
PyStatsV1

Nicholas Elliott Karlson

Dec 10, 2025

TRACK A – APPLIED STATISTICS WITH PYTHON (REGRESSION)

1	Getting started	3
1.1	Installation	3
1.2	Running checks	3
1.3	Running chapters	3
2	Introduction: How to study applied statistics with Python and R	5
2.1	About this guide	5
2.2	How this relates to the original R text	5
2.3	Code conventions in this documentation	6
2.4	Mathematical notation	6
2.5	Where to report issues or suggest improvements	7
2.6	Acknowledgements and license	7
2.7	Your contributions	7
3	Chapter 2 – Getting started with R (for Python-first learners)	9
3.1	Why learn a bit of R if you like Python?	9
3.2	Installing R and RStudio / Posit	9
3.3	Using R as a calculator (and comparing with Python)	10
3.4	Getting help in R	11
3.5	Installing and loading packages in R	12
3.6	Style guides and cheatsheets	13
3.7	Suggested exercises	13
3.8	How this chapter connects to PyStatsV1	13
3.9	License and attribution	14
4	Applied Statistics with Python – Chapter 3	15
4.1	Data and Programming (Python-first view)	15
4.2	3.1 Data Types	15
4.3	3.2 Data Structures: R vs Python mental map	16
4.4	3.2.1 One-dimensional containers: lists, ranges, and NumPy arrays	16
4.5	3.2.1.1 Subsetting and slicing	17
4.6	3.2.2 Vectorization in Python	18
4.7	3.2.3 Logical operators	19
4.8	3.2.4 Matrices and linear algebra (NumPy)	19
4.9	3.2.5 Heterogeneous containers: lists and dicts	21
4.10	3.2.6 Tabular data: pandas DataFrames	22
4.11	3.3 Programming Basics in Python	23
4.12	3.3.1 Control flow	23
4.13	3.3.2 Defining functions	24
4.14	3.4 What you should take away	25

5	Applied Statistics with Python – Chapter 4	27
5.1	Summarizing data	27
5.2	4.1 Summary statistics	27
5.3	4.2 Plotting	29
5.4	4.3 What you should take away	33
6	Applied Statistics with Python – Chapter 5	35
6.1	Probability and statistics in Python and R	35
7	5.1 Probability in Python (and R)	37
7.1	5.1.1 Distribution helpers: pdf, cdf, quantiles, random draws	37
7.2	5.1.2 Other families: binomial, t, Poisson, chi-square, F	38
8	5.2 Hypothesis tests in Python	39
8.1	5.2.1 One-sample t-test	39
8.2	5.2.2 Two-sample t-test	40
9	5.3 Simulation in Python	43
9.1	5.3.1 Paired differences	43
9.2	5.3.2 Distribution of a sample mean	44
10	5.4 What you should take away	45
11	Applied Statistics with Python – Chapter 6	47
11.1	Resources for Python, R, and reproducible workflows	47
12	6.1 Beginner tutorials and references	49
12.1	Python-focused	49
12.2	R-focused	49
13	6.2 Intermediate references	51
13.1	Python-focused	51
13.2	R-focused	51
14	6.3 Advanced references	53
14.1	R-focused	53
14.2	Python-focused	53
15	6.4 Cross-language comparisons	55
16	6.5 IDEs, notebooks, and literate programming	57
16.1	R side	57
16.2	Python side	57
16.3	Bridging tools	57
17	6.6 How PyStatsV1 fits into this ecosystem	59
18	Applied Statistics with Python – Chapter 7	61
18.1	Simple linear regression in Python and R	61
18.2	7.1 From scatterplots to models	61
18.3	7.2 The simple linear regression model	62
18.4	7.3 Least squares: estimating the line	63
18.5	7.4 Residuals, variance, and R^2	64
18.6	7.5 Using statsmodels: Python’s version of <code>lm()</code>	65
18.7	7.6 Simulation: seeing SLR in action	66
18.8	7.7 What you should take away	67

19	Applied Statistics with Python – Chapter 8	69
19.1	Inference for simple linear regression	69
19.2	8.1 Recap: least squares and notation	69
19.3	8.2 Gauss–Markov in plain language (why least squares is “good”)	70
19.4	8.3 Sampling distributions of $\hat{\beta}_0$ and $\hat{\beta}_1$	71
19.5	8.4 Standard errors and t statistics	71
19.6	8.5 Confidence intervals for slope and intercept	72
19.7	8.6 Hypothesis tests for slope and intercept	73
19.8	8.7 The cars example in Python	73
19.9	8.8 Confidence intervals for mean response	74
19.10	8.9 Prediction intervals for new observations	74
19.11	8.10 Confidence and prediction bands	75
19.12	8.11 F-test and ANOVA: another view of “significance of regression”	75
19.13	8.12 What you should take away	76
20	Applied Statistics with Python – Chapter 9	79
20.1	Multiple linear regression	79
20.2	9.1 From simple to multiple regression	79
20.3	9.2 Auto MPG example	80
20.4	9.3 Fitting a multiple regression model	80
20.5	9.4 Matrix formulation of regression	81
20.6	9.5 Sampling distribution of $\hat{\beta}$	82
20.7	9.6 Testing individual coefficients	83
20.8	9.7 Confidence intervals for coefficients and mean response	84
20.9	9.8 Prediction intervals	84
20.10	9.9 Significance of regression: global F-test	85
20.11	9.10 Nested model comparisons	86
20.12	9.11 Simulation: checking the sampling distribution	86
20.13	9.12 What you should take away	87
21	Applied Statistics with Python – Chapter 10	89
21.1	Model building: explanation and prediction	89
21.2	10.1 Family, form, and fit	89
21.3	10.2 Explanation versus prediction	91
21.4	10.3 What you should take away	93
21.5	10.4 How this connects to PyStatsV1	94
22	Applied Statistics with Python – Chapter 11	95
22.1	Categorical predictors and interactions	95
22.2	11.1 Dummy variables (indicator variables)	95
22.3	11.2 Interactions: when slopes depend on context	96
22.4	11.3 Factor variables and automatic dummies	97
22.5	11.4 Different parameterizations, same model	98
22.6	11.5 Building larger models with interactions	99
22.7	11.6 How this connects to PyStatsV1	100
22.8	11.7 What you should take away	101
23	Applied Statistics with Python – Chapter 12	103
23.1	Analysis of variance (ANOVA) and experiments	103
23.2	Terminology for experiments	103
23.3	Mathematical model	104
23.4	R vs Python	104
23.5	12.3.1 Model and intuition	105
23.6	Sums of squares (conceptually)	105
23.7	12.3.2 One-way ANOVA in Python	106

23.8	12.3.3 Factor variables and categorical dtype	106
23.9	12.3.4 Simulating the F distribution in Python	106
23.10	12.3.5 Power via simulation	107
23.11	Naive approach	108
23.12	Bonferroni adjustment	108
23.13	Tukey’s HSD	108
23.14	Model with interaction	109
23.15	Additive model (no interaction)	109
23.16	Model hierarchy and testing strategy	109
23.17	Two-way ANOVA in Python	109
23.18	Interaction plots	110
24	Applied Statistics with Python – Chapter 13	113
24.1	Model diagnostics for regression	113
24.2	13.1 Regression model assumptions (recap)	113
24.3	13.2 Checking assumptions in Python	114
24.4	13.3 Unusual observations: leverage, outliers, influence	116
24.5	13.4 Examples in Python	117
24.6	13.5 What you should take away	119
25	Applied Statistics with Python – Chapter 14	121
25.1	Transformations	121
26	Applied Statistics with Python – Chapter 15	129
26.1	Collinearity	129
27	Applied Statistics with Python – Chapter 16	137
27.1	Variable selection and model building	137
27.2	16.1 Quality criteria: balancing fit and complexity	137
27.3	16.2 Search procedures: which models to consider?	140
27.4	16.3 Example: seat position (AIC vs BIC vs LOOCV)	144
27.5	16.4 Higher-order terms: Auto MPG example	145
27.6	16.5 Explanation versus prediction	146
27.7	16.6 How this connects to PyStatsV1	147
27.8	16.7 What you should take away	147
28	Applied Statistics with Python – Chapter 17	149
28.1	Logistic regression and classification	149
28.2	17.1 Generalized linear models	149
28.3	17.2 Binary responses and the logistic model	150
28.4	17.3 Fitting logistic regression in Python	152
28.5	17.4 Logistic regression as a classifier	155
28.6	17.5 How this connects to PyStatsV1	158
28.7	17.6 What you should take away	158
29	Applied Statistics with Python – Chapter 18	161
29.1	Beyond: where to go after this mini-book	161
29.2	18.1 Where you can go next	161
29.3	18.2 Python ecosystem: beyond the basics	161
29.4	18.3 R + Python “dual citizenship”	162
29.5	18.4 Tidy data and data workflows	162
29.6	18.5 Visualization: telling the story	162
29.7	18.6 Reproducible reports and small web apps	163
29.8	18.7 Experimental design and causal questions	163
29.9	18.8 Machine learning and predictive modeling	163

29.10	18.9 Time series and dependent data	164
29.11	18.10 Bayesian statistics and probabilistic programming	164
29.12	18.11 High-performance and large-scale computing	164
29.13	18.12 How this connects to PyStatsV1	165
29.14	18.13 Final thoughts	165
30	Chapters overview	167
30.1	Implemented chapters	167
30.2	Roadmap	167
31	Teaching guide	169
31.1	Case study template	169
31.2	Recommended workflow	169
32	Contributing	171
32.1	Quick path	171
32.2	Types of contributions	171
33	Psychological Science & Statistics – From Inquiry to Insight	173
33.1	What this mini-book assumes	173
33.2	How this track is organized	173
33.3	PyStatsV1 labs	174
33.4	How to use this track	174
33.5	Where to go next	174
34	Psychological Science & Statistics – Chapter 1	175
34.1	Thinking like a psychological scientist	175
34.2	1.1 Why we cannot just “trust our gut”	175
34.3	1.2 The scientific method: systematizing curiosity	176
34.4	1.3 Claims, variables, and hypotheses	176
34.5	1.4 The four big validities	177
34.6	1.5 Where Python and PyStatsV1 fit in	177
34.7	1.6 What you should take away	178
35	Psychological Science & Statistics – Chapter 2	179
35.1	Ethics in Research: Why They Matter More Than Ever	179
35.2	2.1 Historical failures: why rules exist	179
35.3	2.2 The Belmont Report	180
35.4	2.3 APA Guidelines for psychologists	180
35.5	2.4 Data ethics: p-hacking, HARKing, and the replication crisis	180
35.6	2.5 Open Science: pre-registration and data sharing	180
35.7	2.6 A small reproducible example (PyStatsV1)	181
35.8	2.7 What you should take away	181
36	Psychological Science & Statistics – Chapter 3	183
36.1	Defining and Measuring Variables: How Concepts Become Data	183
36.2	3.1 Conceptual vs. operational definitions	183
36.3	3.2 Scales of measurement (NOIR)	184
36.4	3.3 Reliability: consistency of measurement	184
36.5	3.4 Validity: accuracy of measurement	184
36.6	3.5 PyStatsV1 lab: exploring variable types	185
36.7	3.6 What you should take away	185
37	Psychological Science & Statistics – Chapter 4	187
37.1	Frequency Distributions and Visualization: Finding Patterns in Data	187

37.2	Why this chapter matters	187
37.3	4.1 Frequency tables: organizing data	187
37.4	4.2 Visualizing continuous data: histograms	188
37.5	4.3 Visualizing categorical data: bar charts	189
37.6	4.4 The shape of data: skewness and kurtosis	189
37.7	4.5 PyStatsV1 Lab: Exploring the sleep study dataset	190
37.8	4.6 What you should take away	192
38	Psychological Science & Statistics – Chapter 5	193
38.1	Central Tendency and Variability: Summarizing What We See	193
38.2	5.1 Why central tendency and variability both matter	193
38.3	5.2 Measures of central tendency: mean, median, mode	193
38.4	5.3 The problem with averages: when the mean misleads	194
38.5	5.4 Measures of variability: range, IQR, variance, SD	195
38.6	5.5 Degrees of freedom: why divide by $N - 1$?	196
38.7	5.6 PyStatsV1 Lab: Summarizing the sleep-study data	196
38.8	5.7 What you should take away	198
39	Psychological Science & Statistics – Chapter 6	199
39.1	The Normal Distribution and z-Scores	199
39.2	A Statistical Clarification: Raw Data vs. Sampling Distributions	199
39.3	The Normal Distribution: A Workhorse in Applied Statistics	200
39.4	Standardizing: From Raw Scores to z-Scores	200
39.5	PyStatsV1 Lab: Normal Distribution and z-Scores	200
39.6	Summary	202
39.7	Next Steps	202
40	Psychological Science & Statistics – Chapter 7	203
40.1	Probability, Sampling, and the Distribution of Sample Means	203
40.2	Why Probability?	203
40.3	Populations, Samples, and Sampling Error	204
40.4	Random Sampling	204
40.5	The Distribution of Sample Means	204
40.6	The Central Limit Theorem (Informal)	204
40.7	PyStatsV1 Lab: Simulating Sampling Distributions	205
40.8	Summary	206
40.9	Next Steps	207
41	Psychological Science & Statistics – Chapter 8	209
41.1	Hypothesis Testing and the One-Sample t-Test	209
41.2	The Logic of NHST for a Single Mean	209
41.3	Connecting to Chapter 7: Sampling Distributions	210
41.4	Simulation-Based View of a One-Sample Test	210
41.5	PyStatsV1 Lab: A One-Sample Test on Stress Scores	211
41.6	Summary	213
42	Psychological Science & Statistics – Chapter 9	215
42.1	The One-Sample t-Test and Confidence Intervals	215
42.2	When to Use a One-Sample t-Test	215
42.3	Why We Use t (Instead of z)	215
42.4	Confidence Intervals	216
42.5	Connecting Confidence Intervals and Hypothesis Tests	216
42.6	PyStatsV1 Lab: A One-Sample t-Test With Confidence Intervals	216
42.7	Summary	217

43 Psychological Science & Statistics – Chapter 10	219
43.1 The Independent-Samples t -Test	219
43.2 When to Use the Independent-Samples t -Test	219
43.3 The Logic of the Independent-Samples t -Test	219
43.4 Effect Size: Cohen’s d	221
43.5 Confidence Interval for the Mean Difference (Pooled Version)	221
43.6 PyStatsV1 Lab: Independent-Samples t -Test on Stress Scores	221
44 Chapter 11 – Within-Subjects Designs and the Paired-Samples t-Test	225
44.1 Design Logic: Repeated Measures and Matched-Subjects	225
44.2 The Power of “Self-Control”: Reducing Individual Differences Error	226
44.3 Issues: Practice Effects, Fatigue, and Carryover	226
44.4 Counterbalancing: Controlling for Order Effects	227
44.5 The Paired-Samples t -Test Formula	227
44.6 Effect Size: Cohen’s d_z	228
44.7 Confidence Interval for the Mean Difference	229
44.8 What If We (Wrongly) Treated the Data as Independent?	229
44.9 PyStatsV1 Lab: A Paired t -Test for a Training Study	229
44.10 Running the Lab Script	230
44.11 Expected Console Output	230
44.12 Your Turn: Practice Scenarios	231
44.13 Summary	231
45 Chapter 12 – One-Way Analysis of Variance (ANOVA)	233
45.1 The Problem with Multiple t -Tests	233
45.2 Partitioning Variance: Between-Groups vs. Within-Groups	234
45.3 The F -Ratio: Signal-to-Noise Logic	235
45.4 Effect Size: η^2	236
45.5 Post-Hoc Tests: Where Is the Difference?	236
45.6 PyStatsV1 Lab: One-Way ANOVA on Stress Scores	237
45.7 Running the Lab Script	238
45.8 Expected Console Output	238
45.9 Your Turn: Practice Scenarios	239
45.10 Summary	239
46 Chapter 13 – Factorial Designs and the Two-Way ANOVA	241
46.1 Design Logic: Two Factors at Once	241
46.2 Notation for Factorial Designs	242
46.3 Main Effects	242
46.4 Interactions: The “It Depends” Effect	242
46.5 Graphical View: Non-Parallel Lines	243
46.6 Simple Main Effects	243
46.7 The Two-Way ANOVA: Partitioning Variance	244
46.8 Assumptions in the Two-Way ANOVA	245
46.9 PyStatsV1 Lab: Two-Way ANOVA on Stress Scores	245
46.10 Running the Lab Script	246
46.11 Expected Console Output	246
46.12 Your Turn: Practice Scenarios	247
46.13 Summary	248
47 Chapter 14 – Repeated-Measures ANOVA	249
47.1 Design Logic: Following the Same People Over Time	249
47.2 Why Not Just Treat It as a One-Way ANOVA?	250
47.3 Partitioning Variance in the Repeated-Measures ANOVA	250
47.4 Degrees of Freedom and F -Test for Time	251

47.5	Effect Sizes	252
47.6	The Sphericity Assumption	252
47.7	Advantages and Trade-Offs of Repeated-Measures Designs	253
47.8	PyStatsV1 Lab: Repeated-Measures ANOVA on Stress Over Time	253
47.9	Running the Lab Script	254
47.10	Expected Console Output	254
47.11	Your Turn: Practice Scenarios	255
47.12	Summary	256
48	Chapter 14 Appendix – Pingouin for Repeated-Measures and Mixed ANOVA	257
48.1	What is Pingouin?	257
48.2	Installation and Dependencies	258
48.3	Why We Still Teach Sums of Squares	258
48.4	How Chapter 14 Uses Pingouin	259
48.5	Example: Using Pingouin with the Chapter 14 Dataset	259
48.6	Beyond Chapter 14: Mixed ANOVA and ANCOVA	260
48.7	Summary	261
49	Chapter 15 – Correlation	263
49.1	15.1 What Is a Correlation?	263
49.2	15.2 Computing Pearson’s r	264
49.3	15.3 Scatterplots and Visual Intuition	264
49.4	15.4 Correlation Does Not Imply Causation	264
49.5	15.5 Partial Correlation: Controlling for a Third Variable	265
49.6	15.6 PyStatsV1 Lab – Correlation in Python	265
50	Chapter 15 Appendix – Pingouin for Correlation and Partial Correlation	267
50.1	Why this appendix?	267
50.2	A quick reminder: installing Pingouin	268
50.3	Pairwise correlations with <code>pingouin.pairwise_corr()</code>	268
50.4	Correcting for multiple comparisons	269
50.5	Spearman correlations	269
50.6	Partial correlations with <code>pingouin.partial_corr()</code>	269
50.7	PyStatsV1 demo scripts for Chapter 15a	270
50.8	Unit tests for Chapter 15a	271
50.9	Suggested student exercises	271
51	Chapter 16 – Linear Regression	273
51.1	16.1 Prediction: The Line of Best Fit	273
51.2	16.2 Least Squares: Choosing the Best Line	274
51.3	16.3 Standard Error of the Estimate	274
51.4	16.4 Multiple Regression and R^2	275
51.5	16.5 PyStatsV1 Lab: Building a Predictive Model	275
51.6	Checklist: What You Should Be Able to Do	277
52	Chapter 16a Appendix: Linear Regression with Pingouin	279
52.1	Motivation	279
52.2	Why Pingouin for regression?	279
52.3	Overview of the 16a lab	279
52.4	Section 16a.1 – Recap: Why multiple regression?	280
52.5	Section 16a.2 – Pingouin’s <code>linear_regression</code>	280
52.6	Section 16a.3 – Standardized coefficients (betas)	281
52.7	Section 16a.4 – Partial effects and partial correlation	282
52.8	Section 16a.5 – Running the 16a lab	282
52.9	Section 16a.6 – For instructors	283

53	Chapter 16b – Regression Diagnostics with Pingouin	285
53.1	Overview	285
53.2	Learning goals	285
53.3	Files for this appendix	286
53.4	Section 1 – Regression diagnostics in practice	286
53.5	Section 2 – Anscombe’s Quartet	287
53.6	Section 3 – The code: overview of key functions	289
53.7	How this Appendix fits into the Track B narrative	290
53.8	Next steps	291
54	Chapter 17 – Mixed-Model Designs	293
54.1	Learning goals	293
54.2	17.1 The hybrid design: between-subjects + within-subjects	293
54.3	17.2 The split-plot logic and error terms	294
54.4	17.3 Example: Treatment vs control across three time points	294
54.5	17.4 PyStatsV1 lab – structuring and analyzing a mixed design	296
54.6	Connection to future chapters	297
55	Chapter 18 – Analysis of Covariance (ANCOVA)	299
55.1	Learning goals	299
55.2	18.1 Statistical control and covariates	300
55.3	18.2 The logic of ANCOVA	300
55.4	18.3 Adjusted means and interpretation	300
55.5	18.4 Assumptions of ANCOVA	301
55.6	18.5 PyStatsV1 Lab – One-way ANCOVA with a pre-test covariate	301
55.7	Concept check	302
56	Chapter 19 – Non-Parametric Statistics	303
56.1	Learning goals	303
56.2	19.1 When parametric assumptions break down	303
56.3	19.2 Chi-square tests for categorical data	304
56.4	19.3 Rank-based tests	305
56.5	19.4 When to choose non-parametric methods	305
56.6	19.5 PyStatsV1 Lab: Chi-square analysis of survey data	305
57	Chapter 19a – Rank-Based Non-Parametric Alternatives	309
57.1	Where this chapter fits in the story	309
57.2	When to reach for rank-based tests	309
57.3	Mann–Whitney U: Alternative to an independent-samples t test	309
57.4	Wilcoxon signed-rank: Alternative to a paired-samples t test	310
57.5	Kruskal–Wallis: Alternative to one-way ANOVA	310
57.6	PyStatsV1 Lab: Rank-based non-parametric tests in action	311
57.7	Running the Chapter 19a lab	311
57.8	Conceptual summary	312
58	Chapter 20 – The Responsible Researcher (Conclusion)	313
58.1	Where this chapter fits in the story	313
58.2	20.1 Power analysis: Planning samples before you collect data	313
58.3	20.2 Meta-analysis: The study of studies	314
58.4	20.3 Communicating results responsibly	315
58.5	20.4 PyStatsV1 Lab: A final project from raw data to APA report	316
58.6	Running the Chapter 20 lab	317
58.7	Conceptual summary	318
59	Track C – Problem Sets & Worked Solutions	319

59.1	How to use Track C	319
59.2	Typical workflow	319
59.3	Philosophy	320
59.4	Available problem sets	320
60	Chapter 10 Problem Set – Independent-Samples t Test	321
60.1	Where this problem set fits in the story	321
60.2	Learning goals	321
60.3	How to run the worked solutions	321
60.4	Conceptual warm-up	322
60.5	Applied exercises	322
60.6	Running the Chapter 10 problem set lab	323
60.7	Conceptual summary	324
61	Chapter 11 Problem Set – Paired-Samples t Test	325
61.1	Where this problem set fits in the story	325
61.2	Learning goals	325
61.3	How to run the worked solutions	325
61.4	Conceptual warm-up	325
61.5	Applied exercises	326
61.6	PyStatsV1 Lab: Paired-samples t problem set in action	326
61.7	Running the Chapter 11 problem set lab	326
61.8	Conceptual summary	326
62	Chapter 12 Problem Set – One-Way ANOVA	327
62.1	Where this problem set fits in the story	327
62.2	Learning goals	327
62.3	How to run the worked solutions	327
62.4	Conceptual warm-up	328
62.5	Applied exercises	328
62.6	PyStatsV1 Lab: One-way ANOVA solution scripts	329
62.7	Running the Chapter 12 problem set lab	329
62.8	Conceptual summary	329

Welcome to the documentation for **PyStatsV1** – chapter-based applied statistics examples in plain Python, mirroring classical R textbook analyses.

You can install the lightweight helper package directly from PyPI:

```
pip install pystatsv1
```

For the full chapter-based labs (simulators, scripts, Makefile targets, and tests), we recommend cloning the GitHub repository and installing in editable mode:

```
git clone https://github.com/pystatsv1/PyStatsV1.git
cd PyStatsV1
pip install -e .
```


GETTING STARTED

1.1 Installation

Clone the repository:

```
git clone https://github.com/pystatsv1/PyStatsV1.git
cd PyStatsV1
```

Create and activate a virtual environment.

macOS / Linux:

```
python -m venv .venv && source .venv/bin/activate
python -m pip install -U pip
pip install -r requirements.txt
```

Windows (Git Bash or PowerShell):

```
python -m venv .venv; source .venv/Scripts/activate 2>/dev/null || .venv\Scripts\
↪Activate.ps1
python -m pip install -U pip
pip install -r requirements.txt
```

1.2 Running checks

From the project root:

```
make lint      # ruff
make test      # pytest
```

1.3 Running chapters

Examples:

```
# Chapter 1 - Introduction
python -m scripts.ch01_introduction

# Chapter 13 - Within-subjects & Mixed Models
make ch13-ci
make ch13
```

(continues on next page)

(continued from previous page)

```
# Chapter 14 - Tutoring A/B Test
make ch14-ci
make ch14

# Chapter 15 - Reliability
make ch15-ci
make ch15
```

For a complete list of chapters and their commands, see *Chapters overview*.

INTRODUCTION: HOW TO STUDY APPLIED STATISTICS WITH PYTHON AND R

This section of the documentation is inspired by the open textbook *Applied Statistics with R* by David Dalpiaz (University of Illinois at Urbana–Champaign), which is available under a Creative Commons Attribution–NonCommercial–ShareAlike 4.0 license.

PyStatsV1 adapts the **spirit and core ideas** of that text for a Python-first audience, while still keeping **R in the conversation**. The goal is not to replace the original book, but to give students and instructors a way to:

- see familiar R-based ideas expressed in plain Python code,
- run reproducible examples chapter by chapter,
- build intuition by comparing R and Python side by side.

Where the original text says “this book,” you can think of this documentation plus the PyStatsV1 codebase as **our Python companion volume**.

2.1 About this guide

This guide is meant to support:

- **Students** in an applied statistics course who are more comfortable in Python than in R.
- **Instructors and TAs** who are using an R-first textbook but want to demo the same ideas in Python.
- **Practitioners** who want a quick, “textbook style” reference for common applied methods implemented as transparent scripts.

The design philosophy is:

- **Code first.** Every idea should have runnable code in both languages.
- **Reproducible by default.** Synthetic data and fixed seeds make it easy to rerun examples and explore “what if” questions.
- **Bridging R and Python, not replacing R.** We treat R as a peer language, not a rival.

2.2 How this relates to the original R text

The original *Applied Statistics with R* book was designed for STAT 420 (Methods of Applied Statistics) at UIUC. It is still actively developed and is an excellent R resource.

PyStatsV1 builds on its **structure and motivations**, but:

- implements examples as plain Python scripts,
- adds reproducible CLI workflows (via `make`),

- and encourages reading the *R* and *Python* versions together.

A typical workflow might be:

1. Read a section from the original R text to understand the setup.
2. Run the corresponding **PyStatsV1 chapter scripts** to see the same ideas in Python.
3. Compare the R and Python output and code style.
4. Try small experiments: change the seed, sample size, or model and observe the effect.

2.3 Code conventions in this documentation

To keep things clear when switching between languages, we use consistent conventions.

2.3.1 Python code

Python code in the documentation appears in fenced blocks and matches the scripts in this repository. For example:

```
# Python example
import numpy as np

rng = np.random.default_rng(123)
x = rng.normal(loc=0, scale=1, size=100)
x.mean()
```

2.3.2 R code

Occasionally we will show R snippets for comparison. These will be clearly labeled and follow the usual R console style:

```
# R example
set.seed(123)
x <- rnorm(100, mean = 0, sd = 1)
mean(x)
```

When you see both versions together, the idea is:

- **Same statistical idea, different syntax.**
- Focus on the model and reasoning, not just the language.

2.4 Mathematical notation

As in the original text, we occasionally use symbols like p to denote the number of β parameters in a linear model. You do not need to memorize every symbol immediately; the important point is to connect:

- the **model equation**,
- the **R implementation**, and
- the **Python implementation**.

2.5 Where to report issues or suggest improvements

Just like the original book, this project is a work in progress. You may encounter:

- typos or unclear explanations,
- small discrepancies between R and Python output,
- places where the documentation could use another example.

If you do, we would love to hear from you.

- Use **GitHub issues** on the PyStatsV1 repository: <https://github.com/pystatsv1/PyStatsV1/issues>
- For general questions or teaching stories, use **GitHub Discussions**: <https://github.com/pystatsv1/PyStatsV1/discussions>

Helpful ways to contribute:

- Suggest rewording a confusing paragraph.
- Point out where the Python code could better match the R text.
- Propose a new small example or diagnostic plot.
- Submit a pull request if you are comfortable with Git and GitHub.

2.6 Acknowledgements and license

This guide owes a major intellectual debt to:

- **David Dalpiaz**, author of *Applied Statistics with R*.
- The STAT 420 teaching team and contributors acknowledged in the original text.

The original book is available at:

- Repository: <https://github.com/daviddalpiaz/appliedstats>
- PDF: http://daviddalpiaz.github.io/appliedstats/applied_statistics.pdf

Our adaptation for PyStatsV1 follows the [Creative Commons Attribution–NonCommercial–ShareAlike 4.0 International License](#) of the original work. In particular:

- You are free to share and adapt this material for **non-commercial** purposes.
- You must provide appropriate attribution to the original author.
- Derivative works must use the **same license**.

PyStatsV1 extends this by adding:

- Python implementations of textbook-style analyses,
- command-line workflows and CI tests,
- and documentation tailored to a Python + R learning environment.

2.7 Your contributions

If you contribute a substantial improvement to this documentation or the associated code and would like to be acknowledged, feel free to open an issue or pull request and indicate how you would like your name to appear. We are happy to recognize contributors and, if desired, link to a GitHub or personal website.

If you care about **open, reproducible statistics education** and want to learn *why* methods work by seeing them in both R and Python, you're exactly the audience this guide was written for.

CHAPTER 2 – GETTING STARTED WITH R (FOR PYTHON-FIRST LEARNERS)

This chapter is adapted from Chapter 2 (*Introduction to R*) of *Applied Statistics with R* by David Dalpiaz, reworked for a Python-first audience and the PyStatsV1 project. The original text is available at

- Repository: <https://github.com/daviddalpiaz/appliedstats>
- PDF: http://daviddalpiaz.github.io/appliedstats/applied_statistics.pdf

Our goal here is to give you **enough R** to:

- understand R-based textbook examples, and
- compare them with the Python scripts in PyStatsV1.

You *do not* need to become an R expert to use this project, but seeing the same ideas in both languages is extremely valuable.

3.1 Why learn a bit of R if you like Python?

Many applied statistics courses and textbooks are written with R code. If you only know Python, it can feel like you are always “translating in your head”.

PyStatsV1 is designed to reduce that friction:

- the *ideas* come from an R-first textbook,
- the *code you actually run* is Python,
- and R appears as a reference point.

A small investment in R gives you:

- direct access to the original examples,
- a second way to check your understanding,
- and a broader mental model of how statistical software works.

3.2 Installing R and RStudio / Posit

R is both:

- a **language**, and
- a **software environment** for statistical computing.

To follow along with the original book or to compare outputs, you will usually install two pieces of software:

1. **R** from the Comprehensive R Archive Network (CRAN).
2. **RStudio Desktop** (now maintained by Posit), an IDE that provides a console, editor, plots pane, and more.

