). Sampling distributions model the variability of the estimates we might obtain from random samples from a population. Knowing the shape of the sampling distribution allows us to quantify the uncertainty in the estimates we compute from a given sample \mathbf{x} .

Student's t -distribution, the χ^2 -distribution, and the Fisher–Snedecor F -distribution that we saw in Section 2.6 are essential building blocks for the classical (frequentist) inferential statistics topics like confidence intervals and hypothesis testing procedures:

- We'll use Student's t -distributions when doing inference for the unknown population mean μ_X , based on the sample mean $\bar{\mathbf{x}}$ and sample standard deviation $s_{\mathbf{x}}$.
- We'll use the χ^2 -distribution for statistical inferences about variance-like quantities. For example, the sampling distribution of the sample variance estimator $s_{\mathbf{x}}^2 = \mathbf{var}(\mathbf{x})$ is a scaled χ^2 -distribution with $\nu = n - 1$ degrees of freedom.
- The F -distribution will come up in the analysis of variance (ANOVA) hypothesis tests in Chapter 3, and for some of the model checks we'll learn in Chapter 4.

Your knowledge of sampling distributions will be very helpful when learning about *confidence intervals* and *hypothesis testing* in Chapter 3. You already know how the probability machinery works, so you'll be able to focus on learning the logic of statistical inference.

The likelihood function

Consider some unknown population that we assume is described by the data model $f_{X|\theta} = \mathcal{M}(\theta)$. We use the notation $f_{X|\theta}$ instead of f_X because we want to make the model parameters θ visible in the following calculations. The joint probability distribution for

an independent, identically distributed (i.i.d.) random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$ from this population is the n -fold product of $f_{X|\theta}$ s:

$$f_{\mathbf{X}|\theta}(\mathbf{x}|\theta) = f_{X|\theta}(x_1|\theta)f_{X|\theta}(x_2|\theta) \cdots f_{X|\theta}(x_n|\theta) = \prod_{i=1}^n f_{X|\theta}(x_i|\theta).$$

The distribution $f_{\mathbf{X}|\theta}$ describes the probability of observing different samples \mathbf{X} from the population $\mathcal{M}(\theta)$. See Figure 2.94 (a).

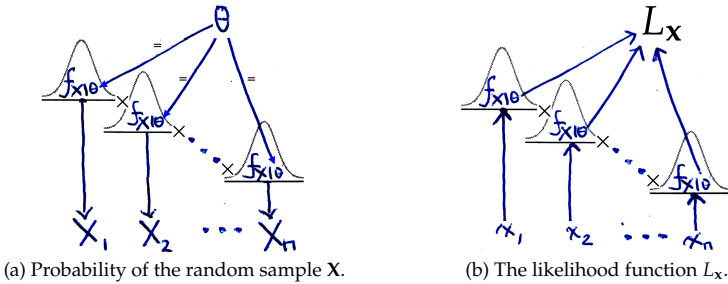


Figure 2.94: Two use cases of the data model $f_{\mathbf{X}|\theta} = \mathcal{M}(\theta)$. Part (a) of the figure shows how we use the joint probability distribution in the “forward” direction to obtain the probability of the random sample \mathbf{X} . Part (b) illustrates the likelihood function $L_{\mathbf{x}}$, which uses the data model $f_{\mathbf{X}|\theta}$ in the “backward” direction to evaluate the likelihood of different parameters θ , given the observed sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$.

Figure 2.94 (b) illustrates a different scenario in which we have observed the data sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$ from a population with unknown parameters θ . The *likelihood function* $L_{\mathbf{x}}$ is defined as

$$L_{\mathbf{x}}(\theta) = f_{X|\theta}(x_1|\theta)f_{X|\theta}(x_2|\theta) \cdots f_{X|\theta}(x_n|\theta) = \prod_{i=1}^n f_{X|\theta}(x_i|\theta).$$

The likelihood $L_{\mathbf{x}}$ is a function of θ . It tells us which values of θ are more or less likely to have produced the data $\mathbf{x} = (x_1, x_2, \dots, x_n)$.

Note the likelihood function $L_{\mathbf{x}}$ is defined by the same math expression as the joint probability distribution $f_{\mathbf{X}|\theta}$, but when $\mathbf{x} = (x_1, x_2, \dots, x_n)$ is fixed and θ is variable. It might be helpful to think of the joint probability distribution $f_{\mathbf{X}|\theta}$ and the likelihood function $L_{\mathbf{x}}$ as alternative directions of information flowing through the data model $f_{\mathbf{X}|\theta}$, as illustrated by the arrows in Figure 2.94.

Calculating the likelihood of different parameter values is a general principle that can be used to “learn” the parameters θ of any statistical model. We’ll see likelihood functions come up in Chapter 4 when fitting linear models, and as part of the Bayesian update procedure we’ll learn in Chapter 5.

Knowledge representations in Bayesian statistics

Probability distributions can be used to represent our *knowledge* or *belief* about the unknown population parameters. The “width” of a probability distribution is a natural way to describe our uncertainty about the parameters. For example, we can use a probability distribution f_{Θ} to describe our knowledge about the parameters θ .

In Chapter 5, we’ll learn about Bayesian statistics, which is a different approach to statistical inference based on the Bayes’ rule formula, which we saw in Section 2.5 (see page 210). We’ll use the probability models we saw in Section 2.6 as building blocks inside Bayesian statistical models for different statistical analysis scenarios. Your familiarity with probability distributions will help you understand Bayesian ideas because you are already familiar with the probability math details.

Half the climb is already done

The probability theory concepts you learned in this chapter are the foundation for statistical inference topics we’ll learn in Part 2 of the book. Look back to Figure 2 on page vi for a visual representation of your progress. The fact that you got to the end of the chapter, and you survived this intense “probability theory bootcamp” featuring lots of math formulas and code examples means you are prepared to face the statistical inference “uphills” in Part 2 of the book.

Other applications of probability theory

The topics we covered in this chapter were specifically selected to prepare you for the study of statistical inference. Probability theory has applications in many other domains. Here are some areas where probability theory and random variables are used.

- **Probabilistic machine learning.** Many machine learning algorithms are constructed using probability building blocks. Your knowledge of the probability models that we discussed in this chapter will be helpful for further studies in machine learning.
- **Gambling and game theory.** Some of the earliest uses of probability were for the analysis of games of chance. Probability theory can help us determine optimal strategies in certain games. We can calculate the expected value of any gambling game to see whether it’s worth playing this game or not.
- **Randomized algorithms.** Algorithms are sequences of instructions for solving a particular type of problem. For some prob-

lems, using randomness as part of the instructions produces algorithms that solve problems more reliably.

- **Information theory.** Probability theory is an essential tool for studying how information is encoded and transmitted. For example, *data compression* procedures use the probability of different symbols in a data stream to build an efficient encoding that minimizes redundancy. In contrast, *error-correcting codes* add a certain amount of redundancy to messages that allows the information to be decoded even if parts of the message are lost due to noise.
- **Cryptography.** Randomness is a key building block in cryptography, which is the study of secure communication. Randomness plays a practical role in key generation and in the definitions of security concepts.
- **Finance.** The price changes of individual stocks and stock portfolios are often analyzed using probability models.
- **Weather forecasting.** The results of meteorological models are often presented as probabilistic predictions, like the probability of precipitation tomorrow.
- **Insurance.** Actuarial science develops probability models for rare events like deaths, accidents, and natural disasters. These models are then used to set the price of insurance policies.
- **Inventory management.** Businesses that sell physical products must constantly keep track of stock levels in the supply chain. Probabilistic models are used to forecast demand, avoid gaps in supplies, and optimize logistics.

The material you learned in this chapter opens the door for you to study all these subjects.

2.10 Probability problems

To have a good time when learning statistics, you need to be fluent with probability theory calculations. The best way to gain this fluency is to solve practice problems. Reading is not enough—you need to get some hands-on experience with probability distributions. How convenient is it that the next pages contain a bunch of probability practice problems. Go grab a coffee and dive into the problems!

P2.1 The *addition rule* of probability states that

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B),$$

for any two sets of outcomes A and B . The symbol \cup describes the *union* of two sets, which means all elements that are either A or B . The symbol \cap describes the *intersection* of the two sets, meaning the elements that are in A and B .

Verify the addition rule applies to the die roll random variable D , and the sets of outcomes $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, by computing the probabilities of all terms in the above equation.

P2.2 If you have a very large number of samples from a given distribution, you can use the empirical distribution to do probability calculations. Assume the sample of 2000 observations in `datasets/kombuchapop.csv` are samples from the random variable K . Compute the probability $\Pr(\{K \geq 1005\})$.

P2.3 The dataset `mortality.csv` contains the age at which 10000 individuals have died. Use the empirical distribution to calculate: a) average life expectancy; b) average life expectancy given the person has lived to age 20; c) average life expectancy given your age.

P2.4 Given a random variable X with three possible outcomes $\{1, 2, 3\}$ and probability distribution $f_X = (p_1, p_2, p_3)$, prove that $p_1 \leq 1$.

Hint: Use the Kolmogorov's axioms and build a proof by contradiction.

P2.5 The probability of heads for a fair coin is $p = \frac{1}{2}$. What is the probability of getting heads four times in a row?

P2.6 You have a biased coin that lands on heads with probability p , and consequently lands on tails with probability $(1 - p)$. Suppose you want to flip the coin until you get heads. Define the random variable N as the number of tosses required until the first heads outcome. What is the probability mass function $f_N(n)$ for success on the n^{th} toss? Confirm that the formula is a valid probability distribution by showing $\sum_{n=1}^{\infty} f_N(n) = 1$.

Hint: Find the probabilities for cases $n = 1, 2, 3, \dots$ and look for a pattern.

P2.7 A mathematician walks over to a roulette table in a casino. The roulette wheel has 101 numbers: 50 are black, 50 are red, and the number zero is green. If the mathematician bets \$1 on black and the roulette ball stops on a black number, the payout is \$2, otherwise the bet is lost. Calculate the

End matter

Conclusion

Congratulations on completing the first half of the book! In Part 1, we learned the essential prerequisites for the statistics topics that we'll study in Part 2 of the book. Statistics is a notoriously complicated subject, but only for people who are not adequately prepared. If you've made it this far in the book, it means you have experience with both data management and probability calculations, which are the foundation for understanding statistics.

Review of key ideas

Let's review the key ideas we learned in Part 1 of the book.

Data generating process

The fundamental idea from Chapter 1 you need to remember is the data generating process that explains how the sample was collected. We discussed the *random selection* procedure which produces samples with no systematic bias. We also discussed the concept of *random assignment* used in statistical experiments. Random selection and random assignment are key tools that enable statistical inference.

Probability models

In Chapter 2, we learned about random variables and probability distributions that allow us to model population variability. For example, Figure 2.92 on page 280 shows the probability model $W \sim \mathcal{N}(\mu_W, \sigma_W)$, which is an approximation to the real-world distribution of weights of adult males in a city.

The key ideas from chapters 1 and 2 form the two pillars on which we'll build statistical inference in Part 2 of the book.

Review of practical skills

In addition to the foundation of ideas about populations, samples, and probability models, we learned a lot of practical computing skills that will prove useful in Part 2 of the book.

Using Python as a calculator We saw a lot of Python code examples in this book. I hope the code examples, exercises, and problems helped you gain experience using Python as a calculator. Feel free to revisit the Python tutorial in Appendix C anytime you want to refresh your Python knowledge.

Data manipulation using Pandas We saw many examples of data manipulation using Pandas in Chapter 1. I hope you feel comfortable with loading datasets, filtering data, simple transformations, and computing descriptive statistics. To review and deepen your understanding of data management tasks, you can consult the Pandas tutorial in Appendix D.

Data visualization using Seaborn We used the Seaborn library to visualize datasets and probability distributions. Data visualization is an important skill that will facilitate all your work, because you can literally *see* what data you're working with, and visualize the shapes of probability distributions. See the Seaborn tutorial in Appendix E to review the functions used to generate data visualizations.

Probability calculations using SciPy In Chapter 2, we saw many examples of probability calculations using the computer models for random variables defined in the `scipy.stats` module. The module `scipy.stats` will continue playing an important role in all the probability calculations in Part 2 of the book. For a complete list of all the probability distributions available in `scipy.stats`, see page 307 in Appendix B. See also the list of methods for discrete random variables in Table 2.2 on page 164 and continuous random variables in Table 2.3 on page 182.

Further reading

I've compiled the following list of recommended learning resources for readers who want to broaden their knowledge base about data management and probability theory.

Probability theory

[*Introduction to Probability* by Grinstead and Snell]

<https://math.dartmouth.edu/~prob/prob/prob.pdf>

Appendix A

Answers and solutions

Chapter 1 solutions

Answers to exercises

E1.3 4245. **E1.5** mean = 33.75; standard deviation = 14.28. **E1.13** `mean(efforts)` = 8.9, `min(efforts)` = 5.21, `max(efforts)` = 12.0, `range(efforts)` = 6.79. **E1.14** `Q1(efforts)` = 7.76, `med(efforts)` = 8.69, `Q3(efforts)` = 10.35. **E1.15** The frequencies within the bins are 2, 6, 5, and 2. **E1.19** `Freq(debate)` = 8, `Freq(lecture)` = 7, `RelFreq(debate)` = 0.53, `RelFreq(lecture)` = 0.47. **E1.20** The mode is debate with frequency 8.

Solutions to selected exercises

E1.5 Assuming you used the default options when loading the data file `players.csv`, the `age` variable will fall in the column C of the spreadsheet. You can then compute the average age using the formula `=AVERAGE(C2:C13)` and the standard deviation of the ages using `=STDEV(C2:C13)`.

E1.9 Note you can also view the Pandas documentation online, see <https://bit.ly/pdsumdocs>.

E1.13 We first use `efforts = students["effort"]` to extract the effort variable, then use the `.mean()`, `.min()`, and `.max()` methods. Compute the range by calling `efforts.max()-efforts.min()`.

E1.14 Extract the effort variable, `efforts = students["effort"]`, then compute `efforts.quantile(q=0.25)`, `efforts.median()`, and `efforts.quantile(q=0.75)`.

E1.15 Run `efforts.value_counts(bins=bins).sort_index()` where `bins = [5,7,9,11,13]`.

E1.16 Create the scatter plot by making a data frame, then call `sns.scatterplot`:

```
code >>> df = pd.DataFrame([(2,2), (3,3), (4,3), (5,5),
A.0.1      (6,4), (5,4), (7,6), (8,5)],
                           columns=["x", "y"])
>>> sns.scatterplot(data=df, x="x", y="y")
```

E1.18 Use `sns.countplot(data=students, x="curriculum")` to draw a bar plot.

Appendix B

Notation

This appendix contains a summary of the math notation used in this book. The tables are provided for easy reference: whenever you encounter some math symbol whose meaning you don't know, you can look it up here to learn what it is called and what it means.

Math notation

Math notation is a precise language for describing variables, expressions, and functions. We'll now provide a condensed summary of the notation used for basic math concepts as a review for readers whose math skills might be a little *rusty*.

Variables and constants The usual names we use for variables are taken from the end of the alphabet x , y , z , m , n , etc. In contrast, we denote constants using letters from the beginning of the alphabet a , b , c , or using Greek letters. Yeah, we use the Greek alphabet a lot in statistics, but I don't want you to freak out about this. For example, the population mean is denoted μ (the Greek letter *mu*), and the population standard deviation is denoted σ (the Greek letter *sigma*). The Greek letters *alpha* α and *beta* β are the equivalents of a and b . Other Greek letters we'll use in this book include: *lambda* λ , *delta* Δ , *chi* χ (pronounced *khai*), *epsilon* ϵ , *gamma* γ , *nu* ν , *rho* ρ , *tau* τ , and *theta* θ .

We'll use the subscript notation to denote several variables of the same type, for example we can denote three observations of the variable x as x_1 , x_2 , x_3 .

Math operations The book uses the standard math notation for arithmetic operations, as summarized in the following table.

Expression	Read as	Geometric interpretation
$a + b$	a plus b	sum of the lengths a and b
$a - b$	a minus b	difference between the lengths a and b
$-a$	negative a	the length a in the opposite direction to $+a$
$a \cdot b = ab$	a times b	area of a rectangle with sides a and b
$a^2 = aa$	a squared	area of a square of side length a
$a^3 = aaa$	a cubed	volume of a cube of side length a
a^n	a exponent n	a multiplied by itself n times
$\sqrt{a} = a^{\frac{1}{2}}$	square root of a	the side length of a square of area a
$a/b = \frac{a}{b}$	a divided by b	a parts of a length split into b parts
$a^{-1} = \frac{1}{a}$	one over a	division by a
%	percent	proportions of a total; $a\% \stackrel{\text{def}}{=} \frac{a}{100}$

We denote multiplication between the numbers a and b using the centre-dot symbol $a \cdot b$, or sometimes the multiplication symbol $a \times b$, but most commonly don't use any symbol at all. The expression ab uses *implicit multiplication*: the default interpretation when two variables are placed side by side is that they are multiplied together.

Equations and inequalities Most mathematical statements describe some relation between two math expressions. Here are the most common types of relations we'll see in this book.

Symbol	Read as	Used to denote
$=$	is equal to	expressions that have the same value
$\stackrel{\text{def}}{=}$	is defined as	definition of a new variable
\approx	is approximately	expressions that are almost equal
$<$	less than	$a < b$ says a is strictly less than b
\leq	less than or equal	$a \leq b$ says a is less than or equal to b
$>$	greater than	$a > b$ says a is strictly greater than b
\geq	greater than or equal	$a \geq b$ says a is greater than or equal to b

An equation is a mathematical statement that says the expression on the left of the equality sign “=” is equal to the expression on the right side of the equality sign. The symbol $\stackrel{\text{def}}{=}$, read “equal by definition,” is a special kind of equality sign that we use to define new variables. We use the symbol \approx to denote approximations, which are similar to equations, but the equality doesn't hold exactly—the left side and the right side are only approximately equal. For example, the fraction $\frac{5}{6}$ corresponds to an infinitely long decimal $\frac{5}{6} = 0.8333333333 \dots$

In order for the equality to hold, we would need to write infinitely many 3s in the decimal expansion, which is impractical. Instead, we can write $\frac{5}{6} \approx 0.833$, which tells us the two quantities are approximately equal.

Sets A *set* is a collection of math objects. We denote sets using the curly brackets $\{\text{<desc>}\}$, where <desc> describes the elements in this set. For example the expression $S \stackrel{\text{def}}{=} \{1, 2, 3\}$ defines the set S that consists of the three numbers 1, 2, and 3. We can also describe sets using *interval notation*, which specifies a range of numbers using its endpoints enclosed in $[$ or $($ brackets. For example, the interval $[a, b]$ describes the set of numbers between a and b , including the endpoints a and b . The interval $[a, b)$ describes the set of numbers between a and b that includes the endpoint a but not the endpoint b .

Here are some important number sets that mathematicians give special names to and denote using a special font.

Symbol	Name	Used to denote
\mathbb{N}	naturals	the set of natural numbers $\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, 3, \dots\}$
\mathbb{Z}	integers	the set of integers $\mathbb{Z} \stackrel{\text{def}}{=} \{\dots, -2, -1, 0, 1, 2, 3, \dots\}$
\mathbb{Q}	rationals	the set of fractions $\frac{m}{n}$ where m and n are integers
\mathbb{R}	reals	the set of real numbers $\mathbb{R} \stackrel{\text{def}}{=} (-\infty, \infty)$
\mathbb{R}_+	non-neg. reals	the set of non-negative real numbers $\mathbb{R}_+ \stackrel{\text{def}}{=} [0, \infty)$

All the number sets listed above contain an infinite number of elements. The notation \dots in the definition of the naturals describes an infinite sequence of integers. The set of real numbers includes the naturals \mathbb{N} , the integers \mathbb{Z} , the rationals \mathbb{Q} , as well as irrational numbers like π , e , $\sqrt{2}$, etc. Essentially, the set of real numbers includes any numbers you might encounter in the real world.

Sets notation One of the most intimidating aspect of math notation are the symbols mathematicians use to describe set relations (\in , \notin , \subset) and set operations (\cup , \cap , \setminus). One of my students referred to these characters as “alien symbols,” and this is an accurate description if you’re seeing set notation for the first time. I’ve tried to limit the use of the alien symbols to a minimum in this book, but I use set notation in some of the problem solutions, so I figured I should explain how to read each symbol and what it means.

Symbol	Read as	Meaning
$a \in S$	a in S	a is an element of the set S
$b \notin S$	b not in S	b is not an element of the set S
\subset	subset	one set strictly contained in another
\subseteq	subset or equal	containment or equality
\cup	union	the combined elements from two sets
\cap	intersection	the elements two sets have in common
$S \setminus T$	S set minus T	the elements of S that are not in T
$\forall x$	for all x	a statement that holds for all x
$\exists x$	there exists x	an existence statement
$\nexists x$	there doesn't exist x	a non-existence statement
$ $	such that	describe or restrict the elements of a set

The purpose of the alien symbols is to express mathematical statements as precisely and as concisely as possible. Whenever you want to say something using mathematics, it's important to make your statements as concise as possible, so they can fit on a single line. Short, precise statements make it easy to "see" what is going on. This is why mathematicians came up with all those symbols: they are not trying to make life difficult for you, on the contrary, they are trying to make things as simple as possible.

This condensed symbolic notation is most useful in proofs and derivations, which are sequences of math statement that show why some statement is true by reducing or simplifying some expression. It would be very annoying to have to write every mathematical proof in natural language, writing out mathematical statements as full English sentences. Math proofs would take entire pages! Basically, you have to trust me that these complicated math symbols actually make things simpler, not more complicated.

Let's look at some examples of the alien symbols in use. The symbols \in (read *in*) and \notin (read *not in*) are used to describe set membership. For example, the statement $\sqrt{2} \in \mathbb{R}$ reads "the square root of two is an element of the set of real numbers," or "the square root of two is a real number." The statement $\sqrt{2} \notin \mathbb{Q}$ reads "the square root of two is not an element of the set of rational numbers," or more concisely, "the square root of two is not a rational number."

The subset symbols allow us to concisely describe set containment. For example, the math statement $\mathbb{Q} \subset \mathbb{R}$ reads "the set of rational numbers is a subset of the set of real numbers," or more concisely "the reals contain the rationals." This means every rational number $q \in \mathbb{Q}$ is also a real number $q \in \mathbb{R}$.

Set builder notation The *set-builder* notation is a common pattern mathematicians use to describe subsets. We can describe an arbitrary

subset of the set S using the notation $\{x \in S \mid \text{<conditions>}\}$, where <conditions> is some expression that defines the conditions satisfied by all elements x in this subset. The vertical bar symbol “ \mid ” is read “such that,” so the whole set-builder statement reads as “the set of numbers x in S such that <conditions>.” For example, the interval $[a, b]$ is defined as $[a, b] \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a \leq x \leq b\}$, where the condition $a \leq x \leq b$ describes precisely the numbers between a and b , inclusively. The interval $[a, b)$, is defined as $[a, b) \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid a \leq x < b\}$, which reads “the set of real numbers that are greater than or equal to a and strictly less than b .”

Functions We use the notation $f : A \rightarrow B$ to describe a function f that takes inputs from the set A and produces outputs in the set B . The most common type of function used in this book has the form $f : \mathbb{R} \rightarrow \mathbb{R}$, which means it is a function that takes real numbers as inputs and produces real numbers as outputs. We often denote the function input as x and the function output as $f(x)$ or y . The following table shows some examples of commonly used functions.

Definition	Name	Used to denote
$f(x) = mx + b$	linear	line with slope m and y -intercept b
$f(x) = x^2$	quadratic	square of x
$f(x) = \sqrt{x}$	square root	square root of x
$f(x) = x $	absolute value	absolute value of x
$f(x) = e^x$	exponential	exponential function base e
$f(x) = \ln(x)$	natural log of x	logarithm base e
$f(x) = 10^x$	power of 10	exponential function base 10
$f(x) = \log_{10}(x)$	log base 10 of x	logarithm base 10

You can think of these functions as basic building blocks for describing arbitrary relationships between an input variable x and an output variable $y = f(x)$.

We use the notation $f(x)$, read “ f of x ,” to describe the output of the function f applied to the input x . For example, if we define the function f as $f(x) \stackrel{\text{def}}{=} 3x + 5$, we can then write the expression $f(a)$, which is equivalent to the function’s output $3a + 5$.

When solving equations involving the function $f(x)$, it is useful to know the *inverse function* of the function f , which we denote f^{-1} . The inverse function f^{-1} acts as the “undo” operation for the function f . If you apply f^{-1} to the output of f , you get back the original x you started from: $f^{-1}(f(x)) = x$. For example, the inverse of the function $f(x) \stackrel{\text{def}}{=} 3x + 5$ is the function $f^{-1}(x) = \frac{1}{3}(x - 5)$. Try choosing any number $x = a$ and calculate $f(a)$, the plug the number $f(a)$ into the inverse function f^{-1} to confirm that you get back to

the value a you started from. Here are some other examples of functions and their inverses. The inverse of the square root function $g(x) \stackrel{\text{def}}{=} \sqrt{x}$, is the quadratic function: $g^{-1}(x) = x^2$. The inverse of the exponential $h(x) = e^x$, is logarithm base e : $h^{-1}(x) = \ln(x)$. An inverse relationship also exists between the exponential function base 10, and the logarithm base 10.

* * *

If this is the first time you're seeing the math notation we presented in the previous pages, you might benefit from a more thorough review of high school math concepts. I'd like to use this opportunity to shamelessly plug the *No Bullshit Guide to Mathematics*[Sav18], since it is precisely on this topic.

Calculus notation

Expression	Denotes
$f(x)$	a function of the form $f : \mathbb{R} \rightarrow \mathbb{R}$
$f'(x)$	derivative of $f(x)$
$\frac{d}{dx}$	derivative operator: $\frac{d}{dx} [f(x)] = f'(x)$
$\int_a^b f(x) dx$	integral of $f(x)$ between $x = a$ and $x = b$
$F(x)$	the integral function of $f(x)$
a_k	sequence $a_k : \mathbb{N} \rightarrow \mathbb{R}$, also denoted $(a_0, a_1, a_2, a_3, \dots)$
$\sum_{k=r}^s a_k$	the summation $a_r + a_{r+1} + a_{r+2} + \dots + a_{s-1} + a_s$

Data notation

We denote samples using a boldface symbol $\mathbf{x} = (x_1, x_2, x_3, \dots, x_n)$. You can think of \mathbf{x} as the measurements of the variable x collected from n individuals. The sample \mathbf{x} will often be stored as a column named \mathbf{x} in a Pandas data frame.

For a sample of numerical values, we can compute the following *descriptive statistics* to capture its essential characteristics.

Symbol	Read as	Denotes
n	sample size	number of observations in the sample \mathbf{x}
mean	mean	average value
med	median	middle value of the dataset
mode	mode	the most frequently observed value
var	variance	average squared deviation from the mean
std	standard deviation	the square root of the variance
Q_1	first quartile	one quarter of values smaller than Q_1
Q_2	second quartile	middle value of the sample \mathbf{x}
Q_3	third quartile	one quarter of values are larger than Q_3
IQR	interquartile range	span of the middle fifty percent of the data
min	minimum	the smallest value in the sample \mathbf{x}
max	maximum	the largest value in the sample \mathbf{x}
range	range	difference between the max and min

You should think of the descriptive statistics as functions you compute from the sample \mathbf{x} . For example, the minimum value in the sample \mathbf{x} is denoted $\min(\mathbf{x})$. Refer to Table 1.4 (page 1.4) to see Pandas methods we use compute descriptive statistics. The sample mean, variance, and standard deviation are the most common statistics, so we use the special shorthand notation for them: $\bar{x} = \text{mean}(\mathbf{x})$, $s_x^2 = \text{var}(\mathbf{x})$, and $s_x = \text{std}(\mathbf{x})$.

Bivariate dataset A bivariate dataset consists of pairs of observations from two variables, $[\mathbf{x}, \mathbf{y}] = [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$. The *covariance* is denoted $\text{cov}(\mathbf{x}, \mathbf{y})$ and it measures the joint variability of \mathbf{x} and \mathbf{y} . Another descriptive statistics is the *correlation*, which is denoted $\text{corr}(\mathbf{x}, \mathbf{y})$ and measures the degree of linear relatedness between \mathbf{x} and \mathbf{y} .

Categorical data Consider now a sample $\mathbf{x} = (x_1, x_2, x_3, \dots, x_n)$ of categorical values. The frequency of the value v in the data \mathbf{x} is denoted $\text{freq}_v(\mathbf{x})$, and is the number of v s in the data \mathbf{x} . The *relative frequency* is denoted $\text{relfreq}_v(\mathbf{x})$, and counts the number of v s in a the data \mathbf{x} divided by n .

Probability notation

Expression	Denotes
$\{\text{descr}\}$	an outcome described by the condition “descr”
$\Pr(\{\text{descr}\})$	the probability of the outcome $\{\text{descr}\}$
$\Pr(\{\text{descr}\} \{\text{cond}\})$	conditional probability of outcome $\{\text{descr}\}$ given $\{\text{cond}\}$
$\mathbb{I}_{\{\text{descr}\}}$	indicator function for the outcome $\{\text{descr}\}$
X	a random variable (henceforth abbreviated as r.v.)
\mathcal{X}	the set of possible outcomes for the r.v. X
x	a particular outcome of the r.v. X
$f_X(x) \stackrel{\text{def}}{=} \Pr(\{X = x\})$	<i>probability mass function</i> (pmf) of a discrete r.v. X , or <i>probability density function</i> (pdf) of a continuous r.v. X .
$X \sim f_X$	X is distributed according to f_X
$F_X(x) \stackrel{\text{def}}{=} \Pr(\{X \leq x\})$	cumulative distribution function (CDF) of the r.v. X
$\mathbb{E}_X[\cdot]$	expectation operator with respect to r.v. X
$\mu \stackrel{\text{def}}{=} \mathbb{E}_X[X]$	the <i>mean</i> of X
$\sigma^2 \stackrel{\text{def}}{=} \mathbb{E}_X[(X - \mu)^2]$	the <i>variance</i> of X ; also denoted $\mathbf{Var}(X)$
$\sigma = \sqrt{\sigma^2}$	the <i>standard deviation</i> of X
$\mathcal{N}(\mu, \sigma)$	the <i>normal</i> distribution with mean μ and standard deviation σ . The probability density function for the r.v. $X \sim \mathcal{N}(\mu, \sigma)$ is $f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$
$Z \sim \mathcal{N}(0, 1)$	the <i>standard normal distribution</i>
$\Gamma(z)$	the Gamma function $\Gamma(z) \stackrel{\text{def}}{=} \int_{y=0}^{y=\infty} y^{z-1} e^{-y} dy$
$\mathbf{X} \stackrel{\text{def}}{=} (X_1, X_2, \dots, X_n)$	random sample of size n . Each $X_i \sim f_X$
$\mathbb{E}_{\mathbf{X}}[\cdot]$	expectation with respect to random sample \mathbf{X}
$\mathbf{x} \stackrel{\text{def}}{=} (x_1, x_2, \dots, x_n)$	sample of n observations
$f_{\mathbf{x}}$	empirical pmf of sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$
$F_{\mathbf{x}}$	empirical CDF of sample $\mathbf{x} = (x_1, x_2, \dots, x_n)$
$F_{\mathbf{x}}^{-1}$	inverse of the empirical CDF of the sample \mathbf{x}

Probability distributions reference

The following table shows the relevant probability distributions that can be imported from `scipy.stats`.

Math notation	Code notation	Parameters	Notes
$\mathcal{U}_d(\alpha, \beta)$	<code>randint(alpha, beta+1)</code>	$\alpha, \beta \in \mathbb{Z}$	
$\text{Bernoulli}(p)$	<code>bernoulli(p)</code>	$p \in [0, 1]$	
$\text{Binom}(n, p)$	<code>binom(n, p)</code>	$n \in \mathbb{N}_+, p \in [0, 1]$	
$\text{Pois}(\lambda)$	<code>poisson(0, 1/lam)</code>	$\lambda \in \mathbb{R}_+$	
$\text{Geom}(p)$	<code>geom(p)</code>	$p \in (0, 1)$	
$\text{NBinom}(r, p)$	<code>nbinom(r, p)</code>	$r \in \mathbb{N}_+, p \in (0, 1)$	
$\text{Hypergeom}(a, b, n)$	<code>hypergeom(a+b, a, n)</code>	$a, b, n \in \mathbb{N}_+$	
$\mathcal{U}(\alpha, \beta)$	<code>uniform(a, b-a)</code>	$\alpha, \beta \in \mathbb{R}$	
$\text{Expon}(\lambda)$	<code>expon(0, 1/lam)</code>	$\lambda \in \mathbb{R}_+$	
$\mathcal{N}(\mu, \sigma)$	<code>norm(mu, sigma)</code>	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}_+$	
$Z \sim \mathcal{N}(0, 1)$	<code>norm(0, 1)</code>	$\mu = 0, \sigma = 1$	
$T_\nu \sim \mathcal{T}(\nu)$	<code>tdist(df)</code>	$\nu \in \mathbb{R}_+$	alias for <code>t</code>
$Q_\nu \sim \chi^2(\nu)$	<code>chi2(df)</code>	$\nu \in \mathbb{N}_+$	
$F \sim \mathcal{F}(\nu_1, \nu_2)$	<code>fdist(df1, df2)</code>	$\nu_1, \nu_2 \in \mathbb{R}_+$	alias for <code>f</code>
$\text{Gamma}(\alpha, \lambda)$	<code>gammadist(a, 0, 1/lam)</code>	$\alpha, \lambda \in \mathbb{R}_+$	alias for <code>gamma</code>
$\text{Beta}(\alpha, \beta)$	<code>betadist(a, b)</code>	$\alpha, \beta \in \mathbb{R}_+$	alias for <code>beta</code>
$\text{LogNorm}(\mu, \sigma)$	<code>lognorm(mu, sigma)</code>	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}_+$	
$\text{Laplace}(\mu, b)$	<code>laplace(mu, b)</code>	$\mu \in \mathbb{R}, b \in \mathbb{R}_+$	

The notes column shows the actual model family name in `scipy.stats`, which we alias-import to avoid possible confusion. For example, when imposing Student's t -distribution, we use `from scipy.stats import t` as `tdist`.

Each of these models provides you with the same set of methods for computing probabilities, confidence intervals, and generating random values: `.pmf(x)` or `.pdf(x)`, `.cdf(b)`, `.expect(f)`, `.mean()`, `.std()`, `.rvs(n)`, etc.

Appendix C

Python tutorial

Click this link bit.ly/pytut3 to run this notebook interactively.

Abstract: In this tutorial, I'll introduce you to the basics of the Python programming language. Don't worry, learning Python is not complicated. You're not going to become a programmer or anything like that, I'm just going to show you **how to use Python as a calculator**. We'll start by setting up a Python coding environment on your computer (**JupyterLab Desktop**). We'll then cover basic building blocks like *expressions, variables, functions, lists, for-loops*, etc. By the end of this tutorial, you'll be familiar with the essential parts of the Python syntax needed to use the most kick-ass, powerful calculator that ever existed.

C.1 Introduction

Python is like a fancy calculator

Python commands are similar to the commands you give to a calculator when you want to compute something. The same way a calculator has different buttons for the various arithmetic operations, the Python language has a number of commands you can “run” or “execute.” Knowing how to use Python gives you access to hundreds of useful libraries and thousands of functions, which are like calculator buttons specialized for different domains: math, science, probability, statistics, etc.

Sales pitch about why you should learn Python

Here are some examples of the things you can do using Python:

- You can use Python as a basic calculator for doing arithmetic calculations using math operations like `+`, `-`, `*`, `/`, and other math functions.
- Python is also a scientific calculator since it provides functions like `sin`, `cos`, `log`, etc.
- You can use Python as a graphical calculator to plot functions and visualize data.
- Python is an extensible, programmable calculator that allows you to define your own functions and operations.
- Python provides powerful libraries for numerical computing (`numpy`), scientific computing (`scipy`), and symbolic math calculations (`sympy`).
- Python has libraries for data management (`pandas`), data visualization (e.g. `seaborn`), and statistics (e.g. `statsmodels`).

If any of these seems useful to you, then read on to get started learning Python!

How to learn Python

The fastest way to learn Python is through hands-on experimentation in an interactive coding environment like a Jupyter notebook. By running various Python commands and seeing the results they produce, you can quickly learn what Python functions (calculator buttons) are available, and what they do.

This tutorial is intended for absolute beginners with no prior programming experience. I'll guide you step-by-step through the code examples, and explain what the code does in plain English. You can then explore the Python code examples, run them on your own, or “play” with them by changing the commands and running them with different inputs to see how the outputs change. I've also prepared some exercises for you, so you can practice what you're learning.

C.2 Getting started

Follow the six steps outlined below to setup the JupyterLab Desktop computational environment on your computer that will allow you to run this tutorial interactively on your laptop or desktop. The process takes ten minutes, but it will greatly improve your Learning User Experience (LUX) for the rest of the tutorial.

Installing JupyterLab Desktop

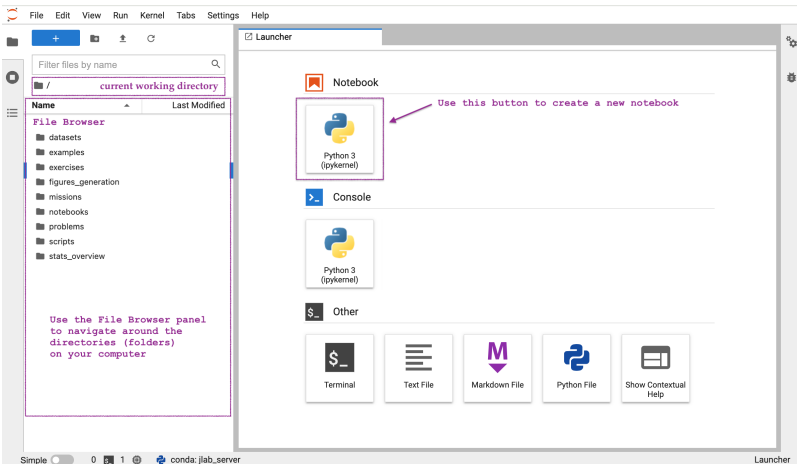
JupyterLab is a computing platform that makes it easy to run Python code. JupyterLab provides a notebook interface that allows you to run and edit Python commands interactively. **JupyterLab Desktop** is a convenient all-in-one application that you can install on your computer to run JupyterLab.

STEP 1: Download JupyterLab Desktop

Visit the web page github.com/jupyterlab/jupyterlab-desktop and choose the download link for your operating system. Complete the installation steps.

STEP 2: Start JupyterLab Desktop

Launch the JupyterLab Desktop application and choose the **New session...** option from the menu. You might be prompted to install a “bundled Python environment,” which you should accept. After this is done, you should see a window similar to the one shown below.



STEP 3: Create a new notebook

Click the **Python 3 (ipykernel)** button to create a new Jupyter notebook.

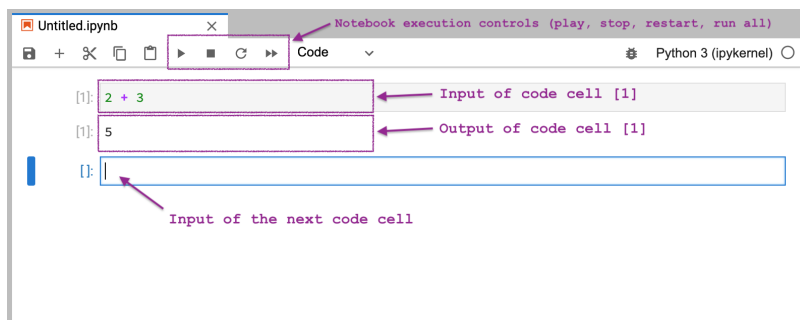
Notebooks are interactive documents

A Jupyter notebook consists of a sequence of cells, similar to how a text document consists of a series of paragraphs. The notebook you

created in STEP 3 has a single empty code cell, which is ready to accept Python commands.

STEP 4: Try a numerical calculation

Enter the expression `2 + 3` into the code cell, then press **SHIFT+ENTER** to run the code. You should see the result of the calculation displayed on a new line immediately below your input. The cursor will automatically move to the next code cell, as shown in the screenshot below.



Notebook execution controls The menu bar at the top of the notebook contains the notebook execution control buttons. The **play** button executes the code in the current cell, and is equivalent to pressing **SHIFT+ENTER**. The **stop** and **restart** buttons can be used to interrupt a computation that is stuck or taking too long. The **run all** button is useful when you want to rerun all the cells in the notebook from start to finish.

Code cells contain Python commands

Each code cell in a notebook is a command prompt that allows you to enter Python commands and “run” them by pressing **SHIFT+ENTER**, or by clicking the **play** button in the toolbar. Let’s look at the sample command from screenshot again. We can make Python compute the sum of two numbers by entering `2+3` in a code cell, then pressing **SHIFT+ENTER**, as show below.

```
2 + 3
```

```
5
```

When you *run* a code cell, you’re telling the computer to **evaluate** the Python command it contains. Python then displays the result of the

computation immediately below. In the above example, the input is the expression $2 + 3$, and the output is the number 5.

Let's now compute a more complicated math expression $(1 + 4)^2 - 3$. The Python equivalent of this math expression is shown below.

```
2*(1+4) - 3
```

7

Note the Python syntax for math operations is similar to the notation we use in math: addition is `+`, subtraction is `-`, multiplication is `*`, division is `/`, and we use the parentheses `(` and `)` to tell Python to compute $1+4$ first.

Running a code cell is similar to using the **EQUALS** button on a calculator: whatever math expression you entered, the calculator will compute its value and display its result. Our interaction with the Python calculator follow a similar script: we input some Python code, then press **SHIFT+ENTER** to make Python run the code and display the result.

STEP 5: Your turn to try!

Type in some expression involving numbers and operations in a new code cell, then run it by pressing **SHIFT+ENTER** or by using the **play** button in the toolbar.

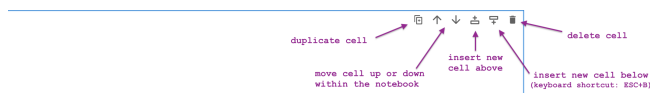
STEP 6: Download the tutorial notebook

Click this link bit.ly/pytut3nb to download the notebook `python_tutorial.ipynb` and save it to a permanent location on your computer (e.g. create a new folder `Documents/LearningPython/` and place the tutorial notebook in that folder). Switch back to the JupyterLab window and use the **File Browser** panel to navigate to the location on your computer where you saved the notebook `python_tutorial.ipynb`, then open it.

If you are reading this tutorial in print, I expect you to **follow along with the notebook on your computer for the remainder of this tutorial**, so that you can play with the code examples interactively. The tutorial notebook contains many pre-filled code cells with examples of Python commands for you to run. I encourage you to try editing the contents of the code cells and re-run them to see how the output

changes. Using this trial-and-error is the best way to learn Python commands.

Cell operations menu In the top-right corner of each code cell, there are buttons for performing several useful cell operations.



You can use the **insert new cell below** button to create a blank code cell at any place in the notebook and experiment with some variation of the code you are learning. Try running the code with different numbers, or come up with your own examples. **Think of this notebook as your personal playground for exploring Python commands.** Don't worry you can't break anything!

Learning on hard mode If you are feeling particularly ambitious, you can try manually typing-out all the code examples into a blank notebook. In general, I recommend that you type things out by yourself rather than relying on copy-paste. It's generally considered a good to practice to manually re-type code examples by yourself, because it improves your general typing skills, and builds you "muscle memory." You also get practice entering special characters like `[]`, `{ }`, `>`, `<`, `#`, which are often used in Python code.