In very rough steps:

- visit CRAN,
- download the latest R version for your operating system,
- then install RStudio Desktop and let it find your R installation.

(Exact installation instructions change over time; the official websites will have the current details.)

Once installed, you can:

- run R directly from a terminal by typing `R`, or
- open RStudio and work in its console and editor.

In this guide we will focus on the *language*, not the IDE, but RStudio is an excellent environment for learning.

3.3 Using R as a calculator (and comparing with Python)

Both R and Python can be used as fancy calculators. To warm up, we will mirror a few basic operations. The point is not to memorize every line, but to see how close the two languages really are.

3.3.1 Basic arithmetic

Addition, subtraction, multiplication, and division:

- **Python:**

```
3 + 2
3 - 2
3 * 2
3 / 2    # 1.5
```

- **R:**

```
3 + 2
3 - 2
3 * 2
3 / 2    # 1.5
```

3.3.2 Exponentiation and roots

- **Python** (`**` for powers):

```
3 ** 2          # 9
2 ** -3         # 0.125
100 ** 0.5      # 10
import math
math.sqrt(100)  # 10
```

- **R** (`^` for powers):

```
3 ^ 2      # 9
2 ^ (-3)   # 0.125
100 ^ 0.5  # 10
sqrt(100)  # 10
```

3.3.3 Constants and logarithms

R and Python both provide common mathematical constants and log functions.

- **Python** (using `math`):

```
import math

math.pi      # 3.14159...
math.e        # 2.71828...

math.log(math.e)      # natural log
math.log10(1000)      # base 10
math.log2(8)          # base 2
```

- **R**:

```
pi            # 3.14159...
exp(1)        # 2.71828...

log(exp(1))   # natural log
log10(1000)   # base 10
log2(8)       # base 2
log(16, base = 4) # log base 4
```

3.3.4 Trigonometry

Angles are in radians in both languages.

- **Python**:

```
import math

math.sin(math.pi / 2) # 1.0
math.cos(0)           # 1.0
```

- **R**:

```
sin(pi / 2) # 1
cos(0)      # 1
```

The takeaway: if you can read basic Python math, R's syntax is only a small step away.

3.4 Getting help in R

In the examples above we used functions such as `sqrt()`, `exp()`, `log()` and `sin()`. R has built-in documentation for these.

To open the help page for a function in R, type a **question mark** in front of its name at the console:

```
?log  
?sin  
?paste  
?lm
```

RStudio displays the documentation in the *Help* pane. This help system is extremely rich and is often the best first place to look when you are unsure what a function does.

When that is not enough, general advice from the original text applies equally to R and Python:

- Start by searching the exact error message.
- If you ask another human for help, include: - what you expected the code to do, - what actually happened (including the full error), - and enough code to reproduce the problem.

This mindset carries over directly to PyStatsV1. When you open a GitHub issue, the same principles make it much easier for maintainers and instructors to help you.

3.5 Installing and loading packages in R

Base R ships with many functions and datasets, but one of its biggest strengths is the **package system**. Packages provide:

- new functions,
- new data sets,
- or sometimes entire modeling frameworks.

There are two steps to using a package:

1. **Install** it (once per machine), and
2. **Load** it in each R session where you need it.

3.5.1 Installation

Use `install.packages()` to download and install from CRAN:

```
install.packages("ggplot2")
```

Think of this as adding a new cookbook to your shelf.

3.5.2 Loading

Each time you start R, call `library()` for any packages you want to use in that session:

```
library(ggplot2)
```

This is like taking the cookbook off the shelf and opening it. When you exit R, the package is no longer loaded, but it remains installed.

3.5.3 Python analogy

If you are used to Python:

- `install.packages("ggplot2")` is similar to `pip install matplotlib seaborn`.
- `library(ggplot2)` is similar to:


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

You install once per environment, but you **import** / **load** in every session.

3.6 Style guides and cheatsheets

The original R text recommends using a **style guide** and checking out the RStudio / Posit “cheatsheets” for base R. The exact style you follow is less important than being **consistent**.

For R, two well-known guides are:

- Hadley Wickham’s style guide from *Advanced R*.
- Google’s R style guide.

PyStatsV1 does not enforce a particular R style, but we do:

- encourage consistency within an analysis,
- follow PEP 8–style conventions on the Python side,
- and use tools like **ruff** and **pytest** to keep scripts clean.

In your own work, it is worth choosing a small set of rules for:

- how you name variables (snake_case vs CamelCase),
- where you put spaces around operators,
- and how you format function calls.

3.7 Suggested exercises

To solidify the ideas from this chapter, try the following:

1. **Mirror basic calculations.** Open a Python REPL and an R console side by side. Recreate the examples above and invent a few of your own. Verify that both languages give the same results.
2. **Explore documentation.** In R, run `?log` and `?lm` and skim the help pages. In Python, use `help(math.log)` or check the NumPy / SciPy docs for similar functions. Notice the similarities in structure.
3. **Install and load a package.** In R, install and load `ggplot2`. In Python, install and import a plotting library you like (Matplotlib, Seaborn, Plotnine, ...) and create a tiny plot in each language.
4. **Practice asking for help.** Take an error message from either language (you can even create one on purpose), and write a short, clear question that you would be comfortable posting on a help forum or sending to an instructor.

3.8 How this chapter connects to PyStatsV1

In later chapters of PyStatsV1, you will see:

- R-based textbook examples,
- matching Python scripts in this repository,
- and, where helpful, R snippets for comparison.

This chapter is your “minimal R toolkit” for understanding those connections. Whenever you see an R command that you do not recognize, you can:

- revisit this chapter,

- consult the original *Applied Statistics with R* text, or
- open a GitHub Discussion to ask a question.

3.9 License and attribution

This chapter is a derivative work based on content from *Applied Statistics with R* by David Dalpiaz, used under the terms of the Creative Commons Attribution–NonCommercial–ShareAlike 4.0 International License. You can view the full license at:

- <http://creativecommons.org/licenses/by-nc-sa/4.0/>

We thank David Dalpiaz and contributors to the original text. PyStatsV1 extends their work by providing Python implementations, command-line workflows, and teaching-focused documentation for a Python + R environment.

APPLIED STATISTICS WITH PYTHON – CHAPTER 3

4.1 Data and Programming (Python-first view)

This chapter is the Python companion to the “Data and Programming” chapter from the R notes. The *statistical* ideas are the same:

- You need a small set of **data types** (numbers, text, booleans).
- You store them in **data structures** (vectors/arrays, matrices, tables).
- You use **programming tools** (control flow + functions) to glue analyses together.

The R book uses R’s vocabulary (vectors, matrices, lists, data frames). Here we’ll use the Python stack that maps to the same concepts:

- Core Python (built-in types and control flow)
- NumPy (for arrays, vectorization, and linear algebra)
- pandas (for tabular data like R data frames / tibbles)

The goal is not to turn you into a software engineer. The goal is:

Think “what is the data?” and “what operation am I doing?” and then choose the Python object that matches that mental model.

4.2 3.1 Data Types

R has numeric, integer, complex, logical, character. Python has very similar building blocks:

- `int` – integers: 1, 42, -3
- `float` – real numbers (double precision): 1.0, 3.14, -0.001
- `complex` – complex numbers: 4+2j
- `bool` – logical values: True or False
- `str` – text: "a", "Statistics", "1 plus 2"

A few quick parallels:

- R’s TRUE / FALSE Python’s True / False
- R’s NA Python’s None (missing in general) or `numpy.nan` (missing numeric)
- R’s automatic coercion (e.g., mixing numbers and strings in a vector) in Python, lists *can* hold mixed types, but numerical containers like NumPy arrays and pandas columns are usually homogeneous.

4.3 3.2 Data Structures: R vs Python mental map

R distinguishes between “homogeneous” (everything the same type) and “heterogeneous” (mixed types). Same idea in Python, just with different names.

Dimension	Homogeneous (R)	Homogeneous (Python)
1D	vector	NumPy ndarray (1D), pandas Series
2D	matrix	NumPy 2D ndarray, pandas DataFrame
3D+	array	higher-dim NumPy ndarray

Dimension	Heterogeneous (R)	Heterogeneous (Python)
1D	list	Python list, dict, dataclass
2D	data frame	pandas DataFrame

We’ll mostly use:

- **Python lists** for small, generic sequences.
- **NumPy arrays** when we mean “numeric vector/matrix.”
- **pandas DataFrames** when we mean “rectangular data with named columns.”

4.4 3.2.1 One-dimensional containers: lists, ranges, and NumPy arrays

4.4.1 Python list: flexible sequence

This is the closest analogue to an R “generic” vector (but can hold mixed types):

```
x = [1, 3, 5, 7, 8, 9]
x[0]      # 1 (0-based indexing in Python)
x[2]      # 5
x[-1]     # 9 (last element)
```

Remember: **Python indexes from 0**, not 1. That’s one of the biggest mental differences from R.

4.4.2 Creating sequences

R uses `c()`, `:` and `seq()`. Python equivalents:

```
# Explicit list
x = [1, 3, 5, 7, 8, 9]

# A sequence of integers (like 1:100 in R)
y = list(range(1, 101)) # 1, 2, ..., 100

# A sequence with a step (like seq(1.5, 4.2, by = 0.1))
import numpy as np

seq = np.arange(1.5, 4.3, 0.1) # up to (but not including) 4.3
```

4.4.3 Repetition

R has `rep()`. In Python:

```
["A"] * 10          # ['A', 'A', ..., 'A']
x * 3               # repeats the list x three times

# with NumPy for numeric work:
x_arr = np.array(x)
rep_arr = np.tile(x_arr, 3) # repeat the vector x three times
```

4.4.4 Vector length

R: `length(x)`

Python:

```
len(x)           # length of a list
len(x_arr)       # length of a NumPy array
```

4.5 3.2.1.1 Subsetting and slicing

R uses `x[1]`, `x[1:3]`, negative indices to drop elements, and logical vectors. Python has similar ideas but with different syntax.

4.5.1 Indexing by position

```
x = [1, 3, 5, 7, 8, 9]

x[0]      # 1 (first element)
x[2]      # 5 (third element)
x[1:4]    # [3, 5, 7] (slice: start inclusive, stop exclusive)
x[:3]     # [1, 3, 5]
x[3:]     # [7, 8, 9]
x[-1]     # 9 (last)
x[-2:]    # [8, 9] (last two)
```

NumPy arrays support exactly the same slice notation:

```
x_arr = np.array(x)
x_arr[0]      # 1
x_arr[1:4]    # array([3, 5, 7])
```

4.5.2 Boolean indexing (logical subsetting)

This is where NumPy and pandas line up very nicely with R.

R:

```
x[x > 3]
x[x != 3]
```

NumPy:

```
mask = x_arr > 3          # array([False, False, True, True, True, True])
x_arr[mask]              # array([5, 7, 8, 9])

x_arr[x_arr != 3]        # array([1, 5, 7, 8, 9])
```

4.6 3.2.2 Vectorization in Python

The R chapter emphasises that R is “vectorized”: operations apply to whole vectors at once. Same idea in the scientific Python stack:

- Pure Python lists: arithmetic is **not** vectorized.
- NumPy arrays and pandas objects: arithmetic **is** vectorized.

Compare:

```
x_list = [1, 2, 3, 4, 5]

# NOT vectorized - this concatenates lists
x_list + [1]          # [1, 2, 3, 4, 5, 1]

# Vectorized: use NumPy arrays
x = np.array([1, 2, 3, 4, 5])

x + 1                  # array([2, 3, 4, 5, 6])
2 * x                  # array([ 2,  4,  6,  8, 10])
2 ** x                 # powers, elementwise
np.sqrt(x)
np.log(x)
```

Same mental model as in R:

“If I apply a numeric function to a whole vector, I get a vector back.”

4.6.1 Length recycling vs broadcasting

In R, $x + y$ can silently **recycle** the shorter vector and even warn if lengths don’t match nicely.

In NumPy:

- Shapes must be **compatible** for broadcasting.
- Shape mismatch gives an error instead of a warning (which is usually safer).

Example:

```
x = np.array([1, 3, 5, 7, 8, 9])
y = np.arange(1, 61)

x + y          # works: NumPy broadcasts x along y's length (6 divides 60)

# If shapes truly don't match, you'll get a ValueError instead of a “silent” recycle.
```

4.7 3.2.3 Logical operators

R operators: <, >, <=, >=, ==, !=, !, &, |.

Python has very similar operators:

```
x = np.array([1, 3, 5, 7, 8, 9])

x > 3      # array([False, False,  True,  True,  True,  True])
x < 3      # array([ True, False, False, False, False, False])
x == 3     # array([False,  True, False, False, False, False])
x != 3     # array([ True, False,  True,  True,  True,  True])
```

A few important notes:

- For **NumPy arrays**, use & and | for elementwise AND/OR, with parentheses:

```
(x > 3) & (x < 8)    # both conditions
(x == 3) | (x == 9)  # either condition
```

- For pure Python booleans (not arrays), use and / or:

```
(3 < 4) and (42 > 13)
```

4.7.1 Counting and coercion

R shows that logical values act like 0/1 in numeric calculations (`sum(x > 3)`). Same in Python/NumPy:

```
mask = x > 3
mask      # array([False, False,  True,  True,  True,  True])

mask.sum()    # 4 (True acts like 1, False like 0)
np.sum(mask)  # also 4

mask.astype(int) # array([0, 0, 1, 1, 1, 1])
```

4.8 3.2.4 Matrices and linear algebra (NumPy)

R uses `matrix()`, `%%%`, `t()`, `solve()`, `diag()` and friends. In Python, these live in NumPy:

4.8.1 Creating matrices

```
x = np.arange(1, 10)          # 1..9
X = x.reshape(3, 3, order="F") # like R's column-major matrix()
X

# array([[1, 4, 7],
#        [2, 5, 8],
#        [3, 6, 9]])

Y = x.reshape(3, 3, order="C") # row-wise (byrow = TRUE in R)
Y

Z = np.zeros((2, 4))          # 2x4 matrix of zeros
```

4.8.2 Subsetting

```
X[0, 1]      # element in first row, second column (4)
X[0, :]      # first row
X[:, 1]      # second column
X[1, [0, 2]] # row 2, columns 1 and 3
```

4.8.3 Matrix operations

Elementwise operations:

```
X + Y
X - Y
X * Y      # elementwise product
X / Y      # elementwise division
```

Matrix multiplication and linear algebra:

```
# matrix multiplication (like R's %*%)
X @ Y
np.matmul(X, Y)

# transpose
X_T = X.T

# identity and diagonal matrices
np.eye(3)      # 3x3 identity
np.diag([1, 2, 3]) # diagonal with 1,2,3 on the diagonal

# inverse (if invertible)
Z = np.array([[9, 2, -3],
              [2, 4, -2],
              [-3, -2, 16]])

Z_inv = np.linalg.inv(Z)

Z_inv @ Z
# approximately the identity matrix
```

4.8.4 Floating point equality

R uses `all.equal` to compare floating-point matrices. NumPy equivalent:

```
np.allclose(Z_inv @ Z, np.eye(3)) # True
(Z_inv @ Z == np.eye(3)).all()    # often False due to tiny round-off
```

4.8.5 Dot product and outer product

R uses `a_vec %*% b_vec` and `a_vec %o% b_vec`; also `crossprod`.

Python:

```
a_vec = np.array([1, 2, 3])
b_vec = np.array([2, 2, 2])
```

(continues on next page)

(continued from previous page)

```

# Inner (dot) product
a_vec @ b_vec          # 12
np.dot(a_vec, b_vec)    # 12

# Outer product
np.outer(a_vec, b_vec)

# "crossprod(X, Y)" ( $X^T Y$ ) in NumPy:
C_mat = np.array([[1, 2, 3],
                  [4, 5, 6]])
D_mat = np.array([[2, 2, 2],
                  [2, 2, 2]])

C_mat.T @ D_mat        # like crossprod(C_mat, D_mat)
np.allclose(C_mat.T @ D_mat, C_mat.T.dot(D_mat))

```

4.9 3.2.5 Heterogeneous containers: lists and dicts

The R chapter introduces **lists** as “one-dimensional containers that can hold anything”: vectors, matrices, functions, etc.

In Python we have:

- **list** – ordered sequence (can be mixed types)
- **dict** – mapping from names to values (key–value store)

An R list like:

```

ex_list = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 42) { print("Hello World!") },
  e = diag(5)
)

```

could be represented roughly as:

```

def say_hello(arg=42):
    print("Hello World!")

ex_dict = {
    "a": np.array([1, 2, 3, 4]),
    "b": True,
    "c": "Hello!",
    "d": say_hello,
    "e": np.diag(np.arange(1, 6))
}

```

Accessing elements:

```
ex_dict["e"]      # matrix
ex_dict["a"]      # array
ex_dict["d"](arg=1)
```

4.10 3.2.6 Tabular data: pandas DataFrames

R's data frame / tibble Python's pandas DataFrame.

Minimal example:

```
import pandas as pd

example_data = pd.DataFrame({
    "x": [1, 3, 5, 7, 9, 1, 3, 5, 7, 9],
    "y": ["Hello"] * 9 + ["Goodbye"],
    "z": [True, False] * 5
})

example_data
example_data.head()      # first rows
example_data.info()      # structure, types
example_data.shape       # (n_rows, n_cols)
example_data.columns     # column names
```

Reading from CSV (similar to `read_csv` in R):

```
cars = pd.read_csv("data/example-data.csv")

# glimpse the data
cars.head(10)
cars.info()
```

4.10.1 Subsetting rows and columns

Like R:

```
# single column as a Series
example_data["x"]

# multiple columns as a DataFrame
example_data[["x", "y"]]

# Boolean filter: "fuel efficient cars"
mask = example_data["x"] > 5
example_data[mask]

# Equivalent to subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"
# →)):
mpg = cars # imagine we loaded the mpg data
mpg[mpg["hwy"] > 35][["manufacturer", "model", "year"]]
```

You can also use query for more R-like syntax:

```
mpg.query("hwy > 35")[["manufacturer", "model", "year"]]
```

4.11 3.3 Programming Basics in Python

Now we connect data structures with basic programming tools: control flow and functions.

4.12 3.3.1 Control flow

4.12.1 If / elif / else

R:

```
if (x > y) {
  # ...
} else {
  # ...
}
```

Python:

```
x = 1
y = 3

if x > y:
    z = x * y
    print("x is larger than y")
else:
    z = x + 5 * y
    print("x is less than or equal to y")
```

There is also a short expression form (similar spirit to ifelse for scalars):

```
result = 1 if 4 > 3 else 0      # 1
```

4.12.2 Vectorized “if” with NumPy/pandas

R’s `ifelse(condition, value_if_true, value_if_false)` is used for vectors.

In Python we use `np.where` or pandas methods:

```
fib = np.array([1, 1, 2, 3, 5, 8, 13, 21])
np.where(fib > 6, "Foo", "Bar")
# array(['Bar', 'Bar', 'Bar', 'Bar', 'Bar', 'Foo', 'Foo', 'Foo'], dtype='<U3')
```

For pandas Series:

```
mpg["label"] = np.where(mpg["hwy"] > 35, "Efficient", "Regular")
```

4.12.3 For loops vs vectorization

The R chapter shows that explicit loops are often replaced by vectorized code.

Same in Python:

```
# Loop version
x = [11, 12, 13, 14, 15]
for i in range(len(x)):
    x[i] = x[i] * 2

# Vectorized version with NumPy
x_arr = np.array([11, 12, 13, 14, 15])
x_arr = x_arr * 2
```

4.13 3.3.2 Defining functions

4.13.1 Basic structure

R:

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

Python:

```
import numpy as np

def standardize(x: np.ndarray) -> np.ndarray:
    """
    Standardize a numeric vector/array:
    subtract the mean and divide by the sample standard deviation.
    """
    m = x.mean()
    s = x.std(ddof=1) # ddof=1 for sample SD (like R's sd)
    return (x - m) / s
```

Test it:

```
sample = np.random.normal(loc=2, scale=5, size=10)
z = standardize(sample)

z.mean() # close to 0
z.std(ddof=1) # close to 1
```

4.13.2 Default arguments

R:

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

Python:

```
def power_of_num(num, power=2):
    return num ** power

power_of_num(10)           # 100
power_of_num(num=10, power=2) # 100
power_of_num(power=3, num=2) # 8
```

4.13.3 Variance example (biased vs unbiased)

The R notes define two forms of variance: unbiased (divide by $n-1$) and biased (divide by n).

We can mirror this:

```
def sample_variance(x: np.ndarray, biased: bool = False) -> float:
    """
    Compute the variance of x.

    biased = False -> divide by (n-1) (unbiased, like R's var)
    biased = True  -> divide by n      (ML / population variance)
    """
    x = np.asarray(x)
    n = x.size
    ddof = 0 if biased else 1
    return x.var(ddof=ddof)

sample = np.random.normal(size=10)
sample_variance(sample)           # unbiased (n-1)
sample_variance(sample, True)     # biased (n)
```

4.14 3.4 What you should take away

By the end of this chapter (R + Python versions), you should be comfortable with:

- Distinguishing **data types** (int, float, bool, str, complex).
- Choosing an appropriate **data structure**:
 - list vs NumPy array vs pandas DataFrame
 - when you want homogeneity (numeric computation) vs heterogeneity.
- Using **vectorized operations** instead of unnecessary loops:
 - arithmetic on whole arrays
 - logical masks and boolean indexing
 - basic linear algebra with NumPy.
- Writing **small helper functions** with clear arguments and defaults to standardize repeated analysis steps.

In later PyStatsV1 chapters, you'll see these ideas used to:

- build reusable simulation functions,
- manipulate data for case studies,
- and express models in a compact, vectorized way.

If any of the Python code in this chapter feels new, it's worth experimenting interactively in a notebook or Python shell:

- create a small vector or DataFrame,
- try out indexing and filtering,
- write a tiny function and call it on real data.

That practice will pay off quickly in the applied chapters.

APPLIED STATISTICS WITH PYTHON – CHAPTER 4

5.1 Summarizing data

This chapter parallels the “Summarizing Data” chapter from the R notes. The statistical ideas are the same:

- For **numeric variables**, we summarize the distribution using measures of **center** and **spread**.
- For **categorical variables**, we summarize using **counts** and **proportions**.
- We then use **plots** to visualize those summaries.

In the R version you see functions like `mean()`, `median()`, `sd()`, `IQR()`, `hist()`, `boxplot()`, and `plot()` for scatterplots. Here we’ll use Python, NumPy, pandas, and Matplotlib to achieve the same goals.

Throughout, imagine we have a DataFrame `mpg` that mirrors the R `ggplot2::mpg` dataset, with columns like:

- `cty` – city miles per gallon
- `hwy` – highway miles per gallon
- `drv` – drivetrain (“f”, “r”, “4”)
- `displ` – engine displacement in liters

You could obtain this DataFrame in several ways, for example:

```
import pandas as pd
import seaborn as sns # only needed if you want to load the example

# Option 1: seaborn's built-in mpg dataset
mpg = sns.load_dataset("mpg")

# Option 2: read from a CSV bundled with your project
# mpg = pd.read_csv("data/mpg.csv")
```

5.2 4.1 Summary statistics

We start with **summary statistics**: numbers that describe center, spread, and distribution shape for a variable.

5.2.1 4.1.1 Numeric variables: center and spread

In R you saw a table of summaries like:

- mean
- median

- variance
- standard deviation
- interquartile range (IQR)
- minimum, maximum, range

We can compute the same quantities with pandas:

```
import numpy as np
import pandas as pd

# City miles per gallon
cty = mpg["cty"]

# Center
cty_mean = cty.mean()      # average
cty_median = cty.median()  # median

# Spread
cty_var = cty.var(ddof=1)  # sample variance (n-1)
cty_sd = cty.std(ddof=1)   # sample standard deviation (n-1)
cty_iqr = cty.quantile(0.75) - cty.quantile(0.25)

cty_min = cty.min()
cty_max = cty.max()
cty_range = cty_max - cty_min

summary = {
    "mean": cty_mean,
    "median": cty_median,
    "variance": cty_var,
    "sd": cty_sd,
    "IQR": cty_iqr,
    "min": cty_min,
    "max": cty_max,
    "range": cty_range,
}

summary
```

A quick shortcut for many of these is `describe`:

```
mpg["cty"].describe()
```

which returns count, mean, standard deviation, quartiles, min, and max.

Conceptual recap

- Mean: arithmetic average; sensitive to outliers.
- Median: middle value; robust to outliers.
- Variance/SD: average squared (or square-rooted) distance from the mean.
- IQR: distance between the 25th and 75th percentiles (middle 50% of data).
- Min/Max/Range: show the extremes of the distribution.

Python vs R differences:

- R's `var()` and `sd()` use $n-1$ by default (unbiased estimators).
- pandas uses `ddof=1` for `DataFrame.var` and `DataFrame.std` by default.
- NumPy's `np.var` and `np.std` default to `ddof=0` (divide by n). Use `ddof=1` to match the R textbook.

5.2.2 4.1.2 Categorical variables: counts and proportions

For categorical variables, we care about **how often** each level appears.

In R, you saw `table(mpg$drv)` and relative frequencies with `table(mpg$drv) / nrow(mpg)`.

In pandas:

```
drv_counts = mpg["drv"].value_counts()
drv_props = mpg["drv"].value_counts(normalize=True)

drv_counts
drv_props
```

This gives frequency and proportion for each drivetrain category.

Key ideas:

- `value_counts()` is the pandas analogue of `table()` in R.
- `normalize=True` turns counts into proportions.
- These summaries are the numerical counterpart of a bar chart.

5.3 4.2 Plotting

Numeric tables are useful, but most of the time we learn more from good visualization.

We'll mirror the same four plot types as the R chapter:

- Histograms
- Bar charts
- Boxplots
- Scatterplots

We will use Matplotlib and pandas plotting helpers. These examples assume:

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

5.3.1 4.2.1 Histograms

When you have **one numeric variable**, a histogram is the workhorse plot.

In R: `hist(mpg$cty)` and a more polished version with axis labels, title, breaks, colors.

In Python/pandas:

```
fig, ax = plt.subplots()

mpg["cty"].hist(
```

(continues on next page)

(continued from previous page)

```

    bins=12,                # similar idea to breaks =
    color="dodgerblue",
    edgecolor="darkorange",
    ax=ax,
)

ax.set_xlabel("Miles per gallon (city)")
ax.set_ylabel("Frequency")
ax.set_title("Histogram of MPG (city)")

plt.tight_layout()

```

Notes:

- bins is analogous to R's breaks argument.
- Always label axes and add a clear title.
- hist gives the familiar histogram shape: bars whose area corresponds to counts (or densities).

5.3.2 4.2.2 Bar charts

Bar charts summarize **categorical** variables (or numeric variables with a small number of distinct values).

R example: `barplot(table(mpg$drv))`.

Python:

```

drv_counts = mpg["drv"].value_counts().sort_index()

fig, ax = plt.subplots()

drv_counts.plot(
    kind="bar",
    color="dodgerblue",
    edgecolor="darkorange",
    ax=ax,
)

ax.set_xlabel("Drivetrain (f = FWD, r = RWD, 4 = 4WD)")
ax.set_ylabel("Frequency")
ax.set_title("Drivetrains")

plt.tight_layout()

```

If you want **proportions** instead of counts, apply `value_counts(normalize=True)`:

```

drv_props = mpg["drv"].value_counts(normalize=True).sort_index()
drv_props.plot(kind="bar", ax=ax) # same idea; y-axis now in [0, 1]

```

5.3.3 4.2.3 Boxplots

Boxplots are ideal when you want to summarize the distribution of a numeric variable, especially **across groups** defined by a categorical variable.

Single boxplot

R: `boxplot(mpg$hwy)`

Python/pandas:

```
fig, ax = plt.subplots()

mpg["hwy"].plot(kind="box", vert=True, ax=ax)

ax.set_ylabel("Miles per gallon (highway)")
ax.set_title("Highway MPG - overall distribution")

plt.tight_layout()
```

Grouped boxplots

R syntax: `boxplot(hwy ~ drv, data = mpg)` – highway MPG by drivetrain.

In pandas, we group then call `boxplot`:

```
fig, ax = plt.subplots()

mpg.boxplot(
    column="hwy",
    by="drv",
    ax=ax,
    grid=False,
)

ax.set_xlabel("Drivetrain (f = FWD, r = RWD, 4 = 4WD)")
ax.set_ylabel("Miles per gallon (highway)")
ax.set_title("MPG (highway) vs drivetrain")
# pandas adds its own super-title; remove if you like:
fig.suptitle("")

plt.tight_layout()
```

Interpretation reminders:

- The box shows the interquartile range (IQR): 25th to 75th percentile.
- The line inside the box is the median.
- Whiskers extend to typical minimum/maximum values.
- Points beyond the whiskers are potential outliers.

In Chapter 4 of the R notes, there is also emphasis on the formula syntax `y ~ x`. The conceptual equivalent here is:

- “Take numeric column `hwy`”
- “Group by drivetrain `drv`”
- “Draw separate boxplots for each group”

5.3.4 4.2.4 Scatterplots

Scatterplots show the relationship between **two numeric variables**.

The R chapter uses

```
plot(hwy ~ displ, data = mpg)
```

We can mirror this with pandas:

```
fig, ax = plt.subplots()

ax.scatter(
    mpg["displ"],
    mpg["hwy"],
    s=30,
    color="dodgerblue",
)

ax.set_xlabel("Engine displacement (liters)")
ax.set_ylabel("Miles per gallon (highway)")
ax.set_title("MPG (highway) vs engine displacement")

plt.tight_layout()
```

Typical interpretation for the mpg data:

- As engine displacement increases, highway MPG tends to decrease.
- The scatterplot shows not only the trend but also variability and potential clusters (e.g., different vehicle types).

A tiny bit of code to add a fitted line (optional, for later chapters):

```
import numpy as np

X = mpg["displ"].to_numpy()
y = mpg["hwy"].to_numpy()

# simple least-squares line via NumPy polyfit
m, b = np.polyfit(X, y, deg=1)

x_grid = np.linspace(X.min(), X.max(), 100)
y_hat = m * x_grid + b

fig, ax = plt.subplots()
ax.scatter(X, y, s=30, color="dodgerblue", alpha=0.7)
ax.plot(x_grid, y_hat, color="darkorange", linewidth=2)

ax.set_xlabel("Engine displacement (liters)")
ax.set_ylabel("Miles per gallon (highway)")
ax.set_title("MPG (highway) vs engine displacement with fitted line")

plt.tight_layout()
```

You do *not* need to understand regression yet; here the line is just a visual summary of the overall trend. Later chapters will unpack the model behind it.

5.4 4.3 What you should take away

By the end of this chapter (R + Python versions), you should be comfortable with:

- Computing basic **summary statistics** for numeric data:
 - mean, median, variance, sd, IQR, min, max, range.
- Computing **frequency tables** and **proportions** for categorical variables using `value_counts` (and `normalize=True` for proportions).
- Matching each summary to an appropriate **plot type**:
 - histogram for one numeric variable,
 - bar chart for one categorical variable,
 - boxplot for numeric vs categorical,
 - scatterplot for two numeric variables.
- Translating R's functions and syntax to Python/pandas/NumPy:
 - mean `Series.mean()`,
 - sd `Series.std(ddof=1)`,
 - IQR quantiles / `Series.quantile`,
 - table `value_counts`,
 - hist / barplot / boxplot / plot Matplotlib / pandas plotting.