C.3 Expressions and variables

Most Python commands involve computing the value of some Python expression and storing the resulting value in a variable. Since our goal is to learn Python, we'll start by explaining what Python expressions are and how to store values into variables.

Expressions

A Python expression is an arbitrary combination of values, operators, and function calls. We write expressions then Python "runs" them to compute their value. Here is an example of an expression involving several math operations, which is similar to the expression we saw earlier.

```
3*5 - 3
```

12

Let's now look at a fancier expression that shows other aspects of the Python syntax like lists and function calls. The following expression computes the sum of a list of three numbers using the function `sum`.

```
sum( [1,2,3] )
```

6

In Python, we define lists using the square brackets `[` and `]`. The code `[1,2,3]` defines a list of three elements: 1, 2, and 3. The function `sum` computes the sum of the list of numbers. The parentheses `(` and `)` are used to delimit the input to the function. We'll have a lot more to say about lists and functions later on in this tutorial. For now, I just wanted to show you an expression that involves something other than arithmetic operations, as a preview of what is to come.

Python comments

Comments are annotations in code cells intended for human readers. We write comments in Python by prefixing them with the character `#`. Python will ignore the `#` character and any text that comes after it on that line.

```
# this is a comment
```

The text `"# this is a comment"` is ignored by the Python interpreter. As far as Python is concerned, the above code cell is completely empty.

I've included many comments in this tutorial to provide an additional "narrative" for each line of code. I highly recommend that you read these comments, even if "technically" they are not part of the code. Some comments will tell you how to read the code (WHAT), explain how the code works (HOW), and, towards the end of the tutorial, switch to explaining the intent of each command (WHY).

Variables

A variable is a **name** we use to refer to some value. This is similar to how variables are used in math. Variables are used to represent constants, function inputs, function outputs, and all kinds of intermediate values in calculations. We use the assignment operator `=` to store values into variables.

The assignment operator =

In all the code examples above, we computed various Python expressions, but we didn't do anything with the results. The more common pattern in Python is to **store the result of an expression evaluation into a variable**.

For example, here is the code that computes the expression $2+3$ and stores the result in a variable named `x`.

```
x = 2+3    # calculate 2+3 and store the result into x
```

The result of this expression is that the value 5 gets saved in the variable `x`. Python evaluates this command from right-to-left: Python first looks at the expression $2+3$ and computes its result 5, then it sees the remainder of the statement `x =`, which means we want to store the result into the variable `x`. The assignment statement doesn't have any output, so running this code cell doesn't display anything.

To display the contents of the variable `x`, we can write its name in a code cell.

```
x    # print x
```

5

Note the value we assigned to the variable `x` is available in this code cell, even though we did the assignment in the previous cell. This is the whole point of the computational notebook environment: **every code cell runs in a “context” that includes the results from all the previous code cells**. When you use the **run all** button in the toolbar, and read the notebook from top-to-bottom, it's as if you're telling a story in which you define different characters (variables), and show them in action (computations).

In a notebook interface, the value of **the last expression in each code block gets displayed** as the output of that cell. The above code cell contains the simple expression `x` that consists only of the variable `x`. Python evaluated this expression to find its value, and displayed 5 as output since this is the value we assigned to this variable earlier in the notebook.

We can combine the commands “assign the result of $2+3$ to `x`” and “display `x`” into a single code cell, as shown below.

```
x = 2+3    # store 2+3 into x
x          # print x
```

The first line in this code block computes the expression $2+3$ and assigns the result to the variable `x`. The second line evaluates the variable `x` to display its contents.

We'll see this pattern of code repeated many times in the remainder of this notebook: code cells usually contain one or more lines of Python commands that compute some quantity of interest and store its value into a some variable, then the last line of the code cells evaluates this variable to show the final result of the computations.

Exercise 1: using the assignment operator

Imitate the above code examples to write the Python code that creates another variable `y` that contains the value of the expression $2(1 + 4) + 3$. On a separate line, evaluate the variable `y` to show the result.

```
# Instructions: put your answer in this code cell
```

Click [here](#) for the solution.

Summary: how to read (pronounce) the assignment operator

The meaning of the assignment operator `=` is not the same as the meaning of the operator `=` used for writing math equations, so you should *not* read `x = 5` as “`x` equals 5.” The assignment statement `x = 5` tells Python to store the value 5 (the right-hand side) into the variable `x` (the left-hand side), so the correct way to read this statement is, “set `x` to 5”, “Put 5 into `x`”, or “record 5 under the name `x`”. The symbol `=` is not a passive statement of equality, but an active verb telling Python to store the result of an expression into a particular variable.

To summarize, the syntax of an assignment statement is as follows:

```
<place> = <some expression>
```

The assignment operator `=` stores the value of the expression `<some expression>` into the memory location `<place>`, which is usually a variable name. Later in this tutorial, we'll learn how to store values inside containers like lists and dictionaries, but for now you can assume that `<place>` is the name of a variable.

Using multi-line expressions for step-by-step calculations

Okay so how does storing values into variables help us with real-world practical calculations? Let me show you an example of a complicated calculation that we can do easily in Python by defining variables to store the intermediate values of the calculation.

Example 1: number of seconds in one week

Let's say we need to calculate how many seconds there are in one week. We know there are 60 seconds in one minute, 60 minutes in one hour, 24 hours in one day, and 7 days in one week. We can calculate the number of seconds in one week step by step, where in each step we perform a simple calculation. We'll define several intermediate variables and give the variable descriptive names to help us keep track of the intermediate quantities used in the calculation.

```
one_min = 60           # number of seconds in one minute
one_hour = one_min * 60 # calc. number of sec in one hour
one_day = one_hour * 24 # calc. number of sec in one day
one_week = one_day * 7  # calc. number of sec in one week
one_week           # print the value of one_week
```

604800

Note we can use the underscore `_` as part of variable names. This is a [common pattern](#) for variable names in Python code, because the name `some_name` is easier to read than `somename`.

Okay, enough theory. It's time to see if you can use your new knowledge about Python variables, expressions, and the assignment operator to solve these exercises.

Exercise 2: weight conversion

Johnny weights 107 kg and wants to know his weight in pounds. One kilogram is equivalent to 2.2 lbs. Can you write the Python expression for computing Johnny's weight in pounds?

Below, I have created a code cell where you can write your answer, and even started filling in the code for you by defining the variable `weight_in_kg` that contains the weight in kilograms. To complete this exercise, I want you to replace the `...` placeholder with a **Python expression that converts the value in the variable `weight_in_kg` to pounds**, so the converted value gets assigned to `weight_in_lb`.

After that, add a new line to the code block to display the variable `weight_in_lb`.

```
# Instructions: replace ... with your answer

weight_in_kg = 107 # store 107 into weight_in_kg
weight_in_lb = ...
```

Click [here](#) for the solution.

Exercise 3: tax calculation

You're buying an item at the store and the price is \$50 dollars. The government imposes a 10% tax on your purchase. Calculate the total you'll have to pay, including the 10% tax.

```
# Instructions: replace the ...s with your calculations

price = 50.0
taxes = ...
total = ...
```

Click [here](#) for the solution.

Exercise 4: temperature conversion

The formula for converting temperatures from Celsius to Fahrenheit is $F = \frac{9}{5} \cdot C + 32$. Given the variable `C` that contains the current temperature in Celsius, write the expression that calculates the current temperature in Fahrenheit and store the answer in a new variable named `F`.

```
# Instructions: replace the ... with a Python expression

C = 20
F = ...
```

Check your formula works by changing the value of `C`, and re-running the code cell. When `C = 100`, the temperature in Fahrenheit should be 212.0.

Click [here](#) for the solution.

Variable types

Every variable in Python has a *type*, which tells you what kind of data it contains, and what kind of operations you can do with it. There

are two types of variables for storing numbers, and another type for storing text. There are also several types of “container variables” like lists and dictionaries.

Here is a list of the most common types of variables in Python:

- **Integers** (int): used to store whole numbers like 42, 65, 78, -4, -200, etc. Python integers are roughly equivalent to the math set of integers \mathbb{Z} .
- **Floating point numbers** (float): used to store decimals like 4.6, 78.5, 1000.0 = 1e3, 0.001 = 1e-3, 123.456 = 1.23456e2, etc.
- **Lists** (list): ordered container for other values. For example, the list [61, 79, 98, 72] contains four integers. The beginning and the end of the list are denoted by the square brackets [and], and its elements are separated by commas.
- **Strings** (str): used to store text like "Hello", "Hello everyone". Strings can be denoted using either double quotes "Hi" or single quotes 'Hi'.
- **Boolean values** (bool): logic variables with only two possible values, True or False.

Let's look at some examples of different types of Python variables, an integer, a floating point number, a list, a string, and a boolean value.

```
score = 98                # an int
average = 77.5            # a float
scores = [61, 79, 98, 72] # a list
message = "Hello everyone" # a str
above_the_average = True  # a bool
```

To see the contents of the variable `score`, we simply type its name.

```
score
```

```
98
```

What type of variable is the variable `score`? To see the *type* of a variable, we can call the function `type` like this:

```
type(score)
```

```
int
```

The output tells us that `score` is of type `int` (an integer).

Exercise 5: values and types of variables

Print the *contents* and the *type* of the variables `average`, `scores`, `message`, and `above_the_average` that were defined above.

```
# Instructions: Show the contents of each variable then
#               use the function `type` to obtain its type.
```

Click [here](#) for the solution.

Technical jargon: objects and methods

The technical term *object* is used to describe the fundamental building blocks we use to store data when programming. We store different types of data into different types of objects, and the integers, floating point numbers, lists, strings, and boolean values are examples of different types of Python objects. Calling the function `type` on the object `obj`, `type(obj)`, tells us the type of the object `obj`.

Objects have certain functions “attached” to them, which we call *methods*. Different types of objects have different kinds of methods, which make it easy to manipulate the data stored in them. Earlier, we defined the Python list object `scores = [61, 79, 98, 72]`, which is a container that holds four numbers. The list `scores` has methods like `.append`, `.insert`, `.extend`, `.reverse`, `.sort`, etc. These methods allow us to perform operations on the underlying data in the list. For example, if we want to sort the values in the list in increasing order, we can call the method `scores.sort()` and the list will become `[61, 72, 79, 98]`.

Every value in Python is an *object*. Learning Python requires being familiar with the different object types and their methods. For example, later on in this tutorial we’ll dedicate a whole section on lists, where we’ll describe their methods and the use cases for each method. There are similar sections for boolean values, strings, and dictionaries. The different types of Python objects are like different types of toys available for you to play with. Remember that you can use the function `type` to find out what kind of toy you have in your hands.

C.4 Functions

Functions are the essential building blocks of Python programs. Functions allow us to encapsulate any sequence of operations as a reusable piece of functionality. You can think of a function as a chunk

of code that you define once, then use multiple times by “calling” it from other places in your code.

Calling functions

The Python syntax for calling the function named “*fun*” with the input “*arg*” is to wrap the input in parentheses: `fun(arg)`. The syntax for calling functions in Python is borrowed from the math notation $f(a)$, which is used to describe evaluating the function f on the input a . A Python function takes certain variable(s) as inputs and produces certain variable(s) as outputs. In programming, function inputs are called *arguments* and the output of the function is called its *return value*.

Python built-in functions

Python functions are like the different calculator buttons you can press to perform computations. Here is a list of the most common Python functions:

- `type(obj)`: tells us the type of the object `obj`.
- `sum(mylist)`: calculates the sum of the values in the list `mylist`.
- `print(obj1,obj2,...)`: displays the values of the objects `obj1`, `obj2`, etc.
- `len(mylist)`: returns the length of the list object `mylist`.
- `help(obj)`: displays help information about the object `obj`.
- `range(a,b)`: creates the list of numbers `[a,a+1,...,b-1]`.
- `str(obj)`: converts the object `obj` into a text string (`str`).

You’ll learn to use all these functions (and many others) in later sections of this tutorial.

Multiple arguments

Some functions can accept multiple arguments as inputs. We use commas to separate the different arguments to a function: `fun(arg1,arg2,arg3)`. For example, if we wanted to print several values on the same line of output, we can call the function `print` with multiple arguments, as shown below.

```
print("The average is", average, "percent.")
```

The average is 77.5 percent.

In the above code cell, we call the function `print` with three arguments: - The first argument is “The average is” (a string) -

The second argument is `average` (a float) - The third argument is `"percent."` (another string)

Calling the function `print` in this way converts all arguments to strings, combines them all together into a single string using space " " as a separator, then displays the result on a single line.

Keyword arguments

Some functions accept *optional arguments* (also called *options* or *keyword arguments*) that modify the function's behaviour. For example, the `print` function accepts the keyword argument `sep` (separator) that specifies the character used to separate the different arguments. The default value for the option `sep` is a single space " ", which is why the number `77.5` (second argument) appears separated by spaces from the string arguments (first and third) in the output of the above cell.

We can specify a different value for the `sep` keyword argument, if we want to use a different separator when printing multiple values. For example, if we want to print different values separated by `=` in the poutput, we can specify the option `sep="="` when calling the `print` fuction, as shown below.

```
print("average", average, sep="=")
```

```
average=77.5
```

Python syntax for defining new functions

Python makes it easy to define your own functions, which is like **adding new buttons to the calculator**. To define a new function called `fun`, we use the following syntax:

```
def fun(arg):           # define the function `fun`
    <calculation step 1> # first line of calculations
    <calculation step 2> # second line of calculations
    return <out>        # set the output of the function
```

Defining the Python function `fun` is like adding the new button `fun` to the Python calculator. Let's go over the code example above line-by-line to explain all the new elements of syntax. We start with the Python keyword `def`, then give the name of the function we want to define, which is `fun` in this case. Next, we specify the arguments that the function expects to receive as inputs inside parentheses. In this example, the function `fun` takes a single input called `arg`. The colon `:` indicates the beginning of the function body. The function

body is an *indented code block* (each line begins with four spaces) that specifies the calculations that the function is supposed to perform on the input `arg`. The last line in the function body is a `return` statement that tells us the output of the function (its *return value*).

Defining the function `fun`

Let's make this example concrete, by showing the definition of the Python function `fun` that corresponds to the math function $f(x) = 10x - 8$.

```
def fun(x):  
    ten_x = 10 * x  
    out = ten_x - 8  
    return out
```

The function `fun` accepts a single input `x`, which we assume is some kind of number (an `int` or a `float`, or some other number-like Python object). The first calculation step `fun` performs on the input `x` will be to multiply it by 10 and assign the result to a new variable aptly named `ten_x`. The second calculation step creates a new variable `out` that contains the value of the first calculation minus 8, so $10 \times x - 8$, which is the value we want the function to produce as output. The last line in the function body specifies we want the function to return the value of `out` as output.

The variables `ten_x` and `out` that we used to store the intermediate steps of the calculation are defined **inside the function**, which is a different, temporary “space” for variable names that is local to the function's body. Without going into too much details, you just need to know that you have the freedom to define as many temporary variables as you want inside the body of a function definition, and you don't have to worry about these temporary variables “leaking out” or overwriting other variables in the outer context where you are calling the function.

Calling the function `fun`

Once we have defined the function `fun` using the `def`-syntax above, we can **call** the function in code cells below this one. For example, calling `fun(5)` will perform the calculations steps 1 and 2 on the value 5, then return the value output calculated from this input 5, which is:

```
fun(5)
```

42

Evaluating the expression `fun(5)` produces the value 42. This means anywhere you see the expression `fun(5)`, you can mentally replace it with the value 42, which is the output of `fun` when the input is 5. The ability to replace the the function call `fun(5)` by its value 42 is called the *substitution property* in math, and *referential transparency* in computer science.

I know this function may seem complicated at first sight, but I assure you there is nothing new going on here. We're just piggybacking on the math convention for defining functions. If I define the math function f using the equation $f(x) = 10x - 8$, and I ask you to evaluate the math expression $f(5) + 7$, you know that $f(5) = 42$ and use the substitution property to calculate the answer $f(5) + 7 = 42 + 7 = 49$.

Let's look at some examples of Python function definitions and calls.

Example 2: adding the double button to the calculator

Let's define a new Python function called `double`, which takes a number as input and multiplies the number by two.

```
def double(x): # define the function `double` with input `x`  
    y = 2 * x # compute 2 times `x`; store result in `y`  
    return y # return the value of `y` as output
```

Note the body of the function (the second and third lines) is indented by four spaces. In Python, we use indentation to delimit when a code block starts and when it ends. In the above example, the function's body contains only two lines of code, but in general, Python functions can contain dozens or even hundreds of lines of code: any sequence of operations you might want to perform on the input `x` to calculate the output `y`.

To call the function `double`, we use the function name followed by the function's input argument in parentheses, as we saw earlier.

```
double(6) # call the function `double` on the input `6`
```

12

Let's narrate what happened in the above code cell from Python's perspective. When Python sees the expression `double(6)` it recognizes it as a function call. It then looks for the definition of the function `double` and finds the `def` statement in the previous code cell, which tells it what steps need to be performed on the input. It then

performs the steps in the function body with the value 6 stored in the variable `x`, which results in 12 as the return value. Visit the link tinyurl.com/bdzt69s9 to see a visualization of what happens when we call the function `double` with different inputs.

Example 3: tax calculating function

Let's define a function `add_taxes` that adds 10% tax to a given purchase price and returns the total of the price plus taxes.

```
def add_taxes(price):           # define the function `add_taxes`
    taxes = 0.10 * price        # compute 10% taxes
    total = price + taxes        # compute the total
    return total                # return the value in `total`
```

We can use the function `add_taxes` to calculate total cost of purchasing items with different prices.

```
add_taxes(50)    # calculate the total cost of a $50 item
```

55.0

```
add_taxes(100)   # calculate the total cost of a $100 item
```

110.0

Try to narrate what happens when Python evaluates the function call expressions `add_taxes(50)` and `add_taxes(100)`. I want you to imitate the narrative from the point of view of Python like the one for the function call `double(6)` above, but narrating the steps performed by the function `add_taxes` on its input price. This is an important exercise that I highly recommend so you'll become comfortable with the semantics of function calls. Click this link tinyurl.com/muufc5cn to see a visualization of the `add_taxes` function calls.

Example 4: calculating the mean of a list of numbers

The formula for computing the mean (average value) of a list of numbers $[x_1, x_2, \dots, x_n]$ is

$$\bar{x} = (x_1 + x_2 + \dots + x_n) / n.$$

In words, the average value is the sum of the values, divided by the length of the list.

Let's define a Python function called `mean` that computes the mean of the values provided as the input called `values`.

```
def mean(values):           # define the function `mean`
    n = len(values)         # compute the length of the list
    avg = sum(values) / n   # compute the average
    return avg              # return the value in `avg`
```

The calculations this function performs are the same as the math formula. Let's try the function on a list of numbers.

```
mean([1,2,3,4])           # calculate the mean of the values
↪ [1,2,3,4]
```

2.5

Indeed, $(1 + 2 + 3 + 4)/4 = 10/4 = 2.5$, so the function `mean` seems to be working as expected.

Exercise 6: temperature conversion function

Now it's your turn to try using the `def`-syntax for defining Python functions. I want you to define a Python function called `temp_C_to_F` that converts temperatures from Celsius C into Fahrenheit F . The math formula for converting a temperature in Celsius to a temperature in Fahrenheit is $F = \frac{9}{5} \cdot C + 32$.

Hint: You can reuse the code from **Exercise 4**.

```
# Instructions: replace ... with your answer

def temp_C_to_F(C):
    ...
```

Click [here](#) for the solution.

There is a lot more to learn about Python functions, but there are also other topics I want to show you, so we'll have to move on now.

C.5 Python lists and for-loops

Python lists allow us to easily manipulate thousands or millions of values. A `for`-loop is a programming construct used to repeat some operation multiple times. We often use `for`-loops to perform a calculation for each element in a list, which is a very common computational task.

We'll start by describing how Python lists work, then show some examples of using for-loops for list calculations.

Lists

The Python syntax for creating a list starts with an opening square bracket `[`, followed by the list elements separated by commas `,`, and ends with a closing bracket `]`. For example, here is how to define the list `scores` that contains four integers:

```
scores = [61, 79, 98, 72]    # define a list of four numbers
scores
```

```
[61, 79, 98, 72]
```

A list has a *length* property, which you can obtain by calling the function `len` on it.

```
len(scores)    # compute the length of the list `scores`
```

```
4
```

We use the `in` operator to check if a list contains a certain element.

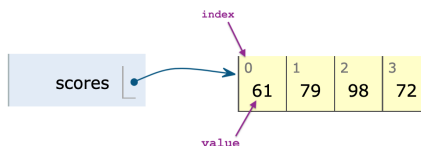
```
98 in scores    # Is the number 98 contained in list `scores`?
```

```
True
```

The result is the boolean value `True`, which means the answer is: “Yes, the number 98 appears in the list `scores`.”

Accessing elements of a list

We access the individual elements of the list `scores` using the square brackets syntax `scores[<idx>]`, where `<idx>` is the 0-based index of the element we want to access. You can think of indices as analogous to the street numbers used to address different locations on a street.



The visualization above shows the different indices and values of the elements in the list `scores`. The first element of the list has index 0,

the second element has index 1, and so on. The last element has an index equal to the length of the list minus one.

Here are some examples of the square brackets syntax for accessing the elements inside the list `scores`.

```
scores[0]    # first element in the list `scores`
```

61

```
scores[1]    # second element in the list `scores`
```

79

```
scores[3]    # last element in the list `scores`
```

72

Note the semantics used to *access* list elements is completely separate from the semantics used to *define* lists, even though both are based on the square brackets syntax `[and]`. The `[and]` in the expression `[1,2,3]` means “create a list,” while the `[and]` in `mylist[0]` means “get the first element of `mylist`.”

List slicing

You can extract a subset of a list using the “slice” syntax `a:b`, which corresponds to the range of indices `a`, `a+1`, ..., `b-1`. For example, if you want to extract the first three elements in the list `scores`, use the slice `0:3`, which is equivalent to the indices 0, 1, and 2.

```
scores[0:3]   # first three elements of the list `scores`
```

[61, 79, 98]

The result of selecting a slice from a list is another list.

List-related Python functions

Here are some Python built-in functions for working with lists:

- `len`: calculate length of the list
- `sum`: add together all the values in a list of numbers
- `max`: find the largest value in a list of numbers
- `min`: find the smallest value in a list of numbers
- `sorted`: return a copy of the list in sorted order
- `print`: print the contents of the list

Use the code cell below to call each of these functions on the list `scores` and check that you get the answer you expect.

```
# e.g. len(scores)
```

List methods

List objects can be modified using their methods: `.sort()`, `.append(x)`, `.pop()`, and `.reverse()`. Remember a *method* is a function “attached” to a given object that we use to perform certain operations on that object.

We’ll now look at some examples that illustrate how to use the dot-syntax for calling list methods.

To sort the list of `scores`, you can call its `.sort()` method.

```
scores.sort()    # sort the list in increasing order
scores           # print `scores` to show it is now sorted
```

```
[61, 72, 79, 98]
```

Use the `.append(newel)` method to add a new element `newel` to the list (added to to the right end).

```
scores.append(22) # add `22` to the end of the list
scores           # print `scores` to show `22` was added
```

```
[61, 72, 79, 98, 22]
```

The method `.pop()` extracts the last element of the list (from the right end).

```
scores.pop()    # pull out the last element of the list
```

```
22
```

You can think of `.pop()` as the “undo operation” of the `.append(newel)` operation.

To reverse the order of elements in the list, call its `.reverse()` method:

```
scores.reverse() # reverse the order of values in the list
scores           # print `scores` to show it is reversed
```

```
[98, 79, 72, 61]
```

Other useful list methods include `.insert(idx,newel)` and `.remove(el)`, which allow you to insert new elements or to remove elements at arbitrary places in the list.

For loops

The `for`-loop is a programming concept that allows us to repeat some operation(s) multiple times. The most common use case of a `for`-loop is to perform some calculation for each element in a list.

The syntax of a Python `for`-loop looks like this:

```
for <element> in <container>:  
    <operation 1 using `element`>  
    <operation 2 using `element`>  
    <operation 3 using `element`>
```

This code tells Python to repeat the operations 1, 2, and 3 **for each** element `<element>` in the list `<container>`. The operations we want to repeat are indented by four spaces, which indicates they are part of the body of the `for`-loop. Recall we saw indented code blocks previously when defining Python functions. The indentation syntax serves the same purpose here: it delimits which three lines are **inside** the `for` loop.

When the `for`-loop is evaluated, the variable `<element>` will take on the value of the different elements in the list `<container>` for each iteration of the `for` loop, and there will be a total of `len(<container>)` iterations.

Example 5: print all the scores

We start with a basic example of a `for`-loop that simply prints the value of each element.

```
scores = [61, 79, 98, 72]    # define a list of four numbers  
for score in scores:        # repeat for each score:  
    print(score)            #     print the value of `score`
```

```
61  
79  
98  
72
```

A `for`-loop describes a certain action (or actions) that you want Python to repeat. The `for`-loop shown in the code cell above instructs Python to repeat the action `print(score)` four times, once

for each score in the list `scores`. The operation we want to repeat is indented by four spaces and hence **inside** the for loop.

Example 6: compute the average score

The math formula for computing the mean (average value) of a list of numbers $[x_1, x_2, \dots, x_n]$ is: $\bar{x} = (x_1 + x_2 + \dots + x_n)/n$. We previously computed the average using the functions `sum` and `len`: `avg = sum(grades)/len(grades)`, but suppose we don't have access to the function `sum` for some reason, and we want to compute the sum of the grades using a for-loop. Here is the code for computing the average score.

```
total = 0                                # used to store cumulative sums
for score in scores:                     # repeat for each score in the list
    total = total + score                 # add score to the running total

# At this point, total contains the sum of the scores.

avg = total / len(scores)               # compute the average
avg                                     # print `avg`
```

77.5

On the first line, we define the temporary variable `total` (initially set to 0), which we use to store the intermediate values of the sum after each iteration of the for-loop. Next, the for-loop tells Python to go through the list `scores`. For each score in the list, we perform the operation `total = total + score`. After the for loop is finished, the sum of all the scores is stored in the variable `total`, as if we calculated `total = sum(scores)`. To obtain the average, we divide `total` by the length of the list.

Visit this link tinyurl.com/3bpx887v to see a step-by-step visualization of the four steps (iterations) of the above for-loop execution, which shows how the variable `total` grows after each iteration. See also this [blog post](#) for more background on for-loops.

Loop variable names are arbitrary The name of the variable you use for the current element of the list during each iteration of a for-loop is totally up to you. We generally try to choose a name that accurately describes the list elements. For example, if the list is called `objs`, it makes sense to use `obj` as the name of the loop variable: “for `obj` in `objs`:”. Given a list `profiles`, we would write the for-loop as “for `profile` in `profiles`:”. Given a list `nodes`, we would use a for-loop like “for `node` in `nodes`:”, etc.

List comprehension (bonus topic)

We often need to transform a list of values by applying the same transformation to each value in the list. Using the standard for-loop syntax to apply the function `fun` to all the elements in a list requires four lines of code:

```
newvalues = []                # storage space for the result
for value in values:          # repeat for each value in the list:
    newvalue = fun(value)     # compute `fun(value)`
    newvalues.append(newvalue) # add to the list `newvalues`
```

Python provides the *list comprehension* syntax, which is shorthand for describing a for loop as a single line of code.

```
newvalues = [fun(value) for value in values]
# apply `fun` to all `values` and put the result in `newvalues`
```

Here is an example of using the list comprehension syntax to apply the function `double` on all the numbers in the list `numbers`, and store the result in a new list called `doubles`.

```
numbers = [1, 2, 3, 4, 5]
doubles = [double(number) for number in numbers]
doubles
```

```
[2, 4, 6, 8, 10]
```

List comprehension is often used express list transformations in Python code, since it easier to understand (reads like an English sentence).

C.6 Boolean variables and conditionals

Boolean variables

Boolean variables represent logical conditions that are either `True` or `False`. We obtain boolean values when we perform numerical comparisons using the operators like `<`, `>=`, `<=`, `==` (equal to), `!=` (not equal to).

```
x = 3    # store 3 in the variable `x`
x > 2    # Is `x` greater than 2?
```

```
True
```


Another context where boolean values come up is when we use the `in` operator to check if an object is part of a list (or other container).

```
3 in [1,2,3,4]    # Is the number 3 contained in [1,2,3,4]?
```

True

Conditional statements

Boolean values are used in conditional statements, which are blocks of Python code that may or may not be executed, depending on the value of a boolean value. The Python keywords `if`, `elif`, and `else` are used to create conditional statements.

Let's start with some examples of `if` statements.

```
if True:
    print("this code will run")

if False:
    print("this code will not run")
```

this code will run

The indented code block inside an `if` statement is executed only when the `if` condition is `True`. We can extend an `if` statement with an `else` clause, which allows us to specify a different code block that will run if the condition is `False`.

```
x = 3

if x > 2:
    print("x is greater than 2")
else:
    print("x is less than or equal to 2")
```

x is greater than 2

We can check multiple conditions by inserting additional `elif` clauses in the middle of the `if-else` statement, as shown in the next example.

```
temp = 8

if temp > 22:
    print("It's hot.")
elif temp < 10:
    print("It's cold.")
```

```
else:
    print("It's OK.")
```

It's cold.

When Python sees this `if-elif-else` statement, it checks all the `if` and `elif` conditions one-by-one, and if it finds a condition that is `True`, it will execute the corresponding code block. If none of the `if` and `elif` conditions are `True`, then the `else` code block will run. At the end of this `if-elif-else` statement, exactly one of the `print` commands will be executed.

Exercise 7: temperature assessment

Add another condition to the temperature code to print `It's very hot!` if the temperature is above 30.

```
temp = 33

# Instructions: edit the code below to add the new condition

if temp > 22:
    print("It's hot.")
elif temp < 10:
    print("It's cold.")
else:
    print("It's OK.")
```

Click [here](#) for the solution.

Boolean expressions

You can use the logical operations `and`, `or`, and `not` to combine individual boolean values into larger boolean expressions that check multiple conditions.

The result of an `and` operator (logical conjunction) is `True` only if both operands are `true`, and `False` otherwise, as shown in the following table.

Boolean expression	Value
True and True	True
True and False	False
False and True	False
False and False	False

The `or` operator (logical disjunction) will result in `True` as long as at least one operand is `True`:

Boolean expression	Value
<code>True or True</code>	<code>True</code>
<code>True or False</code>	<code>True</code>
<code>False or True</code>	<code>True</code>
<code>False or False</code>	<code>False</code>

The `not` operator performs the negation of a boolean value.

Boolean expression	Value
<code>not True</code>	<code>False</code>
<code>not False</code>	<code>True</code>

Here is an example of using the `and` operator to check two conditions simultaneously.

```
x = 3
x >= 0 and x <= 10    # Is `x` between 0 and 10?
```

`True`

Try changing the value of `x` to make the boolean expression `False`.

Exercise 8: water phases

The phase of water depends on temperature. The three possible phases of water are "gas" (water vapour), "liquid" (water), and "solid" (ice). The table below shows the phase of water depending on the temperature `temp`, expressed as math inequalities.

Temperature range	Phase
<code>temp >= 100</code>	gas
<code>0 <= temp < 100</code>	liquid
<code>temp < 0</code>	solid

Let's see if you can fill in the `if-elif-else` statement below to print the correct phase depending on the variable `temp`.

```
temp = 90    # the current water temperature

# Instructions: Fill-in the code of the if-elif-else by
#               replacing the ...s with conditions.

if ...:
    print(...)
elif ...:
    print(...)
else:
    print(...)
```

Click [here](#) for the solution.

Exercise 9: assigning letter grades

Teacher Joelle has computed the final scores of the students as a percentage (a score out of 100). The school where she teaches requires her to convert each student's score to a letter grade according to the following grading scale:

Grade	Numerical score interval
A	90% - 100%
B	70% - 89.999...%
C	50% - 69.999...%
F	0% - 49.999...%

Can you write an if-elif-elif-else statement that looks at the score variable (a number between 0 and 100), and prints the appropriate letter grade for that score.

```
score = 90    # student score

# Instructions: Fill in the if-elif-elif-else statement by
#               replacing the ...s with the right conditions.

if ...:
    print(...)
elif ...:
    print(...)
elif ...:
    print(...)
else:
    print(...)
```

Click [here](#) for the solution.

C.7 Other data structures

We already discussed lists, which are the most common data structure (container for data) in Python. In this section, we'll briefly introduce some other Python data structures you might encounter.

Strings

In Python, strings (type `str`) are the containers we use for storing text. We can create a string by enclosing text in single quotes `'` or double quotes `"`.

```
message = "Hello everyone"  
type(message)
```

```
str
```

String concatenation

We can use the `+` operator to concatenate (combine) two strings.

```
name = "Julie"  
message = "Hello " + name + "!"  
message
```

```
'Hello Julie!'
```

Strings behave like lists of characters

You can think of the string `"abc"` as a list of three characters `["a", "b", "c"]`. We can use list syntax to access the individual characters in the list. To illustrate this list-like behaviour of strings, let's define a string of length 26 that contains all the lowercase Latin letters.

```
letters = "abcdefghijklmnopqrstuvwxyz"  
letters
```

```
'abcdefghijklmnopqrstuvwxyz'
```

```
len(letters)  # length of the string
```

```
26
```

We can access the individual characters within the using the square brackets. For example, the index of the first letter in the string is 0:

```
letters[0]
```

```
'a'
```

The index of the letter "b" in the string `letters` is 1:

```
letters[1]
```

```
'b'
```

The last element in a list of 26 letters has index 25

```
letters[25]
```

```
'z'
```

We can use the slice syntax to extract a substring that spans a certain range of indices. For example, the first four letters of the alphabet are:

```
letters[0:4]
```

```
'abcd'
```

The syntax `0:4` is a shorthand for the expression `slice(0,4)`, which corresponds to the range of indices from 0 (inclusive) to 4 (noninclusive): `[0,1,2,3]`.

Tuples

A tuple is similar to a list, but with fewer features. The syntax for defining a tuples uses the parentheses (and), and the elements of the tuple are separated by commas ,.

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

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

Actually, the parentheses are optional: we can define a tuple by simply using a bunch of comma-separated values.

```
1, 2, 3
```

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

In computational notebooks environments, we can use the comma-separated values tuple syntax to print multiple values as the output

of a code cell. For example, here is how we can display the first, second, and last characters from the string `letters` on a single line.

```
letters[0], letters[1], letters[25]
```

```
('a', 'b', 'z')
```

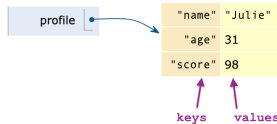
Dictionaries

One of the most useful data structures used in programming are *associative arrays*, which are also known as *lookup tables*, or *hash-tables*. Associative arrays are called dictionaries (type `dict`) in Python. A dictionary is a container of key-value pairs. The syntax for creating a dictionary is based on `"key": value` pairs, separated by commas, wrapped in curly braces `{` and `}`. For example, the code below defines the dictionary `profile` that contains three key-value pairs.

```
profile = {"name": "Julie", "age": 31, "score": 98}
profile
```

```
{'name': 'Julie', 'age': 31, 'score': 98}
```

A dictionary is a mapping from keys to values, as illustrated below.



We use the keys to access the different values in the dictionary. The keys of the `profile` dictionary are `"name"`, `"age"`, and `"score"`.

```
profile.keys()
```

```
dict_keys(['name', 'age', 'score'])
```

The values associated with these keys are: `"Julie"` (a string), `31` (an integer), and `98` (another integer).

```
profile.values()
```

```
dict_values(['Julie', 31, 98])
```

We access the value in the dictionary using the square brackets syntax. For example, to get the value associated with key "score", we use:

```
profile["score"]
```

98

Recall that we used the square bracket syntax earlier for accessing the values within a list. Indeed, lists and dictionaries are both “container” objects, so we use square brackets syntax to access the elements within them.

You can change the value associated with a key by assigning a new value to it, as follows:

```
profile["score"] = 77
profile
```

```
{'name': 'Julie', 'age': 31, 'score': 77}
```

You can also add a new key-value pair to the dictionary by assigning a value to a key that doesn't exist yet:

```
profile["email"] = "julie@site.org"
profile
```

```
{'name': 'Julie', 'age': 31, 'score': 77, 'email': 'julie@site.
  ↳ org'}
```

Note the profile dictionary now has a new key "email" and the value "julie@site.org" stored under that key.

Exercise 10: creating a new profile dictionary

Create a new dictionary called `profile2` with the same keys as dictionary `profile` for the user "Alex" who has age 42 and score 65.

```
profile2 = ... # replace ... with your answer
```

Click [here](#) for the solution.

Type conversions

We sometimes need to convert between variables of different types. The functions for conversing types have the same name as the object types they convert to.

- `int` : converts any expression into an `int`
- `float`: converts any expression into a `float`
- `str`: converts any expression into a `str`

For example, we're starting with the number "42.5" (as a string) stored in the variable `numstr`, and we want to use this number for some arithmetic calculation.

```
numstr = "42.5"  
type(numstr)
```

`str`

Before we can do arithmetic calculations with this input, we need to convert this Python string into a Python numerical type, like a floating point number. We call the function `float` to do this conversion to a `float` type.

```
numfloat = float(numstr)  
numfloat
```

42.5

```
type(numfloat)
```

`float`

Now that we have converted the string "42.5" into the float 42.5 (stored in the variable `numfloat`), we can do numerical operation like `+`, `-`, and `*` with this number.

```
6*numfloat - 7
```

248.0

Example 7: compute the sum of two numbers

Suppose we're given two numbers m and n , and we want to compute their sum $m + n$. However, the numbers m and n are given to us as strings, so we don't get the expected result when we add them together.

```
mstr = "2.1"  
nstr = "3.4"  
mstr + nstr
```

'2.13.4'

This is because the addition operator `+` for strings means concatenate, not add. We have to manually convert the strings to numbers, if we want to add them together as numbers.

```
mfloat = float(mstr)
nfloat = float(nstr)
mfloat + nfloat
```

5.5

Exercise 11: type conversion and sum

Write the Python code that converts values in the list `prices` to floating point numbers and adds them together.

Hint: use a `for`-loop or the list comprehension syntax.

```
prices = ["22.2", "10.1", "33.3"]

# write the code to compute the total price
```

Click [here](#) for the solution.

C.8 Python packages and modules

All the code examples we showed above were using Python *built-in* functions and data types, but that is only a small part of the functionality available in Python. There are hundreds of Python packages and modules that provide additional functions and data types for all kinds of applications. There are Python modules for processing different data files, making web requests, doing efficient numerical computing, calculating statistics, etc. The list is almost endless!

Import statements

We use the `import` keyword to load a Python module and make it available in the current notebook environment. The command to import the module `<module>` in the current context is:

```
import <module>
```

After this import statement, we can use all the functions from the module `<module>` by calling them using the prefix `<module>.`, which is called the “dot notation” for accessing names within the `<module>`

namespace. For example, here is how we can import the `statistics` module and use the function `statistics.mean` to compute the mean of a list of numbers.

```
import statistics
statistics.mean([1,2,3,4])
```

2.5

Alias import statements

A very common trick you'll see in many Python notebooks, is to import python modules under an "alias" name, which is usually a shorter name that is faster to type. The alias-import statement looks like this:

```
import <module> as <mod>
```

For example, importing the `statistics` module under the alias `stats` looks like this.

```
import statistics as stats
stats.mean([1,2,3,4])
```

2.5

The alias-import statement allows us to use the shorter names `stats.mean` and `stats.median` instead of typing out the full module name each time, like `statistics.mean` and `statistics.median`. In data science, it is very common to alias-import the `numpy` module as `np`, the `pandas` module as `pd`, and the `seaborn` module as `sns`.

Selective import statements

It is also possible to import only a specific function from a module using the following syntax.

```
from <module> import <function>
```

This allows you to use `<function>` directly in your code, without the `<module>.` prefix.

```
from statistics import mean
mean([1,2,3,4])
```

2.5

The Python standard library

The Python *standard library* refers to the Python modules that come bundled with every Python installation. Here are some of the most commonly used modules from the standard library:

- `math`: provides math functions like `sqrt`, `sin`, `cos`, etc.
- `random`: used for random number generation
- `statistics`: descriptive statistics functions
- `datetime`: manipulate dates and times
- `urllib`: tools to manipulate URLs
- `json`: read and write JSON files
- `csv`: read and write CSV files
- `sys`: access information about the current process
- `os`: interface with the operating system and file system paths
- `re`: regular expressions patterns for text processing

The easy access to all these modules that contain thousands of useful functions is a big reason why Python is so popular today. Python is often described as a language that comes with “batteries included” because of all the functionality available in the standard library.

Installing packages using pip

In addition to the standard library, the Python ecosystem includes a vast ecosystem of third-party modules, which you can install using the `pip` command-line tool. Normally, using `pip` requires some familiarity with the command-line, but when working in a JupyterLab environment, you can use the magic command `%pip` right within a notebook. To install Python package named `<pkgname>`, run the command `%pip install <pkgname>` in a notebook code cell. For example, here is the `%pip` command for installing the `sympy` package.

```
%pip install sympy
```

```
Requirement already satisfied: sympy in
/Users/ivan/Projects/Minireference/STATSbook/noBSstats/venv/
↳lib/python3.12/site-
packages (1.14.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/Users/ivan/Projects/Minireference/STATSbook/noBSstats/venv/
↳lib/python3.12/site-
packages (from sympy) (1.3.0)
Note: you may need to restart the kernel to use updated
↳packages.
```

In this case, the message “Requirement already satisfied: sympy”

tells us that the sympy package is already installed, so there is nothing to do. If the sympy package was not present on my computer already, the `%pip` command would find the sympy package from the Python Package Index website (see <https://pypi.org/project/sympy/>), download the latest version, and install it so that it becomes available for use (e.g. using `import sympy`).

The breadth and depth of the modules in the Python ecosystem is staggering. Whether you're an astrophysicist analyzing data from deep space, a neuroscientist who is trying to decode neural activity, or an office worker trying to process some obscure file format, there is a good chance that someone has come before you and written some Python code that you can reuse for your task.

A detailed discussion of the Python third-party modules ecosystem is outside the scope of this introductory tutorial, but I'd like to mention a few of the all-star, heavy-hitter libraries for scientific computing and data science.

Scientific computing libraries

Python was adopted by the scientific community very early on, because it is an easy language to learn. Scientific computing usually involves large-scale numerical calculations on vectors and matrices, which led to the development of the following packages for high-performance scientific computing:

- **Numerical Python** (`numpy`) is a library that provides array and matrix objects that make mathematical operations run very fast.
- **Scientific Python** (`scipy`) is a library provides many functions used by scientists and engineers. The module `scipy.stats` contains all the probability models we use in statistics.
- **Symbolic Python** (`sympy`) is a library for symbolic math calculations. Using SymPy you can work with math expressions just like when solving math problems using pen and paper. See the [sympy_tutorial.pdf](#) to learn more about SymPy.

Data science libraries

Another community where Python is very popular is among statisticians and data analysts, which is thanks to the following libraries for data handling and visualization:

- **Pandas** (`pandas`) is a library for tabular data management. Pandas is like a Swiss army knife of data manipulations, and provides functions for loading data from various formats, and

doing data processing, data cleaning, and calculating descriptive statistics. See the [pandas_tutorial.ipynb](#) notebook to learn more about Pandas.

- **Seaborn** (`seaborn`) is a high-level library for statistical plots. Seaborn allows you to generate beautiful data visualizations like strip plots, scatter plots, histograms, bar plots, box plots, etc. See the [seaborn_tutorial.ipynb](#) notebook to learn more about Seaborn.
- **Statsmodels** (`statsmodels`) is a statistical modelling library that includes advanced statistical models and functions for performing statistical tests.