Most importantly:

You can now look at a variable, decide whether it is numeric or categorical, and quickly choose a summary and a plot that make sense.

These skills will be used constantly in later PyStatsV1 chapters—before we fit any models, we will always:

1. **Summarize the data numerically** (center, spread, and counts), and
2. **Visualize** the data with one or more of the plots from this chapter.

APPLIED STATISTICS WITH PYTHON – CHAPTER 5

6.1 Probability and statistics in Python and R

This chapter mirrors the “**Probability and Statistics in R**” chapter from the R notes. We’ll keep the *statistical* ideas the same, but express them in Python-first terms.

By the end of this chapter you should be comfortable with:

- using library functions to work with common distributions,
- translating between **pdf / cdf / quantile / random draws**,
- performing basic **t-tests** in Python (one-sample and two-sample),
- and using **simulation** to understand probability results.

Where the R notes show functions like `dnorm()`, `qbinom()`, or `t.test()`, we’ll highlight the Python equivalents from:

- `scipy.stats`
- `numpy.random`
- and a few plain-Python helper functions.

5.1 PROBABILITY IN PYTHON (AND R)

7.1 5.1.1 Distribution helpers: pdf, cdf, quantiles, random draws

In the R book you see the pattern:

- `dname` – density or pmf (pdf),
- `pname` – cumulative distribution function (cdf),
- `qname` – quantile function,
- `rname` – random draws.

For example, `dnorm()`, `pnorm()`, `qnorm()`, `rnorm()` for the Normal distribution, or `dbinom()`, `pbinom()` and so on.

In Python, the roles are similar but the interface is a little different.

The most convenient toolkit is `scipy.stats`. Each distribution is an object with methods and “frozen” versions.

For a Normal distribution with mean 2 and standard deviation 5:

```
import numpy as np
from scipy import stats

dist = stats.norm(loc=2, scale=5)

# pdf (density) at x = 3
pdf_at_3 = dist.pdf(3)

# cdf at x = 3, P(X <= 3)
cdf_at_3 = dist.cdf(3)

# 97.5th percentile (quantile)
q_975 = dist.ppf(0.975)

# random sample of size n = 10
sample = dist.rvs(size=10, random_state=42)
```

The mapping to the R notation is:

- R `dnorm(x, mean, sd)` → Python `stats.norm(mean, sd).pdf(x)`
- R `pnorm(q, mean, sd)` → Python `stats.norm(mean, sd).cdf(q)`
- R `qnorm(p, mean, sd)` → Python `stats.norm(mean, sd).ppf(p)`
- R `rnorm(n, mean, sd)` → Python `stats.norm(mean, sd).rvs(size=n)`

7.2 5.1.2 Other families: binomial, t, Poisson, chi-square, F

The R chapter lists a family of distributions with the same d/p/q/r pattern. In Python you will typically reach for:

- `scipy.stats.binom` for binomial,
- `scipy.stats.t` for Student's t,
- `scipy.stats.poisson` for Poisson,
- `scipy.stats.chi2` for chi-square,
- `scipy.stats.f` for F.

Example – Binomial probability

Compute $P(Y = 6)$ when $Y \sim \text{Binomial}(n = 10, p = 0.75)$:

```
from scipy import stats

# pmf (probability mass function) for a Binomial
prob_y_eq_6 = stats.binom(n=10, p=0.75).pmf(6)
```

In R the same calculation would use `dbinom(x = 6, size = 10, prob = 0.75)`.

Notice again:

- R's "d" for discrete distributions is really the **pmf**,
- Python exposes this as **pmf** for discrete and **pdf** for continuous.

You can explore other distributions in `scipy.stats` in exactly the same way: create the frozen distribution object, then call **pdf**, **cdf**, **ppf**, and **rvs**.

5.2 HYPOTHESIS TESTS IN PYTHON

The R notes briefly review hypothesis testing and then demonstrate t-tests. We'll match that structure here, but show both:

- the **formula** (to demystify the test statistic), and
- the **convenience function** from `scipy.stats`.

8.1 5.2.1 One-sample t-test

Set-up: we observe x_1, \dots, x_n and want to test

$$H_0 : \mu = \mu_0 \quad \text{vs} \quad H_1 : \mu \neq \mu_0,$$

assuming the data are approximately Normal with unknown variance.

The test statistic is

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \sim t_{n-1} \quad \text{under } H_0,$$

where \bar{x} is the sample mean and s the sample standard deviation.

Python implementation “by hand”

We revisit the cereal-box example from the R notes: boxes labelled “16 oz”, sample of 9 observed weights. (Here we just hard-code the data.)

```
import numpy as np
from scipy import stats

weights = np.array([15.5, 16.2, 16.1, 15.8, 15.6, 16.0, 15.8, 15.9, 16.2])
mu_0 = 16

n = len(weights)
x_bar = weights.mean()
s = weights.std(ddof=1)

t_stat = (x_bar - mu_0) / (s / np.sqrt(n))
df = n - 1

# two-sided p-value
p_value_two_sided = 2 * stats.t.sf(np.abs(t_stat), df=df)
```

For a **one-sided** alternative (for example $H_1 : \mu < \mu_0$):

```
p_value_less = stats.t.cdf(t_stat, df=df)      # P(T <= observed)
p_value_greater = stats.t.sf(t_stat, df=df)    # P(T >= observed)
```

This mirrors what the R code is doing with `t.test()` and `pt()`.

Using SciPy's convenience function

```
# two-sided one-sample t-test against mu_0
t_stat2, p_value2 = stats.ttest_1samp(weights, popmean=mu_0)

# SciPy always reports a two-sided p-value; you can halve it
# for a one-sided test if the sign of t matches your alternative.
p_value_less = p_value2 / 2 if t_stat2 < 0 else 1 - p_value2 / 2
```

The important idea: whether you calculate t by hand or use a helper function, the **model assumptions** and **test structure** are the same as in the R notes.

8.2 5.2.2 Two-sample t-test

Now suppose we have two independent samples:

$$X_1, \dots, X_n \sim N(\mu_x, \sigma^2), \quad Y_1, \dots, Y_m \sim N(\mu_y, \sigma^2),$$

and we want to test

$$H_0 : \mu_x - \mu_y = 0 \quad \text{vs} \quad H_1 : \mu_x - \mu_y > 0,$$

under an equal-variance assumption.

The pooled-variance test statistic is

$$t = \frac{\bar{x} - \bar{y}}{s_p \sqrt{1/n + 1/m}}, \quad s_p^2 = \frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}.$$

Python implementation “by hand”

```
import numpy as np
from scipy import stats

x = np.array([70, 82, 78, 74, 94, 82])
y = np.array([64, 72, 60, 76, 72, 80, 84, 68])

n, m = len(x), len(y)
x_bar, y_bar = x.mean(), y.mean()
s_x = x.std(ddof=1)
s_y = y.std(ddof=1)

s_p = np.sqrt(((n - 1) * s_x**2 + (m - 1) * s_y**2) / (n + m - 2))
t_stat = (x_bar - y_bar) / (s_p * np.sqrt(1 / n + 1 / m))
df = n + m - 2

# one-sided alternative H1: mu_x > mu_y
p_value = stats.t.sf(t_stat, df=df)
```

Using SciPy's two-sample test

```
t_stat2, p_value_two_sided = stats.ttest_ind(  
    x, y, equal_var=True  
)  
  
# convert to one-sided p-value if desired  
p_value_greater = p_value_two_sided / 2 if t_stat2 > 0 else 1 - p_value_two_sided / 2
```

SciPy's `ttest_ind()` mirrors R's `t.test(x, y, var.equal = TRUE)` call.

For practical work you will usually:

1. check assumptions (rough normality, similar spreads),
2. call the SciPy function,
3. interpret t , degrees of freedom, p-value, and confidence interval.

5.3 SIMULATION IN PYTHON

The R chapter ends with simulations that illustrate how probability results behave in practice. We'll take the same examples and rewrite them in NumPy.

The pattern you'll see throughout PyStatsV1 is:

- write small simulation functions,
- loop or vectorize to build many replicates,
- examine histograms and summary statistics.

9.1 5.3.1 Paired differences

We consider two Normal populations with means $\mu_1 = 6$, $\mu_2 = 5$, common variance $\sigma^2 = 4$, and sample size $n = 25$ in each group.

From theory, if \bar{X}_1 and \bar{X}_2 are the sample means and $D = \bar{X}_1 - \bar{X}_2$, then

$$D \sim N(\mu_1 - \mu_2, \sigma^2/n + \sigma^2/n) = N(1, 0.32).$$

We can compute $P(0 < D < 2)$ analytically using the Normal cdf:

```
from scipy import stats
import numpy as np

mu_D = 1.0
var_D = 0.32
sd_D = np.sqrt(var_D)

prob_0_to_2 = stats.norm(mu_D, sd_D).cdf(2) - stats.norm(mu_D, sd_D).cdf(0)
```

Alternatively we can *simulate* many realizations of D and approximate this probability empirically.

```
rng = np.random.default_rng(seed=42)

n = 25
num_samples = 10_000

x1 = rng.normal(loc=6, scale=2, size=(num_samples, n))
x2 = rng.normal(loc=5, scale=2, size=(num_samples, n))

diffs = x1.mean(axis=1) - x2.mean(axis=1)
```

(continues on next page)

(continued from previous page)

```
prob_sim = np.mean((diffs > 0) & (diffs < 2))

diffs_mean = diffs.mean()
diffs_var = diffs.var(ddof=1)
```

Here:

- prob_sim should be close to prob_0_to_2,
- diffs_mean should be near 1,
- diffs_var should be near 0.32.

In later PyStatsV1 chapters we will use this pattern repeatedly to check the behaviour of estimators and test statistics.

9.2 5.3.2 Distribution of a sample mean

The second example in the R notes studies the distribution of the sample mean from a Poisson distribution and connects it to the Central Limit Theorem.

Let $X \sim \text{Poisson}(\mu)$ with $\mu = 10$. We draw samples of size $n = 50$ and compute the sample mean \bar{X} . The CLT suggests

$$\bar{X} \approx N\left(\mu, \frac{\mu}{n}\right)$$

for moderately large n .

In Python:

```
from scipy import stats
import numpy as np

rng = np.random.default_rng(seed=1337)

mu = 10
sample_size = 50
num_samples = 100_000

# each row: one sample of size n
samples = rng.poisson(lam=mu, size=(num_samples, sample_size))
xBars = samples.mean(axis=1)

# empirical mean and variance
mean_emp = xBars.mean()
var_emp = xBars.var(ddof=1)

mean_theory = mu
var_theory = mu / sample_size
```

You can then:

- plot a histogram of xBars,
- overlay a Normal density with mean mean_theory and variance var_theory,
- and check proportions within 1, 2, or 3 standard deviations of the mean.

(We keep plotting code to a minimum here; later chapters will show more matplotlib examples.)

5.4 WHAT YOU SHOULD TAKE AWAY

By the end of this chapter (R + Python versions), you should be comfortable with:

- **Using distribution helpers** to get pdf/pmf, cdf, quantiles, and random samples from common distributions in Python and R.
- **Translating between R and Python:** d^* / p^* / q^* / r^* functions in R correspond to pdf/pmf, cdf, ppf, and rvs in `scipy.stats`.
- **Implementing t-tests** both “by hand” from the formulas and using convenience functions like `scipy.stats.ttest_1samp()` and `scipy.stats.ttest_ind()`.
- **Using simulation** to: - approximate probabilities, - understand sampling distributions, - and verify theoretical results.

If any of the code in this chapter feels new, try it interactively:

- change parameters (means, variances, sample sizes),
- re-run the simulations,
- see how the distribution of statistics (like \bar{X} or the t-statistic) responds.

That experimentation will pay off quickly in the PyStatsV1 applied chapters, where we’ll connect these probability tools directly to real case studies.

APPLIED STATISTICS WITH PYTHON – CHAPTER 6

11.1 Resources for Python, R, and reproducible workflows

The earlier chapters moved quickly through Python, R, and basic statistical ideas. This chapter is not required for the rest of PyStatsV1, but it collects resources you can use to go deeper:

- tutorials for learning Python and R,
- books that go beyond this mini-textbook,
- cross-language “cheatsheets”,
- and tooling for reproducible work (RStudio, Jupyter, Quarto, etc.).

Use this as a menu: you do **not** need to read everything here. Pick one or two resources that match your current level and goals.

6.1 BEGINNER TUTORIALS AND REFERENCES

If you're just getting comfortable with code, you want **short, hands-on** introductions where you can type and run examples immediately.

12.1 Python-focused

- **Official Python Tutorial**

The tutorial in the Python documentation walks through basic syntax, control flow, functions, and modules.

<https://docs.python.org/3/tutorial/>

- **Scientific Python “quick start” guides**

Short introductions to NumPy, pandas, Matplotlib, and the scientific Python ecosystem. Good for bridging from “I know basic Python” to “I can do data analysis.”

<https://numpy.org/doc/stable/user/quickstart.html> https://pandas.pydata.org/docs/getting_started/index.html
<https://matplotlib.org/stable/tutorials/index.html>

- **JupyterLab / notebooks**

Interactive environment for mixing code, narrative, and plots. Very helpful for experimentation and teaching.

<https://jupyter.org/>

12.2 R-focused

These mirror the resources listed in the original R chapter, but with brief comments about how they complement PyStatsV1.

- **Try R (interactive tutorial)**

Short, browser-based introduction to R syntax and objects. Nice if you want to see R once without installing anything.

- **Quick-R (Kabacoff)**

Web-based reference for common R tasks: importing data, basic plots, regression, etc. Handy if you already know the statistics and just want to remember “how do I do this in R?”.

- **R Tutorial (Chi Yau)**

A mix of tutorial and reference that covers core language features plus common data analysis tasks.

- **R Programming for Data Science (Roger Peng)**

Free online book that builds R from the ground up, with an emphasis on good programming habits.

<https://bookdown.org/rdpeng/rprogdatascience/>

6.2 INTERMEDIATE REFERENCES

Once you're comfortable running code and reading error messages, the next step is to learn **data analysis workflows** and **programming patterns**.

13.1 Python-focused

- **Python for Data Analysis (Wes McKinney)**

The pandas “founder’s book”. Great for learning how to manipulate data frames, handle time series, and write reusable analysis code.

- **Think Stats / Think Bayes (Allen Downey)**

Statistics books that use Python from the start, with an emphasis on simulation and computation rather than hand algebra.

- **Statistical Thinking in Python (various tutorials)**

Many online courses and notebooks walk through hypothesis testing, regression, and visualization in Python. PyStatsV1 chapters can be a complementary “worked examples” resource.

13.2 R-focused

- **R for Data Science (Wickham & Grolemund)**

A modern introduction to data wrangling, visualization, and modeling in the **tidyverse** ecosystem. Pairs nicely with PyStatsV1 if you want to see the same ideas in both R and Python.

<https://r4ds.had.co.nz/>

- **The Art of R Programming (Norman Matloff)**

Gentle but thorough introduction to R as a programming language (control flow, functions, object types), as opposed to just a statistics tool.

6.3 ADVANCED REFERENCES

These are for when R or Python has become part of your regular toolkit and you want to think about performance, internals, or large projects.

14.1 R-focused

- **Advanced R (Hadley Wickham)**

Deep dive into R's object system, environments, functional programming, and metaprogramming. Helps explain *why* some R code behaves in surprising ways.

- **The R Inferno (Patrick Burns)**

A humorous but very technical guide to R's "gotchas". Useful if you write a lot of complex R or maintain other people's R code.

- **Efficient R Programming (Gillespie & Lovelace)**

Focuses on writing R code that is fast and scalable, and on using R tools efficiently day to day.

14.2 Python-focused

- **Scientific Python ecosystem docs**

NumPy, SciPy, pandas, and Matplotlib all have detailed documentation that covers vectorization, broadcasting, and performance tips.

- **Probabilistic programming / Bayesian methods**

Libraries like PyMC, Stan (via CmdStanPy), or NumPyro provide powerful tools for Bayesian modeling. These are beyond the scope of PyStatsV1, but worth exploring if you continue into advanced applied statistics.

6.4 CROSS-LANGUAGE COMPARISONS

If you already know another language, sometimes the fastest way to learn is via a “**Rosetta stone**” that shows equivalent idioms side by side.

Examples mentioned in the original R chapter include:

- **Numerical computing comparison**

Cheat sheets comparing MATLAB, NumPy, Julia, and R for common numerical and matrix operations.

- **R vs Stata vs SAS**

Short documents that show how the same data analysis is written in each language.

For Python + R specifically, useful patterns to practice are:

- indexing and slicing in NumPy vs R,
- data frame operations in pandas vs `dplyr`,
- plotting with Matplotlib/Seaborn vs `ggplot2`.

In PyStatsV1 we deliberately write code in a way that makes these parallels easier to see: plain, explicit scripts in both languages rather than opaque one-liners.

6.5 IDES, NOTEBOOKS, AND LITERATE PROGRAMMING

A big part of modern applied statistics is organising code, text, and plots in a single **reproducible document**.

16.1 R side

- **RStudio**

Widely used IDE for R (and now other languages). Integrates console, editor, plots, and package management in one window.

- **RMarkdown**

Framework for combining narrative text, R code, and output into a single report (HTML, PDF, slides, etc.). The original notes for this book are written with RMarkdown.

16.2 Python side

- **Jupyter notebooks / JupyterLab**

Interactive notebooks where you can mix Python code, Markdown, LaTeX equations, and plots. Excellent for exploratory analysis and teaching.

- **VS Code, PyCharm, and other editors**

For larger projects, an editor or IDE with good Python support (linting, debugging, Git integration) can make a big difference.

16.3 Bridging tools

- **Quarto**

A modern, language-agnostic framework for literate programming that supports both Python and R in the same ecosystem. If you like RMarkdown + RStudio, Quarto + VS Code/Jupyter gives a similar experience for Python.

No single tool is “the right one”. The best setup is whatever makes it easy for you to:

1. write clear code,
2. keep data and outputs organised,
3. rerun everything later and get the same results.

6.6 HOW PYSTATSV1 FITS INTO THIS ECOSYSTEM

PyStatsV1 is **not** trying to replace full textbooks or advanced courses. Instead, it aims to be:

- a **bridge** between R-first teaching materials and Python,
- a repository of **worked examples** that you can run, edit, and extend,
- and a place where you can practice **reproducible workflows** without a heavy framework.

You might use this chapter as follows:

- If you're brand new to coding, pair the PyStatsV1 chapters with one of the beginner Python or R tutorials.
- If you already know R well, skim the Python resources to see how the same ideas look in NumPy/pandas.
- If you're comfortable in Python, use the R resources when you need to read or adapt R-heavy applied work.

Most importantly: **don't feel obligated to read everything**. Choose one or two resources that look approachable, and treat PyStatsV1 as your sandbox for trying ideas out in real code.

APPLIED STATISTICS WITH PYTHON – CHAPTER 7

18.1 Simple linear regression in Python and R

This chapter parallels the *Simple Linear Regression* chapter from the R notes. The statistical ideas are the same:

- We have a **numeric predictor** (x) and a **numeric response** (y).
- We want to describe how the *mean* of y changes as x changes.
- We quantify that relationship with a **line** plus **random noise**.

In the R notes, the running example is the classic `cars` dataset:

- `speed` – car speed in miles per hour,
- `dist` – stopping distance in feet.

In PyStatsV1 we reuse the same data, but we work in **Python-first** terms: NumPy, pandas, Matplotlib, and the `statsmodels` regression API.

By the end of this chapter you should be able to:

- write down the **simple linear regression model** and its assumptions,
- compute least-squares estimates by hand (with NumPy),
- fit the same model using a high-level tool (`statsmodels.ols`),
- interpret the slope, intercept, residuals, and R^2 ,
- use the fitted model to make **predictions** (and know when not to trust them),
- understand how the R function `lm()` corresponds to the Python tools we use.

Note

Throughout this chapter we assume you have the PyStatsV1 repository checked out locally and that you can run the chapter script

```
scripts/ch07_simple_linear_regression.py
```

which contains a full, executable version of the examples.

18.2 7.1 From scatterplots to models

We start with a scatterplot: *speed* vs. *stopping distance*.

In Python, with a DataFrame `cars` that has `speed` and `dist` columns:

```
import pandas as pd
import matplotlib.pyplot as plt

cars = pd.read_csv("data/cars.csv") # see PyStatsV1 data folder

fig, ax = plt.subplots()
ax.scatter(cars["speed"], cars["dist"], alpha=0.7)
ax.set_xlabel("Speed (mph)")
ax.set_ylabel("Stopping distance (ft)")
ax.set_title("Stopping distance vs. speed")
plt.show()
```

The plot tells us:

- Faster cars tend to have **longer** stopping distances.
- The points do not fall exactly on a line—there is **random variation**.

We can express this informally as

response = pattern + noise.

In math notation we write

$$Y = f(X) + \varepsilon,$$

where

- X – predictor (speed),
- Y – response (stopping distance),
- $f(\cdot)$ – unknown systematic pattern,
- ε – random error.

In this chapter we **restrict** $f(\cdot)$ to be a *line*.

18.3 7.2 The simple linear regression model

The **simple linear regression (SLR)** model uses a straight line to describe how the mean of Y changes with X :

$$Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad i = 1, \dots, n.$$

Here

- x_i – observed predictor value (fixed, not random),
- Y_i – random response,
- β_0 – intercept,
- β_1 – slope,
- ε_i – random error term.

We assume the errors satisfy the usual **LINE** conditions:

- **L – Linear**: the mean of Y is a straight line in x ,

$$\mathbb{E}[Y_i \mid X_i = x_i] = \beta_0 + \beta_1 x_i.$$

- **I – Independent:** errors ε_i are independent.
- **N – Normal:** errors are normally distributed,

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2).$$

- **E – Equal variance:** all errors share the same variance σ^2 .

Under these assumptions, the conditional distribution of Y_i is

$$Y_i \mid X_i = x_i \sim \mathcal{N}(\beta_0 + \beta_1 x_i, \sigma^2).$$

The three unknown parameters are β_0 , β_1 , and σ^2 . Our task is to estimate them from data.

18.4 7.3 Least squares: estimating the line

Given observed pairs (x_i, y_i) , the **least-squares** idea is:

Choose the line that makes the squared vertical errors as small as possible.

Vertical errors (residuals) are

$$e_i = y_i - (\beta_0 + \beta_1 x_i).$$

We want β_0 and β_1 that minimize

$$\text{SSE}(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_i))^2.$$

Solving the resulting equations gives the familiar closed forms

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}, \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x},$$

where \bar{x} and \bar{y} are the sample means.

18.4.1 7.3.1 Computing $\hat{\beta}_0$ and $\hat{\beta}_1$ in Python

Using NumPy with the cars dataset:

```
import numpy as np
import pandas as pd

cars = pd.read_csv("data/cars.csv")

x = cars["speed"].to_numpy()
y = cars["dist"].to_numpy()

x_bar = x.mean()
y_bar = y.mean()

Sxx = np.sum((x - x_bar) ** 2)
Sxy = np.sum((x - x_bar) * (y - y_bar))

beta1_hat = Sxy / Sxx
beta0_hat = y_bar - beta1_hat * x_bar

print(beta0_hat, beta1_hat)
```

Interpretation (for this dataset):

- $\hat{\beta}_1 \approx 3.93$ – for each +1 mph of speed, the **mean** stopping distance increases by about **3.9 feet**.
- $\hat{\beta}_0 \approx -17.6$ – the predicted mean distance at 0 mph (an extrapolation; not physically meaningful, but needed for the line).

18.4.2 7.3.2 Predictions and interpolation

Once we have $\hat{\beta}_0$ and $\hat{\beta}_1$, the fitted line is

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x.$$

Predictions in Python are straightforward:

```
def predict_stopping_distance(speed_mph: float) -> float:
    return beta0_hat + beta1_hat * speed_mph

predict_stopping_distance(8)    # within data range (interpolation)
predict_stopping_distance(21)   # within range but not observed
predict_stopping_distance(50)   # outside range (extrapolation - be careful!)
```

Key idea:

- **Interpolation** inside the observed **x** range is usually reasonable.
- **Extrapolation** far beyond the data range is risky, even with a good line.

18.5 7.4 Residuals, variance, and R^2

Once the line is fitted, we inspect how far each point is from that line.

18.5.1 Residuals

Residuals are observed minus fitted values:

$$e_i = y_i - \hat{y}_i = y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i).$$

In NumPy:

```
y_hat = beta0_hat + beta1_hat * x
residuals = y - y_hat

residuals[:5]
```

A good first diagnostic is a **residual vs. fitted** plot; you should see:

- no strong curve (supports linearity),
- roughly constant spread (supports equal variance),
- no obvious pattern or clustering (supports independence).

18.5.2 Residual variance and residual standard error

The sum of squared residuals is

$$\text{SSE} = \sum_{i=1}^n e_i^2.$$

We estimate σ^2 by

$$s_e^2 = \frac{\text{SSE}}{n - 2},$$

where $n - 2$ reflects the two parameters we estimated.

In Python:

```
n = len(y)
sse = np.sum(residuals ** 2)
s2_e = sse / (n - 2)
s_e = np.sqrt(s2_e) # residual standard error
```

The residual standard error s_e is in the same units as y (feet here). You can read it as:

“Our fitted mean stopping distances are typically off by about s_e feet.”

18.5.3 Decomposition of variation and R^2

We can decompose total variation in y into explained and unexplained parts:

$$\underbrace{\sum (y_i - \bar{y})^2}_{\text{SST}} = \underbrace{\sum (\hat{y}_i - \bar{y})^2}_{\text{SSReg}} + \underbrace{\sum (y_i - \hat{y}_i)^2}_{\text{SSE}}.$$

Then the **coefficient of determination** is

$$R^2 = \frac{\text{SSReg}}{\text{SST}} = 1 - \frac{\text{SSE}}{\text{SST}}.$$

Interpretation:

- R^2 is the **proportion of variation in y** explained by the regression on x .
- Values near 1 mean the line explains most of the variability; values near 0 mean the line explains little.

In Python:

```
sst = np.sum((y - y_bar) ** 2)
ss_reg = np.sum((y_hat - y_bar) ** 2)
r2 = ss_reg / sst # or 1 - sse / sst
```

For the cars example, R^2 is around 0.65 – about 65% of the variation in stopping distance is explained by speed alone.

18.6 7.5 Using statsmodels: Python’s version of `lm()`

In R, we would write

```
cars <- datasets::cars
fit <- lm(dist ~ speed, data = cars)

summary(fit)
predict(fit, newdata = data.frame(speed = 8))
```

In Python, the closest equivalent is statsmodels’ formula API:

```
import pandas as pd
import statsmodels.formula.api as smf

cars = pd.read_csv("data/cars.csv")

model = smf.ols("dist ~ speed", data=cars).fit()

print(model.params)      # beta_0_hat and beta_1_hat
print(model.rsquared)    # R^2
print(model.summary())   # detailed regression table

model.predict({"speed": [8, 21, 50]})
```

You should recognize many familiar quantities in the summary:

- coefficient estimates ($\hat{\beta}_0, \hat{\beta}_1$),
- their standard errors and t-statistics,
- R^2 and adjusted R^2 ,
- residual standard error (called scale or sigma).

18.6.1 Mapping R → Python

- `lm(dist ~ speed, data=cars) → smf.ols("dist ~ speed", data=cars).fit()`
- `coef(fit) → model.params`
- `resid(fit) → model.resid`
- `fitted(fit) → model.fittedvalues`
- `summary(fit) → model.summary()`
- `predict(fit, newdata=...) → model.predict(new_dataframe)`

18.7 7.6 Simulation: seeing SLR in action

Simulation is a powerful way to *see* what a model means.

Suppose the **true** relationship is

$$Y = 5 - 2x + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 3^2).$$

We can simulate a dataset, fit a line, and compare estimates to truth:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

rng = np.random.default_rng(seed=1)

n = 21
beta0_true = 5.0
beta1_true = -2.0
sigma_true = 3.0
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(0, 10, n)
eps = rng.normal(loc=0.0, scale=sigma_true, size=n)
y = beta0_true + beta1_true * x + eps

sim = pd.DataFrame({"x": x, "y": y})

fit = smf.ols("y ~ x", data=sim).fit()
print(fit.params)
```

If you plot the data and overlay both lines (true and fitted), you will see that the estimated line is close, but not perfect—that’s sampling variability.

This kind of simulation becomes even more useful in later chapters (e.g., to study confidence intervals, hypothesis tests, and power).

18.8 7.7 What you should take away

By the end of this chapter you should be comfortable with:

- thinking in terms of **models**:

$$Y = \beta_0 + \beta_1 X + \varepsilon,$$

and “response = prediction + error”,

- interpreting the **slope** and **intercept** in context,
- computing least-squares estimates $\hat{\beta}_0$ and $\hat{\beta}_1$,
- computing and interpreting **residuals**, **SSE**, **residual standard error**,
- interpreting R^2 as “fraction of variance explained,”
- using `statsmodels` in Python as the counterpart of R’s `lm()`:

```
model = smf.ols("dist ~ speed", data=cars).fit()
model.summary()
model.predict({"speed": [10, 20]})
```

In later PyStatsV1 chapters, we will:

- extend SLR to **multiple** regression,
- build models with **transformed** predictors (e.g., quadratic terms),
- and connect regression more formally to **inference** (tests and confidence intervals).

If any of the algebra feels fuzzy, focus first on the **pictures** and the Python code; the formulas will slowly become familiar as you keep applying them to real datasets.

APPLIED STATISTICS WITH PYTHON – CHAPTER 8

19.1 Inference for simple linear regression

In Chapter 7 you met the **simple linear regression (SLR)** model and learned how to fit a line to data:

$$Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \dots, n.$$

We focused on *estimating* β_0 and β_1 with least squares and interpreting the fitted line.

In this chapter we ask questions like:

- *How variable are $\hat{\beta}_0$ and $\hat{\beta}_1$ from sample to sample?*
- *How sure are we about the true slope?*
- *Is there statistically significant evidence that the slope is non-zero?*
- *How do we build confidence intervals and prediction intervals?*

The R notes use `lm()` and its printed output. Here we work mostly with Python's `statsmodels` and `scipy.stats`, but the statistical ideas are exactly the same.

Throughout, keep the cars example in mind:

- Y – stopping distance (feet),
- X – speed (mph).

We'll use the same fitted model from Chapter 7 to make the ideas concrete.

```
import pandas as pd
import statsmodels.formula.api as smf

cars = pd.read_csv("data/cars.csv")
model = smf.ols("dist ~ speed", data=cars).fit()

print(model.summary())
```

19.2 8.1 Recap: least squares and notation

From Chapter 7:

- The **least squares estimates** $\hat{\beta}_0$ and $\hat{\beta}_1$ are the intercept and slope that minimize the sum of squared residuals

$$\text{RSS}(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_i))^2.$$

- We can write them in terms of centered sums

$$S_{xx} = \sum_{i=1}^n (x_i - \bar{x})^2, \quad S_{xy} = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}).$$

Then

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}}, \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

- The fitted values and residuals are

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i, \quad e_i = y_i - \hat{y}_i.$$

- The residual standard error (RSE) estimates σ :

$$s_e = \sqrt{\frac{1}{n-2} \sum_{i=1}^n e_i^2}.$$

In Python, `model.params` stores $\hat{\beta}_0$ and $\hat{\beta}_1$, and `model.mse_resid` is s_e^2 .