These libraries only scratch the surface of the vast ecosystem of data science and machine learning libraries available in Python.

C.9 Getting comfortable with Python

JupyterLab provides lots of tools for making learning Python easy for beginners, including documentation (help menus) and interactive ways to explore variable properties.

Showing the help info

Every Python object has a “docstring” (documentation string) associated with it, which provides the help info about this object. There are three equivalent ways to view the docstring of any Python object `obj` (value, variable, function, module, etc.):

- `help(obj)`: shows the docstring for the object `obj`
- `obj?`: shortcut for `help(obj)`
- press **SHIFT+TAB** while the cursor is on top of a variable or function inside a code cell.

There are also other methods for getting more detailed info about an object like `obj??`, `%psource obj`, `%pdef obj`, but we won’t discuss these here.

Example 8: learning about the `print` function

Let’s say you want to learn more about the `print` function. To do so, put your mouse cursor somewhere inside the function name, then and press **SHIFT+TAB**.

```
print
```

```
<function print(*args, sep=' ', end='\n', file=None,
flush=False)>
```

We already knew that the `print` function accepts one or more arguments, but the help menu shows the complete information about other keywords arguments (options) you can use when calling this function. We can also use the `help` function to see the same information.

```
help(print)
```

Help on built-in function print in module builtins:

```
print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
        string inserted between values, default a space.
    end
        string appended after the last value, default a newline.
    file
        a file-like object; defaults to the current sys.stdout.
    flush
        whether to forcibly flush the stream.
```

The `*args` in the doc string tells this function accepts multiple arguments. The keyword argument `sep` (separator) dictates the text that is inserted between values when they are printed. The keyword `end` controls what is printed at the end of the line (`\n` means newline). Note we could have obtained the doc string using `print?` or using `print(print.__doc__)`.

Exercise 12: getting help

Display the doc-string of the function `len`.

```
len
```

```
<function len(obj, /)>
```

Click [here](#) for the solution.

Inspecting Python objects

Suppose you have an unknown Python object `obj`. How can you find out what it is and learn what you can do with it?

Displaying the object

There are several ways to display information about the object `obj`.

- `print(obj)`: converts the object into a string representation and prints it
- `type(obj)`: tells you what type of object it is
- `repr(obj)`: similar to `print`, but shows the complete text representation (including quotes). The output of `repr(obj)` usually contains all the information needed to reconstruct the object `obj`.

Auto-complete object's methods

JupyterLab code cells provide an “autocomplete” feature that allows us to interactively exploring what methods are available on a given object. To use the autocomplete functionality, start by writing the name of the object followed by a dot `obj.`, then press the **TAB** button. You'll be presented with a list of all the object's methods.

Use the following code cell to explore the methods available on Python string objects:

```
message = "Hello everyone"  
# message.      # place cursor after the dot and press TAB
```

This autocomplete feature is used often by programmers to find the names of the methods they need. Examples of useful string methods on the `message` object include `message.upper()`, `message.lower()`, `message.split()`, `message.replace(...)`, etc. You don't need to remember all these method names, since you can always write `message.` then press **TAB** to use the autocomplete feature.

Python error messages help you fix mistakes

Sometimes your code will cause an error and Python will display an error message describing the problem it encountered. You need to be mentally prepared for these errors, since they happen *a lot* and can be very discouraging to see. Examples of errors include `SyntaxError`, `NameError`, `TypeError`, etc. Error messages look scary, but really they are there to help you—if you read what the error message is telling you, you'll know exactly what you need to fix in your code.

The error message literally describes the problem!

Let's look at an example expression that causes a Python exception. Suppose you're trying to compute the difference `5-3`, but you forget the minus sign by mistake:

5 3

```
Cell In[87], line 1
    5 3
      ^
SyntaxError: invalid syntax
```

The code cell above shows an example of a `SyntaxError` that occurs because Python expects some operator or separator between the two numbers. Python doesn't know what to do when you just put two numbers side-by-side, and it doesn't want to guess what your intention was (53? 5.3?).

You'll see a threatening-looking red message like this any time Python encounters an error while trying to run the commands you specified in a code cell. We say the code "raises" or "encountered" an exception. This is nothing to be alarmed by. It usually means you made a typo or forgot a required syntax element. The way to read these red messages is to focus on the **explanation message that gets printed on the last line**. The error message usually tells you what you need to fix. The solution will be obvious when you have made a typo or a syntax error, but in more complicated situations, you might need to search for the error message online to find what the problem is. In the above example, the fix is simple: we just need to add the missing minus sign: 5-3. Edit the code cell above and re-run it to make the error message disappear.

The three most common Python error messages are:

- `SyntaxError`: you typed in something wrong (usually missing `,`, `"`, or `]` or some other punctuation).
- `NameError`: raised when a variable is not found in the current context.
- `TypeError`: raised when a function or operation is applied to an object of incorrect type.

You might run into `KeyError` when a key is not found in a dictionary, `ValueError` when a function gets an argument of correct type but improper value, `ImportError` when importing modules, and `AttributeError` when trying to access an object attribute that doesn't exist.

Exercise 13: let's break Python!

Try typing in Python commands that causes each of these exceptions:

1. `SyntaxError` (hint: write an incomplete list expression)
2. `NameError` (hint: try referring to a variable that doesn't exist)
3. `TypeError` (hint: try to sum something that is not a list)

Click [here](#) for the solution.

Python documentation is very good

The official Python website is <https://docs.python.org>. This website provides tons of excellent learning resources and reference information for learning Python. Here are some useful links to essential topics for Python beginners:

- Tutorial: <https://docs.python.org/3/tutorial/index.html>
- Built-in types: `int`, `float`, `list`, `bool`, `str`, `tuple`, `dict`.
- Functions docs: `abs`, `len`, `type`, `print`, `sum`, `min`, `max`.

I encourage you to browse the site to familiarize yourself with the high quality information that is available in the official docs pages.

When searching online, the official Python docs might now show up on the first page of results. Because of Python's popularity, there are hundreds of spammy websites that provide inferior information, wrapped in tons of advertisements and popups. These websites that are much inferior to the official documentation. When learning something new, you should always prefer the official docs, even if they don't appear first in the list of search results. [Stack overflow](#) discussions can be a good place to find answers to common Python questions. ChatGPT is also pretty good with Python code, so you can ask it to give you code examples or provide feedback on code you've written.

C.10 Final review

Let's do a quick review of all the concepts we saw in this tutorial.

Python grammar and syntax review

Learning Python is like learning a new language:

- **nouns**: variables and values of different types
- **verbs**: functions and methods, including basic operators like `+`, `-`, etc.
- **grammar**: rules about how to use nouns and verbs together

Python keywords

Here is a complete list of keywords in the Python language:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

As you can see from the above list, there are some keywords that we didn't cover in this tutorial, because they are not essential to know at first. Here is the shortlist of the keywords you need to remember:

- `def` used to define a new function and the `return` statement that defines the output of the function
- `in` to check if element is part of container
- `for` used in `for`-loops and list-comprehension syntax
- the boolean values `True` and `False`
- `or`, `and`, and `not` to create boolean expressions
- `if`, `elif`, `else` used in conditional statements
- `None` is a special keyword that represents the absence of a value
- `import ...`, `import ... as ...`, and `from ... import ...` statements to import Python modules

Python data types

Here is a reminder of the Python built-in data types:

- `int`: integers
- `float`: decimals
- `list`: list of objects [`obj1`, `obj2`, `obj3`, ...]
- `bool`: boolean values (either `True` or `False`)
- `str`: text strings
- `dict`: associative array between keys and values
- `tuple`: similar to a list, but cannot be modified

Python functions

Here are the essential functions you need to remember:

- `print(arg1, arg2, ...)`: print `str(arg1)`, `str(arg2)`, ... to the screen
- `type(obj)`: tells you what kind of object
- `len(obj)`: length of the object (only for: `str`, `list`, `dict` objs)

- `range(a,b)`: creates a list of numbers `[a,a+1,...,b-1]`
- `help(obj)`: display info about the object, function, or method

Here is another set of build-in functions used for list-like objects:

- `len(mylist)`: length of the list `mylist`
- `sum(mylist)`: sum of the values in the list of numbers `mylist`
- `all(mylist)`: True if all values in the list `mylist` are True
- `any(mylist)`: True if any of the values in the list `mylist` are True
- `enumerate(mylist)`: convert list of values to `mylist` to list of tuples `(i,mylist[i])`. Used to know the index in for-loop, e.g. `for i, item in enumerate(items): ...`
- `zip(list1,list2)`: joint iteration over two lists
- Low-level iterator functions: `iter()` and `next()`

The function for input/output (I/O) operations are:

- `input`: prompt the user for text input
- `print(arg1,arg2, ...)`: print `str(arg1), str(arg2), ...`
- `open(filepath,mode)`: open the file at `filepath` for reading or writing. Use `mode="r"` for reading or `mode="w"` for writing.

Here are some functions you can use when “looking around” to find more information about objects:

- `str(obj)`: display the string representation of the object.
- `repr(obj)`: display the Python representation of the object. Usually, you can copy-paste the output of `repr(obj)` into a Python shell to re-create the object.
- `help(obj)`: display info about the object, function, or method. This is equivalent to printing object’s docstring `obj.__doc__`.
- `dir(obj)`: show complete list of attributes and methods of the object `obj`
- `globals()`: display variables in the Python global namespace
- `locals()`: display local variables (within current scope)

Advanced functions:

- Functional stuff: `map()`, `eval()`, `exec()`
- Meta-programming: `hasattr()`, `getattr()`, `setattr()`
- Object oriented programming: `isinstance()`, `issubclass()`, `super()`

Python punctuation

The most difficult part of learning Python is the use of non-word punctuation characters, which have very specific meaning and take

some time to get used to. Let's review and summarize how the symbols `{`, `*`, `#`, `:` are used in Python.

- The hash symbol `#` is used for comments.
- Asterisk `*` is the multiplication operator, while the double asterisk `**` is the exponent operator.
- Equal sign `=` is used for:
 - assignment statements: `x = 5`
 - pass keyword arguments to functions: `print("ans", 3, sep=": ")`
 - (advanced) specify default keyword argument in function definitions
- The operators `<`, `>`, `<=`, `==` (equal to), and `!=` (not equal to) are used compare relative size of numerical values: `x > 2`.
- Parentheses `()` are used for:
 - function definitions: `def fun(x): ...`
 - calling functions: `print(3)`
 - enforcing order of operations: `(x + y) * z`
 - defining tuples: `(1, 2, 3)`
- Comma `,` is used as:
 - element separator in lists, tuples, sets, etc.
 - separator between `key:value` pairs when defining dicts
 - separate function arguments in function definitions
 - separate function arguments when calling functions
- Curly-brackets (accolades) `{}` are used to
 - define dictionaries: `mydict = {"key": "value", "k2": "v2"}`
 - define sets: `{1, 2, 3}`
- Square brackets `[]` are used for:
 - defining lists: `mylist = [1, 2, 3]`
 - accessing list elements: `mylist[2]`
 - list slicing: `mylist[0:2]` = first two items in `mylist`
 - accessing dictionary values by key: `mydict["key"]`
- Quotes `"` or `'` are used to define string literals. There are several variations you can use:
 - raw strings `r"..."` are used to reduce escape characters
 - f-strings `f"..."` are used to include variables
 - triple quotes `"""` or `'''` are used to define entire paragraphs
- Colon `:` is used for:
 - starting an indented code block in statements like `if`, `elif`, `else`, `for`, etc.
 - key: value separator in dict literals
 - slice notation: `0:2 == slice(0,2) == [0,1]`
- Period `.` is used as:

- decimal separator for floating point literals: `4.2`
 - access object attributes and methods: `msg.upper()`
 - access names within a namespace: `statistics.mean`
- Semicolon `;` can be used to put multiple commands on single line. We can also use `;` at the end of a line to suppress the output of a command in a notebook environment.

C.11 Applications

I want to tell you about some of the cool Python applications you can look forward to if you choose to develop your Python skills further. Python is not just a calculator, but a general-purpose programming language, so it can be used for many applications. We already mentioned Python uses in scientific computing and data science. Here are some other areas where Python programming is popular.

- **Command line scripts:** you can put Python commands into a script, then run them on the command line (terminal in UNIX or `cmd.exe` in Windows). For example, you can write a simple script that downloads music videos from YouTube and saves them as `.mp3` files you can listen to offline.
- **Graphical user interface (GUI) programs:** many desktop applications are written in Python. An example of a graphical, point-and-click application written in Python is [Calibre](#), which is a powerful eBook management library, eBook reader, and eBook converter that supports every imaginable eBook format.
- **Web applications:** the [Django](#) and [Flask](#) frameworks are often used to build web applications. Many of the websites you access every day have as server component written in Python.
- **Machine learning:** most of machine learning (artificial intelligence) research and development is done using Python code.
- **Glue code:** whenever you have some process that needs to take data from one place and bring it into another program, Python is a good choice to automate this process.
- **Cloud automation:** you can use Python scripts to automate the entire IT infrastructure of a company.

I mention these examples so you'll know the other possibilities enabled by Python, beyond the basic "use Python interactively like a calculator" code examples that we saw in this tutorial. We're at the end of this tutorial, but just the beginning of your journey to discover all the interesting things you can do with Python.

Links

I've compiled a list of the best Python learning resources for you.

Python cheat sheets

- <https://gto76.github.io/python-cheatsheet/>
- https://ipgp.github.io/scientific_python_cheat_sheet/
- <https://learnxinyminutes.com/docs/python/>
- <https://www.pythoncheatsheet.org/>

Tutorials

[Python tutorial by Russell A. Poldrack]

<https://statsthinking21.github.io/statsthinking21-python/>

[Programming with Python by Software Carpentry]

<https://swcarpentry.github.io/python-novice-inflammation/>

[Official Python tutorial]

<https://docs.python.org/3/tutorial/>

[Nice tutorial]

<https://www.pythonlikeyoumeanit.com/>

[Python data structures]

<https://devopedia.org/python-data-structures>

[Online tutorial]

<https://www.kaggle.com/learn/python>

Special topics

[Stats-related python functions]

<https://www.statology.org/python-guides/>

[Scientific computing]

<https://devopedia.org/python-for-scientific-computing>

Books

[*Learn Python the Right Way* by Ritza]

<https://learnpythontherightway.com/>

[*Automate the Boring Stuff with Python* by Sweigart]

<https://automatetheboringstuff.com/>

[Object-Oriented Programming in Python]

<https://python-textbok.readthedocs.io/>

Appendix D

Pandas tutorial

The tutorial is still under development. See [this link](#) for the latest version.

D.1 Pandas overview

The Pandas library provides data structures for working with tabular data. Pandas is the standard tool used for real-world data management, data cleaning, and statistical analysis in Python. You can think of Pandas as a Swiss army knife for data manipulations, since it provides a multitude of functions and methods for doing common tasks.

This tutorial will introduce you to the essential parts of the Pandas library, with a focus on the data manipulation tasks used in statistics. We'll start with Pandas essentials like loading data from source into Pandas data frames, and get to know the data frame methods.

Pandas is a Python library, so any prior experience with Python will come in handy. Remember that Appendix C contains a Python tutorial you can use to get up to speed quickly on the syntax, so if you haven't checked that out yet now would be a good time.

Installing Pandas

First, let's make sure pandas is installed using the `%pip` Jupyter command.

```
%pip install -q pandas
```


Note: you may need to restart the kernel to use updated `↵` packages.

We then import the pandas library under the alias `pd`, which is a widespread convention that makes Pandas commands short and fast to type.

```
import pandas as pd
```

All the Pandas functionality is now available behind the alias `pd`.

Pandas is a high-level toolbox for manipulating data. In particular, if you learn how to work with list-like `pd.Series` objects, and table-like `pd.DataFrame` objects, then you'll know most of what you need to know about data management.

We'll also import the NumPy module and issue a command to control the display of numbers in the rest of the notebook.

```
import numpy as np
# simple int and float __repr__
np.set_printoptions(legacy='1.25')
```

D.2 Series

Pandas `Series` objects are list-like containers of values. We work with series whenever performing calculations on individual columns (variables) of a data frame. We'll start by creating a standalone `pd.Series` object, and defer the discussion about data frames until the next subsection.

The code line below shows how to create a series from a list of four numbers. Pandas `pd.Series` objects are similar to Python lists `[3, 5, 7, 9]`. They are containers for series of values.

```
s = pd.Series([3, 5, 7, 9])
```

We stored the series object into a variable named `s`, which is short for series.

We can print the series `s` by simply typing its name in a new code cell. Recall that the notebook interface automatically prints the last expression evaluated in a code cell.

```
s
```

```
0    3
1    5
2    7
3    9
dtype: int64
```

The numbers printed on the left are called the *index* of the series, while the numbers on the right are the *values* of the series. The last line in the output shows some additional information about the series. The series *s* contains integers, so its *dtype* (data type) is *int64*.

We use integer indices to identify the elements in the series: the first element is at index 0, the second element is at index 1, and so on, until the last element which is at index `len(s) - 1`. Here is an example that shows accessing individual values of the series using the default 0-based indexing.

```
print("First:  index =", 0, " value =", s[0])
print("Second: index =", 1, " value =", s[1])
print("Last:   index =", len(s)-1, " value =", s[len(s)-1])
```

```
First:  index = 0  value = 3
Second: index = 1  value = 5
Last:   index = 3  value = 9
```

The series *index* attribute tells you all the possible indices for the series.

```
s.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

```
list(s.index)
```

```
[0, 1, 2, 3]
```

The series *s* uses the default index `[0, 1, 2, 3]`, which consists of a range of integers, starting at 0, just like the index of a Python list with four elements.

The *values* attributes of the series tells you the underlying values without the index.

```
s.values
```

```
array([3, 5, 7, 9])
```

You can access the individual elements of the series using the square brackets syntax based on the index labels for the series. The first element in the series is at index 0, so we access it as follows:

```
s[0]
```

```
3
```

We can select a range of elements from the list using the the square brackets and slice notation for the indices:

```
s[0:3]
```

```
0    3
1    5
2    7
dtype: int64
```

The slice notation 0:3 refers to the list of indices [0, 1, 2]. The result of `s[0:3]` is a new series that contains a subset of the original series that contains the first three elements.

Calculations Pandas series have methods for performing common calculations. For example, the method `.count()` tells us length of the series:

```
s.count() # == len(s)
```

```
4
```

The method `.sum()` computes the sum of the values in the series.

```
s.sum()
```

```
24
```

You can perform arithmetic operations like `+`, `-`, `*`, `/` with series. For example, we can convert the counts in the series `s` to proportions, but dividing the series `s` by the sum of the values.

```
s / s.sum()
```

```
0    0.125000
1    0.208333
2    0.291667
3    0.375000
```

```
dtype: float64
```

Series have methods for computing *descriptive statistics* like `.min()`, `.max()`, `.mean()`, `.median()`, `.var()`, `.std()`, `.quantile()`, etc. For example, the mean (average value) and the standard deviation (dispersion from the mean) are two common statistics we want to compute from data. We can calculate the arithmetic mean of the values in the series `s` by calling its `.mean()` method.

```
s.mean()
```

```
6.0
```

To find the sample standard deviation of the values in the series `s`, we use the `.std()` method.

```
s.std()
```

```
2.581988897471611
```

Pandas makes it really easy to compute all descriptive statistics!

TODO: TABLE showing all methods on Series objects

We can also use arbitrary numpy functions on series, and Pandas will apply the function to the values in the series.

```
import numpy as np
np.log(s)
```

```
0    1.098612
1    1.609438
2    1.945910
3    2.197225
dtype: float64
```

Bonus material 1 A series can contain float values.

```
s2 = pd.Series([0.3, 1.5, 2.2])
s2
```

```
0    0.3
1    1.5
2    2.2
dtype: float64
```

Here is another example of a series that contains strings (categorical variables).

```
s3 = pd.Series(["a", "b", "b", "c"])
s3
```

```
0    a
1    b
2    b
3    c
dtype: object
```

Bonus material 2 Pandas series allow arbitrary labels to be used as the index, not just integers. For example, we can use string labels like ("x", "y", etc.) as the index of a series.

```
s4 = pd.Series(index=["x", "y", "z", "t"],
               data=[ 3,   5,   7,   9 ])
s4
```

```
x    3
y    5
z    7
t    9
dtype: int64
```

```
s4.index
```

```
Index(['x', 'y', 'z', 't'], dtype='object')
```

```
s4.values
```

```
array([3, 5, 7, 9])
```

We can now use the string labels to access the individual elements in the series.

```
s4["y"]
```

```
5
```

In other words, Pandas series also act like Python dictionary objects with arbitrary keys. Indeed any quantity that can be used as a key in a dictionary (a Python hashable object), can also be used as a label in a Pandas series. The list of keys of a Python dictionary is the same as the index of a Pandas series.

D.3 Data frames

A Pandas data frame (`pd.DataFrame`) is a container for tabular data similar to a spreadsheet. You can also think of Pandas data frames as collections of Pandas series.

The most common way to create a data frame is to read data from a CSV (Comma-Separated-Values) file. Here is the raw contents of the sample data file `minimal.csv`.

```
!cat "datasets/minimal.csv"
```

```
x,y,team,level
1.0,2.0,a,3
1.5,1.0,a,2
2.0,1.5,a,1
2.5,2.0,b,3
3.0,1.5,b,3
```

The code sample below shows how to load the data file `datasets/minimal.csv` into a data frame called `df`, which is a common name we use for data frames.

```
df = pd.read_csv("datasets/minimal.csv")
df
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

The function `pd.read_csv()` reads the contents of the data file `datasets/minimal.csv` and automatically determines the names of the columns based on the first line in the CSV file, which is called the *header* row.