```
beta0_hat, beta1_hat = model.params
se = model.mse_resid**0.5
n = model.nobs

print(beta0_hat, beta1_hat, se, n)
```

19.3 8.2 Gauss–Markov in plain language (why least squares is “good”)

The **Gauss–Markov theorem** is one of the main theoretical results behind ordinary least squares.

Under the SLR assumptions (in particular, linear mean and constant variance), the least squares estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ are:

- **Linear**: they are linear combinations of the responses Y_i .
- **Unbiased**: their expectations equal the true parameters,

$$\mathbb{E}[\hat{\beta}_0] = \beta_0, \quad \mathbb{E}[\hat{\beta}_1] = \beta_1.$$

- **Best** among all linear unbiased estimators: they have the **smallest variance** in that class.

This is often summarized as:

OLS estimates are BLUE – Best Linear Unbiased Estimators.

You do *not* need to prove Gauss–Markov to use regression, but you should remember the practical message:

If the linear model assumptions are reasonable, least squares is a very sensible default – you are not throwing away precision by using it.

19.4 8.3 Sampling distributions of $\hat{\beta}_0$ and $\hat{\beta}_1$

Because the errors ε_i are normal and independent, the least squares estimates themselves are random variables with normal distributions.

Their exact variances (in terms of the unknown σ^2) are:

$$\text{Var}(\hat{\beta}_1) = \frac{\sigma^2}{S_{xx}}, \quad \text{Var}(\hat{\beta}_0) = \sigma^2 \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}} \right).$$

So, under the model,

$$\hat{\beta}_1 \sim \mathcal{N}\left(\beta_1, \frac{\sigma^2}{S_{xx}}\right), \quad \hat{\beta}_0 \sim \mathcal{N}\left(\beta_0, \sigma^2 \left(\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}} \right)\right).$$

These distributions describe how the slope and intercept *would vary* if you could repeatedly collect new samples of size n from the same population.

19.4.1 A quick simulation check in Python

You can verify this by simulation, just as the R notes do.

```
import numpy as np

rng = np.random.default_rng(42)

n = 100
x = np.linspace(-1, 1, n)
Sxx = np.sum((x - x.mean())**2)

beta0_true = 3.0
beta1_true = 6.0
sigma_true = 2.0

num_samples = 10_000
beta0_hats = np.empty(num_samples)
beta1_hats = np.empty(num_samples)

for i in range(num_samples):
    eps = rng.normal(loc=0.0, scale=sigma_true, size=n)
    y = beta0_true + beta1_true * x + eps

    sim_model = smf.ols("y ~ x", data={"y": y, "x": x}).fit()
    beta0_hats[i] = sim_model.params["Intercept"]
    beta1_hats[i] = sim_model.params["x"]

print(beta1_hats.mean(), beta1_true)
print(beta1_hats.var(), sigma_true**2 / Sxx)
```

The empirical mean and variance of the simulated slopes should be very close to the theoretical values.

19.5 8.4 Standard errors and t statistics

In practice we do not know σ^2 , so we replace it with s_e^2 . This gives **standard errors** for the estimates:

$$\text{SE}(\hat{\beta}_1) = \frac{s_e}{\sqrt{S_{xx}}}, \quad \text{SE}(\hat{\beta}_0) = s_e \sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}.$$

If you standardize with these estimated standard deviations, you get t -distributed statistics:

$$T_{\beta_1} = \frac{\hat{\beta}_1 - \beta_1}{SE(\hat{\beta}_1)} \sim t_{n-2}, \quad T_{\beta_0} = \frac{\hat{\beta}_0 - \beta_0}{SE(\hat{\beta}_0)} \sim t_{n-2}.$$

This is why regression output uses the t distribution with $n - 2$ degrees of freedom for tests and intervals.

In Python, `model.bse` stores the standard errors:

```
se_beta0, se_beta1 = model.bse
print(se_beta0, se_beta1)
```

19.6 8.5 Confidence intervals for slope and intercept

The generic shape of a confidence interval is

$$\text{estimate} \pm (\text{critical value}) \times \text{standard error}.$$

For the regression coefficients we get

$$\hat{\beta}_1 \pm t_{\alpha/2, n-2} SE(\hat{\beta}_1), \quad \hat{\beta}_0 \pm t_{\alpha/2, n-2} SE(\hat{\beta}_0),$$

where $t_{\alpha/2, n-2}$ is a critical value from the t_{n-2} distribution.

In Python you can compute these either by hand or via helper methods:

```
from scipy import stats

alpha = 0.01 # 99% CI
df = int(model.df_resid)
tcrit = stats.t.ppf(1 - alpha/2, df=df)

ci_beta1 = (
    beta1_hat - tcrit * se_beta1,
    beta1_hat + tcrit * se_beta1,
)
print("99% CI for beta1:", ci_beta1)
```

Statsmodels also offers a convenience method:

```
print(model.conf_int(alpha=0.01))
```

19.6.1 Interpretation in the cars example

For the slope β_1 (change in mean stopping distance per 1 mph increase in speed), a 99% CI might look like

$$[2.82, 5.05] \text{ feet per mph.}$$

We read this as:

“We are 99% confident that each 1 mph increase in speed is associated with between about 2.8 and 5.0 additional feet of average stopping distance.”

Notice that this interval is **entirely above 0**, which is closely tied to the significance tests in the next section.

19.7 8.6 Hypothesis tests for slope and intercept

We often want to test hypotheses such as:

- Is the slope zero? (Is there any linear relationship?)
- Is the intercept equal to some value?

The generic t statistic has the form

$$t = \frac{\text{estimate} - \text{hypothesized value}}{\text{standard error}}.$$

19.7.1 Testing whether the slope is zero

The most common regression test is

$$H_0 : \beta_1 = 0 \quad \text{vs} \quad H_1 : \beta_1 \neq 0.$$

Under H_0 , the model reduces to $Y_i = \beta_0 + \varepsilon_i$; the response does *not* depend linearly on x .

The test statistic is

$$t = \frac{\hat{\beta}_1 - 0}{\text{SE}(\hat{\beta}_1)} \sim t_{n-2} \quad \text{under } H_0.$$

In Python:

```
t_beta1 = beta1_hat / se_beta1
p_beta1 = 2 * stats.t.sf(abs(t_beta1), df=df)
print(t_beta1, p_beta1)
```

The same values appear in `model.summary()` in the row for speed: `coef`, `std err`, `t`, and `P>|t|`.

If the p-value is very small (for the cars data it is tiny), we reject H_0 and conclude there is a statistically significant linear relationship between speed and stopping distance.

19.8 8.7 The cars example in Python

Here is a compact version of the “coefficients table” extraction that parallels the R code in the original notes:

```
coefs = model.summary2().tables[1]
print(coefs)

beta0_hat = coefs.loc["Intercept", "Coef."]
se_beta0 = coefs.loc["Intercept", "Std.Err."]
t_beta0 = coefs.loc["Intercept", "t"]
p_beta0 = coefs.loc["Intercept", "P>|t|"]

beta1_hat = coefs.loc["speed", "Coef."]
se_beta1 = coefs.loc["speed", "Std.Err."]
t_beta1 = coefs.loc["speed", "t"]
p_beta1 = coefs.loc["speed", "P>|t|"]

print(beta1_hat, se_beta1, t_beta1, p_beta1)
```

This mirrors the R output:

- Estimate \rightarrow Coef.
- Std. Error \rightarrow Std.Err.
- t value \rightarrow t
- $\Pr(>|t|) \rightarrow P>|t|$

19.9 8.8 Confidence intervals for mean response

Sometimes we want an interval for the **mean response** at a given predictor value x_0 :

$$\mu(x_0) = \mathbb{E}[Y \mid X = x_0] = \beta_0 + \beta_1 x_0.$$

Our point estimate is $\hat{y}(x_0) = \hat{\beta}_0 + \hat{\beta}_1 x_0$. Its variance (and therefore its standard error) accounts for uncertainty in the fitted line:

$$\text{SE}(\hat{y}(x_0)) = s_e \sqrt{\frac{1}{n} + \frac{(x_0 - \bar{x})^2}{S_{xx}}}.$$

A $(1 - \alpha) \times 100\%$ confidence interval for the mean response at x_0 is

$$\hat{y}(x_0) \pm t_{\alpha/2, n-2} \text{SE}(\hat{y}(x_0)).$$

In statsmodels you can obtain these with `get_prediction(..., obs=False)` or by specifying "confidence" in the original R workflow:

```
new_speeds = pd.DataFrame({"speed": [5, 21]})
mean_pred = model.get_prediction(new_speeds)
print(mean_pred.summary_frame(alpha=0.01)) # 99% CI
```

The output includes columns like `mean`, `mean_ci_lower`, and `mean_ci_upper`.

19.10 8.9 Prediction intervals for new observations

A **prediction interval** describes the likely range of a *new individual* observation Y_{new} at x_0 .

There are two sources of variability:

1. Uncertainty in the fitted line (same as the mean response case).
2. The random noise ε around the line.

This adds an extra σ^2 term inside the square root:

$$\text{SE}_{\text{pred}}(x_0) = s_e \sqrt{1 + \frac{1}{n} + \frac{(x_0 - \bar{x})^2}{S_{xx}}}.$$

The prediction interval has the same basic form but is *wider*:

$$\hat{y}(x_0) \pm t_{\alpha/2, n-2} \text{SE}_{\text{pred}}(x_0).$$

In statsmodels you can request prediction intervals via `get_prediction(...).summary_frame()` and use the `obs_ci_*` columns:

```

pred = model.get_prediction(new_speeds)
frame = pred.summary_frame(alpha=0.01)

print(frame[["mean", "mean_ci_lower", "mean_ci_upper",
             "obs_ci_lower", "obs_ci_upper"]])

```

Compare the **mean** intervals with the **observation** intervals: the latter are always wider.

19.11 8.10 Confidence and prediction bands

Instead of intervals at a single x_0 , we can compute intervals across a grid of x values to form **bands**.

```

import numpy as np
import matplotlib.pyplot as plt

speed_grid = np.linspace(cars["speed"].min(),
                        cars["speed"].max(), 200)
grid_df = pd.DataFrame({"speed": speed_grid})

pred_grid = model.get_prediction(grid_df).summary_frame(alpha=0.01)

mean_lwr = pred_grid["mean_ci_lower"]
mean_upr = pred_grid["mean_ci_upper"]
obs_lwr = pred_grid["obs_ci_lower"]
obs_upr = pred_grid["obs_ci_upper"]

plt.scatter(cars["speed"], cars["dist"], alpha=0.5, label="Data")
plt.plot(speed_grid, pred_grid["mean"], label="Fitted line")
plt.plot(speed_grid, mean_lwr, "--", label="99% mean CI")
plt.plot(speed_grid, mean_upr, "--")
plt.plot(speed_grid, obs_lwr, ":", label="99% prediction band")
plt.plot(speed_grid, obs_upr, ":")
plt.xlabel("Speed (mph)")
plt.ylabel("Stopping distance (ft)")
plt.legend()
plt.show()

```

Things to notice (mirroring the R version):

- The bands are narrowest near \bar{x} (the mean speed).
- Prediction bands are wider than confidence bands.
- Both bands flare out toward the extremes of the x range.

19.12 8.11 F-test and ANOVA: another view of “significance of regression”

In simple linear regression there are two mathematically equivalent ways to test the significance of the slope:

- A **t-test** on $\beta_1 = 0$.
- An **F-test** for the overall regression.

The F -test uses the decomposition of total variability:

$$\underbrace{\sum_{i=1}^n (y_i - \bar{y})^2}_{\text{SST}} = \underbrace{\sum_{i=1}^n (y_i - \hat{y}_i)^2}_{\text{SSE}} + \underbrace{\sum_{i=1}^n (\hat{y}_i - \bar{y})^2}_{\text{SSReg}}.$$

The **math: F statistic** is

$$F = \frac{\text{SSReg}/1}{\text{SSE}/(n-2)} \sim F_{1, n-2} \quad \text{under } H_0 : \beta_1 = 0.$$

In simple linear regression,

$$F = t^2,$$

where t is the t statistic for the slope test. So the p -values from the t -test and the F -test always agree.

In statsmodels you can see this in two ways:

1. In `model.summary()`, the bottom of the table shows the F -statistic and its p -value.
2. You can construct an ANOVA table:

```
import statsmodels.api as sm

anova_table = sm.stats.anova_lm(model, typ=1)
print(anova_table)
```

You will see rows for speed and Residual, with the familiar “sum of squares”, “mean square”, F and PR(>F) columns.

19.13 8.12 What you should take away

By the end of this chapter you should be comfortable with:

- How least squares estimates behave as random variables (their **sampling distributions**).
- How to compute **standard errors** for the slope and intercept.
- How to build **confidence intervals** for β_0 and β_1 .
- How to run and interpret:
 - t -tests for individual coefficients,
 - the “significance of regression” test ($H_0 : \beta_1 = 0$),
 - and the equivalent F -test / ANOVA view.
- The difference between:
 - **Confidence intervals for a mean response** at a given x ,
 - **Prediction intervals for a new observation** at that x .
- How to construct **bands** over a grid of x values to visualize uncertainty around the fitted line.

In later PyStatsV1 chapters, these tools will appear repeatedly:

- to compare different models on the same data,
- to quantify uncertainty in estimated effects,
- and to connect simulation results back to theoretical distributions.

If any of the formulas feel abstract, revisit the `cars` example in a Python shell:

- compute the statistics by hand from `model`'s attributes,
- verify that the printed summary matches your calculations,
- and try changing the model (e.g., adding a new predictor) to see how the tests and intervals change.

APPLIED STATISTICS WITH PYTHON – CHAPTER 9

20.1 Multiple linear regression

This chapter mirrors the “**Multiple Linear Regression**” chapter from the original R notes. The statistical ideas are the same, but we’ll express them in Python-first language and keep R alongside as a translation layer.

By the end of this chapter you should be comfortable with:

- Building and interpreting linear regression models with **more than one predictor**.
- Understanding the **matrix formulation** of regression.
- Using summary output to:
 - obtain **standard errors** and **t-tests** for each coefficient,
 - construct **confidence intervals** and **prediction intervals**,
 - run **global F-tests** for the significance of regression,
 - and compare **nested models** with ANOVA F-tests.
- Connecting these ideas back to simulation: checking a theoretical sampling distribution with code.

Throughout we will:

- Treat R’s `lm()` as the reference implementation, and
- Show Python equivalents using `pandas` and `statsmodels`.

20.2 9.1 From simple to multiple regression

In simple linear regression (SLR) we had one predictor:

$$Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad \varepsilon_i \sim N(0, \sigma^2).$$

Graphically, this is a **line** through a cloud of points.

In multiple linear regression (MLR) we allow several predictors:

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_{p-1} x_{i,p-1} + \varepsilon_i.$$

For two predictors you can picture a **plane** in 3-D space; for more predictors, geometry is harder to visualize but the algebra scales cleanly.

Key idea: each β_j is a **partial slope**:

- β_j measures the change in the mean response for a one-unit change in x_j , **holding all other predictors fixed**.

That “holding others fixed” interpretation is new in MLR and central to reading regression output correctly.

20.3 9.2 Auto MPG example

We'll again use a car-fuel-efficiency dataset. In the R notes it's read directly from the UCI Machine Learning Repository and cleaned into a data frame `autompg` with columns:

- `mpg` – miles per gallon (response),
- `cyl` – number of cylinders,
- `disp` – engine displacement,
- `hp` – horsepower,
- `wt` – weight,
- `acc` – acceleration,
- `year` – model year.

In PyStatsV1, you can either:

- read the data from a local CSV we include, or
- follow the R code closely by reading from the URL and cleaning in Python.

A minimal Python version might look like:

```
import pandas as pd

url = (
    "http://archive.ics.uci.edu/ml/machine-learning-databases/"
    "auto-mpg/auto-mpg.data"
)

cols = ["mpg", "cyl", "disp", "hp", "wt", "acc", "year", "origin", "name"]
autompg = pd.read_csv(
    url,
    delim_whitespace=True,
    names=cols,
    na_values="?",
    comment="#",
)

# Drop rows with missing horsepower and the Plymouth Reliant row,
# then drop unused columns to match the R notes
autompg = autompg.dropna(subset=["hp"])
autompg = autompg[autompg["name"] != "plymouth reliant"]
autompg = autompg[["mpg", "cyl", "disp", "hp", "wt", "acc", "year"]]
autompg["hp"] = autompg["hp"].astype(float)

autompg.info()
```

This gets us to the same cleaned structure as in R.

20.4 9.3 Fitting a multiple regression model

We'll start with a two-predictor model: `mpg` as a function of weight and year.

Model:

$$Y_i = \beta_0 + \beta_1 \text{wt}_i + \beta_2 \text{year}_i + \varepsilon_i.$$

20.4.1 R version

```
mpg_model <- lm(mpg ~ wt + year, data = autompg)
summary(mpg_model)
```

20.4.2 Python version (statsmodels)

We'll use the formula API, which understands "mpg ~ wt + year" like `lm()`.

```
import statsmodels.formula.api as smf

mpg_model = smf.ols("mpg ~ wt + year", data=autompg).fit()
print(mpg_model.summary())
```

You should see:

- estimates for Intercept, wt, and year,
- standard errors and t-values,
- residual standard error (`sigma`),
- R^2 and adjusted R^2 ,
- and an F-statistic testing whether the regression as a whole is useful.

20.4.3 Interpreting the coefficients

Using the R output notation, suppose we get:

- $\hat{\beta}_0 \approx -14.6$
- $\hat{\beta}_1 \approx -0.0066$ for weight
- $\hat{\beta}_2 \approx 0.76$ for year

Interpretations:

- **Intercept:** expected mpg for a car of weight 0 made in year 0. This is physically meaningless, but needed algebraically. Intercepts often have poor real-world meaning but are still part of the model.
- **Weight slope:** for a one-unit increase in weight (one pound in these data), expected mpg **decreases** by about 0.0066, **holding model year fixed**.
- **Year slope:** for a one-unit increase in model year (one year newer), expected mpg **increases** by about 0.76, **holding weight fixed**.

Notice how every slope interpretation now includes “for a fixed value of the other predictor”.

20.5 9.4 Matrix formulation of regression

Matrix notation lets us handle any number of predictors cleanly.

We stack the response values into a vector Y , the predictors into a design matrix X , the coefficients into a vector β , and the errors into a vector ε :

$$Y = X\beta + \varepsilon.$$

- Y is $n \times 1$,
- X is $n \times p$ (first column all ones for the intercept),
- β is $p \times 1$,
- ε is $n \times 1$.

The least squares estimate of β is:

$$\hat{\beta} = (X^\top X)^{-1} X^\top y.$$

The fitted values and residuals are:

$$\hat{y} = X\hat{\beta}, \quad e = y - \hat{y}.$$

The residual variance estimate (mean squared error) is:

$$s_e^2 = \frac{e^\top e}{n - p},$$

where n is the sample size and p is the number of β parameters (including the intercept). Compare this to SLR:

- SLR had $p = 2$ and denominator $n - 2$,
- here we allow general p , so the denominator is $n - p$.

20.5.1 Python check

We can verify that statsmodels is doing exactly this:

```
import numpy as np

y = autmpg["mpg"].to_numpy()
X = np.column_stack([
    np.ones(len(autmpg)), # intercept
    autmpg["wt"].to_numpy(),
    autmpg["year"].to_numpy(),
])

XtX_inv = np.linalg.inv(X.T @ X)
beta_hat = XtX_inv @ X.T @ y

print(beta_hat) # matches mpg_model.params

y_hat = X @ beta_hat
e = y - y_hat
n, p = X.shape
s_e = np.sqrt((e @ e) / (n - p))
print(s_e, mpg_model.mse_resid**0.5)
```

20.6 9.5 Sampling distribution of $\hat{\beta}$

Under the normal-error assumptions of the linear model:

- $\varepsilon \sim N(0, \sigma^2 I)$,

we can show (using multivariate normal theory) that:

$$\hat{\beta} \sim N(\beta, \sigma^2(X^\top X)^{-1}).$$

Consequences:

- $E[\hat{\beta}] = \beta$ – each coefficient estimate is **unbiased**.
- The covariance matrix of $\hat{\beta}$ is $\sigma^2(X^\top X)^{-1}$. The diagonal elements give the variances of individual coefficients.

If we call $C = (X^\top X)^{-1}$, then

$$\text{Var}(\hat{\beta}_j) = \sigma^2 C_{jj}, \quad SE(\hat{\beta}_j) = s_e \sqrt{C_{jj}}.$$

Each coefficient has a t-distribution after standardization:

$$\frac{\hat{\beta}_j - \beta_j}{s_e \sqrt{C_{jj}}} \sim t_{n-p}.$$

In practice we rarely compute C by hand; we read standard errors from software.

- R: `summary(mpg_model)$coef`
- Python: `mpg_model.params` and `mpg_model.bse`

20.7 9.6 Testing individual coefficients

To test whether a specific predictor is useful in the presence of the others, we typically test

$$H_0 : \beta_j = 0 \quad \text{vs} \quad H_1 : \beta_j \neq 0.$$

Test statistic:

$$t = \frac{\hat{\beta}_j - 0}{SE(\hat{\beta}_j)} = \frac{\hat{\beta}_j}{s_e \sqrt{C_{jj}}} \sim t_{n-p} \quad \text{under } H_0.$$

20.7.1 R version

```
summary(mpg_model)$coef
```

The Estimate, Std. Error, t value, and Pr(>|t|) columns give exactly this test.

20.7.2 Python version

```
mpg_model.params      # estimates
mpg_model.bse         # standard errors
mpg_model.tvalues     # t statistics
mpg_model.pvalues     # two-sided p-values
```

If the p-value for `wt` is tiny, we reject $H_0 : \beta_{wt} = 0$ and conclude weight is a useful predictor, *given that year is already in the model*.

This “given that ...” interpretation is crucial: the t-test evaluates a coefficient **conditional on the other predictors in the model**, not in isolation.

20.8 9.7 Confidence intervals for coefficients and mean response

20.8.1 Intervals for coefficients

From the sampling distribution we get a $(1 - \alpha)$ confidence interval:

$$\hat{\beta}_j \pm t_{\alpha/2, n-p} \cdot SE(\hat{\beta}_j).$$

R:

```
confint(mpg_model, level = 0.99)
```

Python:

```
mpg_model.conf_int(alpha=0.01)
```

20.8.2 Intervals for mean response

Suppose we want the mean mpg for cars with specified predictors $x_0 = (1, x_{01}, \dots, x_{0,p-1})^\top$. The fitted mean is

$$\hat{y}(x_0) = x_0^\top \hat{\beta}.$$

Its standard error is

$$SE(\hat{y}(x_0)) = s_e \sqrt{x_0^\top (X^\top X)^{-1} x_0}.$$

A $(1 - \alpha)$ confidence interval is

$$\hat{y}(x_0) \pm t_{\alpha/2, n-p} \cdot s_e \sqrt{x_0^\top (X^\top X)^{-1} x_0}.$$

R:

```
new_cars <- data.frame(wt = c(3500, 5000),
                      year = c(76, 81))

predict(mpg_model, newdata = new_cars,
        interval = "confidence", level = 0.99)
```

Python:

```
new_cars = pd.DataFrame({"wt": [3500, 5000],
                        "year": [76, 81]})

pred = mpg_model.get_prediction(new_cars)
pred.summary_frame(alpha=0.01)[["mean", "mean_ci_lower", "mean_ci_upper"]]
```

20.9 9.8 Prediction intervals

Prediction intervals account for both:

- uncertainty in the mean, and
- variability of individual observations around that mean.

Standard error:

$$SE_{\text{pred}}(\hat{y}(x_0)) = s_e \sqrt{1 + x_0^\top (X^\top X)^{-1} x_0}.$$

Prediction interval:

$$\hat{y}(x_0) \pm t_{\alpha/2, n-p} \cdot s_e \sqrt{1 + x_0^\top (X^\top X)^{-1} x_0}.$$

R:

```
predict(mpg_model, newdata = new_cars,
        interval = "prediction", level = 0.99)
```

Python:

```
pred.summary_frame(alpha=0.01)[
    ["obs_ci_lower", "obs_ci_upper"]
]
```

20.9.1 Warning about extrapolation

With multiple predictors, extrapolation can be subtle. Each new point must be reasonable **in the joint predictor space**, not just in each coordinate separately. A car with an unusual combination of weight and year may be far from the existing data cloud even if its weight and year separately look “within range”.

Plotting predictors against each other and marking new points is a good sanity check.

20.10 9.9 Significance of regression: global F-test

In SLR, the t-test for the slope and the F-test for the model were equivalent. In MLR they separate:

- Individual t-tests: “Is this particular predictor useful, given the others?”
- Global F-test: “Is **any** linear relationship present at all?”

Null hypothesis for the global F-test:

$$H_0 : \beta_1 = \beta_2 = \dots = \beta_{p-1} = 0.$$

Under H_0 the model reduces to

$$Y_i = \beta_0 + \varepsilon_i,$$

an intercept-only model. The F-statistic is based on the variance decomposition

$$SST = SS_{\text{Reg}} + SSE,$$

and compares mean squares:

$$F = \frac{SS_{\text{Reg}}/(p-1)}{SSE/(n-p)} \sim F_{p-1, n-p} \quad \text{under } H_0.$$

Software gives this automatically.

R:

```
summary(mpg_model)$fstatistic # F and df
# or
null_mpg_model <- lm(mpg ~ 1, data = autmpg)
anova(null_mpg_model, mpg_model)
```

Python:

```
mpg_model.fvalue, mpg_model.f_pvalue, mpg_model.df_model, mpg_model.df_resid
```

A tiny p-value says: “At least one predictor has a non-zero slope; the regression as a whole explains a meaningful amount of variation.”

20.11 9.10 Nested model comparisons

Often we want to compare two models where one is a **subset** of the other.

Example:

- **Reduced (null) model:** $\text{mpg} \sim \text{wt} + \text{year}$
- **Full model:** $\text{mpg} \sim \text{wt} + \text{year} + \text{cyl} + \text{disp} + \text{hp} + \text{acc}$

Here the reduced model is nested inside the full model. The null hypothesis is:

$$H_0 : \beta_{\text{cyl}} = \beta_{\text{disp}} = \beta_{\text{hp}} = \beta_{\text{acc}} = 0.$$

We use an F-test based on the extra sum of squares explained by the full model.

R:

```
null_mpg_model <- lm(mpg ~ wt + year, data = autmpg)
full_mpg_model <- lm(mpg ~ wt + year + cyl + disp + hp + acc,
                     data = autmpg)

anova(null_mpg_model, full_mpg_model)
```

Python (statsmodels):

```
import statsmodels.api as sm

null_mpg_model = smf.ols("mpg ~ wt + year", data=autmpg).fit()
full_mpg_model = smf.ols("mpg ~ wt + year + cyl + disp + hp + acc",
                        data=autmpg).fit()

sm.stats.anova_lm(null_mpg_model, full_mpg_model)
```

If the p-value is large, the extra predictors don’t improve the model significantly given *wt* and *year*.

The global significance-of-regression test from the previous section is a special case where the reduced model is intercept-only.

20.12 9.11 Simulation: checking the sampling distribution

The R notes close with a simulation study that:

- fixes a design matrix X ,

- simulates many response vectors from a known linear model,
- refits the regression each time,
- and looks at the empirical distribution of one coefficient.

You can implement the same idea in Python. Sketch:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm

rng = np.random.default_rng(1337)

n = 100
x1 = np.linspace(1, 10, n)
x2 = np.linspace(1, 10, n)[::-1]

beta_true = np.array([5.0, -2.0, 6.0])
sigma = 4.0

X = np.column_stack([np.ones(n), x1, x2])

num_sims = 10_000
beta2_vals = np.empty(num_sims)

for i in range(num_sims):
    eps = rng.normal(0, sigma, size=n)
    y = X @ beta_true + eps

    df = pd.DataFrame({"x1": x1, "x2": x2, "y": y})
    fit = smf.ols("y ~ x1 + x2", data=df).fit()
    beta2_vals[i] = fit.params["x2"]

beta2_vals.mean(), beta2_vals.var()
```

You should find:

- the mean of `beta2_vals` is close to the true β_2 ,
- the variance matches $\sigma^2 C_{22}$ where $C = (X^\top X)^{-1}$,
- a histogram of `beta2_vals` overlaid with the corresponding Normal density looks very similar.

This is a concrete demonstration of the theoretical sampling distribution results from Section 9.5.

20.13 9.12 What you should take away

By the end of this chapter (and its R + Python versions), you should be able to:

- Write down and interpret a **multiple linear regression model** with several predictors.
- Understand and use the **matrix formulation**:
 - $Y = X\beta + \varepsilon$,
 - $\hat{\beta} = (X^\top X)^{-1} X^\top y$,
 - residual variance $s_e^2 = e^\top e / (n - p)$.

- Read software output to obtain:
 - coefficient estimates, standard errors, t-tests, and p-values,
 - confidence intervals for coefficients and for mean responses,
 - prediction intervals for new observations,
 - global F-tests for significance of regression,
 - and F-tests comparing **nested models**.
- Recognize that:
 - coefficient interpretations are **conditional** (“holding others fixed”),
 - extrapolation in multiple dimensions can be subtle,
 - and simulation is a powerful way to check theoretical results.

In later PyStatsV1 chapters, these tools will underpin more advanced models (logistic regression, models with interactions, etc.) and many of the case studies.

APPLIED STATISTICS WITH PYTHON – CHAPTER 10

21.1 Model building: explanation and prediction

In earlier chapters we focused on **fitting a single model**:

- simple linear regression (Chapters 7–8),
- multiple linear regression (Chapter 9).

Here we step back and ask a bigger question:

How do we choose *which* model to use?

We will:

- separate the ideas of **family**, **form**, and **fit**,
- distinguish between models aimed at **explanation** vs **prediction**,
- see how **overfitting** and **train–test splits** enter the picture.

Throughout, you can imagine the familiar Auto MPG example:

- response y = miles per gallon (mpg),
- predictors x_1, x_2, \dots = car attributes (weight, horsepower, ...).

21.2 10.1 Family, form, and fit

When we say “build a model”, there are really *three* choices hiding inside:

1. **Family** – the broad class of models we are willing to consider.
2. **Form** – the specific predictors and transformations included.
3. **Fit** – the numerical values of the parameters, estimated from data.

We will mostly stay inside one family:

Family: linear models

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_{p-1} x_{p-1} + \varepsilon,$$

with ε capturing noise or unexplained variation.

Other families exist (trees, smoothers, neural nets), but linear models are:

- the **standard starting point**,
- easy to fit and interpret,

- an excellent gateway to more advanced methods.

21.2.1 10.1.1 Fit

Suppose we choose a simple form with one predictor:

$$y = \beta_0 + \beta_1 x_1 + \varepsilon.$$

To **fit** this model in Python we choose a loss function and minimize it. In this course we almost always use **least squares**:

$$\min_{\beta_0, \beta_1} \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_{1i}))^2.$$

In practice:

- with `statsmodels`, this is done by `smf.ols(...).fit()`;
- with `sklearn`, by `LinearRegression().fit(X, y)`.

The result is a **fitted model**:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1,$$

which we can use for **interpretation** or **prediction**.

21.2.2 10.1.2 Form

The **form** of a linear model is determined by:

- which predictors are included,
- which transformations and interactions we use.

Examples, using `mpg` as the response:

- Simple linear regression:

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \varepsilon.$$

- Multiple linear regression:

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \beta_2 \text{horsepower} + \beta_3 \text{year} + \varepsilon.$$

- Model with a transformation and an interaction:

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \beta_2 \text{weight}^2 + \beta_3 \text{year} + \beta_4 \text{weight} \times \text{year} + \varepsilon.$$

All of these are still **linear models**: linear in the parameters β_j . The form controls *flexibility*:

- more predictors and terms \rightarrow more flexibility,
- but also more risk of **overfitting** and harder interpretation.

21.2.3 10.1.3 Family

The **family** is the broad modeling approach. Some examples:

- linear regression,
- generalized linear models (logistic, Poisson, ...),
- non-parametric smoothers,
- trees and ensembles (random forests, boosting).

In this mini-textbook we focus on the **linear regression family** because:

- it is the standard tool for many applied problems,
- it has a rich theory of **inference** (standard errors, t-tests, F-tests),
- many ideas (design matrices, loss functions, regularization) carry directly into more advanced models.

You should keep a mental picture:

- **family** = which toolbox?
- **form** = which tools from that box?
- **fit** = how we use data to tune the tools (estimate parameters).

21.2.4 10.1.4 Assumed model vs fitted model

When we write a formula like

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \beta_2 \text{horsepower} + \varepsilon,$$

we are specifying the **assumed model**:

- linear family,
- particular form (which variables and interactions),
- often with additional assumptions about ε (e.g. Normal errors with constant variance).

After fitting, we obtain a **fitted model** such as:

$$\widehat{\text{mpg}} = 46.2 - 3.1 \text{weight} - 0.02 \text{horsepower}.$$

Important:

- Fitting only gives the **best model within the chosen form**.
- If the family or form is poorly chosen, even a perfectly fitted model can be misleading.

21.3 10.2 Explanation versus prediction

Why are we building a model?

- To **explain** how predictors relate to the response?
- Or to **predict** future responses as accurately as possible?

The distinction matters. The modeling steps can look similar, but:

- For **explanation**, we prioritize *interpretability* and valid inference.
- For **prediction**, we prioritize *accuracy on new data* and resistance to overfitting.

21.3.1 10.2.1 Explanation

For explanation we want models that are:

- **small** – using as few predictors as reasonably possible,
- **interpretable** – each coefficient has a clear story,
- **well-behaved** – assumptions are at least approximately satisfied.

In linear regression, we often:

- start from a **full model** with many predictors,
- use: * t-tests for individual coefficients, * F-tests / ANOVA for comparing nested models, * residual plots to check model assumptions,
- gradually simplify to a **parsimonious model** that still fits well.

Example goals for the Auto MPG data:

- quantify how **weight** and **year** relate to fuel efficiency,
- understand which car attributes matter *most*,
- communicate results to non-statisticians.

Here, even if a larger model slightly improves prediction, we may prefer a **simpler model** that tells a clearer story.

21.3.2 10.2.1.1 Correlation and causation

A crucial warning for explanatory models:

Correlation does not imply causation.

Linear models detect **associations** between variables. They do *not* prove that one variable *causes* another.

- Observational data (like Auto MPG) can show that higher horsepower is associated with lower fuel efficiency.
- But this does not prove that “increasing horsepower by 10 automatically reduces mpg by 3” in a causal sense.

To argue for causation we usually need:

- a carefully designed **experiment**,
- or strong subject-matter reasoning and supporting evidence.

In PyStatsV1, we will often treat our models as tools for **description** and **exploration**, with appropriate caution about causal claims.

21.3.3 10.2.2 Prediction

For prediction, the priority is different:

- We care about how well the model predicts **new, unseen data**.
- We are less concerned with: * whether each coefficient is statistically significant, * whether the model is easy to explain in words.

We need a **numerical measure of prediction error**. A common choice is root mean squared error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}.$$

In Python, given arrays `y_true` and `y_pred`:


```
import numpy as np

def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))
```

Lower RMSE means better predictive performance on the data we are evaluating.

21.3.4 10.2.2.1 Train–test split and overfitting

A key problem in predictive modeling is **overfitting**:

- A very flexible model can track the noise in the training data.
- It will have **low error on the data it saw**, but **high error on new data**.

To detect overfitting we mimic the “magic extra data” thought experiment by splitting our data:

- **training set** – used to fit the model,
- **test set** – held out and only used to evaluate predictions.

In code, using scikit-learn:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import numpy as np

X = mpg_df[["weight", "horsepower", "year"]].to_numpy()
y = mpg_df["mpg"].to_numpy()

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

model = LinearRegression().fit(X_train, y_train)

y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

train_rmse = rmse(y_train, y_train_pred)
test_rmse = rmse(y_test, y_test_pred)

print(train_rmse, test_rmse)
```

Typical pattern:

- As we add predictors and complexity: * **train RMSE** almost always decreases, * **test RMSE** may first decrease, then increase once we overfit.
- The best **predictive** model is often the one with the **lowest test RMSE**, even if it is not the largest or most complex.

21.4 10.3 What you should take away

By the end of this chapter (and its R + Python versions), you should be able to:

- distinguish clearly between: * **family** of models, * **form** of a model, * **fit** of a model;

- explain the difference between models aimed at: * **explanation** – small, interpretable, inference-friendly, * **prediction** – chosen to minimize error on new data;
- understand why: * linear models are often the **first choice**, * more complex models can **overfit**;
- compute and interpret: * **RMSE**, and * **train vs test** prediction error;
- describe why a train–test split is essential for honest assessment;
- articulate the warning: * “Correlation does not imply causation” in the context of regression.

21.5 10.4 How this connects to PyStatsV1

In PyStatsV1 you will see these ideas used repeatedly:

- **Explanatory models**
 - `statsmodels` regressions with detailed summaries,
 - ANOVA tables and F-tests for comparing nested models,
 - clean, compact models that are easy to discuss in class.
- **Predictive checks**
 - simple train–test splits for case studies,
 - side-by-side train vs test RMSE,
 - examples where a smaller model outperforms a more complex one on held-out data.

As you work through the code in later chapters, keep asking:

- “Am I trying to **explain** or **predict** here?”
- “Have I thought about **family**, **form**, and **fit** separately?”

That habit will pay off in any future modeling you do, whether with linear models, machine learning methods, or more advanced tools.

APPLIED STATISTICS WITH PYTHON – CHAPTER 11

22.1 Categorical predictors and interactions

So far our regression chapters have mostly used **numeric predictors**: continuous variables like horsepower, weight, or displacement.

In practice we also care about:

- **Categorical predictors** – e.g., transmission type (automatic vs manual), number of cylinders, country of origin.
- **Interactions** – situations where the effect of one predictor depends on the level of another.

The original R chapter for this material uses datasets like `mtcars` and `autompg`, and leans heavily on R's treatment of *factors* and the formula syntax. In this Python-first version we will:

- mirror the main ideas in **NumPy, pandas, and statsmodels**,
- show how **dummy variables** and **interactions** appear in formulas, and
- connect back to the matrix notation from Chapter 9.

Throughout, you can think in three parallel languages:

- the **statistical model** in symbols (e.g. $Y = \beta_0 + \beta_1 x_1 + \dots$),
- the **R formula** (e.g. `mpg ~ hp + am`),
- the **Python formula** for `statsmodels` (e.g. `"mpg ~ hp + am_manual"`).

The goal is not to memorize syntax, but to see how the *same ideas* travel between R and Python.

22.2 11.1 Dummy variables (indicator variables)

A **dummy variable** is a numerical 0/1 variable that encodes a *binary* category. For example, in the classic `mtcars` data:

- `mpg` – fuel efficiency (miles per gallon, our response),
- `hp` – horsepower (numeric predictor),
- `am` – transmission (0 = automatic, 1 = manual).

In R, you might fit

```
lm(mpg ~ hp + am, data = mtcars)
```

In Python, the same idea with `statsmodels` looks like:

```
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

# mtcars from the R datasets shipped with statsmodels
mtcars = sm.datasets.get_rdataset("mtcars", "datasets").data

# am is already coded 0/1, but let's make the meaning explicit
mtcars["am_manual"] = (mtcars["am"] == 1).astype(int)

model = smf.ols("mpg ~ hp + am_manual", data=mtcars).fit()
print(model.params)
```

Statistically the model is

$$Y = \beta_0 + \beta_1 \text{hp} + \beta_2 \text{am_manual} + \varepsilon,$$

where `am_manual` is 1 for manual and 0 for automatic.

Interpretation:

- β_1 – change in mean mpg for a one-unit increase in horsepower, holding transmission type fixed.
- β_2 – *difference* in mean mpg between manual and automatic transmissions at the same horsepower.
- β_0 – mean mpg for an automatic car with `hp = 0` (not realistic, but the algebraic intercept).

Notice how the dummy variable lets us write **two lines** with a shared slope:

- automatic: $Y = \beta_0 + \beta_1 x_1 + \varepsilon$,
- manual: $Y = (\beta_0 + \beta_2) + \beta_1 x_1 + \varepsilon$.

Same slope, different intercepts.

In PyStatsV1-style code, you'll usually see:

- a pandas column containing 0/1 indicators, and
- a statsmodels formula that includes that column as an extra predictor.

22.3 11.2 Interactions: when slopes depend on context

Dummy variables give us **two parallel lines**. Often we want something more flexible: different *slopes* for different groups.

22.3.1 A simple example

Suppose we have a cleaned `autompg` DataFrame with columns:

- `mpg` – response,
- `disp` – engine displacement,
- `domestic` – 1 if the car is built in the US, 0 if foreign.

An *additive* model with a dummy variable is

$$Y = \beta_0 + \beta_1 \text{disp} + \beta_2 \text{domestic} + \varepsilon.$$

Both domestic and foreign cars share the same slope β_1 .

An *interaction* model allows different slopes:

$$Y = \beta_0 + \beta_1 \text{ disp} + \beta_2 \text{ domestic} + \beta_3 \text{ disp} \cdot \text{domestic} + \varepsilon.$$

In Python's formula syntax:

- `disp + domestic` – additive model (no interaction),
- `disp * domestic` – additive terms **plus** `disp:domestic` interaction.

```
mpg_disp_add = smf.ols("mpg ~ disp + domestic", data=autompg).fit()
mpg_disp_int = smf.ols("mpg ~ disp * domestic", data=autompg).fit()

# compare nested models (F-test)
f_stat, p_value, _ = mpg_disp_int.compare_f_test(mpg_disp_add)
print(p_value)
```

If the p-value is very small, the interaction term meaningfully improves fit, and we prefer the more flexible model.

22.3.2 Interpreting the interaction

Write the interaction model separately for foreign (`domestic = 0`) and domestic (`domestic = 1`) cars:

- foreign: $Y = \beta_0 + \beta_1 \text{ disp} + \varepsilon$,
- domestic: $Y = (\beta_0 + \beta_2) + (\beta_1 + \beta_3) \text{ disp} + \varepsilon$.

So:

- **Intercepts** differ by β_2 .
- **Slopes** differ by β_3 .

Graphically you now have **two lines that can cross**, not just parallel lines. This is the key idea: *interactions let the effect of one variable depend on the value of another*.

22.3.3 Numeric–numeric interactions

Interactions are not just “dummy × numeric”. You can also interact two numeric predictors, for example:

```
model = smf.ols("mpg ~ disp * hp", data=autompg).fit()
print(model.summary().tables[1]) # coefficient table
```

The term `disp:hp` corresponds to $\beta_3 x_1 x_2$ in

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon.$$

A quick algebra trick:

$$Y = \beta_0 + (\beta_1 + \beta_3 x_2) x_1 + \beta_2 x_2 + \varepsilon.$$

The “slope in `disp`” is now $\beta_1 + \beta_3 x_2$, which **depends on `hp`**. That is exactly what “interaction” means.

22.4 11.3 Factor variables and automatic dummies

R has a special data type for categorical variables: **factors**. When you use a factor in a formula, R silently creates dummy variables for all but one reference level.

Python has the same idea split in two pieces:

- `pandas.Categorical` (and `dtype="category"`) to mark categorical data,
- `statsmodels` formulas that call `C(name)` to apply categorical coding.

22.4.1 Example: number of cylinders (4, 6, 8)

Suppose `autompg["cyl"]` has values 4, 6, and 8.

In R you might write:

```
lm(mpg ~ disp * cyl, data = autompg)
```

In Python, the equivalent is:

```
autompg["cyl"] = autompg["cyl"].astype("category")

model_add = smf.ols("mpg ~ disp + C(cyl)", data=autompg).fit()
model_int = smf.ols("mpg ~ disp * C(cyl)", data=autompg).fit()
```

Behind the scenes `statsmodels` creates dummy columns such as `C(cyl)[T.6]` and `C(cyl)[T.8]`; 4 cylinders is the reference level by default (because “4” is first in the sorted order).

Interpretation:

- **Intercept** – mean mpg for 4-cylinder cars when `disp = 0`.
- `C(cyl)[T.6]` – difference in intercept between 6- and 4-cylinder cars.
- `C(cyl)[T.8]` – difference in intercept between 8- and 4-cylinder cars.
- In the interaction model, additional terms like `disp:C(cyl)[T.6]` modify the slope for 6-cylinder cars relative to 4-cylinder cars.

This matches the R output conceptually, even though the labels look slightly different.

22.4.2 Changing the reference level

Just as in R you can relevel a factor, in Python you can change which category is treated as the baseline. One simple pattern is to reorder categories:

```
autompg["cyl"] = autompg["cyl"].cat.reorder_categories(["6", "4", "8"])
model = smf.ols("mpg ~ disp * C(cyl)", data=autompg).fit()
```

Now 6 cylinders is the reference group, so all coefficient interpretations shift accordingly—but the **fitted values and residuals are identical**.

22.5 11.4 Different parameterizations, same model

The same regression surface can be written in many algebraically equivalent ways.

For example, with 4/6/8 cylinders you could:

- use **two** dummy variables (6 vs not 6, 8 vs not 8) plus an intercept, or
- use **three** dummy variables and **no intercept**, or
- let `statsmodels` construct dummies via `C(cyl)` with your chosen reference level.

All of these parameterizations represent the same family of three lines, one for each cylinder count.

A check in Python:

```
# model 1: intercept + C(cyl) parameterization
m1 = smf.ols("mpg ~ disp * C(cyl)", data=autompg).fit()

# model 2: explicit dummies, no intercept
dummies = pd.get_dummies(autompg["cyl"], prefix="cyl", drop_first=False)
df2 = pd.concat([autompg[["mpg", "disp"]], dummies], axis=1)

m2 = smf.ols("mpg ~ 0 + disp:cyl_4 + disp:cyl_6 + disp:cyl_8 "
            "+ cyl_4 + cyl_6 + cyl_8", data=df2).fit()

import numpy as np
np.allclose(m1.fittedvalues, m2.fittedvalues) # should be True
```

Takeaway: **coefficients depend on parameterization; fitted values do not.** When comparing models, always compare their predictions or residuals, not just raw coefficient values.

22.6 11.5 Building larger models with interactions

Once you are comfortable with dummy variables and the `*` / `:` syntax, you can specify fairly rich models compactly.

22.6.1 A “big” three-way interaction model

Using the same variables as above:

- `mpg` – response,
- `disp` – displacement (numeric),
- `hp` – horsepower (numeric),
- `domestic` – 1 if US-built, 0 otherwise,

we might start with the full three-way interaction:

```
big = smf.ols("mpg ~ disp * hp * domestic", data=autompg).fit()
print(big.summary().tables[1])
```

The formula

$$\text{mpg} \sim \text{disp} * \text{hp} * \text{domestic}$$

expands to:

$$\text{disp} + \text{hp} + \text{domestic} + \text{disp:hp} + \text{disp:domestic} + \text{hp:domestic} + \text{disp:hp:domestic}.$$

This matches the **hierarchy principle**: if you include a higher-order interaction, you also include all lower-order pieces that lead up to it.

22.6.2 Model simplification with nested F-tests

Just like in the R notes, we typically *remove* high-order interactions unless they truly help.

In Python, you can compare nested models with `compare_f_test`:

```
# full three-way interaction
big = smf.ols("mpg ~ disp * hp * domestic", data=autompg).fit()

# all two-way interactions, no three-way
two_way = smf.ols(
    "mpg ~ disp * hp + disp * domestic + hp * domestic",
    data=autompg
).fit()

# additive model (no interactions)
additive = smf.ols("mpg ~ disp + hp + domestic", data=autompg).fit()

# (1) do we need the three-way interaction?
print(big.compare_f_test(two_way))    # (F, p-value, df_diff)

# (2) do we need any interactions at all?
print(two_way.compare_f_test(additive))
```

Typical workflow:

1. Start with a rich, but sensible model (respecting hierarchy).
2. Compare against a simpler nested model with an F-test.
3. If the p-value is large, prefer the simpler model (fewer parameters, easier to interpret).
4. Stop when further simplification clearly harms fit or removes important structure.

You can also inspect:

- residual standard error / RMSE,
- R^2 and adjusted R^2 ,
- domain knowledge (does the simpler model still make scientific sense?).

22.7 11.6 How this connects to PyStatsV1

In PyStatsV1, these ideas show up repeatedly:

- **Dummy variables** are how we bring categorical information (treatment vs control, male vs female, exposed vs not exposed) into linear models.
- **Interactions** are how we let effects vary across groups, or across levels of another variable.
- **Factor / categorical coding** is how we bridge from the R ecosystem (where factors are everywhere) to Python's pandas + statsmodels world.

When you read or write PyStatsV1 code for regression chapters, watch for:

- columns that are 0/1 (hand-made dummies),
- formulas that contain `C(variable)` (automatic categorical coding),
- `*` and `:` in model formulas (interactions).

These tools are the building blocks for:

- logistic regression with categorical predictors,
- ANOVA-style comparisons between treatments,

- models with policy indicators and time trends,
- and many of the case studies we'll add on top of this mini-textbook.

22.8 11.7 What you should take away

By the end of this chapter (R + Python versions), you should be able to:

- Construct **dummy variables** and interpret their coefficients as *differences* between groups.
- Distinguish between:
 - additive models (parallel lines, shared slopes) and
 - interaction models (group-specific slopes and intercepts).
- Use Python's formula interface to:
 - include categorical predictors via `C(...)`,
 - add interactions with `*` and `:`,
 - and change the reference level when needed.
- Read regression output and translate coefficients back to:
 - intercepts and slopes for specific groups, and
 - differences between those groups.
- Recognize that different **parameterizations** (different dummy codings) can represent the *same* model, even though the coefficient labels differ.
- Use nested model F-tests (or `compare_f_test` in `statsmodels`) to decide whether higher-order interactions are worth keeping.

In later PyStatsV1 chapters, these ideas will underpin:

- regression models with categorical predictors and interactions,
- comparisons of several models on the same data,
- and case studies where interpretation of group differences really matters.

APPLIED STATISTICS WITH PYTHON – CHAPTER 12

23.1 Analysis of variance (ANOVA) and experiments

This chapter parallels the “Analysis of Variance” material from the R notes, but is written for a Python-first audience.

So far in the mini-book we have mostly:

- worked with **numeric predictors** (chapters on simple and multiple regression),
- treated real data sets as if they were “just given,” without asking how they were produced.

Here we:

- draw a sharp line between **observational** and **experimental** data,
- show how experiments naturally lead to **ANOVA models**,
- connect classical ANOVA to the regression tools you have already seen,
- and map the R functions to Python:
 - R: `t.test()`, `aov()`, `pairwise.t.test()`, `TukeyHSD()`
 - Python: `scipy.stats`, `statsmodels` formula API, and `statsmodels.stats.multicomp`.

The key message:

Regression tools are not only for observational data; in experimental settings we use the same modelling ideas, but we are allowed to make much stronger causal statements because the predictors are under our control.

23.1.1 12.1 Experiments: observational vs experimental data

The most important question about a data set is **how it was generated**.

Observational study

Both predictors and response are observed. No one controlled who received which level of a predictor.

Examples: hospital records, government surveys, user-behaviour logs.

Experiment

The analyst **chooses** the levels of one or more predictors, applies them to subjects, then observes the response.

Examples: drug vs placebo trials, A/B tests on a website, training interventions for athletes.

23.2 Terminology for experiments

In experimental settings we rename things slightly:

- **Factors** – predictors that are controlled by the experimenter (treatment group, diet, machine setting, etc.).

- **Levels** – possible values of a factor (control vs treatment, A/B/C, low / medium / high).
- **Subjects** – experimental units (people, animals, plots of land, machines...).
- **Randomization** – subjects are randomly assigned to factor levels.

Randomization is crucial:

- it balances **unobserved** variables across groups on average;
- it justifies treating observations as independent draws from a model.

Throughout this chapter we will:

- write models in symbols,
- show the equivalent R formula (for reference),
- and give the Python version with `statsmodels`.

23.2.1 12.2 Two-sample t-test as a tiny experiment

The simplest experimental design is a **two-group experiment**:

- one factor (for example “group”) with two levels: control and treatment;
- subjects randomly assigned to a level;
- one numeric response measured after treatment.

23.3 Mathematical model

We assume

$$Y_{1j} \sim N(\mu_1, \sigma^2), \quad Y_{2j} \sim N(\mu_2, \sigma^2),$$

for subjects in groups 1 and 2 respectively.

We test

$$H_0 : \mu_1 = \mu_2 \quad \text{vs} \quad H_1 : \mu_1 \neq \mu_2.$$

In Chapter 5 we met the **two-sample t-test** for this situation.

23.4 R vs Python

R (formula interface):

```
t.test(sleep ~ group, data = melatonin, var.equal = TRUE)
```

Python (using `scipy.stats`):

```
import numpy as np
from scipy import stats

control = melatonin.loc[melatonin["group"] == "control", "sleep"]
treatment = melatonin.loc[melatonin["group"] == "treatment", "sleep"]

t_stat, p_value = stats.ttest_ind(control, treatment, equal_var=True)
```

The important conceptual link:

The two-sample t-test is a special case of one-way ANOVA with two groups. The next section generalizes to any number of groups.

23.4.1 12.3 One-way ANOVA

23.5 12.3.1 Model and intuition

Suppose we have one factor with g levels (groups), such as four diets A, B, C, and D. With n_i observations in group i , we write

$$Y_{ij} = \mu_i + \varepsilon_{ij}, \quad \varepsilon_{ij} \sim N(0, \sigma^2), \quad i = 1, \dots, g.$$

Equivalently,

$$Y_{ij} = \mu + \alpha_i + \varepsilon_{ij}, \quad \sum_{i=1}^g \alpha_i = 0.$$

Here:

- μ is the overall mean.
- α_i is the **effect** of group i (difference from the overall mean).
- σ^2 is the common within-group variance.

We test whether the group means are all equal:

$$H_0 : \mu_1 = \mu_2 = \dots = \mu_g \quad \text{vs} \quad H_1 : \text{at least one } \mu_i \text{ is different.}$$

ANOVA works by decomposing variability:

- **Between-group** variability – how far the group means are from the overall mean.
- **Within-group** variability – how much observations vary around their own group mean.

If between-group variation is large relative to within-group variation, the group means are unlikely to be all the same.

23.6 Sums of squares (conceptually)

We can write three key quantities:

- Total variation:

$$SS_T = \sum_{i=1}^g \sum_{j=1}^{n_i} (y_{ij} - \bar{y})^2.$$

- Between-group variation:

$$SS_B = \sum_{i=1}^g n_i (\bar{y}_i - \bar{y})^2.$$

- Within-group variation:

$$SS_W = \sum_{i=1}^g \sum_{j=1}^{n_i} (y_{ij} - \bar{y}_i)^2.$$

These satisfy $SS_T = SS_B + SS_W$.

The ANOVA F statistic compares mean squares:

$$MS_B = \frac{SS_B}{g-1}, \quad MS_W = \frac{SS_W}{N-g}, \quad F = \frac{MS_B}{MS_W},$$

where $N = \sum_i n_i$ is the total sample size.

Under H_0 , F has an $F_{g-1, N-g}$ distribution.

23.7 12.3.2 One-way ANOVA in Python

Suppose we have a DataFrame with columns:

- **response** – numeric outcome,
- **group** – categorical factor (diet, treatment, etc.).

Using `statsmodels`:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.ols("response ~ C(group)", data=df).fit()
anova_table = sm.stats.anova_lm(model, typ=2)
print(anova_table)
```

Key points:

- `C(group)` tells `statsmodels` to treat `group` as categorical.
- The ANOVA table includes sums of squares, degrees of freedom, mean squares, F statistic and p-value.

R uses

```
aov(response ~ group, data = df)
```

The underlying ideas are the same; both are just linear models with dummy variables for the groups.

23.8 12.3.3 Factor variables and categorical dtype

In R you must ensure the grouping variable is a **factor**; otherwise ANOVA silently becomes a regression with a numeric predictor.

In Python / pandas:

- Either wrap the variable in `C()` in the formula, **or**
- set `df["group"] = df["group"].astype("category")`.

If you forget and leave it numeric, the model becomes “line through the codes” (1, 2, 3, ...) rather than separate means per group.

23.9 12.3.4 Simulating the F distribution in Python

A good sanity check is to simulate from a null model (equal means) and verify that the empirical distribution of F matches the theoretical F distribution.

Skeleton code:

```

import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

rng = np.random.default_rng(seed=123)

def sim_oneway_F(n_per_group=10, g=4, sigma=1.0):
    # Null model: all means = 0
    groups = np.repeat(np.arange(g), n_per_group)
    y = rng.normal(loc=0.0, scale=sigma, size=g * n_per_group)
    df = pd.DataFrame({"y": y, "group": groups})
    model = smf.ols("y ~ C(group)", data=df).fit()
    table = sm.stats.anova_lm(model, typ=2)
    return table.loc["C(group)", "F"]

f_stats = np.array([sim_oneway_F() for _ in range(5000)])

```

You can then plot a histogram of `f_stats` and overlay an `scipy.stats.f` density to see the agreement.

23.10 12.3.5 Power via simulation

Because experiments cost time and money, we care about **power**:

$$\text{Power} = P(\text{reject } H_0 \mid H_0 \text{ false}).$$

For one-way ANOVA this depends on:

- effect sizes (how far the group means are apart),
- noise level σ ,
- sample size and balance (equal n_i helps),
- significance level α .

We can use almost the same simulation function as above, but now draw from unequal means and record how often the ANOVA p-value is below α .

Sketch:

```

from scipy import stats

def sim_oneway_p(mu, n_per_group=10, sigma=1.0):
    mu = np.asarray(mu)
    g = len(mu)
    groups = np.repeat(np.arange(g), n_per_group)
    means = np.repeat(mu, n_per_group)
    y = rng.normal(loc=means, scale=sigma, size=g * n_per_group)
    df = pd.DataFrame({"y": y, "group": groups})
    model = smf.ols("y ~ C(group)", data=df).fit()
    table = sm.stats.anova_lm(model, typ=2)
    return table.loc["C(group)", "PR(>F)"]

p_vals = np.array([sim_oneway_p(mu=[-1, 0, 0, 1], sigma=1.5)

```

(continues on next page)

(continued from previous page)

```

        for _ in range(1000)]])

power_05 = np.mean(p_vals < 0.05)
power_01 = np.mean(p_vals < 0.01)

```

By varying μ , σ , and sample size you can explore how design choices affect power.

23.10.1 12.4 Post-hoc comparisons and multiple testing

ANOVA answers a global question:

“Are all group means equal?”

If the F-test is significant, we often want to know **which means differ**.

23.11 Naive approach

Run all pairwise two-sample t-tests and look at their p-values. But if there are many groups, this inflates the **family-wise error rate (FWER)**: the probability of at least one false positive among the whole family of tests.

Example: with 10 independent tests at $\alpha = 0.05$, the probability of at least one false positive is much larger than 0.05.

23.12 Bonferroni adjustment

One simple fix:

Either use a stricter per-test level α/m for m tests, *or* multiply each p-value by m (capped at 1).

Python:

```

from statsmodels.stats.multitest import multipletests

# pvals: array of unadjusted p-values from pairwise t-tests
reject, pvals_adj, _, _ = multipletests(pvals, alpha=0.05,
                                       method="bonferroni")

```

23.13 Tukey’s HSD

For “all pairwise comparisons of group means after one-way ANOVA” there is a classical procedure: **Tukey’s Honest Significant Difference**.

In Python, statsmodels provides:

```

from statsmodels.stats.multicomp import pairwise_tukeyhsd

tukey = pairwise_tukeyhsd(endog=df["response"],
                          groups=df["group"],
                          alpha=0.05)

print(tukey.summary())

```

This reports:

- which pairs of means differ significantly,
- adjusted p-values,

- simultaneous confidence intervals for mean differences.

23.13.1 12.5 Two-way ANOVA and interactions

One-way ANOVA handles one factor. Real experiments often manipulate **two or more factors at once**.

Example design:

- factor A: three types of antibiotic (I, II, III),
- factor B: four treatments (A, B, C, D),
- response: survival time of a bacteria cluster.

23.14 Model with interaction

With factors A and B, levels $i = 1, \dots, I$, $j = 1, \dots, J$, and replicates $k = 1, \dots, K$ per cell, we write

$$Y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijk},$$

where $\varepsilon_{ijk} \sim N(0, \sigma^2)$.

- α_i – main effect of factor A,
- β_j – main effect of factor B,
- $(\alpha\beta)_{ij}$ – **interaction** between A and B.

Interpretation of interaction:

The effect of A depends on the level of B (or vice versa).

23.15 Additive model (no interaction)

If all interaction terms are zero we have

$$Y_{ijk} = \mu + \alpha_i + \beta_j + \varepsilon_{ijk}.$$

The difference between two levels of factor A is the same at every level of factor B.

23.16 Model hierarchy and testing strategy

Typically we proceed in this order:

1. Fit the **full interaction model**.
2. Test whether the interaction is significant.
3. If interaction *is* significant, stop there and interpret it.
4. If interaction is *not* significant, drop it and fit the **additive model** with main effects only.
5. Within the additive model you can test main effects and use Tukey-type post-hoc comparisons on each factor.

23.17 Two-way ANOVA in Python

Assume a DataFrame `bacteria` with columns `time` (response), `antibiotic` (factor A), and `treat` (factor B).