We'll use the data frame `df` for many of the examples in the remainder of this tutorial. The data corresponds to five players in a computer game. The columns `x` and `y` describe the position of the player, the variable `team` indicates which team the player is part of, and the column `level` specifies the character's strength. The meaning of the variables will not be important, since we'll be focussing on the technical aspects of the data manipulation procedures.

Pandas provides methods for loading data from many data file formats. The function `pd.read_excel()` can be used to load data

from spreadsheet files. There are also functions `pd.read_html()`, `pd.read_json()`, and `pd.read_xml()` for reading data from other file formats. We'll talk more about various data formats later in this section.

Data frame properties

Let's explore the attributes and methods of the data frame `df`. First let's use the Python function `type` to confirm that `df` is indeed a data frame object.

```
type(df)
```

```
pandas.core.frame.DataFrame
```

The above message tells us that the `df` object is an instance of the `DataFrame` class defined in the Python module `pandas.core.frame`.

Every data frame has the attributes `index` and `columns`, as illustrated in the following figure:

The diagram shows a DataFrame with 5 rows and 4 columns. The columns are labeled 'x', 'y', 'team', and 'level'. The rows are indexed 0 through 4. Annotations include: a red box around the column headers labeled 'df.columns', a green box around the row indices labeled 'df.index', a blue box around the 'y' column labeled 'df["y"]', and a purple box around the row with index 2 labeled 'df.loc[2]'.

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

The `index` is used to refer to the rows of the data frame.

```
df.index
```

```
RangeIndex(start=0, stop=5, step=1)
```

The data frame `df` uses the “default” range index that consists of a sequence integer labels: `[0,1,2,3,4]`, similar to the 0-based indexing used to access the elements of Python lists.

The `columns-index` attribute `.columns` tells us the names of the columns (variables) in the data frame.

```
df.columns
```

```
Index(['x', 'y', 'team', 'level'], dtype='object')
```

Column names usually consist of short textual identifiers for the variable (Python strings). Note that spaces and special characters can appear in column names. Column names like "x position" and "level (1 to 3)" are allowed, but generally discouraged since complicated column names make data manipulation code more difficult to read.

Another important property of the data frame is its shape.

```
df.shape
```

```
(5, 4)
```

The shape of the data frame `df` is 5×4 , which means it has five rows and four columns.

The `.dtypes` (data types) attribute tells us what type of data is stored in each of the columns.

```
df.dtypes
```

```
x          float64
y          float64
team       object
level      int64
dtype: object
```

We see that the columns `x` and `y` contain floating point numbers, the column `team` can contain arbitrary Python objects (in this case Python strings), and the column `level` contains integers.

The method `.info()` provides additional facts about the data frame object `df`, including information about missing values (null values) and the total memory usage.

```
df.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
#   Column  Non-Null Count  Dtype
---  -
0   x       5 non-null        float64
1   y       5 non-null        float64
2   team    5 non-null        object
3   level   5 non-null        int64
```



```
dtypes: float64(2), int64(1), object(1)
memory usage: 502.0 bytes
```

The data frame `df` takes up 538 bytes of memory, which is not a lot. You don't have to worry about memory usage for any of the datasets we'll analyze in this book, since they are all small- and medium-sized. You might have to think about memory usage if you work on larger data sets like logs or databases.

When exploring a data frame, we often want to print the first few rows of the data frame to see what they look like. The data frame method `.head(k)` prints the first `k` rows.

```
df.head(2)
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2

You can also call `df.tail(k)` to print the last `k` rows of the data frame, or call `df.sample(k)` to select a random sample of size `k` from the data frame.

Accessing and selecting values

The `.loc[]` attribute is used to access individual values, rows, or columns from the data frame.

Accessing individual values The syntax `df.loc[row,col]` is used to select the value that corresponds to the row label `row` and column label `col` in the data frame `df`. For example, to extract the value of the variable `y` for the third row, we use:

```
df.loc[2, "y"]
```

```
1.5
```

Selecting entire rows To select rows from a data frame, we use `df.loc[row,:]` where the `:` syntax is shorthand for “all columns.”

```
row2 = df.loc[2,:]
row2
```

x	2.0
y	1.5
team	a

```
level      1
Name: 2, dtype: object
```

The rows of a data frame are series objects and their index is the same as the columns index of the data frame.

```
type(row2), row2.index
```

```
(pandas.core.series.Series, Index(['x', 'y', 'team', 'level'], dtype='object'))
```

```
row2.values
```

```
array([2.0, 1.5, 'a', 1], dtype=object)
```

```
row2["y"]
```

```
1.5
```

Selecting entire columns We use the syntax `df[col]` to select the column `col` from a data frame.

```
ys = df["y"]
ys
```

```
0    2.0
1    1.0
2    1.5
3    2.0
4    1.5
Name: y, dtype: float64
```

The column `ys` is a series and its index is the same as `df.index`.

```
type(ys), ys.index
```

```
(pandas.core.series.Series, RangeIndex(start=0, stop=5, step=1))
```

The column-selector syntax `df[col]` is shorthand for the expression `df.loc[:,col]`, which means “select all the rows for the column `col`.” We can verify that `df["y"]` equals `df.loc[:, "y"]` using the `.equals()` method.

```
df["y"].equals( df.loc[:, "y"] )
```

True

Selecting multiple columns We can extract multiple columns from a data frame by passing a list of column names inside the square brackets.

```
df[["x", "y"]]
```

```
   x    y
0  1.0  2.0
1  1.5  1.0
2  2.0  1.5
3  2.5  2.0
4  3.0  1.5
```

The result is a new data frame object that contains only the x and y columns from the original df.

Selecting only certain rows A common task when working with Pandas data frames is to select the rows that fit one or more criteria. We usually carry out this selection procedure using in two-step process:

- Build a “selection mask” series that consists of boolean values (True or False). The mask series contains the value True for the rows we want to keep, and the value False for the rows we want to filter out.
- Select the subset of rows from the data frame using the mask. The result is a new data frame that contains only the rows that correspond to the True values in the selection mask.

For example, to select the rows from the data frame that are part of team b, we first build the selection mask.

```
mask = df["team"] == "b"
mask
```

```
0    False
1    False
2    False
3     True
4     True
Name: team, dtype: bool
```

The rows that match the criterion “team column equal to b” correspond to the True values in the mask, while the remaining values

are False.

The actual selection is done by using the mask inside the square brackets.

```
df[mask]
```

	x	y	team	level
3	2.5	2.0	b	3
4	3.0	1.5	b	3

The result is a new data frame that contains only the rows that correspond to the True values in the mask series.

We often combine the two steps we described above into a single expression `df[df["team"]=="b"]`. This combined expression is a little hard to read at first, since it contains two pairs of square brackets and two occurrences of the data frame name `df`, but you'll quickly get used to it, because you'll see this type of selection expressions very often.

```
df[df["team"]=="b"]
```

	x	y	team	level
3	2.5	2.0	b	3
4	3.0	1.5	b	3

We can use the Python bitwise boolean operators `&` (AND), `|` (OR) and `~` (NOT) to build selection masks with multiple criteria. For example, to select the rows with `team` is `b` where the `x` value is greater or equal to 3, we would use the following expression.

```
df[(df["team"] == "b") & (df["x"] >= 3)]
```

	x	y	team	level
4	3.0	1.5	b	3

The selection mask consists of two terms (`df["team"]=="b"`) and (`df["x"]>=3`) that are combined with the bitwise AND operator `&`. Note the use of extra parentheses to ensure the masks for the two conditions are computed first before the `&` operation is applied.

If we want to select multiple values of a variable, we can use the `.isin()` method and specify a list of values to compare with. For example, to build a mask that select all the observations that have `level` equal to 2 or 3, we can use the following code.

```
df["level"].isin([2,3])
```

```
0      True
1      True
2     False
3      True
4      True
Name: level, dtype: bool
```

We see the above expression has correctly selected all observations except the one at index 2, which has `level` equal to 1.

Creating data frames from lists

We sometimes obtain data in the form of regular Python objects like lists and dictionaries. If we want to use Pandas functions to manipulate this data, we'll need to put that data into a data frame object.

One way to create a data frame object is to load Python dictionary whose keys are the column names, and its values are lists of the data in each column. The code below shows how to create a data frame `df2` by initializing a `pd.DataFrame` from a columns dictionary.

```
dict_of_columns = {
    "x": [1.0, 1.5, 2.0, 2.5, 3.0],
    "y": [2.0, 1.0, 1.5, 2.0, 1.5],
    "team": ["a", "a", "a", "b", "b"],
    "level": [3, 2, 1, 3, 3],
}

df2 = pd.DataFrame(dict_of_columns)
df2
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

The data frame `df2` that we created above is identical to the data frame `df` that we loaded from the CSV file earlier. We can confirm this by calling the `.equals()` method.

```
df2.equals(df)
```

```
True
```

Indeed, you can think of data frame objects as dictionary-like containers whose keys are the column names, and whose values are

Pandas series objects (columns of values).

We can also create a data frame from a list of observation records. Each record (row) corresponds to the data of one observation.

```
list_records = [
    [1.0, 2.0, "a", 3],
    [1.5, 1.0, "a", 2],
    [2.0, 1.5, "a", 1],
    [2.5, 2.0, "b", 3],
    [3.0, 1.5, "b", 3],
]
columns = ["x", "y", "team", "level"]

df3 = pd.DataFrame(list_records, columns=columns)
df3
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

When using the list-of-records approach, Pandas can't determine the names of the columns automatically, so we must pass in the `columns` argument with a list of the column names we want for the data frame. The data frame `df3` created from the lists of records is identical to the data frame `df` that we loaded from the CSV file.

```
df3.equals(df)
```

True

Another way to create a data frame is to pass in a list of dicts:

```
dict_records = [
    dict(x=1.0, y=2.0, team="a", level=3),
    dict(x=1.5, y=1.0, team="a", level=2),
    dict(x=2.0, y=1.5, team="a", level=1),
    dict(x=2.5, y=2.0, team="b", level=3),
    dict(x=3.0, y=1.5, team="b", level=3),
]
df4 = pd.DataFrame(dict_records)
df4
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

Once again, the data frame `df4` we obtain is identical to `df` we loaded from `minimal.csv`.

```
df4.equals(df)
```

True

In this section, we illustrated the most common data frame manipulation techniques you might need. Try solving the following exercises before continuing with the rest of the tutorial.

Exercise 1: select team a

Select the rows from the data frame `df` that correspond to the players on team a.

```
# Instructions: write your Pandas code in this cell
```

Click [here](#) for the solution.

D.4 Sorting, grouping and aggregation

Sorting

TODO

Group by and aggregation

A common calculation we need to perform in statistics is to compare different *groups* of observations, where the grouping is determined by one of the variables. The Pandas method `.groupby()` is used for this purpose. For example, we can group the observations in the data frame `df` by the value of the `team` variable using the following code.

```
df.groupby("team")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x10a7969f0>
```

The result of calling the `.groupby()` method is a `DataFrameGroupBy` object that contains the subsets of the data frame that correspond to the different values of the `team` variable, `df[df["team"]=="a"]` and `df[df["team"]=="b"]`.

We can use the `DataFrameGroupBy` object to do further selection of variables and perform computations. For example, to compute the mean value of the `x` variable within the two groups, we run the following code.

```
df.groupby("team")["x"].mean()
```

```
team
a    1.50
b    2.75
Name: x, dtype: float64
```

The result is a series containing the mean for the two groups. The first row contains the value `df[df["team"]=="a"]["x"].mean()`. The second row's value is `df[df["team"]=="b"]["x"].mean()`. Note the result is a series whose index is the `team` variable (the labels `a` and `b`).

We often want to compute several quantities for each group. We can use the `.aggregate()` method after the `groupby()` method to compute multiple quantities. For example, to compute the sum, the count, and the mean value in each group we use the following code.

```
df.groupby("team")["x"].aggregate(["sum", "count", "mean"])
```

```
      sum  count  mean
team
a      4.5      3  1.50
b      5.5      2  2.75
```

The above code sample is an example of the “method chaining” pattern, which is used often in Pandas calculations. We start with the data frame `df`, call its `.groupby()` method, select the `x` column using the square brackets `["x"]`, then call the method `.aggregate()` on the result.

We can “chain” together any number of Pandas methods to perform complicated data selection and aggregation operations. The above examples show chaining just two methods, but it is common to chain together three or more methods as well. This ability to carry out advanced data manipulations using a sequence of simple method applications is one of the main benefits of using Pandas for data pro-

cessing. Method chaining operations work because Pandas Series, DataFrames, and GroupBy objects all offer the same methods, so the output of one calculation can be fed into the next.

When using method chaining for data manipulations, the command chains tend to become very long and often don't fit on a single line of input. We'll therefore split Pandas expressions on multiple lines using the Python line-continuation character `\`, as shown in the code example below.

```
df.groupby("team")["x"] \
    .aggregate(["sum", "count", "mean"])
```

	sum	count	mean
team			
a	4.5	3	1.50
b	5.5	2	2.75

The result of the above code cell is identical to the result of code cell that precedes it, however writing the code on two lines using the line-continuation character `\` makes the operations easier to read. It is customary (but not required) to indent the second line by a few spaces so the dots line up. The indentation gives a visual appearance of a “bullet list” of operation we apply to a data frame.

Another way to get the benefits of multi-line commands is to wrap the entire expression in parentheses.

```
(df
    .groupby("team")["x"]
    .agg(["sum", "count", "mean"])
)
```

	sum	count	mean
team			
a	4.5	3	1.50
b	5.5	2	2.75

The result is identical to the result in previous two code cells. This works because we're allowed to wrap any Python expression in parentheses without changing its value, so wrapping the command in parentheses doesn't do anything. The benefit of the parentheses is that new lines are ignored for expressions inside parentheses, so we're allowed to break the expression onto multiple lines without the need to add the character `\` at the end of each line.

Don't worry too much about the line-continuation and parentheses tricks for multi-line expressions. Most of the Pandas expressions

you'll see in this tutorial will fit on a single line, but I wanted you to know about multi-line expressions syntax in case you see it in certain places.

Using crosstab

TODO

Using pivot and pivot_table

TODO

D.5 Data transformations

So far we talked about selecting subsets of a data frame, but what if we want to modify the data frame? Pandas provides dozens of methods for modifying the shape, the index, the columns, and the data types of data frames. All the methods we'll show in this section return a new data frame, so technically speaking we're not modifying the data frames but returning new, transformed versions of the data frame, which we usually save under a new variable name.

Below is a list of common data transformations you're likely to encounter in getting "raw" source data into the shape needed for statistical analysis.

- **Renaming:** change the column names of a data frame.
- **Reshaping and restructuring** the way the data is organized.
- **Imputation:** filling in missing values based on the surrounding data or a fixed constant.
- **Merging** data from multiple sources to form a combined dataset. For example, we can merge a dataset about downloads per country with another dataset about countries' populations, so we can compute downloads-per-capita statistics.
- **Filtering:** selecting only a subset of the data we're interested in.
- **Splitting** columns.
- **Data cleaning:** various procedures for identifying and correcting bad data. Data cleaning procedures include dealing with missing values like `None`, `NaN` (not a number), and `<NA>` (not available), detecting coding errors, duplicate observations, or other inconsistencies in the data.

- **Outlier detection and removal:** used to reject certain observations that fall outside the range of expected values.

We'll now show some examples of common data transformations, which are worth knowing about. We'll talk about data cleaning and outlier detection in the next section.

Transpose

The *transpose* transformation flips a data frame through the diagonal, turning the rows into columns, and columns into rows.

```
dfT = df.transpose()
dfT
```

	0	1	2	3	4
x	1.0	1.5	2.0	2.5	3.0
y	2.0	1.0	1.5	2.0	1.5
team	a	a	a	b	b
level	3	2	1	3	3

After the transpose operation, the index `df.index` becomes the column index `dfT.columns`, while the columns index `df.columns` becomes the rows index `dfT.index`.

Other data shape transformation methods include `.melt()`, `.stack()`, `.unstack()`, `.merge()`, etc. We'll discuss shape-transformation methods later in the tutorial as needed.

Adding new columns

The most common modification to a data frame is to add a new column. We can add a new column to the data frame `df` by assigning data to a new column name as shown below.

```
df["xy"] = df["x"] * df["y"]
df
```

	x	y	team	level	xy
0	1.0	2.0	a	3	2.0
1	1.5	1.0	a	2	1.5
2	2.0	1.5	a	1	3.0
3	2.5	2.0	b	3	5.0
4	3.0	1.5	b	3	4.5

On the right side of the assignment operator `=` we compute the product of the `x` and `y` columns, which is a series. We assign this series to

a new column called `xy`. The result of this assignment is a modified data frame `df` with an additional column.

To remove a column, we use the `.drop` method.

```
df = df.drop(columns=["xy"])
df
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

In the code cells above, we were modifying the data frame `df` without changing its name. This is considered bad coding practice because the same name refers to different objects. In earlier code cells, the data frame `df` had columns `["x", "y", "team", "level"]`, then we added the `xy` column so the data frame `df` has columns `["x", "y", "team", "level", "xy"]`, and after that we dropped the column `xy` again. This type of “in place” modifications can lead to confusion, since they force us to keep track of the history of operations we performed on `df`.

To avoid the need to keep track of the history of operations, it's best to use a different each time you modify a data frame. For example, you can create a copy of `df` under the new name `dfxy` and perform the modifications on the copy as shown below.

```
dfxy = df.copy()
dfxy["xy"] = df["x"] * df["y"]
```

The resulting `dfxy` is the modified `df` with the extra `xy` column, while the original data frame remains unchanged. Using the new name `dfxy` for the modified data frame makes it clear it is not the same as the original `df`.

The `.assign()` method is a convenient way to add new columns to a data frame, which automatically makes the copy for us.

```
dfxy = df.assign(xy = df["x"] * df["y"])
```

The `.assign()` method returns a new data frame without modifying the original `df`. We save the result to a new variable `dfxy`, so the end result the same as the data frame `dfxy` we obtained above using the copy-then-modify approach.

The `.assign()` method is very useful for performing multiple transformations to a data frame using method chaining. Here is an example that shows how we can perform four different transformation to the data frame `df` in a single statement.

```
import numpy as np
df5 = df.assign(xy = df["x"] * df["y"]) \
        .assign(z = 1) \
        .assign(r = np.sqrt(df["x"]**2 + df["y"]**2)) \
        .assign(team = df["team"].str.upper())
df5
```

	x	y	team	level	xy	z	r
0	1.0	2.0	A	3	2.0	1	2.236068
1	1.5	1.0	A	2	1.5	1	1.802776
2	2.0	1.5	A	1	3.0	1	2.500000
3	2.5	2.0	B	3	5.0	1	3.201562
4	3.0	1.5	B	3	4.5	1	3.354102

The method chain contains four `.assign()` operations. The effect of the first operation is to add the `xy` column containing the product of `x` and `y` values. The second operation adds a new constant column `z` equal to 1. The third operation adds the column `r` obtained from the math formula $r = \sqrt{x^2 + y^2}$, which corresponds to the distance from the origin of a point with coordinates (x, y) . Note we used the function `np.sqrt` from the NumPy module to perform the square root operation. The last transformation changes the values of the `team` column to be in uppercase letters. The result of these four `assign` statements is stored as the new variable `df5`, which we then display.

Dropping rows and and columns

Pandas provides several “drop” methods for removing unwanted rows or columns from a data frame. For example, to drop the first (index 0), third (index 2), and fifth (index 4) rows of the data frame `df` we can use the `.drop()` method and pass in the list of indices to remove to the `index` argument.

```
df.drop(index=[0,2,4])
```

	x	y	team	level
1	1.5	1.0	a	2
3	2.5	2.0	b	3

The result is a new data frame that contains only the second (index 1) and fourth (index 3) rows.

To remove columns from a data frame, use `.drop()` method with the `columns` option. Here is the code to delete the column `level`.

```
df.drop(columns=["level"])
```

	x	y	team
0	1.0	2.0	a
1	1.5	1.0	a
2	2.0	1.5	a
3	2.5	2.0	b
4	3.0	1.5	b

The result is a new data frame that no longer has the `level` column. Note another way to obtain the same result is by selecting the three columns that we want to keep using the code `df[["x", "y", "team"]]`.

Pandas also provides the methods `.dropna()` for removing rows with missing values and `.drop_duplicates()` for removing rows that contain duplicate data. We'll learn more about these methods in the section on data cleaning below.

Renaming columns and values

To change the column names of a data frame, we can use the `.rename()` method and pass in to the `columns` argument a Python dictionary of the replacements we want to make. For example, the code below renames the columns `team` and `level` to use uppercase letters.

```
df.rename(columns={"team":"TEAM", "level":"LEVEL"})
```

	x	y	TEAM	LEVEL
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

The dictionary `{"team":"TEAM", "level":"LEVEL"}` contains the old:new replacement pairs.

To rename the values in the data frame, we can use the `.replace()` method, passing in a Python dictionary of replacements we want to do on the values in each column. For example, here is the code for replacing the values in the `team` column to uppercase letters.

```
team_mapping = {"a": "A", "b": "B"}
df.replace({"team": team_mapping})
```

	x	y	team	level
0	1.0	2.0	A	3
1	1.5	1.0	A	2
2	2.0	1.5	A	1
3	2.5	2.0	B	3
4	3.0	1.5	B	3

The dictionary `{"a": "A", "b": "B"}` contains the old:new replacement pairs, similar to the dictionary we used above for renaming columns.

```
# # ALT. use str-methods to get uppercase letter
# df.assign(team = df["team"].str.upper())
```

Reshaping data frames

One of the most common transformations we need to do with data frames, is to convert them from “wide” format to “long” format. Data tables in “wide” format contain multiple observations in each row, with the column header indicating some property for the observations in each column. The code example below shows a sample data frame with the viewership numbers for a television series. The viewership numbers are organized by season (rows) and by episode (columns).

```
views_data = {
    "season": ["Season 1", "Season 2"],
    "Episode 1": [1000, 10000],
    "Episode 2": [2000, 20000],
    "Episode 3": [3000, 30000],
}
tvwide = pd.DataFrame(views_data)
tvwide
```

	season	Episode 1	Episode 2	Episode 3
0	Season 1	1000	2000	3000
1	Season 2	10000	20000	30000

This organization is very common for data collected in spreadsheets, since it's easy for humans to interpret values based on the columns they appear in.

Data structured in “wide” format is useful for data entry and display purposes, but it makes data selection, grouping, and filtering oper-

ations more complicated to perform. Wide data is not a good shape for statistical analysis. Instead, we prefer all data to be in “long” format where each row corresponds to a single observation, and each column corresponds to a single variable, like the minimal dataset we discussed above.

The Pandas operation for converting “wide” data to “long” data is called `melt`, in reference to melting a wide block of ice into a vertical stream of water. The method `.melt()` requires several arguments to specify how to treat each of the columns in the input data frame, and the names we want to assign to the columns in the output data frame. Let’s look at the code first, and explain the meaning of the arguments after.

```
tvlong = tvwide.melt(id_vars=["season"],
                     var_name="episode",
                     value_name="views")
tvlong
```

	season	episode	views
0	Season 1	Episode 1	1000
1	Season 2	Episode 1	10000
2	Season 1	Episode 2	2000
3	Season 2	Episode 2	20000
4	Season 1	Episode 3	3000
5	Season 2	Episode 3	30000

The argument `id_vars` specifies a list of identifier variables, which are the columns that already contains values for the whole row. All other columns are treated as value variables that apply to the values in that column. The arguments `var_name` and `value_name` determine the name of the variable and value columns of the melted data frame. The result `tvlong` has six rows, one for each observation in the original data frame `tvwide`. Each row corresponds to one episode of the TV show, and each column corresponds to a different variable for that episode (the season, the episode, and the number of views). This means the data frame `tvlong` is in *tidy data* format.

BONUS EXPLAINER 1: SORT AFTER MELT The rows in the data frame `tvlong` appear out of order after the `melt` operation. We can fix this using the `.sort_values()` method and specifying the column names by which we want the data to be sorted.

```
tvlong.sort_values(by=["season", "episode"]) \
    .reset_index(drop=True)
```


	season	episode	views
0	Season 1	Episode 1	1000
1	Season 1	Episode 2	2000
2	Season 1	Episode 3	3000
3	Season 2	Episode 1	10000
4	Season 2	Episode 2	20000
5	Season 2	Episode 3	30000

The data is now sorted by season first then by episode, which is the natural order for this data. The extra method `.reset_index()` is used to re-index the rows using the sequential numbering according to the new sort order.

BONUS EXPLAINER 2: UNDO MELT WITH PIVOT The method for the opposite transformation (converting long data to wide data) is called `pivot` and works like this:

```
tvlong.pivot(index="season",
              columns="episode",
              values="views")
```

episode	Episode 1	Episode 2	Episode 3
season			
Season 1	1000	2000	3000
Season 2	10000	20000	30000

```
# # ALT. to get *exactly* the same data frame as `twide`
# tvlong.pivot(index="season",
#               columns="episode",
#               values="views") \
#   .reset_index() \
#   .rename_axis(columns=None)
```

Tidy data

The concept of *tidy data* is a convention for structuring data sets that makes them easy to work with. A tidy dataset has the following characteristics:

- Each variable corresponds to a separate column.
- Each observation corresponds to a separate row.
- Each data cell contains a single value.

This structure makes it easy to select arbitrary subsets of the data based on values of the variables, and perform arbitrary transformations and aggregations. Tidy data is also very useful for data visualizations.

TODO: examples of non-tidy data + explain what is the problem + how to fix it

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” —Hadley Wickham

The Seaborn data visualization library works well with data in tidy format, allowing you to plot the values from any column, and using other columns as grouping variables. We’ll make sure to convert all datasets into tidy format to benefit from Seaborn visualizations.

String methods

TODO .str namespace

The .str prefix can be used to access other string manipulation methods like .str.join(), .str.split(), .str.startswith(), .str.strip(), etc. Any operation we can perform on a Python string, we can also perform the same operation on an entire Pandas series by calling the appropriate str-method.

Another transformation we might have to do is to split values that combine multiple variables. For example, medical records often contain the age and sex information as a single string 32F, which we would split into the variable age with value 32 and the variable sex with value F.

Merging data frames

TODO

Type conversions

TODO .astype

Dummy coding of categorical variables

TODO

I hope you got a general idea of the transformations that exist.

D.6 Data cleaning

Before a dataset can be used for statistical analysis, we must often perform various pre-processing steps on the data to ensure it is consistently formatted. The term *data cleaning* describes various procedures for correcting various data problems so that the statistical anal-

ysis procedures will not “choke” on it, or lead to erroneous results. Data cleaning procedures include detecting coding errors, removing duplicate observations, and fixing other inconsistencies in the data.

Standardize categorical values A very common problem that occurs for categorical variables, is the use of multiple codes to represent the same concept. Consider the following series containing what-type-of-pet-is-it data in which “dog” is encoded using different values.

```
pets = pd.Series(["D", "dog", "Dog", "doggo"])
pets
```

```
0      D
1     dog
2     Dog
3  doggo
dtype: object
```

This type of inconsistent encoding will cause lots of trouble down the line. For example, performing a `.groupby()` operation on this variable will result in four different groups, even though all these pets are dogs.

We can fix this encoding problem by standardizing on a single code for representing dogs and replacing all other values with the standardized code, as shown below.

```
dogsubs = {"D": "dog", "Dog": "dog", "doggo": "dog"}
pets.replace(dogsubs)
```

```
0     dog
1     dog
2     dog
3     dog
dtype: object
```

The method `.value_counts()` is helpful for detecting coding errors and inconsistencies. Looking at the counts of how many times each value occurs can help us notice exceptional values, near-duplicates, and other problems with categorical variables.

```
pets.replace(dogsubs).value_counts()
```

```
dog      4
Name: count, dtype: int64
```

Exercise 2: cleaning the cats data

Write the Pandas code required to standardize all the cat labels to be cat.

```
# Instructions: clean the cat labels in the following series
pets2 = pd.Series(["cat", "dog", "Cat", "CAT"])
pets2
```

```
0    cat
1    dog
2    Cat
3    CAT
dtype: object
```

Click [here](#) for the solution.

Number formatting errors The text string "1.2" corresponds to the number 1.2 (a float). We can do the conversion from text string to floating point number as follows.

```
float("1.2")
```

```
1.2
```

When loading data, Pandas will automatically recognize numerical expression like this and load them into columns of type float or int.

There are some common number formatting problems you need to watch out for. Many languages use the comma as the decimal separator instead of a decimal point, so the number 1.2 might be written as the text string "1,2", which cannot be recognized as a float.

```
float("1,2")
```

```
ValueError                                Traceback (most recent call last)
Cell In[80], line 1
----> 1 float( )

ValueError: could not convert string to float: '1,2'
```

To fix this issue, we replace the comma character in the string with a period character as follows.

```
"1,2".replace(",", ".")
```

```
'1.2'
```

Another example of a problematic numeric value is "1. 2", which is not a valid float because of the extra spaces. We can fix this by getting rid of the space using "1. 2".replace(" ", "").

Let's now learn how to perform this kind of string manipulations when working with Pandas series and data frames.

```
rawns = pd.Series(["1.2", "1,2", "1. 2"])
rawns
```

```
0    1.2
1    1,2
2    1. 2
dtype: object
```

The series `rawns` contains strings with correct and incorrect formatting. Note the series `rawns` has `dtype` (data type) of `object`, which is what Pandas uses for strings.

Let's try to convert this series to a numeric values (`floats`). We can do this by calling the method `.astype()` as shown below.

```
# uncomment to see the ERROR
# rawns.astype(float)
```

Calling the method `.astype(float)` is essentially the same as calling `float` on each of the values in the series, so it shouldn't be surprising that we see an exception since the string "1,2" is not a valid float.

We can perform the string replacements on the data series `rawns` using the "str-methods" as shown below.

```
rawns.str.replace(",", ".") \
    .str.replace(" ", "") \
    .astype(float)
```

```
0    1.2
1    1.2
2    1.2
dtype: float64
```

After performing the replacements of commas to periods and removing unwanted spaces, the method `.astype(float)` succeeds.

Note the method we used in the above example is `.str.replace()` and not `.replace()`.

Other types of data that might have format inconsistencies include dates, times, addresses, and postal codes. We need to watch out when processing these types of data, and make sure all the data is in a consistent format before starting the statistical analysis.

Use the `.drop_duplicates()` method to remove duplicated rows.

Dealing with missing values

Real-world datasets often contain missing values that can occur as part of the data collection process. Missing values in Pandas are indicated with the special symbol `NaN` (Not a Number), or sometimes using the symbol `<NA>` (Not Available). In Python, the absence of a value corresponds to the value `None`. The term *null value* is a synonym for missing value.

In order to show some examples of dealing with missing values, let's load the "raw" dataset located at `datasets/raw/minimal.csv`, which has several "holes" in it.

```
rawdf = pd.read_csv("datasets/raw/minimal.csv")
rawdf
```

	x	y	team	level
0	1.0	2.0	a	3.0
1	1.5	1.0	a	2.0
2	2.0	1.5	a	1.0
3	1.5	1.5	a	NaN
4	2.5	2.0	b	3.0
5	3.0	1.5	b	3.0
6	11.0	NaN	NaN	2.0

The data frame `rawdf` contains the same data we worked on previously, and some additional rows with "problematic" values that we need to deal with. Specifically, the rows with index 3 is missing the `level` variable, and row 6 is missing the values for the `y` and `team` variables. The missing values are indicated by the `NaN` symbol, which is a special float value that stands for Not a Number.

Note the variable `level` has been converted to a `float` data type in order to use the `NaN` for representing missing values. This is because integers do not have a natural way to represent missing values. Some datasets might use special codes like 999 or -1 to indicate missing integer values. You should use the `.replace()` method to change these values to `float("NaN")` or `pd.NA` in order to avoid the special code being used in numeric calculations by mistake.

We can use the `.info()` method to get an idea of the number of missing values in a data frame.

```
rawdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 4 columns):
#   Column  Non-Null Count  Dtype
---  -
0    x         7 non-null    float64
1    y         6 non-null    float64
2   team         6 non-null    object
3   level        6 non-null    float64
dtypes: float64(3), object(1)
memory usage: 356.0+ bytes
```

The data frame has a total of 7 rows, but the columns `level`, `y`, and `team` have only 6 non-null values, which tells us these columns contain one missing value each.

We can use the method `.isna()` to get a complete picture of all the values that are missing (not available).

```
rawdf.isna()
```

	x	y	team	level
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	True
4	False	False	False	False
5	False	False	False	False
6	False	True	True	False

The locations in the above data frame that contain the value `True` correspond to the missing values in the data frame `rawdf`.

We can summarize the information about missing values, by computing the row-sum of the above data frame, which tells us count of the missing values for each variable.

```
rawdf.isna().sum(axis="rows")
```

x	0
y	1
team	1
level	1

```
dtype: int64
```

We can also compute the column-sum to count the number of missing values in each row.

```
rawdf.isna().sum(axis="columns")
```

```
0    0
1    0
2    0
3    1
4    0
5    0
6    2
```

```
dtype: int64
```

```
# # ALT. convert to data types that have support <NA> values
# tmpdf = pd.read_csv("datasets/raw/minimal.csv")
# rawdf2 = tmpdf.convert_dtypes()
# rawdf2.dtypes
```

```
float("NaN")
```

```
nan
```

We can also compute `rawdf.isna().sum(axis="columns")` to get the count of missing values in each row.

The most common way of dealing with missing values is to exclude them from the dataset, by dropping all rows that contain missing values. The method `.dropna()` filters out any rows that contain null values and return a new “clean” data frame with no NaNs.

```
cleandf = rawdf.dropna()
cleandf
```

	x	y	team	level
0	1.0	2.0	a	3.0
1	1.5	1.0	a	2.0
2	2.0	1.5	a	1.0
4	2.5	2.0	b	3.0
5	3.0	1.5	b	3.0

We can provide arguments to the `.dropna()` method to give more specific instructions about which rows to drop. For example, missing values in one of the columns you don’t plan to use for a statistical analysis are not a problem.

Another approach for dealing with missing values is to use *imputation*, which is the process of “filling in” values based on our “best guess” of the missing values. Imputation is a tricky process since it involves changing the data, which can affect any subsequent statistical analysis, so we try to avoid it as much as possible. The common approaches for filling in missing values include using the mean or median of the variable, or guessing a missing value based on the neighbouring values.

```
# Recode level as integer
cleandf.loc[:, "level"] = cleandf["level"].astype(int)
cleandf
```

	x	y	team	level
0	1.0	2.0	a	3.0
1	1.5	1.0	a	2.0
2	2.0	1.5	a	1.0
4	2.5	2.0	b	3.0
5	3.0	1.5	b	3.0

```
# ALT. Convert the level column to `int`
# cleandf = cleandf.assign(level = cleandf["level"].
→astype(int))
```

```
# # Confirm `cleandf` is the same as `df` we saw earlier
# cleandf.reset_index(drop=True).equals(df)
```

Identifying and removing outliers

Outliers are data values that are much larger or much smaller than other values in the data. Sometimes extreme observations can result from mistakes in the data collection process like measurement instrument malfunctions, data entry typos, or measurements of the wrong subject. For example, if we intend to study the weights of different dog breeds but somehow end up including a grizzly bear in the measurements. These mistakes are usually identified when a measurement is obviously impossible. For example, if your dataset contains the value 600kg for a dog weight, then you know this is probably a mistake.

Sometimes we can “correct” outlier values by replacing them with the correct measurement. For example, if the data was manually transcribed from a paper notebook, we could find the notebook and see what the correct value is supposed to be. Other times, we can’t fix the mistake so we drop that observation.

In any case, it's important for you to know if your data contains outliers before you do any statistical analysis on this data, otherwise the presence of outliers might "break" the statistical inference machinery, and lead you to biased or erroneous results.

Let's look at some example data that contains an outlier and some procedures for identifying the outlier and filtering it out. Consider the data sample $x = [1, 2, 3, 4, 5, 6, 50]$, which consists of seven values, one of which is much much larger than the others.

```
xs = pd.Series([1, 2, 3, 4, 5, 6, 50], name="x")
xs
```

```
0    1
1    2
2    3
3    4
4    5
5    6
6   50
Name: x, dtype: int64
```

We can easily see the value 50 is the outlier here, but the task may not be so easy for larger datasets with hundreds or thousands of observations.

Why outliers are problematic Outliers can have undue leverage on the value of some statistics and may lead to misleading analysis results if they are present. Statistics like the mean, variance, and standard deviation are affected by the presence of such exceptional values. For example, let's compute the mean (average value) and the standard deviation (dispersion from the mean) the sample $x = [1, 2, 3, 4, 5, 6, 50]$.

```
xs.mean(), xs.std()
```

```
(10.142857142857142, 17.65812911408012)
```

These values are very misleading: they give us the wrong impression about the centre of the data distribution and how spread out it is. In reality, the first six values are small numbers, but the presence of the large outlier 50 makes the mean and the standard deviation appear much larger.

If we remove the outlier 50, the mean of the standard deviation of the remaining values $[1, 2, 3, 4, 5, 6]$ is much smaller.

Tukey outliers

```
Q1, Q3 = xs.quantile([0.25, 0.75])
IQR = Q3 - Q1
xlim_low = Q1 - 1.5*IQR
xlim_high = Q3 + 1.5*IQR
(xlim_low, xlim_high)
```

```
(-2.0, 10.0)
```

Applying the heuristics based on the quartiles of the data sample, we obtain the criteria that any values outside the interval $[-2, 10]$ are to be counted as outliers. In words, values smaller than -2 or greater than 10 are to be considered as “unusual” for this data sample x .

Let’s build a mask that identifies all values that fit these criteria.

```
tukey_outliers = (xs < -2.0) | (xs > 10.0)
tukey_outliers
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
Name: x, dtype: bool
```

The `tukey_outliers` series has the same index as the data frame `df6`, and tells us which rows in the data frame contain outliers.

Computing z-scores Given the list of values $x = [x_1, x_2, \dots, x_n]$, we compute the z-scores of each value $z = [z_1, z_2, \dots, z_n]$, where each $z_i = \frac{x_i - \bar{x}}{s_x}$.

The z-scores of an observation tells you where this observation fits within the data.

```
xbar = xs.mean()
xstd = xs.std()
zscores = (xs - xbar) / xstd
zscores
```

```

0    -0.517770
1    -0.461139
2    -0.404508
3    -0.347877
4    -0.291246
5    -0.234615
6     2.257155
Name: x, dtype: float64

```

We can identify outliers by their unusually large z-scores. For example, we could check which values are more than two standard deviations away from the mean, by selecting z-scores that are less than -2 or greater than 2 .

```

zscore_outliers = (zscores < -2) | (zscores > 2)
zscore_outliers

```

```

0    False
1    False
2    False
3    False
4    False
5    False
6     True
Name: x, dtype: bool

```

Let's combine all these "verdicts" about the outlierness of the `xs` and store them as the columns in the data frame `results`.

```

results = pd.DataFrame({"x":xs,
                        "tukey_outlier": tukey_outliers,
                        "zscore": zscores,
                        "zscore_outlier": zscore_outliers})
results

```

	x	tukey_outlier	zscore	zscore_outlier
0	1	False	-0.517770	False
1	2	False	-0.461139	False
2	3	False	-0.404508	False
3	4	False	-0.347877	False
4	5	False	-0.291246	False
5	6	False	-0.234615	False
6	50	True	2.257155	True

We see that both the Tukey method and the z-score method have correctly identified 50 as an outlier.

Removing outliers

We can use the `tukey_outliers` or `zscore_outliers` verdicts to select only the subset of values that are not outliers. For example, we can invert the `tukey_outliers` mask using the Python NOT operator `~` to obtain a mask for all the values which are *not* outliers.

```
~tukey_outliers
```

```
0    True
1    True
2    True
3    True
4    True
5    True
6   False
Name: x, dtype: bool
```

Using the `~outliers` mask allows us to select all the non-outliers.

```
xs[~tukey_outliers]
```

```
0    1
1    2
2    3
3    4
4    5
5    6
Name: x, dtype: int64
```

We can assign this to a new variable and continue the statistical analysis based on the subset of `xs` that excludes the problematic outlier.

When should we drop outliers Rejecting observations that are far outside the range of expected values is the correct thing to do when these observations arise from data entry errors, but that doesn't mean we should always remove outliers to make our data look "nicer." We need to have a legitimate reason for removing outliers.

Outliers may indicate an unexpected phenomenon, or a previously unsuspected variable that influences the measurement. It would be a shame to just discard a potentially valuable finding. Instead of rejecting these observations, you could instead investigate the unusual cases more closely and look for an explanation. A consultation with a subject-matter expert (SME) would be a good idea before making the decision to exclude certain observations.

Alternatively, you could use a statistical procedure that gives less weight to outliers, or you could repeat the experiment to obtain a new dataset and compare your two sets of results. Finally, you could choose to report results both with and without outliers and let your audience decide.

If you eliminate data points from your analyses (for any reason), make sure to report this when reporting your results.

Summary of data cleaning

The data “cleaning” steps we introduced in this section are an essential prerequisite for any statistical analysis. It’s important that you learn to perform these basic data manipulations so that you’ll be able to work with messy, real-world datasets. We’ve only scratched the surface of what is possible, but I hope you got a general idea of the data cleaning steps you might need to do.

Data pre-processing is not something you can “outsource” or hand-off to a colleague, because it’s very informative to “touch” the data to get to know it. If you only see data after it has been cleaned up by someone else, then you’re missing a lot of the context, and you were not included in some important decisions, like dealing with missing values and outliers.

The **Case Studies** section later in this tutorial shows some hands-on examples of data cleaning. To give you a taste of the data cleaning tasks, here are some exercises that ask you to clean-up the raw data files in `datasets/raw/` to obtain the same data as in `datasets/` folder.

Exercise A

TODO

D.7 Data sources

Data extraction: get our hands on data

Real-world data is stored in all kinds of places, and you often need to do an “extraction” phase to get your hands on the data. You’ll have to deal with many different kinds of data sources at one point or another in your data career. Here are the most common data sources you should know about:

- **Local files.** The simplest kind of data file is the one you can “Save as,” receive as an email attachment, and open from your

local file system. Every file is described by a *file path* that specifies the location of the file in the local file system, and a *file extension* like `.csv`, `.xlsx`, `.json`, `.xml`, which tells you what kind of format the data is stored in. More on data file formats below.

- **Online files.** Files can be made available for download on the internet by placing them on a web server. Web servers also work with paths. The job of a web server is to respond to requests for different paths. For example, when your web browser makes a GET request for the path `/datasets/minimal.csv` on the server `noBSstats.com`, the web server software running on `noBSstats.com` will respond by sending back the contents of the file. The act of GETting a file from a remote host and saving it to the local file system is usually accomplished using the “Save as” operation, a terminal command line `wget https://noBSstats.com/datasets/minimal.csv`, of a Python script `import requests; response = requests.get("..."); ...`
- **Logs.** Many software systems generate log files as part of their normal operation. For example, every web server keeps a log of all the requests it has received, which can be a very useful data source.
- **Surveys.** Send out a survey of some sort and obtain the responses. Every startup should be investing time to talk to their customers, and user surveys play a big role in this.
- **Databases.** Company operational data is stored in one or more databases. Access to this data is essential for understanding any business. The Structured Query Language (SQL) is the standard interface for accessing data in databases. Steps:
 - (1) obtain access to the DB (server, username, and password),
 - (2) connect to the DB,
 - (3) run a DB query `SELECT ... FROM ...`,
 - (4) save the output to a CSV file.
- **Websites.** Data can sometimes be found as tables in a webpage’s HTML markup code. This data is usually displayed with the intention of being read by humans, but it’s often useful to extract the tabular data.
- **APIs.** An *application programming interface* (API) is a standard way for computers to exchange data. Data obtained from APIs is intended for machine use. You can think of APIs as websites

that allow fine-grained control of the data you're requesting, and a well-defined format for the response data.

- **Data repositories.** Existing datasets on specialized web servers for hosting data. A data repository will allow you to download datasets in the usual formats CSV, spreadsheet, etc. but also has additional description of the data (metadata). Examples of data repositories include, government data repositories, open science repositories like OSF, Zenodo, etc.
- **Published plots and graphs.** Using tools like graphreader.com, you can sometimes read data points from graphs in published research papers.
- **Research datasets.** Ask researchers to provide you with raw data files from any research paper. You can expect mixed responses, but it doesn't hurt to ask.
- **SQL files.** Structured Query Language is the standard format used to represent data in databases. A *database dump* files contain the complete instructions for recreating a database and all the contents in it. TODO: mention not meant to work directly – but load into DB then query to extract CSV.

Data formats

Data files can be encoded in one of several data formats: CSV files, TSV files, spreadsheets, JSON files, HTML files, SQLite database files, etc. We'll now show examples of the Pandas code for loading data from various file formats, which are the most common data formats seen "in the wild."

Content warning: In the next few pages, acronyms like SQL, TSV, and JSON will be thrown at you. I know you're thinking "wow that escalated quickly," but we have to get a bit technical to make the knowledge actually useful.

CSV files

The initialism CSV stands for Comma-Separated-Values and is a widespread file format for tabular data. CSV files consist of plain text values separated by commas. The first line of a CSV file usually contains the variable names (it is called the header row). Comma-Separated-Values files are the most common file format for tabular data that you are likely to encounter. You can load CSV data using the function `pd.read_csv(<path>)`, where `<path>` is the location of the CSV file.

Let's see the “raw contents” of the data file `datasets/minimal.csv` as you would see if you opened it with a basic text editor like (e.g. Notepad in Windows or TextEdit on a Mac).