Full interaction model:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

model_int = smf.ols("time ~ C(antibiotic) * C(treat)", data=bacteria).fit()
anova_int = sm.stats.anova_lm(model_int, typ=2)
print(anova_int)
```

If the antibiotic:treat row has a small p-value, keep this model and examine estimated means for each combination.

To get cell means:

```
import itertools
import pandas as pd

levels_antibiotic = bacteria["antibiotic"].unique()
levels_treat = bacteria["treat"].unique()

grid = pd.DataFrame(
    list(itertools.product(levels_antibiotic, levels_treat)),
    columns=["antibiotic", "treat"],
)
cell_means = model_int.predict(grid)
grid["mean_time"] = cell_means
print(grid)
```

If the interaction is not significant, fit the additive model:

```
model_add = smf.ols("time ~ C(antibiotic) + C(treat)", data=bacteria).fit()
anova_add = sm.stats.anova_lm(model_add, typ=2)

print(anova_add)
```

You can then run Tukey's HSD separately on antibiotic and treat.

23.18 Interaction plots

Before fitting any models, it is helpful to **plot group means**.

In Python you can approximate R's `interaction.plot` using a line plot of mean response vs one factor, with a separate line for each level of the other factor (for example with `seaborn's pointplot`).

Parallel lines suggest an additive model; clearly crossing lines suggest an interaction.

23.18.1 12.6 What you should take away

By the end of this chapter you should be comfortable with:

- clearly distinguishing **observational** vs **experimental** data,
- explaining why experiments (with randomization) are needed for causal statements,
- writing down the models for
 - two-sample t-tests,
 - one-way ANOVA,

- two-way ANOVA with and without interaction,
- reading an ANOVA table: sums of squares, degrees of freedom, mean squares, F statistics, and p-values,
- using Python tools to perform
 - two-sample t-tests (`scipy.stats`),
 - one-way ANOVA (`statsmodels` with `C(group)`),
 - post-hoc comparisons with multiple-testing corrections,
 - two-way ANOVA and interaction tests,
- understanding the idea of **power** and how simulation can help plan sample sizes,
- appreciating that “statistically significant” is not automatically “scientifically important” – you still need to think about effect sizes.

23.18.2 12.7 How this connects to PyStatsV1

In later PyStatsV1 chapters and case studies you will see these ideas appear in several ways:

- using ANOVA as a special case of the linear models you already know,
- connecting experimental designs to regression models with dummy variables,
- simulating power curves before you “run an experiment in code,”
- contrasting what you can say from experimental data versus observational data using the same modelling tools.

If you work in psychology, sports science, education, or any field with interventions, this chapter provides the conceptual bridge from PyStatsV1’s regression techniques to the world of **experimental design** and **evidence-based practice**.

APPLIED STATISTICS WITH PYTHON – CHAPTER 13

24.1 Model diagnostics for regression

This chapter parallels the *Model Diagnostics* chapter from the R notes, but uses a Python–first workflow. The statistical ideas are the same: we fit a regression model, then carefully check whether its assumptions are reasonable before trusting p-values, confidence intervals, or predictions. :contentReference[oaicite:0]{index=0}

By the end of this chapter (R + Python versions), you should be able to:

- state the core assumptions of a linear regression model,
- diagnose violations of these assumptions using residual plots and tests,
- understand leverage, outliers, and influential points,
- compute and interpret standardized residuals and Cook’s distance,
- and know what to do next when diagnostics look “off”.

24.2 13.1 Regression model assumptions (recap)

We work with the multiple linear regression model

$$Y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_{p-1} x_{i,p-1} + \varepsilon_i, \quad i = 1, \dots, n,$$

or in matrix form

$$\mathbf{Y} = X\beta + \varepsilon.$$

The least–squares estimator is

$$\hat{\beta} = (X^\top X)^{-1} X^\top y.$$

The *assumptions* live in the error term ε_i :

- **Linearity** – the mean of Y is a linear function of the predictors.
- **Independence** – errors ε_i are independent.
- **Normality** – errors are Normally distributed.
- **Equal variance** – errors have constant variance σ^2 for all combinations of predictors (homoscedasticity). :contentReference[oaicite:1]{index=1}

If these assumptions hold, our familiar t–tests, F–tests, and confidence intervals are valid. If they fail badly, we can still *compute* them, but the results are not trustworthy.

In Python, these assumptions underlie models such as `statsmodels.api.OLS` and `statsmodels.formula.api.ols()`.

24.3 13.2 Checking assumptions in Python

In this section we mirror the R diagnostic tools using NumPy, pandas, Matplotlib, SciPy, and statsmodels.

Throughout, imagine we have already fit a regression model

```
import statsmodels.formula.api as smf

model = smf.ols("y ~ x1 + x2", data=df).fit()

fitted = model.fittedvalues
resid = model.resid
```

24.3.1 13.2.1 Fitted versus residuals plots

A **fitted vs residuals** plot (sometimes called *residuals vs fitted*) is our workhorse diagnostic.

- x-axis: fitted values \hat{y}_i
- y-axis: residuals $e_i = y_i - \hat{y}_i$

In Python:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(fitted, resid, alpha=0.6)
ax.axhline(0, color="black", linewidth=1)
ax.set_xlabel("Fitted values")
ax.set_ylabel("Residuals")
ax.set_title("Fitted vs residuals")
```

What to look for:

- **Linearity**
 - At each fitted value, residuals should be centered around 0.
 - A clear curve (e.g. U-shape) suggests the *form* of the model is wrong (missing polynomial terms, interactions, or other nonlinear structure).
- **Equal variance**
 - Spread of residuals should be roughly constant along the x-axis.
 - “Funnel” shapes (narrow then wide) suggest heteroscedasticity (non-constant variance).

For simple linear regression you might see problems directly in the (x, y) scatterplot, but for multiple regression this fitted-vs-residuals plot is essential.

24.3.2 13.2.2 Breusch–Pagan test for constant variance

The Breusch–Pagan test provides a formal check of homoscedasticity:

- H_0 : error variance is constant (homoscedastic).
- H_1 : error variance depends on the predictors (heteroscedastic).

In Python (statsmodels):

```

from statsmodels.stats.diagnostic import het_breuschpagan

bp_stat, bp_pvalue, _, _ = het_breuschpagan(
    model.resid,
    model.model.exog,  # design matrix X
)

print(f"BP statistic = {bp_stat:.3f}, p-value = {bp_pvalue:.3g}")

```

Interpretation (typical conventions):

- **Large p-value** (say > 0.05): no strong evidence against constant variance.
- **Small p-value**: evidence that variance changes with the predictors; consider transformations, adding missing structure, or using robust standard errors.

24.3.3 13.2.3 Histograms of residuals

To check the normality assumption, a simple starting point is a histogram of residuals:

```

fig, ax = plt.subplots()
ax.hist(resid, bins=20, edgecolor="black")
ax.set_xlabel("Residuals")
ax.set_ylabel("Frequency")
ax.set_title("Histogram of residuals")

```

You are looking for a roughly symmetric, bell-shaped distribution. However, histograms can be ambiguous (especially with small samples), so we usually follow up with Q–Q plots and formal tests. :contentReference[oaicite:4]{index=4}

24.3.4 13.2.4 Normal Q–Q plots

A **Normal Q–Q plot** (quantile–quantile plot) compares the sorted residuals to what we would expect if they were sampled from a Normal distribution.

In Python, we can use SciPy or statsmodels:

```

import statsmodels.api as sm

fig = sm.qqplot(resid, line="45")
plt.title("Normal Q-Q plot of residuals")

```

Guidelines:

- Points close to the line \rightarrow residuals are plausibly Normal.
- Systematic curvature (e.g. S-shape) or heavy tails (points far from the line at extremes) \rightarrow Normality assumption is questionable.
- With small n , random noise in the plot is expected; with large n , even small deviations become visible.

24.3.5 13.2.5 Shapiro–Wilk normality test

The **Shapiro–Wilk test** is a widely used test of Normality. :contentReference[oaicite:5]{index=5}

In Python (SciPy):

```
from scipy.stats import shapiro

W_stat, pvalue = shapiro(resid)
print(f"Shapiro-Wilk W = {W_stat:.3f}, p-value = {pvalue:.3g}")
```

Interpretation:

- H_0 : the data are sampled from a Normal distribution.
- Small p-value \rightarrow residuals are unlikely to be Normal.
- Large p-value \rightarrow no strong evidence against Normality.

As always, combine this with visual tools (histogram, Q-Q plot); a test alone does not tell the whole story.

24.4 13.3 Unusual observations: leverage, outliers, influence

Diagnostics are not only about assumptions; we also care about **unusual data points** that can distort a regression:

- **High leverage** points – unusual predictor values (extreme in X).
- **Outliers** – points with large residuals (poorly fit by the model).
- **Influential** points – observations that substantially change the fitted model when removed. :contentReference[oaicite:6]{index=6}

24.4.1 13.3.1 Leverage and the hat matrix

Recall the fitted values

$$\hat{y} = X\hat{\beta} = X(X^\top X)^{-1}X^\top y = Hy,$$

where

$$H = X(X^\top X)^{-1}X^\top$$

is the **hat matrix**. Its diagonal entries h_i are the leverage values:

$$h_i = H_{ii}, \quad i = 1, \dots, n.$$

Properties:

- $0 \leq h_i \leq 1$,
- $\sum_i h_i = p$, the number of regression parameters.

Heuristic:

- Average leverage $\bar{h} = p/n$.
- Points with $h_i > 2\bar{h}$ are often flagged as **high leverage**.

In Python, statsmodels makes this easy:

```
influence = model.get_influence()
leverage = influence.hat_matrix_diag

avg_h = leverage.mean()
high_lev = leverage > 2 * avg_h
df.loc[high_lev, :]
```


24.4.2 13.3.2 Standardized residuals and outliers

Raw residuals have different variances for different h_i . Under the model assumptions we have

$$\text{Var}(e_i) = (1 - h_i) \sigma^2.$$

We therefore look at **standardized (studentized) residuals**

$$r_i = \frac{e_i}{s_e \sqrt{1 - h_i}},$$

which are approximately $N(0, 1)$ when the model is correct. [:contentReference\[oaicite:7\]{index=7}](#)

Rule of thumb:

- $|r_i| > 2 \rightarrow$ potentially an outlier in the regression sense.
- $|r_i| > 3 \rightarrow$ very suspicious.

In Python:

```
std_resid = influence.resid_studentized_internal
outliers = abs(std_resid) > 2
df.loc[outliers, :]
```

Remember: an outlier is defined *relative to the model*. A point may be perfectly reasonable in the original data scale but still be an outlier for a particular regression.

24.4.3 13.3.3 Cook's distance: measuring influence

Cook's distance combines leverage and residual size into a single measure of how much each point influences the fitted model. [:contentReference\[oaicite:8\]{index=8}](#)

Heuristic rule:

$$D_i > \frac{4}{n} \rightarrow \text{observation } i \text{ is influential.}$$

In Python:

```
cooks_d, _ = influence.cooks_distance
influential = cooks_d > 4 / len(cooks_d)

df.loc[influential, :]

# Optionally, sort by Cook's distance
df.assign(cooks_d=cooks_d).sort_values("cooks_d", ascending=False).head()
```

Influential points are not automatically “bad”, but they deserve extra scrutiny: they may be data entry errors, unusual cases that require a different model, or scientifically interesting exceptions.

24.5 13.4 Examples in Python

Here we sketch how the textbook examples translate into Python. The exact datasets and scripts live in the PyStatsV1 repository.

24.5.1 13.4.1 Example: *mtcars* additive model

Consider the model

$$\text{mpg} = \beta_0 + \beta_1 \text{hp} + \beta_2 \text{am} + \varepsilon,$$

where hp is horsepower and am is a 0/1 indicator for manual transmission.

In Python:

```
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

mtcars = pd.read_csv("data/mtcars.csv") # or similar helper in PyStatsV1
mpg_hp_add = smf.ols("mpg ~ hp + am", data=mtcars).fit()

print(mpg_hp_add.summary())
```

Diagnostics:

```
influence = mpg_hp_add.get_influence()
leverage = influence.hat_matrix_diag
std_resid = influence.resid_studentized_internal
cooks_d, _ = influence.cooks_distance

# How many high leverage points?
high_lev = leverage > 2 * leverage.mean()
print("High leverage:", high_lev.sum())

# How many large residuals?
large_resid = abs(std_resid) > 2
print("Large standardized residuals:", large_resid.sum())

# Influential points by Cook's distance
influential = cooks_d > 4 / len(cooks_d)
print("Influential points:", influential.sum())
```

The R version labels specific cars; in Python you can inspect those rows by index and refit the model excluding them to see how much the coefficients change.

24.5.2 13.4.2 Example: a large interaction model for Auto MPG

The R notes end with a “big” interaction model for an Auto MPG dataset and show that diagnostics can look poor when the model is over-complex or the data contain many influential points. :contentReference[oaicite:9]{index=9}

In Python the pattern is the same:

```
autompg = pd.read_csv("data/autompg.csv")

big_model = smf.ols(
    "mpg ~ disp * hp * domestic",
    data=autompg,
).fit()

sm.qqplot(big_model.resid, line="45")
```

(continues on next page)

(continued from previous page)

```
plt.title("Q-Q plot: big_model")
plt.show()

# Many influential points?
infl = big_model.get_influence()
cooks_d, _ = infl.cooks_distance
influential = cooks_d > 4 / len(cooks_d)
print("Number of influential points:", influential.sum())
```

You can then refit on a subset (for example, excluding the most influential points) or, better, reconsider the model structure (transformations, simpler interaction structure, different predictors).

24.6 13.5 What you should take away

By the end of this chapter you should be comfortable with:

- stating the assumptions of a linear regression model and where they live in $Y = X\beta + \varepsilon$,
- using fitted vs residuals plots to check linearity and constant variance,
- using histograms, Q–Q plots, and the Shapiro–Wilk test to assess Normality,
- running and interpreting the Breusch–Pagan test for heteroscedasticity,
- computing leverage, standardized residuals, and Cook’s distance to detect high-leverage, outlying, and influential observations,
- and combining these diagnostics to decide whether your model is reasonable or needs to be revised.

In later PyStatsV1 chapters and case studies, these tools will underpin more advanced modeling (logistic regression, generalized linear models, mixed effects) and will be part of a standard “checklist” whenever we fit a model to real data.

If any of the diagnostics here feel abstract, try the following in a Python shell or notebook:

- simulate data where you *know* a model is correct, and verify that the diagnostics look good;
- then deliberately break assumptions (non-constant variance, nonlinear relationship, heavy-tailed noise) and see how the plots and tests react;
- finally, apply the same tools to real datasets in PyStatsV1 and compare.

That hands-on experimentation will make the ideas in this chapter stick.

APPLIED STATISTICS WITH PYTHON – CHAPTER 14

25.1 Transformations

In Chapter 13 we focused on *diagnosing* regression models: checking assumptions, looking for non-constant variance, and finding unusual observations.

This chapter asks a natural follow-up question:

What can we do when the diagnostics say our model is not OK?

A major tool is to transform variables:

- transform the **response** to stabilize variance or make residuals more nearly Normal;
- transform **predictors** to capture non-linear relationships while keeping a linear model;
- use **polynomial terms** to fit smooth curves.

By the end of this chapter you will be able to:

- explain what a variance-stabilizing transformation is and why it matters;
- fit and interpret regression models with a log-transformed response;
- use the Box–Cox family to choose a transformation automatically;
- transform predictors (for example, log–log relationships);
- fit polynomial regression models using `statsmodels` (and understand how Patsy’s `I()` works);
- recognize over-fitting and the dangers of extrapolating high-degree polynomials.

25.1.1 14.1 Response transformations

We start from a simple but common problem: the residual variance grows with the mean.

Example: salaries at Initech

Suppose we have data on salary versus years of experience at a fictional company Initech, stored in `data/initech.csv`:

```
import pandas as pd
import statsmodels.formula.api as smf

initech = pd.read_csv("data/initech.csv")
initech.head()
```

We first fit a simple linear regression,

$$Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i,$$

where Y_i is salary and x_i is years of experience.

```
lin_mod = smf.ols("salary ~ years", data=initech).fit()
print(lin_mod.summary())
```

If you reuse the residual plots from Chapter 13 you will typically see:

- the *mean* relationship is roughly linear;
- the *variance* of the residuals increases with the fitted value.

This violates the constant variance assumption ($\text{Var}[\varepsilon_i] = \sigma^2$ for all i).

Variance-stabilizing transformations

In symbols, our assumption is

$$\varepsilon \sim N(0, \sigma^2), \quad \text{so } \text{Var}[Y \mid X = x] = \sigma^2.$$

But in the Initech plot the variance looks like it depends on the mean:

$$\text{Var}[Y \mid X = x] = h(\mathbb{E}[Y \mid X = x]),$$

for some increasing function h .

A **variance-stabilizing transformation** is a function g of the response such that

$$\text{Var}[g(Y) \mid X = x] \approx c,$$

a constant that no longer changes with the mean.

In practice we often try simple monotone transforms:

- log: $g(y) = \log y$;
- square root: $g(y) = \sqrt{y}$;
- reciprocal: $g(y) = 1/y$.

A good rule of thumb:

- If the response is strictly positive and spans multiple orders of magnitude, trying a log transform is almost always reasonable.

Log-transforming salary

For Initech we try

$$\log Y_i = \beta_0 + \beta_1 x_i + \varepsilon_i.$$

In Python we can express this directly in the model formula:

```
import numpy as np

log_mod = smf.ols("np.log(salary) ~ years", data=initech).fit()
print(log_mod.summary())
```

Note a few things:

- The *response* is now `np.log(salary)`. We did **not** create a new column; Patsy (the formula library used by statsmodels) evaluates the NumPy call on the fly.
- The model is still *linear* in the parameters β_0 and β_1 .

To visualize:

- scatterplot of `years` versus `np.log(salary)` with the fitted straight line;
- residual plots for `log_mod`.

You should see:

- the residual spread is much more constant;
- the Normal Q–Q plot looks closer to a straight line.

Interpreting coefficients on the original scale

On the transformed scale we have

$$\log \hat{y}(x) = \hat{\beta}_0 + \hat{\beta}_1 x.$$

Exponentiating both sides,

$$\hat{y}(x) = \exp(\hat{\beta}_0) \exp(\hat{\beta}_1 x).$$

Each **additional year of experience** multiplies the *median* salary by

$$\exp(\hat{\beta}_1).$$

For example, if $\hat{\beta}_1 = 0.079$, then

$$\exp(0.079) \approx 1.08,$$

meaning “about an 8% increase in salary per year of experience”.

Comparing fit on the original scale

“Smaller residual standard error” is not comparable across models that use different scales. Instead, compare the root mean squared error on the original salary scale:

```
# original model
rmse_lin = np.sqrt(np.mean((initech["salary"] - lin_mod.fittedvalues) ** 2))

# log model, transformed back to dollars
rmse_log = np.sqrt(
    np.mean((initech["salary"] - np.exp(log_mod.fittedvalues)) ** 2)
)

rmse_lin, rmse_log
```

Typically `rmse_log` is smaller, supporting the transformed model.

14.1.1 The Box–Cox family

The log transform works well here, but we can also *let the data* suggest a transformation.

The **Box–Cox family** of transforms for a strictly positive response y is

$$g_{\lambda}(y) = \begin{cases} \frac{y^{\lambda} - 1}{\lambda}, & \lambda \neq 0, \\ \log y, & \lambda = 0. \end{cases}$$

The idea:

- For each candidate λ , transform the response with g_{λ} , fit a linear model, and compute the log-likelihood.
- Choose the λ that maximizes this likelihood (or a nearby “nice” value, like 0, 0.5, 1, -0.5, etc.).
- Optionally, build a confidence interval for λ to decide whether a simple transformation such as log is adequate.

In Python you can use `scipy.stats.boxcox` to obtain the MLE of λ :

```
from scipy import stats

y = initech["salary"].to_numpy()
# returns transformed y and the MLE lambda_
y_bc, lambda_ = stats.boxcox(y)
lambda_
```

You can then fit a model to `y_bc` instead of `salary` and compare diagnostics as before.

For the Initech data the Box–Cox profile (not shown here) typically puts $\lambda = 0$ (log) very near the maximum and inside its confidence interval, justifying our simpler choice.

25.1.2 14.2 Transforming predictors and using polynomials

So far we transformed the *response*. We can also transform **predictors**:

- to make non-linear relationships look linear;
- to build more flexible models while staying within the linear regression framework.

Log–log relationships

Recall the Auto MPG data used in earlier chapters. Suppose we want to model fuel economy `mpg` as a function of horsepower `hp`:

```
auto = pd.read_csv("data/autompkg.csv")
auto.head()
```

A simple linear model often shows curvature and non-constant variance:

```
mpg_hp_lin = smf.ols("mpg ~ hp", data=auto).fit()
mpg_hp_lin.summary()
```

Diagnostic plots usually reveal:

- `mpg` decreases as `hp` increases;
- the relationship is not quite linear;
- residual variance grows for large `hp`.

A common fix is to log-transform one or both variables.


```
mpg_hp_logy = smf.ols("np.log(mpg) ~ hp", data=auto).fit()
mpg_hp_loglog = smf.ols("np.log(mpg) ~ np.log(hp)", data=auto).fit()
```

The log–log model

$$\log(\text{mpg}) = \beta_0 + \beta_1 \log(\text{hp}) + \varepsilon$$

has a convenient interpretation:

$$\beta_1 \approx \text{elasticity of mpg with respect to hp,}$$

the percent change in mpg for a 1% change in horsepower (holding other variables fixed).

In practice, residual plots for the log–log model are often much cleaner than for the raw variables.

14.2.1 Polynomial regression

Another powerful tool is to include **polynomial terms** of a predictor.

Example: marketing and diminishing returns

Suppose `data/marketing.csv` contains monthly sales of a product (`sales`) and the advertising budget (`advert`), both measured in tens of thousands of dollars.

Plotting the data suggests that

- sales increase with advertising,
- but with *diminishing returns*.

We first fit a straight line:

```
marketing = pd.read_csv("data/marketing.csv")

mod_lin = smf.ols("sales ~ advert", data=marketing).fit()
mod_lin.summary()
```

A linear model ignores curvature. To capture diminishing returns we add a quadratic term:

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i.$$

In Patsy / statsmodels we can write

```
mod_quad = smf.ols("sales ~ advert + I(advert**2)", data=marketing).fit()
print(mod_quad.summary())
```

Key points:

- `I(advert**2)` tells Patsy to treat `advert**2` as a new predictor, not as part of formula syntax—`I()` stands for “inhibit”.
- With the linear term already in the model, the `I(advert**2)` term is highly significant.
- Residual plots are much cleaner; the fitted curve bends the right way.

You can continue this pattern with higher-order terms, but beware of over-fitting and strange behavior outside the data range.

Over-fitting and extrapolation

In principle we can fit a polynomial of any degree:

$$Y_i = \beta_0 + \beta_1 x_i + \cdots + \beta_{p-1} x_i^{p-1} + \varepsilon_i.$$

With degree $p - 1 = n - 1$ (one less than the number of points) a polynomial can *perfectly interpolate* the data: residuals are exactly zero.

That is rarely useful:

- the fit becomes extremely wiggly;
- predictions outside the observed range (extrapolation) can be absurd;
- standard errors explode because the design matrix is nearly singular.

The moral:

- Use low-degree polynomials (quadratic, maybe cubic, occasionally quartic).
- Always check residuals and, importantly, **plots of the fitted curve versus data**, not just R^2 .

Example: fuel economy versus speed

Consider experimental data on a car's fuel efficiency at different speeds, stored in `data/fuel_econ.csv` with columns `mph` and `mpg`.

We expect:

- mpg increases with speed up to some optimal point,
- then decreases again—roughly a smooth hump-shaped curve.

We can start with

```
econ = pd.read_csv("data/fuel_econ.csv")

fit1 = smf.ols("mpg ~ mph", data=econ).fit()
fit2 = smf.ols("mpg ~ mph + I(mph**2)", data=econ).fit()
fit4 = smf.ols("mpg ~ mph + I(mph**2) + I(mph**3) + I(mph**4)", data=econ).fit()
fit6 = smf.ols(
    "mpg ~ mph + I(mph**2) + I(mph**3) + I(mph**4) + I(mph**5) + I(mph**6)",
    data=econ,
).fit()
```

Use residual plots and F-tests for nested models to decide how far to go:

```
from statsmodels.stats.anova import anova_lm

anova_lm(fit4, fit6)
```

If the p -value is moderate but the residuals look clearly better for `fit6`, you might keep the degree-6 model *but* still be cautious about extrapolation beyond the range of observed speeds.

Orthogonal polynomials (optional)

High-degree raw polynomials (`x`, `x**2`, `x**3`, ...) are often highly correlated, which can lead to numerical instability and large standard errors.

In R, `poly(x, degree)` constructs **orthogonal polynomials**. The Patsy library used by `statsmodels` has similar tools but they are less commonly used in basic workflows.

If you need high-degree polynomials in Python, a pragmatic strategy is:

- use `numpy.vander` or `sklearn.preprocessing.PolynomialFeatures` to generate columns;
- standardize predictors (center and scale) before forming high powers;
- keep degrees fairly small unless you have a strong reason and plenty of data.

25.1.3 14.3 Examples in Python

This section sketches complete Python workflows that combine the ideas above. (Feel free to open a notebook and run them step by step.)

Example 1: fixing non-constant variance via log

1. Load Initech salary data and fit the straight-line model $\text{salary} \sim \text{years}$.
2. Reuse the diagnostic helper from Chapter 13 to make residual plots.
3. Fit the log-response model $\text{np.log(salary)} \sim \text{years}$.
4. Compare residual plots and RMSE on the original dollar scale.
5. Translate $\hat{\beta}_1$ into a “percent change per year” interpretation.

Example 2: log–log relationship between mpg and horsepower

1. Load Auto MPG data from `data/autompg.csv`.
2. Fit three models: $\text{mpg} \sim \text{hp}$, $\text{np.log(mpg)} \sim \text{hp}$, and $\text{np.log(mpg)} \sim \text{np.log(hp)}$.
3. Compare:
 - R^2 and residual standard error;
 - residual plots;
 - interpret the slope in the log–log model.

Example 3: polynomial fuel-economy curve

1. Load `data/fuel_econ.csv`.
2. Fit models with degrees 1, 2, 4, and 6 in `mph`.
3. For each model:
 - draw the fitted curve overlaid on the scatterplot;
 - inspect residual plots.
4. Use `anova_lm` or an information criterion (AIC) to compare nested models.
5. Choose a final degree (for example 4 or 6) and interpret its implications for “best speed for fuel economy”.

25.1.4 14.4 How this connects to PyStatsV1

Transformations and polynomial terms are building blocks that reappear throughout PyStatsV1:

- In later material on **model selection** we will compare many models that differ only in which transformed variables they include.
- In **logistic regression** and generalized linear models, link functions play a role very similar to response transformations here.

- In chapters on **experimental design**, we often build models that include polynomial terms in quantitative factors (for example, quadratic response-surface models) and interactions between factors.
- In applied work, much of the “art” of modeling is about choosing sensible transformations that make diagnostics (Chapter 13) look good without sacrificing interpretability.

PyStatsV1 examples and exercises will encourage you to:

- try simple transformations when diagnostics suggest problems;
- check that you can still explain the model in context (e.g., percent changes instead of additive changes);
- avoid blindly adding very high-degree polynomials just because they increase R^2 .

25.1.5 14.5 What you should take away

By the end of this chapter you should be comfortable with:

- **When and why** to transform the response:
 - to stabilize variance;
 - to make residuals more nearly Normal;
 - to convert additive effects into multiplicative (percentage) effects.
- Using the **log transform** for strictly positive variables and interpreting coefficients back on the original scale.
- The **Box–Cox family** as a way to choose a transformation:
 - the special role of $\lambda = 0$ (log);
 - reading the profile of log-likelihood vs λ .
- Transforming **predictors**:
 - log and other monotone transforms;
 - building **polynomial regression** models with $I(x^{**2})$, $I(x^{**3})$, etc.;
 - using nested model comparisons (or AIC) to decide how much complexity is warranted.
- Recognizing **over-fitting** and the dangers of extrapolating high-degree polynomials.
- The distinction between:
 - the *statistical model* (what assumptions we make about $Y | X$);
 - the *formula* language we use to tell Python which transformed variables to include.

Most importantly:

Diagnostics come first; transformations are tools you apply in response to what the diagnostics show.

APPLIED STATISTICS WITH PYTHON – CHAPTER 15

26.1 Collinearity

This chapter parallels the *Collinearity* chapter from the R notes. Here we keep the statistical ideas the same, but express them in Python-first terms.

By the end of this chapter you should be able to:

- Recognize **exact collinearity** (perfect linear dependence) in a design matrix.
- Diagnose **high collinearity** between predictors using correlations and variance inflation factors (VIFs).
- Explain how collinearity affects regression **coefficients, standard errors, and interpretation**.
- Distinguish between the impact of collinearity on **explanation** versus **prediction**.
- Use **partial correlation** and **added-variable plots** to decide whether a new predictor is worth adding to an existing model.

26.1.1 15.1 Exact collinearity: when the design matrix is singular

We start with an extreme case: one predictor is an exact linear combination of the others.

In R, the book defines a function that generates three predictors where

$$x_3 = 2x_1 + 4x_2 + 3,$$

while the response y only depends on x_1 and x_2 .