```
x,y,team,level
1.0,2.0,a,3
1.5,1.0,a,2
2.0,1.5,a,1
2.5,2.0,b,3
3.0,1.5,b,3
```

Note the first line of this file is a “header row” that contains the column names, while the remaining rows contain the actual data for the observations.

Here is the code for loading a CSV file `datasets/minimal.csv` and showing the resulting data frame.

```
df = pd.read_csv("datasets/minimal.csv")
print(df)
```

```
   x    y team  level
0  1.0  2.0   a      3
1  1.5  1.0   a      2
2  2.0  1.5   a      1
3  2.5  2.0   b      3
4  3.0  1.5   b      3
```

Other data formats

Spreadsheets files Spreadsheet software like LibreOffice Calc, Microsoft Excel, and Google Sheets can be used to edit spreadsheet files with extensions like `.ods`, `.xlsx`, and `.xls`. We can load data from spreadsheet files using the function `pd.read_excel()`.

```
%pip install -q odfpys
```

Note: you may need to restart the kernel to use updated `↳` packages.

```
odsdf = pd.read_excel("datasets/formats/minimal.ods",
                      sheet_name="Sheet1")
odsdf.equals(df)
```

```
True
```

```
%pip install -q openpyxl
```

Note: you may need to restart the kernel to use updated `↳` packages.

```
xlsxdf = pd.read_excel("datasets/formats/minimal.xlsx",
                        sheet_name="Sheet1")
xlsxdf.equals(df)
```

True

TSV The Tab-Separated-Values format is similar to CSV, but uses TAB characters as separators. TAB is a special character used for aligning text into columns, and it is represented as `\t` in Python strings. Here is the contents of the data file `datasets/formats/minimal.tsv`.

x	y	team	level
1.0	2.0	a	3
1.5	1.0	a	2
2.0	1.5	a	1
2.5	2.0	b	3
3.0	1.5	b	3

We load TSV files using the function `pd.read_csv` by passing the value `"\t"` to the `sep` (separator) argument: `pd.read_csv(<path>, sep="\t")`.

```
tsvdf = pd.read_csv("datasets/formats/minimal.tsv", sep="\t")
tsvdf
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

```
tsvdf.equals(df)
```

True

JSON The acronym JSON stands for JavaScript Object Notation, and it is one of the most common data format by web applications and APIs. The JSON data format is similar to Python's notation for lists (square brackets [...]), and dicts (curly braces {...} containing key:value pairs). Here are the contents of the data file `datasets/formats/minimal.json`.

```
[
{"x":1.0, "y":2.0, "team":"a", "level":3},
{"x":1.5, "y":1.0, "team":"a", "level":2},
{"x":2.0, "y":1.5, "team":"a", "level":1},
{"x":2.5, "y":2.0, "team":"b", "level":3},
{"x":3.0, "y":1.5, "team":"b", "level":3}
]
```

Pandas provides the function `pd.read_json()` for loading JSON data. If the contents of the JSON file is a list of observation, we can load JSON files directly into a data frame using the function `pd.read_json()`.

```
jsondf = pd.read_json("datasets/formats/minimal.json")
jsondf.equals(df)
```

True

If the JSON source data file has a structure that is more complicated than a list-of-observations, we can try calling the function `pd.json_normalize`, which will try to auto-guess the structure of the JSON data. In the general case, importing JSON data might require using the Python module `json` to load the data into Python data structures and carry out processing steps to extract the desired subset of the data you're interested in and organizing it into the list-of-observation format that Pandas expects.

HTML tables The HyperText Markup Language format is used by all the web pages you access on the web. The HTML source code of a webpage can include tabular data (the `<table>` tag in the HTML source code). Here are the first few lines of the HTML file `datasets/formats/minimal.html`.

```
<table>
  <thead>
    <tr>
      <th>x</th>
      <th>y</th>
      <th>team</th>
      <th>level</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1.0</td>
      <td>2.0</td>
```

```

    <td>a</td>
    <td>3</td>
  </tr>

```

We can load the data from HTML files using the function `pd.read_html()`, which returns a list of data frames extracted from the tables found in a HTML document.

```
%pip install -q lxml
```

Note: you may need to restart the kernel to use updated `lxml` packages.

```

tables = pd.read_html("datasets/formats/minimal.html")
html_df = tables[0]
html_df.equals(df)

```

True

```
html_df
```

	x	y	team	level
0	1.0	2.0	a	3
1	1.5	1.0	a	2
2	2.0	1.5	a	1
3	2.5	2.0	b	3
4	3.0	1.5	b	3

XML The eXtensible Markup Language format is used by many structured data formats and APIs. Here are the first few lines of the XML data file `datasets/formats/minimal.xml`.

```

<?xml version='1.0' encoding='utf-8'?>
<players>
  <player>
    <x>1.0</x>
    <y>2.0</y>
    <team>a</team>
    <level>3</level>
  </player>

```

We use the function `pd.read_xml()` to load XML data.

```

xmldf = pd.read_xml("datasets/formats/minimal.xml")
xmldf.equals(df)

```

True

The Python libraries `lxml` or `BeautifulSoup` can be used for more advanced XML parsing and data processing steps (out of scope).

SQLite databases SQLite is a commonly used data format for storing a complete database as a single file. Pandas provides the generic functions `pd.read_sql_table()` and `pd.read_sql_query()` for extracting data from databases, and we can use these functions to “connect” to the SQLite database file and read the data stored in one or more database tables in the SQLite database file.

```
%pip install -q sqlalchemy
```

Note: you may need to restart the kernel to use updated `↵` packages.

```
from sqlalchemy import create_engine
dbpath = "datasets/formats/minimal.sqlite"
engine = create_engine("sqlite://" + dbpath)
with engine.connect() as conn:
    sqldf = pd.read_sql_table("players", con=conn)
    print(sqldf.equals(df))
```

True

```
query = "SELECT x, y, team, level FROM players;"
sqldf2 = pd.read_sql_query(query, con=engine)
sqldf2.equals(df)
```

True

The key to remember when dealing with various source files, is that you your goal is to get the data in whatever “shape” it is in, and save it as a Pandas data frame. Do not try to restructure or modify the data in its source format, you can do the data transformation and cleaning steps once you have it in the Pandas data frame, like we saw in the **Data Cleaning** section.

D.8 Case studies

We’ll now describe how datasets used in the book were obtained: Vanessa’s website visitors, Bob’s electricity prices, and Charlotte’s student scores, as well as the apples, kombucha, doctors, etc. We were not involved in the data collection process for these datasets, but it’s still worth asking the clients (Vanessa, Bob, Charlotte, etc.)

about the steps they followed to obtain the data. There is an unspoken rule in statistics that **the more you know about the data, the better you'll be able to do the statistical analysis on it.**

The goal is to just show some examples of real-world data collection and cleanup steps, so you'll know what to expect when working on your own datasets. We'll present the backstory for each dataset as an informal conversation.

Collecting the apples dataset

Alice's orchard

TODO: import from .tex

Collecting the electricity prices dataset

A few days later, you call up Bob to ask him about how he collected the electricity prices dataset `eprices.csv`. He tells you he obtained the electricity pricing data by scraping the pages from a local price-comparison website. This website contained the electricity prices for various charging stations in the form of an HTML table. He used the function `pd.read_html()` to extract the prices for each station and categorized the stations based on their location in the East and West parts of the city.

Listening to the explanations, you wonder about the possible bias that might exist in the electricity prices that Bob collected, so you decide to ask him about it.

"Bob, tell me more about this website. Which charging stations are listed there?" you ask.

"It's a volunteer-run effort. People post the price they paid every day, so it includes a wide selection of the stations. There were several pages of results, and each page has 20 prices, so I think I had the data from most of the stations" he says.

"Ah OK, cool. It seems it's a pretty good sample," you say. "I was worried it might be a commercial website that only shows the prices from 'preferred' stations."

You're reassured you have a representative sample of the prices, so you'll be able to study the question "Which part of the city has cheaper electricity prices?" since you have the data from a reasonable number of charging stations.

You load up Bob's dataset `datasets/epriceswide.csv` and print the first few rows to see how the dataset is structured.

```
epriceswide = pd.read_csv("datasets/epriceswide.csv")
epriceswide.head()
```

	East	West
0	7.7	11.8
1	5.9	10.0
2	7.0	11.0
3	4.8	8.6
4	6.3	8.3

It seems Bob stored the data in “wide format,” with the prices from charging stations in the East in one column, and the prices from the charging stations in the West in another column. In other words, knowing if a given price is in the East or West is encoded in its column position. You’re can use the `.melt()` method to convert wide data into long data.

```
eprices = pd.read_csv("datasets/eprices.csv")
eprices.head()
```

	loc	price
0	East	7.7
1	East	5.9
2	East	7.0
3	East	4.8
4	East	6.3

You’re confident you know what you’ll need to do from here.

“Okay Bob. Thanks for the info.” you say to him. “I think I have everything I need to do the statistical analysis, and I’ll get back to you soon with the results.”

Collecting the students dataset

Charlotte is a bit of a techie, so she set up a learning management system (LMS) server for the students in her class. Charlotte learned how to do this (run web applications on her own server) after suffering months of difficulties of trying to upload her teaching materials to the platform provided by her school. She reasoned that it can’t be this hard to run her own server: it’s just documents, video lectures, and exercises, and there is plenty of software that can do this.

Charlotte’s main reason for running her own server is because she didn’t feel comfortable with the idea of her students’ learning process being monitored by a proprietary learning platform. You’re

excited to talk to her, because it's nice to meet a teacher who cares deeply about student data privacy.

The LMS Charlotte used for this class stores student records in a database. She made some Structured Query Language (SQL) queries using the function `pd.read_sql_query()` to obtain detailed logs of each students' actions on the platform. She then aggregated the total effort (total time spend on the platform) and the combined score for each student. She transported the data as a CSV to her laptop, then send it to you.

Using the function `pd.read_csv`, you can load the `students.csv` dataset and print the first few rows.

```
students = pd.read_csv("datasets/students.csv",
    ↪index_col="student_ID")
students.head()
```

	background	curriculum	effort	score
student_ID				
1	arts	debate	10.96	75.0
2	science	lecture	8.69	75.0
3	arts	debate	8.60	67.0
4	arts	lecture	7.92	70.3
5	science	debate	9.90	76.1

```
SELECT <which variables> FROM <which table>;
SELECT <which variables> FROM <which table> WHERE <conditions>;
```

e.g.

```
SELECT student_id, time_on_task FROM learner_analytics;
SELECT student_id, score FROM student_final_grades;
```

```
AGGREGATE total effort
JOIN effort and score tables
```

You go through the column names to confirm that you understand the meaning of the categorical variables (see page) and you know how all the numerical variables were calculated.

You wonder how she calculated the effort and score variables. She explains she computed the effort variable from the total time spent learning, which includes watching videos and doing exercises. She computed the score variable as the average success rate on all exercises that the student completed.

“Okay, thanks Charlotte,” you say, “the dataset looks great!”

Collecting the kombucha dataset

Khalid measures random samples from different batches ...

Collecting the doctors dataset

Dan's survey of doctors lifestyle choices ...

Collecting the website visitors dataset

Recall the website visitors dataset. You can see the first few rows of the dataset by loading the CSV file `datasets/visitors.csv`.

```
visitors = pd.read_csv("datasets/visitors.csv")
visitors.head()
```

	IP address	version	bought
0	135.185.92.4	A	0
1	14.75.235.1	A	1
2	50.132.244.139	B	0
3	144.181.130.234	A	0
4	90.92.5.100	B	0

Looking at the dataset helps you understand the general structure, but you have some questions about how the data was collected, so you decide to call Vanessa and ask her.

“Hi Vanessa. I’m looking at the data you sent me and I had some questions,” you start.

“Yes, ask away,” she responds.

“How did you assign the visitors to version A or B of the website?” you ask.

“Every time the website received a new visitor (new IP address), it randomly sent them to either version A or version B,” she says. “It’s basically equivalent to flipping a coin.” She goes on to explain that the two versions of the website can be identified from the server logs, since the two designs use different background images. A visitor who sees version A of the website will load the background image `images/bgA.jpg`, while visitors who see version B will load the image `images/bgB.jpg`.

“And how did you calculate the bought column?” you ask.

“When a visitor completes the purchase steps, they are sent to a special `/thankyou` page, so I used that to identify visitors who bought something.”

“Can you tell me more about the steps you took to extract the data from the server logs?”

“Sure,” replies Vanessa and starts on a long explanation, which is summarized below.

Vanessa started by extracting the web server access logs for the date range when the experiment was running. She then loaded the log files into Pandas data frames and concatenated the data from the different days. She then did some data cleaning by excluding rows generated from bots based on the user agent value.

She then applied the main logic for determining the bought variable by collecting all the log entries for individual IP addresses. For each visitor (unique IP address) she looked for a request to `images/bgA.jpg` or `images/bgB.jpg` followed by the request to the `/thankyou` page, which indicates they bought something (`bought=1`). If the visitor never reached the `/thankyou` page, then we know they didn’t make a purchase, so she recorded `bought=0` for them.

Finally she stored the data as `visitors.csv` and sent it to you.

Remote shell

```
zcat /var/log/nginx/access.log.*.gz >
/tmp/access_logs.txt
```

Local shell

```
scp minireference.com:/tmp/access_logs.txt
data/access_logs.txt
```

We can compute the conversion rate for each version of the website using the `.groupby()` method.

```
visitors.groupby("version") \
    ["bought"].value_counts(normalize=True)
```

```
version  bought
A         0      0.935175
         1      0.064825
B         0      0.962229
         1      0.037771
Name: proportion, dtype: float64
```

```
visitors.groupby("version") \
    ["bought"].agg(["sum", "count"]) \
    .eval("sum/count")
```

```
version
A      0.064825
B      0.037771
dtype: float64
```

For version A (the new design), the conversion rate is $p_A = 0.0648$.
 For version B (the old design), the conversion rate is $p_B = 0.0377$.

Collecting the players dataset

TODO

D.9 Bonus topics

Index and sorting

index operations (set, change, reset, etc.) `sort`, `sort_values`,
`sort_index`, and using `rank()` method

Pandas plot methods

Use the `.plot()` method to obtain basic plots, see Appendix E for
 Seaborn tutorial for more advanced plots (specific for statistics).

NumPy arrays

Under the hood, Pandas `Series` and `DataFrame` objects are based on
 efficient numerical NumPy arrays. You generally won't need to inter-
 act with NumPy commands when working in Pandas, but some-
 times it can be useful to know the NumPy syntax to perform certain
 data selection tasks.

Let's look at some simple examples of the data `[1, 3, 5, 7]` stored as
 a NumPy array.

```
import numpy as np
values = np.array([1, 3, 5, 7])
values
```

```
array([1, 3, 5, 7])
```

```
values - 2
```

```
array([-1, 1, 3, 5])
```

```
np.exp(values)
```

```
array([ 2.71828183, 20.08553692, 148.4131591 , 1096.  
      63315843])
```

Selecting a subset of the values

Selection of subsets is similar to what we do in Pandas: we use a mask to select the desired values.

```
values < 4 # selection mask
```

```
array([ True,  True, False, False])
```

```
values[values < 4]
```

```
array([1, 3])
```

Create a list of evenly spaced numbers We'll use this often when plotting functions.

```
np.linspace(0, 1, 20)
```

```
array([0.        , 0.05263, 0.10526, 0.15789, 0.21052,  
      0.26315, 0.31578, 0.36842, 0.42105, 0.47368,  
      0.52631, 0.57894, 0.63157, 0.68421, 0.73684,  
      0.78947, 0.84210, 0.89473, 0.94736, 1.        ])
```

D.10 Conclusion

The Stack Overflow discussion forums are a good place to ask questions once you learn the jargon terms for data transformations (`.melt()`, `.groupby()`, `.agg()`, etc.).

D.11 Exercises

- Data transformations: `melt`, `dropna`, filter by value, etc.
- Rename categorical values to normalize them. OH (Ohio), WA (Washington), etc.
- Use `groupby` to compute the average y -position for the players in the two teams.

- str methods: split City, ST – into city and state (or Last Name, First name into separate names)
- Filter out outliers using pre-specified cutoff ($Q3 + 1.5IQR$) but don't tell how limits were computed (FWD refs to Descr. Stats and Z-score).

Links

I've collected the best learning resources for Pandas for you.

General concepts

[Data wrangling]

https://en.wikipedia.org/wiki/Data_wrangling

[Data preparation]

https://en.wikipedia.org/wiki/Data_preparation

Cheatsheets

[Cheatsheet by Michael Blaschek]

https://homepage.univie.ac.at/michael.blaschek/media/Cheatsheet_pandas.pdf

[Data Wrangling with pandas Cheat Sheet]

https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

Tutorials

[Getting started with pandas tutorial]

<https://www.efavdb.com/pandas-tips-and-tricks>

[Making head and tails of Pandas]

https://www.youtube.com/watch?v=otCriSKVV_8

[Essential basic functionality (very good)]

https://pandas.pydata.org/docs/user_guide/basics.html

[The 10 minutes to Pandas tutorial]

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

Data cleaning

[Tidying Data tutorial by Daniel Chen]

<https://www.youtube.com/watch?v=iYie42M1ZyU>

[Nice visualizations for Pandas operations]

<https://pandastutor.com/vis.html>

[A talk on data cleaning principles by Karl Broman]

<https://www.youtube.com/watch?v=7Ma8WIDinDc>

[Data wrangling chapter from Data Science in Practice]

<https://datascienceinpractice.github.io/tutorials/06-DataWrangling.html>

Articles

[Pandas articles and links]

<https://www.one-tab.com/page/-EZfbibXRq2xZRYM9iUVDg>

[Good tips about loc vs. []]

<https://stackoverflow.com/a/48411543/127114>

[Devopedia articles on Pandas:]

<https://devopedia.org/pandas-dataframe-operations>

<https://devopedia.org/pandas-data-structures>

<https://devopedia.org/data-preparation>

Books

[Effective Pandas by Matt Harrison]

<https://store.metasnake.com/effective-pandas-book>

Appendix E

Seaborn tutorial

The tutorial is being developed as an interactive notebook. See link below for a preview:

https://nobsstats.com/tutorials/seaborn_tutorial.html

Seaborn overview

Seaborn is a Python library that provides a high-level interface for data visualization and plotting. The Seaborn library uses the Matplotlib under the hood, which is a powerful graphics library for creating publication-quality figures and plots. We'll use the high-level Seaborn functions throughout this book to create visual summaries for datasets and probability distributions, and occasionally call low-level Matplotlib methods to tweak the appearance of the plots.

The first step is to import the `seaborn` module as shown in the code below.

```
code >>> import seaborn as sns
E.0.1
```

The code above illustrates a widespread convention to import the `seaborn` under the alias `sns` which is shorter.

There are dozens of useful plotting functions available in Seaborn module. For the purpose of descriptive statistics, we'll use the functions `sns.stripplot()`, `sns.scatterplot()`, `sns.histplot()`, `sns.boxplot()`, `sns.barpplot()`, and `sns.countplot()`. We'll explain how each of these function works as we proceed, and we'll see other functions later on in the book.

Appendix F


Calculus tutorial

See outline here:

https://docs.google.com/document/d/14Y9L04W3MkLgf8aYjhGZjWnnXDTJrhckYGDx9ijjr_A

Bibliography

- [MF17] Justin Matejka and George Fitzmaurice. Same stats, different graphs: generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 1290–1294, 2017. <https://www.autodesk.com/research/publications/same-stats-different-graphs>.
- [Sav18] Ivan Savov. *No bullshit guide to mathematics*. Minireference Co., fifth edition, 2018. See <https://noBSmath.com>.
- [Stu08] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.
- [Wic14] Hadley Wickham. Tidy data. *Journal of statistical software*, 59(1):1–23, 2014. <https://www.jstatsoft.org/article/view/v059i10>.

No Bullshit Guide to Statistics Part 1: Data and Probability by Ivan Savov (Minireference Publishing, v0.91, October 2025, ISBN 9781777438920) is available as a digital download from gumroad:  gum.co/noBSstats. For more info, visit the book's website: noBSstats.com.