In Python, we can do the same:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)

def gen_exact_collin_data(num_samples: int = 100) -> pd.DataFrame:
    x1 = rng.normal(loc=80, scale=10, size=num_samples)
    x2 = rng.normal(loc=70, scale=5, size=num_samples)
    x3 = 2 * x1 + 4 * x2 + 3
    y = 3 + x1 + x2 + rng.normal(loc=0, scale=1, size=num_samples)
    return pd.DataFrame({"y": y, "x1": x1, "x2": x2, "x3": x3})

exact_collin_data = gen_exact_collin_data()
exact_collin_data.head()
```

Here y really only “needs” x_1 and x_2 .

If we try to fit a model with all three predictors using the usual normal equations, the matrix $X^T X$ is not invertible:

```
import numpy.linalg as la

X = np.column_stack(
    [
        np.ones(len(exact_collin_data)),      # intercept
        exact_collin_data[["x1", "x2", "x3"]].to_numpy(),
    ]
)

la.inv(X.T @ X)    # raises LinAlgError: singular matrix
```

Statsmodels hides this detail for us and simply drops one column:

```
fit = smf.ols("y ~ x1 + x2 + x3", data=exact_collin_data).fit()
print(fit.summary())
```

You will see that one coefficient is reported as *not defined* because the columns are linearly dependent. Internally, statsmodels has fitted an equivalent model that only uses two of the three predictors.

Key points:

- With **exact collinearity**, there are infinitely many regression coefficient vectors $\hat{\beta}$ that produce the *same fitted values* \hat{y} .
- The model still predicts well, but the **individual coefficients are not uniquely defined**, so they cannot be interpreted.

To see this, fit three smaller models:

```
fit1 = smf.ols("y ~ x1 + x2", data=exact_collin_data).fit()
fit2 = smf.ols("y ~ x1 + x3", data=exact_collin_data).fit()
fit3 = smf.ols("y ~ x2 + x3", data=exact_collin_data).fit()

np.allclose(fit1.fittedvalues, fit2.fittedvalues)
np.allclose(fit2.fittedvalues, fit3.fittedvalues)
```

The fitted values are identical, but

```
fit1.params
fit2.params
fit3.params
```

give quite different coefficients. This is the hallmark of exact collinearity: **same predictions, wildly different coefficient stories**.

26.1.2 15.2 Collinearity in practice: highly correlated predictors

Exact collinearity is rare in real data. More commonly, we see **strong but not perfect correlation** between predictors. This is usually called *multicollinearity*.

The R notes use the `seatspos` dataset from the `faraway` package. In PyStatsV1 we will ship a CSV version of this dataset as `data/seatspos.csv`:

```
import pandas as pd

seatpos = pd.read_csv("data/seatpos.csv")
seatpos.head()
```

The response is `hipcenter` (seat position). Predictors include:

- Age – age in years,
- Weight – weight in pounds,
- Ht – standing height,
- HtShoes – height while wearing shoes,
- Seated, Arm, Thigh, Leg – various body measurements.

15.2.1 Pairwise correlation checks

A first check is to look at pairwise scatterplots and correlations:

```
import matplotlib.pyplot as plt

cols = ["Age", "Weight", "HtShoes", "Ht", "Seated", "Arm", "Thigh", "Leg"]
pd.plotting.scatter_matrix(seatpos[cols], figsize=(8, 8))
plt.tight_layout()

seatpos[cols].corr().round(2)
```

You should see extremely high correlations between many of the size-related variables, especially `HtShoes` and `Ht` (essentially the same measurement).

15.2.2 A “kitchen-sink” regression

Now fit a multiple regression with all predictors:

```
import statsmodels.formula.api as smf

hip_model = smf.ols("hipcenter ~ Age + Weight + HtShoes + Ht + "
                    "Seated + Arm + Thigh + Leg",
                    data=seatpos).fit()
print(hip_model.summary())
```

Typical pattern:

- The **overall F-test** (at the bottom of the summary) says the model is highly significant.
- Yet almost all **individual t-tests** for coefficients have large p-values.
- Some coefficients for very similar predictors can even have **opposite signs** (for example, `Ht` and `HtShoes`).

This is a classic symptom of collinearity:

- Together the predictors explain a lot of variation in `hipcenter`.
- But because they are highly correlated with each other, the model struggles to attribute variation to any single predictor.

15.2.3 Variance inflation factors (VIFs)

To understand the impact on standard errors, we use the **variance inflation factor**.

For a given predictor x_j , let R_j^2 be the R^2 from regressing x_j on all the *other* predictors. Then the variance of its coefficient $\hat{\beta}_j$ can be written as

$$\text{Var}(\hat{\beta}_j) = \sigma^2 \underbrace{\frac{1}{1 - R_j^2}}_{\text{variance inflation factor}} \cdot \frac{1}{S_{x_j x_j}},$$

where $S_{x_j x_j}$ is the sum of squares of x_j around its mean.

The **variance inflation factor (VIF)** is

$$\text{VIF}_j = \frac{1}{1 - R_j^2}.$$

If R_j^2 is close to 1 (x_j is well explained by other predictors), then VIF_j is large and the standard error of $\hat{\beta}_j$ is inflated.

In Python we can compute VIFs using statsmodels:

```
import numpy as np
from statsmodels.stats.outliers_influence import variance_inflation_factor

X = hip_model.model.exog
vif = pd.Series(
    [variance_inflation_factor(X, i) for i in range(X.shape[1])],
    index=hip_model.model.exog_names,
)
vif
```

You should see very large VIFs for several predictors (well above common rules of thumb such as 5 or 10). That tells us:

- The model's **coefficients are unstable and hard to interpret**.
- Small changes in the data can lead to large swings in estimated effects.

15.2.4 Collinearity and small perturbations

To see this instability, add a bit of random noise to the response and refit:

```
rng = np.random.default_rng(1337)
noise = rng.normal(loc=0, scale=5, size=len(seatpos))

hip_model_noise = smf.ols(
    "hipcenter_plus_noise ~ Age + Weight + HtShoes + Ht + "
    "Seated + Arm + Thigh + Leg",
    data=seatpos.assign(hipcenter_plus_noise=seatpos["hipcenter"] + noise),
).fit()

hip_model.params
hip_model_noise.params
```

You will often see:

- Coefficients change a lot, sometimes even **flipping sign**.

- But the **fitted values** are quite similar:

```
plt.scatter(hip_model.fittedvalues,
            hip_model_noise.fittedvalues,
            alpha=0.7)
plt.xlabel("Predicted hipcenter (original)")
plt.ylabel("Predicted hipcenter (with noise)")
plt.axline((0, 0), slope=1, color="k", linestyle="--")
plt.tight_layout()
```

Collinearity therefore:

- Hurts our ability to **explain** the relationship (unstable coefficients),
- But may have **much smaller effect on prediction error**.

15.2.5 A smaller, more stable model

Suppose we fit a simpler model using just a few predictors, for example Age, Arm, and Ht:

```
hip_model_small = smf.ols("hipcenter ~ Age + Arm + Ht",
                          data=seatpos).fit()
print(hip_model_small.summary())

X_small = hip_model_small.model.exog
vif_small = pd.Series(
    [variance_inflation_factor(X_small, i)
     for i in range(X_small.shape[1])],
    index=hip_model_small.model.exog_names,
)
vif_small
```

Now the VIFs are reasonable and the coefficient for Ht has a stable sign.

We can compare the large and small models with an F-test:

```
import statsmodels.api as sm

sm.stats.anova_lm(hip_model_small, hip_model)
```

Often you will find:

- The more complicated model does **not** provide a statistically significant improvement in fit.
- The **simpler model** is therefore preferred: easier to interpret and less sensitive to noise.

26.1.3 15.3 Partial correlation and added-variable plots

Suppose we are considering adding another predictor, say HtShoes, to the smaller model with Age, Arm, and Ht.

A useful diagnostic is the **partial correlation** between the candidate predictor and the response *after* removing the effect of the existing predictors.

Procedure:

1. Fit the *current* model and keep the residuals:

```
resid_y = hip_model_small.resid
```

2. Regress the *candidate* predictor on the existing predictors and keep its residuals:

```
ht_shoes_model_small = smf.ols(
    "HtShoes ~ Age + Arm + Ht",
    data=seatpos
).fit()
resid_ht_shoes = ht_shoes_model_small.resid
```

3. Compute the correlation:

```
np.corrcoef(resid_ht_shoes, resid_y)[0, 1]
```

If this partial correlation is close to 0, then once we know Age, Arm, and Ht, the remaining variation in HtShoes tells us almost nothing about the remaining variation in hipcenter. Adding HtShoes is unlikely to be helpful.

An **added-variable plot** visualizes the same idea:

```
plt.scatter(resid_ht_shoes, resid_y, alpha=0.8)
plt.axhline(0, linestyle="--", color="grey")
plt.axvline(0, linestyle="--", color="grey")

# regression line of residuals on residuals
line = smf.ols(
    "resid_y ~ resid_ht_shoes",
    data=pd.DataFrame({"resid_y": resid_y,
                       "resid_ht_shoes": resid_ht_shoes}),
).fit()
x_grid = np.linspace(resid_ht_shoes.min(), resid_ht_shoes.max(), 100)
y_grid = line.params["Intercept"] + line.params["resid_ht_shoes"] * x_grid
plt.plot(x_grid, y_grid)
plt.xlabel("Residuals of HtShoes (given Age, Arm, Ht)")
plt.ylabel("Residuals of hipcenter (given Age, Arm, Ht)")
plt.tight_layout()
```

A nearly horizontal cloud with a flat regression line indicates that the new predictor adds little explanatory power once the others are in the model.

26.1.4 15.4 How this connects to PyStatsV1

Collinearity is one of the main reasons why “more predictors” does not always mean “better model.”

In later PyStatsV1 chapters and case studies you will see:

- Scripts that compute and **report VIFs** alongside regression summaries.
- Examples where we deliberately drop or combine highly correlated predictors to stabilize coefficients.
- Model-comparison workflows that balance **good fit**, **interpretability**, and **low collinearity**.

When you adapt or extend PyStatsV1 code for your own data, you should:

- check correlations and VIFs early,
- be suspicious of models where small changes in the data flip coefficient signs or dramatically change magnitudes,
- prefer smaller, well-behaved models when the goal is explanation.

26.1.5 15.5 What you should take away

- **Exact collinearity** means some predictor is an exact linear combination of others. The design matrix is singular, the coefficients are not uniquely defined, but predictions can still be fine.
- **High collinearity** (multicollinearity) occurs when predictors are highly correlated. It inflates standard errors and makes coefficients unstable and hard to interpret.
- The **variance inflation factor (VIF)** quantifies how much collinearity inflates the variance of $\hat{\beta}_j$. VIFs above 5–10 are often used as warning flags.
- Collinearity can severely damage our ability to **explain** relationships, while having relatively little impact on **prediction error**.
- Tools such as **partial correlation** and **added–variable plots** help decide whether a new predictor is worth adding to a model that already has several correlated variables.
- When in doubt, prefer **simpler models** with reasonably low collinearity, especially when the goal is to communicate and interpret effects.

APPLIED STATISTICS WITH PYTHON – CHAPTER 16

27.1 Variable selection and model building

The previous chapter showed how **collinearity** between predictors can make regression models unstable. One common remedy is to **fit a smaller model**: drop some predictors so the remaining ones are less redundant.

That raises a new question:

Given many possible models, how do we choose a “good” one?

In the original R notes, this chapter uses tools like `AIC()`, `BIC()`, `step()`, and `regsubsets()` from the **faraway** and **leaps** packages. Here we will:

- translate those ideas into **Python-first** code using `statsmodels` and `scikit-learn`,
- separate **quality criteria** (AIC, BIC, adjusted R^2 , cross-validated RMSE) from **search procedures** (backward, forward, stepwise, exhaustive),
- use a couple of running examples (seat position and Auto MPG) to see model selection in action, and
- revisit the distinction between models for **explanation** and models for **prediction**.

Throughout, you can imagine that we already have two cleaned CSV files:

- `data/seatpos.csv` with a response `hipcenter` and several body measurements as predictors.
- `data/autompg.csv` with `mpg` and car attributes (cylinders, displacement, horsepower, weight, acceleration, model year, domestic / non-domestic).

27.2 16.1 Quality criteria: balancing fit and complexity

So far we have used **residual plots**, R^2 , and RMSE to judge a single model’s fit. If we try to *compare* models using plain R^2 or RMSE, we run into a problem:

Adding predictors can never make R^2 worse, and almost never makes RMSE worse. A huge, overfitted model will always win.

We need criteria that reward **good fit** but **penalize complexity**. In this chapter we will use four:

- **AIC** (Akaike Information Criterion)
- **BIC** (Bayesian Information Criterion)
- **Adjusted R^2**
- **Cross-validated RMSE** (for prediction)

In all cases, “smaller is better” except that **larger** adjusted R^2 is better.

27.2.1 16.1.1 AIC and BIC in Python

For a linear regression with n observations, p parameters (including the intercept), and residual sum of squares RSS, the R notes write AIC and BIC as

$$\text{AIC} = n \log\left(\frac{\text{RSS}}{n}\right) + 2p$$

$$\text{BIC} = n \log\left(\frac{\text{RSS}}{n}\right) + (\log n) p.$$

The first term measures fit; the second term penalizes model size.

In **statsmodels**, both are built in:

```
import pandas as pd
import statsmodels.api as sm

df = pd.read_csv("data/seatpos.csv")
y = df["hipcenter"]
X = df[["Age", "Weight", "HtShoes", "Ht", "Seated", "Arm", "Thigh", "Leg"]]
X = sm.add_constant(X)

model = sm.OLS(y, X).fit()

model.aic      # Akaike Information Criterion
model.bic      # Bayesian Information Criterion
```

Smaller AIC / BIC means “better” under that criterion. BIC usually prefers **smaller** models than AIC because its penalty $(\log n)p$ is larger than $2p$ once $\log n > 2$.

If you have RSS, n , and p directly, you can compute them by hand:

```
import numpy as np

def aic_from_rss(rss: float, n: int, p: int) -> float:
    return n * np.log(rss / n) + 2 * p

def bic_from_rss(rss: float, n: int, p: int) -> float:
    return n * np.log(rss / n) + np.log(n) * p
```

27.2.2 16.1.2 Adjusted R^2

Plain R^2 is

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}.$$

Adjusted R^2 adds a penalty for the number of parameters:

$$R^2_{\text{adj}} = 1 - \frac{\text{SSE}/(n-p)}{\text{SST}/(n-1)} = 1 - \frac{n-1}{n-p}(1 - R^2).$$

Unlike plain R^2 , adjusted R^2 can **decrease** when you add a useless predictor.

In **statsmodels** you get it for free:

```
model.rsquared      # R^2
model.rsquared_adj  # adjusted R^2
```

27.2.3 16.1.3 Cross-validated RMSE

AIC, BIC, and adjusted R^2 all use p explicitly. **Cross-validation** instead asks a predictive question:

How well would this model perform on new data?

The R notes focus on **leave-one-out cross-validation** (LOOCV):

1. For each observation i , fit the model **without** that point.
2. Predict $\hat{y}_{[i]}$ for the left-out point.
3. Compute the LOOCV residual $e_{[i]} = y_i - \hat{y}_{[i]}$.
4. Define

$$\text{RMSE}_{\text{LOOCV}} = \sqrt{\frac{1}{n} \sum_{i=1}^n e_{[i]}^2}.$$

In Python we usually use `scikit-learn` for cross-validation:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, LeaveOneOut

def loocv_rmse(X, y):
    model = LinearRegression()
    loo = LeaveOneOut()
    # By convention, cross_val_score *maximizes* a score, so we use
    # negative MSE and flip the sign.
    neg_mse_scores = cross_val_score(
        model, X, y,
        cv=loo,
        scoring="neg_mean_squared_error",
    )
    mse = -neg_mse_scores.mean()
    return np.sqrt(mse)

X = df[["Age", "Weight", "HtShoes", "Ht", "Seated", "Arm", "Thigh", "Leg"]]
y = df["hipcenter"]

loocv_rmse(X, y)
```

For larger data sets, **5-fold** or **10-fold** cross-validation is more common:

```
from sklearn.model_selection import KFold

def kfold_rmse(X, y, k=5, random_state=0):
    model = LinearRegression()
    kf = KFold(n_splits=k, shuffle=True, random_state=random_state)
    neg_mse_scores = cross_val_score(
        model, X, y,
```

(continues on next page)

(continued from previous page)

```

    cv=kf,
    scoring="neg_mean_squared_error",
)
mse = -neg_mse_scores.mean()
return np.sqrt(mse)

```

27.2.4 16.1.4 Which criterion should I use?

A rule of thumb:

- For **explanation** and **interpretable models**, prefer criteria that strongly penalize complexity:
 - BIC,
 - adjusted R^2 .
- For **prediction**, emphasize **cross-validated RMSE** (or another predictive metric). AIC can also be a compromise.

In the rest of the chapter we will often compare several criteria side-by-side.

27.2.5 16.1.5 R vs Python: quality criteria

Concept	R notes	Python (statsmodels / scikit-learn)
AIC / BIC	AIC(fit), BIC(fit)	fit.aic, fit.bic
Adjusted R^2	summary(fit)\$adj.r.squared	fit.rsquared_adj
LOOCV	manual function	with LeaveOneOut + cross_val_score (or manual loop)
RMSE	hatvalues(fit)	

27.3 16.2 Search procedures: which models to consider?

A **model selection procedure** has two ingredients:

1. A **quality criterion** (AIC, BIC, adjusted R^2 , cross-validated RMSE, ...).
2. A **search strategy** to explore the huge space of possible models.

With p candidate predictors, there are 2^p possible subsets. For $p = 8$, that is already 256 models; for $p = 15$ it is over 32,000.

The R chapter introduces four strategies:

- backward selection,
- forward selection,
- stepwise selection,
- exhaustive (all-subsets) search.

We will sketch Python versions using statsmodels.

27.3.1 Helper: fit a model on a given subset

We'll write a tiny helper that:

- takes a list of predictor names,
- fits a linear regression with an intercept,
- returns a chosen metric (AIC, BIC, adjusted R^2 , or CV-RMSE).

```
import itertools
from typing import List, Literal

Metric = Literal["aic", "bic", "adj_r2", "loocv_rmse"]

def evaluate_subset(df, response: str, predictors: List[str],
                   metric: Metric = "aic") -> float:
    """Fit OLS on a subset of predictors and return a scalar score."""
    y = df[response]
    X = sm.add_constant(df[predictors])

    fit = sm.OLS(y, X).fit()

    if metric == "aic":
        return fit.aic
    if metric == "bic":
        return fit.bic
    if metric == "adj_r2":
        # For consistency with AIC/BIC ("smaller is better"), we return
        # negative adjusted R^2 here.
        return -fit.rsquared_adj
    if metric == "loocv_rmse":
        # Use scikit-learn with the *same* predictors
        X_np = df[predictors].to_numpy()
        y_np = y.to_numpy()
        return loocv_rmse(X_np, y_np)

    raise ValueError(f"Unknown metric: {metric}")
```

27.3.2 16.2.1 Backward selection

Backward selection starts from the **full model** and repeatedly **drops** one predictor at a time if it improves the criterion.

Skeleton implementation:

```
def backward_selection(df, response: str, candidates: List[str],
                      metric: Metric = "aic"):
    remaining = list(candidates)
    best_score = evaluate_subset(df, response, remaining, metric)

    improved = True
    while improved and len(remaining) > 1:
        improved = False
        scores = []

        for predictor in remaining:
```

(continues on next page)

(continued from previous page)

```

        trial = [p for p in remaining if p != predictor]
        score = evaluate_subset(df, response, trial, metric)
        scores.append((score, predictor, trial))

    # Pick the *best* (smallest score) among the candidates
    score, predictor_to_drop, trial = min(scores, key=lambda t: t[0])

    if score + 1e-8 < best_score: # small tolerance
        best_score = score
        remaining = trial
        improved = True

    return remaining, best_score

```

Example (using the seat position data):

```

seat_df = pd.read_csv("data/seatpos.csv")
predictors = ["Age", "Weight", "HtShoes", "Ht", "Seated",
              "Arm", "Thigh", "Leg"]

selected_aic, score_aic = backward_selection(
    seat_df, "hipcenter", predictors, metric="aic"
)
selected_bic, score_bic = backward_selection(
    seat_df, "hipcenter", predictors, metric="bic"
)

print("Backward AIC:", selected_aic)
print("Backward BIC:", selected_bic)

```

Typically:

- AIC keeps **more** predictors.
- BIC keeps a **smaller** set with nearly as good predictive performance.

27.3.3 16.2.2 Forward selection

Forward selection starts from the **intercept-only** model and repeatedly **adds** the predictor that most improves the criterion.

```

def forward_selection(df, response: str, candidates: List[str],
                     metric: Metric = "aic"):
    remaining = list(candidates)
    selected: List[str] = []
    best_score = evaluate_subset(df, response, selected or [remaining[0]], metric)

    improved = True
    while improved and remaining:
        improved = False
        scores = []

        for predictor in remaining:
            trial = selected + [predictor]

```

(continues on next page)

(continued from previous page)

```

    score = evaluate_subset(df, response, trial, metric)
    scores.append((score, predictor, trial))

    score, predictor_to_add, trial = min(scores, key=lambda t: t[0])

    if score + 1e-8 < best_score:
        best_score = score
        selected = trial
        remaining.remove(predictor_to_add)
        improved = True

    return selected, best_score

```

Forward and backward often find *similar* but not identical models.

27.3.4 16.2.3 Stepwise selection

Stepwise selection looks **both directions** at each step:

- try adding each unused predictor;
- also try dropping each currently-included predictor;
- commit the change (add or drop) that best improves the criterion.

Conceptually it is:

```

def stepwise_selection(df, response: str, candidates: List[str],
                      metric: Metric = "aic"):
    selected: List[str] = []
    best_score = float("inf")

    while True:
        changed = False

        # 1. Try adding
        add_scores = []
        for p in candidates:
            if p in selected:
                continue
            trial = selected + [p]
            score = evaluate_subset(df, response, trial, metric)
            add_scores.append((score, ("add", p), trial))

        # 2. Try dropping
        drop_scores = []
        for p in selected:
            trial = [q for q in selected if q != p]
            if not trial:
                continue
            score = evaluate_subset(df, response, trial, metric)
            drop_scores.append((score, ("drop", p), trial))

        all_scores = add_scores + drop_scores

```

(continues on next page)

(continued from previous page)

```

    if not all_scores:
        break

    score, (action, p), trial = min(all_scores, key=lambda t: t[0])

    if score + 1e-8 < best_score:
        best_score = score
        selected = trial
        changed = True

    if not changed:
        break

    return selected, best_score

```

Stepwise tends to behave like a hybrid between forward and backward.

27.3.5 16.2.4 Exhaustive search (all subsets)

For **small** numbers of predictors ($p \leq 10$ or so), you can simply check **all** subsets and pick the best according to a given criterion.

```

def exhaustive_search(df, response: str, candidates: List[str],
                     metric: Metric = "aic"):
    best_score = float("inf")
    best_subset: List[str] | None = None

    for k in range(1, len(candidates) + 1):
        for subset in itertools.combinations(candidates, k):
            subset = list(subset)
            score = evaluate_subset(df, response, subset, metric)
            if score < best_score:
                best_score = score
                best_subset = subset

    return best_subset, best_score

```

This mirrors the R chapter's use of `regsubsets()` from the **leaps** package.

27.4 16.3 Example: seat position (AIC vs BIC vs LOOCV)

In the R version, the **seatpos** data from the **faraway** package is used to compare models for the response **hipcenter**. Assume we have `data/seatpos.csv` with the following columns:

- **hipcenter** – driver hip center (response),
- Age, Weight, HtShoes, Ht, Seated, Arm, Thigh, Leg – body measurements.

A typical workflow in Python:

1. Fit the **full** model with all eight predictors.
2. Run backward selection with **AIC** and **BIC**.
3. Compare adjusted R^2 and LOOCV RMSE across the full and selected models.

Sketch:

```
full_predictors = ["Age", "Weight", "HtShoes", "Ht",
                  "Seated", "Arm", "Thigh", "Leg"]

# Full model
y = seat_df["hipcenter"]
X_full = sm.add_constant(seat_df[full_predictors])
full_fit = sm.OLS(y, X_full).fit()

# Backward AIC / BIC
back_aic, _ = backward_selection(seat_df, "hipcenter", full_predictors, "aic")
back_bic, _ = backward_selection(seat_df, "hipcenter", full_predictors, "bic")

# Fit selected models
X_aic = sm.add_constant(seat_df[back_aic])
X_bic = sm.add_constant(seat_df[back_bic])

fit_aic = sm.OLS(y, X_aic).fit()
fit_bic = sm.OLS(y, X_bic).fit()

# Compare criteria
print("Full    adj R^2:", full_fit.rsquared_adj)
print("AIC     adj R^2:", fit_aic.rsquared_adj)
print("BIC     adj R^2:", fit_bic.rsquared_adj)

print("Full    LOOCV RMSE:", loocv_rmse(X_full.iloc[:, 1:].to_numpy(), y.to_numpy()))
print("AIC     LOOCV RMSE:", loocv_rmse(X_aic.iloc[:, 1:].to_numpy(), y.to_numpy()))
print("BIC     LOOCV RMSE:", loocv_rmse(X_bic.iloc[:, 1:].to_numpy(), y.to_numpy()))
```

Typically you will see:

- both AIC and BIC improve **adjusted R^2** and **LOOCV RMSE** compared to the full model;
- BIC selects a **smaller** subset with only a small loss (or even a gain) in predictive performance.

27.5 16.4 Higher-order terms: Auto MPG example

The final part of the R chapter returns to the **Auto MPG** data and considers:

- quadratic terms such as disp^2 , hp^2 , wt^2 , acc^2 ,
- all two-way interactions between first-order terms.

The idea is to show that:

- we can start with a **very rich model** (many terms),
- then use backward selection (often with BIC) to find a smaller model that still predicts well,
- and compare LOOCV RMSE between the full and selected models.

In Python we can use **formula syntax** with statsmodels:

```
import numpy as np
import statsmodels.formula.api as smf
```

(continues on next page)

(continued from previous page)

```

auto_df = pd.read_csv("data/autompg.csv")

# Log-transform the response as in the R notes
auto_df["log_mpg"] = np.log(auto_df["mpg"])

big_formula = (
    "log_mpg ~ (cyl + disp + hp + wt + acc + year + domestic) ** 2"
    " + I(disp ** 2) + I(hp ** 2) + I(wt ** 2) + I(acc ** 2)"
)

big_fit = smf.ols(big_formula, data=auto_df).fit()
big_fit.summary()

```

Now apply backward selection on this rich model, using BIC or AIC as the criterion. The search logic is identical; the only difference is that the candidate “predictors” are now terms from the formula (main effects, quadratics, interactions) rather than raw columns.

Python does not automatically enforce **hierarchy** (keeping main effects whenever an interaction is included), so in practice you may want to:

- define groups of terms that must stay together,
- or rely on domain knowledge to simplify the final model.

The main lesson:

Sometimes a slightly richer model (interactions, quadratic terms) combined with a strong penalty (BIC or cross-validation) gives the best predictive performance.

27.6 16.5 Explanation versus prediction

A central theme of the R chapter is the distinction between models used for:

- **Explanation** – understanding how predictors relate to the response.
- **Prediction** – making accurate forecasts for new observations.

27.6.1 Explanation

For explanation we care about:

- **Interpretability**: How does each predictor affect the response, holding others fixed?
- **Parsimony**: Smaller models are easier to explain and reason about.
- **Causality vs association**: Does a predictor *cause* changes in the response, or is it just correlated?

We normally:

- favor **BIC** or adjusted R^2 to penalize complexity,
- inspect diagnostic plots (from previous chapters),
- remember that with **observational data** we detect **associations**, not necessarily **causal effects**.

27.6.2 Prediction

For pure prediction we care about:

- **out-of-sample error** (cross-validated RMSE),
- robustness to new data.

We are less worried about:

- exact p-values,
- whether effects are causal,
- whether the model is small.

A model can predict well even if it is not a realistic description of the data-generating process.

A classic analogy: **shoe size** and **reading level** in children are highly correlated (both driven by age). Shoe size does not *cause* reading ability, but knowing shoe size still helps predict reading level. For prediction that is fine; for explanation it is misleading.

27.7 16.6 How this connects to PyStatsV1

In PyStatsV1 we will continually revisit **model selection**:

- Choosing **which predictors to include** in a regression or GLM.
- Comparing different model families (linear vs logistic vs Poisson).
- Balancing **interpretability** against **predictive performance**.

This chapter gives you a toolkit that we can plug into many later examples:

- computing AIC / BIC / adjusted R^2 ,
- using cross-validation to estimate predictive error,
- exploring backward / forward / stepwise / exhaustive search.

In applied case studies we will often start from a **rich model** suggested by subject-matter knowledge, then use these tools to trim it down to something:

- scientifically interpretable,
- numerically stable,
- and reasonably accurate for prediction.

27.8 16.7 What you should take away

By the end of this chapter (and its R + Python versions), you should be able to:

- Explain why **plain R^2 and RMSE** are not enough for model comparison.
- Compute and interpret:
 - AIC and BIC,
 - **adjusted R^2** ,
 - **cross-validated RMSE** (LOOCV or K-fold).
- Implement or adapt **search procedures** in Python:
 - backward selection,

- forward selection,
 - stepwise selection,
 - exhaustive search for small p .
- Use these procedures on real data sets (like **seat position** and **Auto MPG**) to find reasonable models.
- Distinguish between models aimed at **explanation** and those aimed at **prediction**, and choose criteria that match your goal.
- Recognize that:
 - strong penalties (BIC, adjusted R^2) often yield compact, interpretable models,
 - cross-validation is the gold standard for estimating predictive error,
 - observational data supports **association**, not automatic causal claims.

In later PyStatsV1 chapters, these model-selection tools will support both:

- **observational** regression analyses, and
- upcoming material on **experimental design**, where we can combine good designs with good models to make stronger causal statements.

APPLIED STATISTICS WITH PYTHON – CHAPTER 17

28.1 Logistic regression and classification

So far, our regression chapters have focused on *numeric* response variables and ordinary least squares (OLS). In many applications, however, the response is binary:

- disease vs no disease,
- spam vs non-spam,
- success vs failure, and so on.

In this chapter we:

- introduce **generalized linear models (GLMs)** as an extension of OLS,
- develop **logistic regression** for binary responses,
- show how to **fit, interpret, and test** logistic models in Python, and
- use logistic regression as a **classifier**, including evaluation metrics such as misclassification rate, sensitivity, and specificity.

Throughout, we will mirror the R notes but work in Python using:

- **numpy** and **pandas** for data handling,
- **statsmodels** for logistic regression and GLMs, and
- **scikit-learn** for classification workflows and cross-validation.

28.2 17.1 Generalized linear models

Ordinary linear regression assumes

- a *Normal* distribution for the response given the predictors, and
- a mean that is a **linear combination** of the predictors:

$$Y \mid X = x \sim N(\mu(x), \sigma^2), \quad \mu(x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_{p-1} x_{p-1}.$$

A **generalized linear model (GLM)** keeps the linear predictor

$$\eta(x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_{p-1} x_{p-1}$$

but allows:

- different choices of distribution for $Y \mid X = x$, and

- a **link function** $g(\cdot)$ connecting the linear predictor to the mean:

$$\eta(x) = g(E[Y | X = x]).$$

For example:

- **Linear regression**
 - Distribution: Normal
 - Link: identity $g(\mu) = \mu$
- **Poisson regression**
 - Distribution: Poisson (counts)
 - Link: $\log g(\lambda) = \log \lambda$
- **Logistic regression**
 - Distribution: Bernoulli (binary)
 - Link: $\text{logit } g(p) = \log\{p/(1-p)\}$

28.3 17.2 Binary responses and the logistic model

Suppose we have a binary response coded as

$$Y = \begin{cases} 1, & \text{event occurs ("yes", "spam", "disease")} \\ 0, & \text{otherwise ("no", "not spam", "no disease").} \end{cases}$$

Define

$$p(x) = P(Y = 1 | X = x).$$

With a **logistic regression model** we assume

$$\log\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_{p-1} x_{p-1}.$$

The left-hand side is the **log-odds** (logit). Applying the inverse logit gives

$$p(x) = \frac{\exp\{\eta(x)\}}{1 + \exp\{\eta(x)\}} = \frac{1}{1 + \exp\{-\eta(x)\}}.$$

This guarantees $0 < p(x) < 1$, which is exactly what we want from a probability.

28.3.1 Where is the error term?

In the OLS model we write

$$Y = \beta_0 + \beta_1 x_1 + \cdots + \beta_q x_q + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2),$$

or equivalently

$$Y | X = x \sim N(\mu(x), \sigma^2).$$

The Normal distribution has two parameters, $\mu(x)$ and σ^2 , so we estimate both.

In logistic regression the conditional distribution is **Bernoulli** with mean $p(x)$. The distribution has a *single* parameter, so we only need to estimate $p(x)$ (through the β coefficients). There is no separate σ^2 parameter.

28.3.2 17.2.1 Why not OLS on a 0/1 outcome?

If $Y \in \{0, 1\}$, the conditional mean is

$$E[Y \mid X = x] = P(Y = 1 \mid X = x) = p(x).$$

You might think we can simply fit an ordinary linear regression of Y on X and interpret the fitted values as estimated probabilities. The problem:

- the OLS fitted line is linear in x , so fitted values can be **less than 0** or **greater than 1**, which makes no sense as probabilities;
- the Normal error assumption is a poor match to a Bernoulli variable.

Logistic regression solves both issues: probabilities are constrained to $(0, 1)$ and the Bernoulli model correctly reflects the 0/1 nature of the data.

28.3.3 17.2.2 Simulating logistic data in Python

Here is a direct translation of the R simulation used in the original notes.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)

def sim_logistic_data(sample_size=25, beta_0=-2.0, beta_1=3.0) -> pd.DataFrame:
    x = rng.normal(size=sample_size)
    eta = beta_0 + beta_1 * x
    p = 1 / (1 + np.exp(-eta))          # inverse logit
    y = rng.binomial(n=1, p=p, size=sample_size)
    return pd.DataFrame({"y": y, "x": x})

df = sim_logistic_data()
df.head()
```

We can now compare an OLS fit to a logistic fit:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

# ordinary least squares
ols_mod = smf.ols("y ~ x", data=df).fit()

# logistic regression (GLM with binomial family + logit link)
logit_mod = smf.glm(
    "y ~ x", data=df,
    family=sm.families.Binomial()
).fit()

x_grid = np.linspace(df["x"].min(), df["x"].max(), 200)
grid = pd.DataFrame({"x": x_grid})

plt.scatter(df["x"], df["y"], s=20)
plt.plot(x_grid, ols_mod.predict(grid), label="OLS")
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_grid, logit_mod.predict(grid), linestyle="--", label="Logistic")
plt.xlabel("x")
plt.ylabel("Estimated probability")
plt.legend()
plt.grid(True)
```

The logistic curve stays between 0 and 1; the OLS line does not.

28.4 17.3 Fitting logistic regression in Python

We use `statsmodels` to estimate logistic regression models. The syntax parallels `glm()` in R.

28.4.1 Basic fit

```
import statsmodels.formula.api as smf
import statsmodels.api as sm

# binary response y, predictors x1, x2, ...
logit_mod = smf.glm(
    "y ~ x1 + x2",
    data=df,
    family=sm.families.Binomial()
).fit()

print(logit_mod.summary())
```

Key outputs:

- coefficient estimates $\hat{\beta}_j$,
- standard errors and Wald z statistics,
- approximate p-values for tests $H_0 : \beta_j = 0$,
- deviance and information criteria (AIC / BIC).

28.4.2 Wald tests

A single-parameter hypothesis

$$H_0 : \beta_j = 0 \quad \text{vs} \quad H_1 : \beta_j \neq 0$$

is tested with a **Wald statistic**

$$z = \frac{\hat{\beta}_j}{\text{SE}(\hat{\beta}_j)} \approx N(0, 1),$$

reported in the summary as `z` and `P>|z|`.

28.4.3 Likelihood-ratio tests

To compare a **reduced** model to a **full** model, we can use a likelihood-ratio test (LRT), the GLM analogue of the regression F-test.

```

reduced = smf.glm("y ~ x1 + x2", data=df,
                  family=sm.families.Binomial()).fit()

full = smf.glm("y ~ x1 + x2 + x3 + x4", data=df,
               family=sm.families.Binomial()).fit()

lr_stat, lr_pvalue, df_diff = full.compare_lr_test(reduced)
print(lr_stat, df_diff, lr_pvalue)

```

Here:

- `lr_stat` is the deviance difference D ,
- `df_diff` is the difference in degrees of freedom (number of parameters),
- `lr_pvalue` is the p-value under a $\chi^2_{df_diff}$ approximation.

28.4.4 17.3.1 SAheart example: coronary heart disease

The R notes use the SAheart dataset from ElemStatLearn. In PyStatsV1 we will assume a CSV version is available, for example:

- `data/sa_heart.csv`

with columns:

- `chd` – coronary heart disease indicator (1 = present, 0 = absent),
- `sbp` – systolic blood pressure,
- `tobacco` – lifetime tobacco (kg),
- `ldl` – low density lipoprotein cholesterol,
- `adiposity`, `famhist`, `typea`, `obesity`, `alcohol`, `age`.

Reading and preparing the data:

```

sa = pd.read_csv("data/sa_heart.csv")

# make sure chd is 0/1 integers
sa["chd"] = sa["chd"].astype(int)

```

Single-predictor model (LDL only)

```

chd_ldl = smf.glm(
    "chd ~ ldl", data=sa,
    family=sm.families.Binomial()
).fit()

print(chd_ldl.summary())

```

A positive coefficient for `ldl` indicates that higher LDL is associated with higher probability of CHD.

Additive model with all predictors

```
chd_add = smf.glm(
    "chd ~ sbp + tobacco + ldl + adiposity + C(famhist)"
    " + typea + obesity + alcohol + age",
    data=sa,
    family=sm.families.Binomial()
).fit()

chd_add.summary()
```

Note: C(famhist) treats famhist as a categorical predictor.

Comparing models with an LRT:

```
lr_stat, lr_pvalue, df_diff = chd_add.compare_lr_test(chd_ldl)
print(f"LRT stat = {lr_stat:.3f}, df = {df_diff}, p = {lr_pvalue:.3g}")
```

A very small p-value suggests that the larger additive model explains CHD substantially better than LDL alone.

Variable selection (briefly)

For a quick backward stepwise fit using AIC we can loop over predictors or use a small helper; for now we simply note that:

- AIC/BIC from `result.aic / result.bic` can be compared across models.
- We prefer models that balance **good fit** (low deviance) with **parsimony** (few predictors).

Confidence intervals for coefficients

```
chd_sel = chd_add # for now, suppose this is our selected model
ci_99 = chd_sel.conf_int(alpha=0.01)
ci_99.columns = ["lower", "upper"]
ci_99
```

These are profile-likelihood intervals, analogous to the R output from `confint`.

Confidence intervals for mean response

For a new patient we often want an interval for the *predicted probability* of CHD.

```
new_obs = pd.DataFrame(
    {
        "sbp": [148.0],
        "tobacco": [5.0],
        "ldl": [12.0],
        "adiposity": [31.23],
        "famhist": ["Present"],
        "typea": [47],
        "obesity": [28.50],
        "alcohol": [23.89],
        "age": [60],
    }
)
```

(continues on next page)

(continued from previous page)

```
# get eta and its standard error
pred = chd_sel.get_prediction(new_obs)
summary = pred.summary_frame() # includes mean, mean_ci_lower, mean_ci_upper
summary[["mean_ci_lower", "mean", "mean_ci_upper"]]
```

By default, statsmodels returns intervals on the **link scale** (log-odds). Using `transform=True` we can instead request intervals already back-transformed to probabilities:

```
summary_prob = chd_sel.get_prediction(new_obs).summary_frame(transform=True)
summary_prob[["mean_ci_lower", "mean", "mean_ci_upper"]]
```

This provides a confidence interval for $p(x)$ between 0 and 1.

28.5 17.4 Logistic regression as a classifier

Up to now, we have focused on modelling the probability $p(x)$. To turn logistic regression into a **classifier**, we map each observation to a class label.

If we knew the true probabilities, the **Bayes classifier**

$$C_B(x) = \arg \max_k P(Y = k \mid X = x)$$

would minimize the probability of misclassification. For a binary response this reduces to:

$$C_B(x) = \begin{cases} 1, & p(x) > 0.5, \\ 0, & \text{otherwise.} \end{cases}$$

In practice we replace $p(x)$ by its estimate $\hat{p}(x)$ from the logistic model and use the same rule.

28.5.1 Misclassification rate

Given predictions $\hat{C}(x_i)$ on a dataset, the **misclassification rate** is

$$\text{Misclass} = \frac{1}{n} \sum_{i=1}^n I\{y_i \neq \hat{C}(x_i)\}.$$

Equivalently, accuracy is $1 - \text{Misclass}$.

28.5.2 Sensitivity and specificity

A confusion matrix for a binary classifier records:

- true positives (TP),
- false positives (FP),
- true negatives (TN),
- false negatives (FN).

Two important metrics are:

- **Sensitivity** (recall, true positive rate)

$$\text{Sens} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

- **Specificity** (true negative rate)

$$\text{Spec} = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

Changing the probability cutoff c in

$$\hat{C}(x) = \begin{cases} 1, & \hat{p}(x) > c, \\ 0, & \hat{p}(x) \leq c, \end{cases}$$

trades off sensitivity and specificity:

- lowering c increases sensitivity but decreases specificity,
- raising c does the opposite.

In practice, the “best” cutoff depends on the costs of false positives vs false negatives.

28.5.3 17.4.1 Spam example: email classification

The R notes use the classic spam dataset from the UCI machine learning repository. In PyStatsV1 we can assume a CSV file

- data/spam.csv

with a binary column type ("spam" vs "nonspam") and many engineered text features.

Train / test split

Using scikit-learn:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    confusion_matrix, accuracy_score, recall_score
)

spam = pd.read_csv("data/spam.csv")

X = spam.drop(columns=["type"])
y = (spam["type"] == "spam").astype(int) # 1 = spam, 0 = non-spam

X_trn, X_tst, y_trn, y_tst = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

Fit several models of increasing complexity:

```
# simple model using only capitalTotal
caps_mod = LogisticRegression(max_iter=1000)
caps_mod.fit(X_trn[["capitalTotal"]], y_trn)

# a small hand-picked model
few_features = ["edu", "money", "capitalTotal", "charDollar"]
sel_mod = LogisticRegression(max_iter=1000)
sel_mod.fit(X_trn[few_features], y_trn)
```

(continues on next page)

(continued from previous page)

```
# additive model using all predictors
full_mod = LogisticRegression(
    max_iter=1000, n_jobs=-1
)
full_mod.fit(X_trn, y_trn)

# (an "over-parametrized" model with interactions would usually be handled
# via feature engineering; we omit it here for brevity.)
```

Cross-validated misclassification rate

Instead of relying on training error, use cross-validation:

```
from sklearn.model_selection import cross_val_score

def cv_misclass(model, X, y, cv=5):
    acc = cross_val_score(model, X, y, cv=cv, scoring="accuracy")
    return 1 - acc.mean()

cv_caps = cv_misclass(caps_mod, X_trn[["capitalTotal"]], y_trn)
cv_sel = cv_misclass(sel_mod, X_trn[few_features], y_trn)
cv_full = cv_misclass(full_mod, X_trn, y_trn)

print(cv_caps, cv_sel, cv_full)
```

Typically you will see:

- the very simple model **underfits** (higher misclassification),
- a moderate-size model performs better,
- an extremely complex model risks **overfitting**.

Confusion matrix on the test set

Pick a preferred model (say `full_mod`) and evaluate it on the held-out test set:

```
# predicted probabilities for class 1 (spam)
p_hat = full_mod.predict_proba(X_tst)[: , 1]

# default cutoff 0.5
y_pred_50 = (p_hat > 0.5).astype(int)

cm_50 = confusion_matrix(y_tst, y_pred_50)
acc_50 = accuracy_score(y_tst, y_pred_50)
sens_50 = recall_score(y_tst, y_pred_50) # sensitivity = recall for positive class

# specificity needs a small helper
tn, fp, fn, tp = cm_50.ravel()
spec_50 = tn / (tn + fp)

print("Confusion matrix (c = 0.5):")
print(cm_50)
```

(continues on next page)

(continued from previous page)

```
print(f"Accuracy = {acc_50:.3f}, Sensitivity = {sens_50:.3f}, Specificity = {spec_50:.3f}
↪")
```

Changing the cutoff to 0.1 or 0.9 illustrates the trade-off:

```
for c in [0.1, 0.5, 0.9]:
    y_pred = (p_hat > c).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_tst, y_pred).ravel()
    acc = accuracy_score(y_tst, y_pred)
    sens = recall_score(y_tst, y_pred)
    spec = tn / (tn + fp)
    print(f"Cutoff {c:.1f}: acc={acc:.3f}, sens={sens:.3f}, spec={spec:.3f}")
```

This matches the behaviour described in the R notes: lower cutoffs favour sensitivity (catching more spam) at the cost of more false positives; higher cutoffs do the opposite.

28.6 17.5 How this connects to PyStatsV1

Logistic regression and GLMs will reappear throughout PyStatsV1:

- Many modern models used in science, health, and industry are **generalized linear models**, not just ordinary linear regression.
- In later chapters on **experimental data and causal questions**, logistic regression provides a natural way to model binary outcomes (e.g., success vs failure) while controlling for covariates.
- For applied work, logistic regression often serves as a **baseline classifier** before moving to more complex machine-learning methods. PyStatsV1 will show how to compare logistic models with tree-based and other algorithms.
- The ideas of **link functions**, **likelihood-based inference**, and **classification metrics** generalize to many other models (Poisson, multinomial, etc.).

Exercises and examples built on this chapter will encourage you to:

- fit and interpret logistic models for real binary outcomes,
- compare OLS and logistic regression on the same dataset,
- practice using deviance, AIC, and cross-validated misclassification to choose between models, and
- think carefully about which metrics (accuracy, sensitivity, specificity) are appropriate for a given applied problem.

28.7 17.6 What you should take away

By the end of this chapter (and its R + Python versions), you should be able to:

- Explain how **generalized linear models** extend ordinary linear regression.
- Define and work with the **logistic regression model**:
 - log-odds, odds, probabilities,
 - logit and inverse-logit (sigmoid) transformations.
- Fit logistic regression models in Python using **statsmodels** and **scikit-learn**.
 - interpret coefficients in terms of log-odds and probabilities;
 - compute Wald tests and likelihood-ratio tests;

- construct confidence intervals for coefficients and mean response.
- Use logistic regression as a **classifier**:
 - obtain class probabilities $\hat{p}(x)$,
 - convert them to labels using a cutoff,
 - compute misclassification rate, sensitivity, and specificity,
 - understand how changing the cutoff trades off false positives vs false negatives.
- Recognize that many ideas from linear regression (model comparison, variable selection, diagnostics) carry over almost unchanged to the logistic setting.

In short: logistic regression is your first step into the broader world of GLMs and classification methods, and a core tool for applied statistics with PyStatsV1.

APPLIED STATISTICS WITH PYTHON – CHAPTER 18

29.1 Beyond: where to go after this mini-book

You’ve worked through a full mini-sequence on regression:

- basic Python and R workflows,
- simple and multiple linear regression,
- diagnostics, transformations, and model building,
- logistic regression, ANOVA, and experimental design ideas.

That already covers a large chunk of what many “Applied Regression” courses offer. But it’s really just the start.

In this chapter we sketch possible next steps and how PyStatsV1 can support them. Think of this as a **roadmap**, not a checklist: you do *not* need to explore everything here. Pick the paths that match your goals and curiosity.

29.2 18.1 Where you can go next

Broadly, there are three directions you can grow:

- **Deeper modeling.** More regression variants, more careful inference, and richer models for complex data.
- **Stronger computing skills.** Better data workflows, reproducible reports, and tools for working with larger or messier datasets.
- **Domain-focused practice.** Applying these ideas in fields like psychology, ecology, economics, sports science, public health, or business analytics.

PyStatsV1 is designed to help you bridge **R Python** and connect textbook ideas to code, so you can move along any of these paths with less friction.

29.3 18.2 Python ecosystem: beyond the basics

In these notes we mainly used:

- **numpy** for arrays and numerical work,
- **pandas** for tabular data,
- **statsmodels** for regression and ANOVA,
- **matplotlib** for plotting.

From here you might explore:

- **SciPy** for numerical optimization, distributions, and signal processing.

- **Seaborn** or **plotnine** for higher-level, statistically oriented visualizations.
- **scikit-learn** for predictive modeling: cross-validation, pipelines, regularization, trees, ensembles, etc.
- **JupyterLab** or **VS Code** for a smoother notebook / editor workflow.

PyStatsV1 later chapters and case studies will assume you are comfortable moving between plain Python scripts, notebooks, and command-line tools.

29.4 18.3 R + Python “dual citizenship”

Many applied statistics resources are still written with R in mind. Rather than choosing one language forever, it can be powerful to become a **bilingual analyst**:

- Use **R** when you want: * quick, high-level modeling with tidyverse-style data pipelines; * packages that are deeply integrated with specific scientific domains; * RMarkdown / Quarto documents and Shiny apps.
- Use **Python** when you want: * to integrate statistics into larger software systems; * access to the broader machine-learning and data-science ecosystem; * easier deployment to production systems and web backends.

The cross-language patterns in this mini-book (formulas, model objects, simulation code) are meant to make it easy to translate between the two.

29.5 18.4 Tidy data and data workflows

In many of our examples, the data was already “clean”: each row was a single observation, each column a variable, with no missing values or awkward encodings.

Real projects are rarely that kind.

A large part of practical statistics is:

- reshaping data between **wide** and **long** forms;
- handling missing values and outliers;
- joining multiple tables; and
- encoding dates, times, and categorical variables consistently.

In Python, this often means getting comfortable with:

- pandas methods like `melt`, `pivot`, `merge`, and `groupby`,
- writing small, reusable helper functions for common cleaning steps,
- documenting your choices so analyses remain reproducible.

Later PyStatsV1 material will lean more on these “data tidying” skills.

29.6 18.5 Visualization: telling the story

Throughout the chapters we used relatively simple plots: scatterplots, line plots, residual plots, and a few specialized diagnostics.

To go further, you could:

- Learn a **grammar-of-graphics** style library (`plotnine` in Python or `ggplot2` in R) to build complex plots from a small set of ideas (geoms, aesthetics, facets, scales).
- Practice turning model output into **story-driven graphics**: prediction bands, effect plots, partial dependence plots, and before/after comparisons.

- Experiment with **interactive** visualizations for teaching or exploratory work using tools like Altair, Bokeh, or Plotly.

A good exercise: re-implement the regression diagnostics from earlier chapters using a different visualization library and compare what feels easier or harder.

29.7 18.6 Reproducible reports and small web apps

Statistics becomes much more valuable when results can be **shared and re-run** easily:

- For **reports and notes**, you can use: * Jupyter notebooks exported to HTML or PDF, * Quarto documents that mix code and prose in either R or Python, * plain Markdown + Makefiles (as in PyStatsV1) for lightweight automation.
- For **interactive exploration**, you might try: * **Streamlit** or **Dash** in Python, * **Shiny** in R.

A natural extension of PyStatsV1 is to wrap some of the core examples (e.g. Auto MPG, seat position, logistic regression case studies) in small web apps where sliders and dropdowns control model inputs.

29.8 18.7 Experimental design and causal questions

In Chapter 12 we drew a sharp line between **observational** and **experimental** data, and noted that regression alone cannot magically answer causal questions.

To go further you might explore:

- **Classical experimental design**: randomized controlled trials, blocking, factorial designs, and power calculations for experiments.
- **A/B testing** and online experimentation: how tech companies use controlled experiments to choose between design or policy options.
- **Causal inference**: potential outcomes, matching, instrumental variables, and graphical approaches (causal DAGs).

For PyStatsV1, this means:

- case studies where we deliberately distinguish “what the regression says” from “what we’re allowed to conclude causally”,
- simulated experiments where we know the ground truth and can check whether our methods recover it.

29.9 18.8 Machine learning and predictive modeling

Logistic regression is already a simple **classification** method. Many modern machine-learning tools generalize the same ideas:

- regularized linear models (ridge, lasso, elastic net),
- tree-based methods (random forests, gradient boosting),
- support vector machines and kernels,
- cross-validation for honest assessment of predictive performance.

Python’s `sklearn` makes it relatively easy to:

- wrap preprocessing and modeling in **pipelines**,
- tune hyperparameters with grid search or randomized search, and

- evaluate models using cross-validated metrics.

One good next step is to re-visit familiar datasets (Auto MPG, logistic regression examples) and compare simple statistical models to more flexible machine-learning models, being explicit about the trade-off between **insight** and **pure predictive accuracy**.

29.10 18.9 Time series and dependent data

All of our regression work assumed **independent** observations. Many real datasets are not:

- daily sales or web traffic,
- sensor readings over time,
- repeated measurements on the same individual.

Time series analysis introduces tools like:

- autoregressive and moving-average models (AR, MA, ARIMA),
- state-space and Kalman filter models,
- models with seasonal patterns and trend.

Python and R both have rich ecosystems for time series; the main conceptual shift is learning to think about **serial dependence** and forecasting rather than treating each row as unrelated to the others.

29.11 18.10 Bayesian statistics and probabilistic programming

In this mini-book we took a **frequentist** perspective: parameters are fixed, data are random, and uncertainty is summarized with confidence intervals and p-values.

A complementary view is **Bayesian**, where:

- parameters are treated as random quantities with prior distributions,
- inference is performed via posterior distributions,
- uncertainty is expressed as credible intervals and full probability statements about unknowns.

Modern **probabilistic programming** tools (for example, *Stan* via `cmdstanpy` or `pystan`, or Python libraries such as `pymc`) make it possible to:

- write models in a domain-focused way,
- combine complex likelihoods with informative priors,
- propagate uncertainty through hierarchical models.

If you enjoyed the simulation-based arguments in earlier chapters, Bayesian methods are a natural next step.

29.12 18.11 High-performance and large-scale computing

Once models get large or data get big, you may need to think about performance:

- vectorizing and broadcasting operations in NumPy instead of writing Python loops,
- using **Numba** or **Cython** to accelerate critical sections,
- offloading heavy linear algebra to GPUs when appropriate,
- working with out-of-core or distributed data tools (for example, Dask or Spark).

The key idea is the same as in Chapter 3: **measure** where time is spent, then optimize the bottlenecks while keeping code clear and well-tested.

29.13 18.12 How this connects to PyStatsV1

PyStatsV1 is meant to be a **launchpad** for these directions, not an endpoint.

As the project grows, you can expect to see:

- additional chapters and case studies that: * use experimental data and A/B-testing-style designs, * explore generalized linear models and mixed-effects models, * revisit classical examples using Bayesian and machine-learning tools;
- more **teaching-oriented notebooks and scripts** that instructors can drop directly into courses;
- community-contributed examples from different domains (epidemiology, sports analytics, social science, ecology, etc.).

If you'd like to contribute, good starting points include:

- opening a **Discussion** on GitHub describing a course or project you'd like to support;
- filing an issue for: * a new chapter idea, * a missing example or dataset, * or an improvement to the documentation;
- submitting a pull request with: * a small new example script, * a teaching exercise, * or an additional section in this mini-book.

29.14 18.13 Final thoughts

If you've reached this chapter, you've already done something substantial:

- learned to connect mathematical models to real data,
- seen how the same ideas appear in both R and Python,
- practiced reading and writing code that documents your analysis.

From here on, the most important step is simply to **keep using these tools**:

- analyze real datasets that matter to you,
- re-fit models when you learn a new technique,
- explain your results to non-statisticians.

PyStatsV1 is here as a companion—part textbook, part codebase, part community. We hope you'll use it, question it, and help make it better for the next wave of learners.

CHAPTERS OVERVIEW

PyStatsV1 is organized around chapters that mirror classical applied statistics textbook content.

30.1 Implemented chapters

- **Chapter 1 – Introduction**

Basic introduction and environment checks. Run:

```
python -m scripts.ch01_introduction
```

- **Chapter 13 – Within-subjects & Mixed Models**

Within-subjects design and mixed models. Run:

```
make ch13-ci    # tiny CI smoke
make ch13       # full chapter demo
```

- **Chapter 14 – Tutoring A/B Test (two-sample t-test)**

A/B testing with two-sample t-tests. Run:

```
make ch14-ci
make ch14
```

- **Chapter 15 – Reliability (Cronbach's , ICC, Bland–Altman)**

Reliability analysis. Run:

```
make ch15-ci
make ch15
```

30.2 Roadmap

For future chapters and planned topics (e.g., regression extensions, power analysis, epidemiology examples), see the `ROADMAP.md` file in the repository root.

TEACHING GUIDE

PyStatsV1 is designed to be used in courses, workshops, and self-study settings.

31.1 Case study template

The repository provides a case study template at:

```
docs/case_study_template.md
```

This template outlines:

- Learning objectives
- Data description
- Analysis steps
- Discussion questions

31.2 Recommended workflow

- Choose a chapter that matches your lesson (e.g., A/B testing, mixed models, reliability).
- Duplicate the case study template and adapt it to your context.
- Use the chapter scripts as the underlying code for demos or assignments.
- Encourage students to run the scripts locally and experiment with variations.

If you are using PyStatsV1 in teaching, we would love to hear from you via GitHub Discussions ([Teaching stories](#)) or an issue titled `Course report: <institution>`.

CONTRIBUTING

PyStatsV1 welcomes contributions from students, instructors, and practitioners.

32.1 Quick path

1. Read the `CONTRIBUTING.md` file in the repository root.
2. Browse issues labeled `good first issue` or `help wanted`.
3. Fork the repository and clone your fork.
4. Create and activate a virtual environment, then run:

```
pip install -r requirements.txt
make lint
make test
```

5. Make your change, ensure the checks still pass, and open a pull request.

32.2 Types of contributions

- Documentation improvements and teaching notes
- Small refactors and test improvements
- New chapter examples (simulator + analyzer + Makefile targets)
- Bug fixes and robustness improvements

For more details, including coding style and pull request guidelines, see `CONTRIBUTING.md`.

PSYCHOLOGICAL SCIENCE & STATISTICS – FROM INQUIRY TO INSIGHT

This mini-book is **Track B** in the PyStatsV1 documentation. It is written for undergraduate psychology students (and instructors) who want to connect:

- core ideas from research methods and statistics,
- with **real, reproducible analyses in Python**, and
- with examples that look like actual psychology studies.

33.1 What this mini-book assumes

- You have seen basic ideas like variables, hypotheses, and p-values.
- You may have used SPSS, JASP, or jamovi before.
- You are either new to Python, or you have only used it a little.

You do *not* need to be a math expert. We will emphasize:

- research questions and study design,
- good measurement and data collection,
- clear, honest interpretation of results.

33.2 How this track is organized

The chapters are grouped into broad parts:

- **Foundations of psychological science:** why we do experiments, threats to validity, and ethical research.
- **Describing and exploring data:** distributions, visualization, and effect sizes that matter in psychology.
- **Comparing groups:** one-sample, independent, and paired-samples t tests, plus power and planning.
- **Experiments and ANOVA:** one-way and factorial designs, interactions, and planned contrasts.
- **Association and prediction:** correlation, simple and multiple regression, and mediation-style thinking.
- **Advanced designs and capstone projects:** mixed models, repeated measures, nonparametric tests, and full study write-ups.

33.3 PyStatsV1 labs

Most chapters include a **PyStatsV1 lab**:

- a small, psychology-themed dataset (e.g., reaction times, mood ratings, memory scores, or intervention effects),
- Python code that reproduces the main analyses,
- and guidance on how to interpret the output as a researcher.

These labs live in the main PyStatsV1 repository so that you can:

- run them locally in a notebook or Python script,
- modify them for your own course or project,
- and use them as templates for your own studies.

33.4 How to use this track

If you are a **student**:

- Read the conceptual sections first.
- Then open the matching PyStatsV1 lab and run the code yourself.
- Try changing small pieces (sample size, effect size, model specification) and see how the results change.

If you are an **instructor or TA**:

- Treat each chapter as a lecture + lab pairing.
- Use the PyStatsV1 scripts as live demos or homework templates.
- Mix and match chapters with your existing syllabus.

33.5 Where to go next

As we build out this track, new chapters will appear under:

- *Track B – Psychological Science & Statistics (Psych track)*

on the left-hand sidebar.

For more detailed regression theory and diagnostics, you can also follow **Track A – Applied Statistics with Python (Regression)**, which mirrors a classic applied regression course in a language-agnostic way (R Python).

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 1

34.1 Thinking like a psychological scientist

This chapter sets the stage for the rest of the psychology track. Our goals are to connect:

- the *questions* psychologists ask,
- the *studies* they design, and
- the *analyses* they run,

so that statistics feels like part of the scientific process, not a separate math class.

34.2 1.1 Why we cannot just “trust our gut”

Everyday life encourages us to rely on intuition: “It feels true, so it must be true.” In psychological science, that is not enough.

Three classic biases:

- **Confirmation bias**

We notice and remember information that supports what we already believe, and ignore disconfirming evidence.

Example: if you believe “screens before bed ruin sleep,” you may pay attention to the nights you slept badly after using your phone and forget about good nights.

- **Hindsight bias (“I knew it all along”)**

After we know the outcome, we feel as if it was obvious.

Example: once you hear the result of a study, the conclusion feels inevitable, even if you would never have predicted it in advance.

- **Availability heuristic**

We judge how common something is based on how easily examples come to mind.

Example: you may think exam anxiety is “universal” because anxious students are more vivid in memory or more likely to talk about it, even if many students are relatively calm.

These biases are **not character flaws**; they are how human cognition works. The point of scientific methods and statistics is to *protect us from our own minds*.

34.3 1.2 The scientific method: systematizing curiosity

A cartoon version of the scientific method is:

Theory \Rightarrow Hypothesis \Rightarrow Observation \Rightarrow Revision.

In more detail:

1. **Theory**

A framework or story about how the world works.

Example: using social media late at night increases arousal and interferes with sleep, which increases next-day anxiety.

2. **Hypothesis**

A specific, testable prediction derived from the theory.

Example: students who restrict screen time in the hour before bed will report lower test anxiety than students who do not.

3. **Observation / Study**

You design a study, collect data, and analyze the results.

4. **Revision**

You compare the results to the theory:

- If the data support the hypothesis, your confidence in the theory may increase.
- If the data do **not** support it, you refine or replace the theory.

Psychological science is not a straight line from theory to proof. It is a looping process of **asking, testing, and revising**.

34.4 1.3 Claims, variables, and hypotheses

Research methods courses often distinguish three broad kinds of claims:

- **Frequency claims** – how often something happens. - Example: “About 30% of students report test anxiety before exams.”
- **Association claims** – how two measured variables move together. - Example: “Students with higher sleep quality tend to report lower anxiety.”
- **Causal claims** – whether one variable *causes* a change in another. - Example: “Mindfulness training reduces test anxiety.”

To study any of these, we need **variables**:

- A *variable* is something that can take different values (score, condition, gender, reaction time, etc.).
- A *construct* is the underlying idea (test anxiety, motivation, working memory).
- An *operational definition* is how we turn the construct into data. - Example: “Test anxiety” \rightarrow total score on a 10-item questionnaire.

A **hypothesis** links constructs in a testable way, using variables we can measure:

- Construct level: “Higher sleep quality leads to lower anxiety.”
- Operational level: “Students with higher scores on the sleep quality scale will have lower scores on the anxiety questionnaire.”

Throughout this track we will move between:

- *words* (psychological story),
- *variables* (how we measure it), and
- *numbers* (what we analyze in Python).

34.5 1.4 The four big validities

When psychologists evaluate a study, they often ask about four types of *validity*: how well does the study support the claim being made?

- **Construct validity**

Are we really measuring what we think we are?

- Does our test anxiety questionnaire actually capture “anxiety about tests” rather than general stress or depression?

- **External validity**

Do the findings generalize beyond this specific sample and setting?

- Would the results hold at a different university, age group, or culture?

- **Statistical validity**

Do the numbers support the claim?

- Are the analyses appropriate for the design and variables?
- Is the sample big enough to detect realistic effects?
- Are we interpreting p-values, confidence intervals, and effect sizes correctly?

- **Internal validity**

Did A really cause B?

- Are there alternative explanations (confounds)?
- Was there random assignment, or could pre-existing differences explain the results?

No single study is perfect on all four dimensions. The real skill is to:

- match the **claim** to the **design** and **analysis**, and
- be honest about which kinds of validity are strongest (and weakest).

34.6 1.5 Where Python and PyStatsV1 fit in

So far this chapter has been mostly conceptual. That is intentional: you need a strong sense of **questions, claims, and validities** before statistics and code really make sense.

In this mini-book, Python is the environment where we will:

- store and clean psychological data,
- compute descriptive statistics and effect sizes,
- fit models (e.g. t tests, regression, logistic regression),
- create plots that tell clear stories, and
- keep an executable record of what we did.

PyStatsV1 provides:

- example datasets that look like real psychology studies,
- reusable analysis scripts and “labs,” and
- documentation (like this mini-book) that connects theory to code.

For now, we just want a **gentle “hello world”**.

```
import pandas as pd

# Replace this path with the actual location in PyStatsV1 once the labs are set up
data = pd.read_csv("data/study1_sleep_anxiety.csv")

print(data.head())

print(data.dtypes)
```

This tiny script already brings several ideas together:

- a dataset with rows = participants and columns = variables,
- variable *types* (numeric vs categorical) in `dtypes`,
- a repeatable analysis (you can rerun the same code on the same file).

In later chapters we will build full **PyStatsV1 labs** where:

- each chapter has a corresponding dataset and notebook/script,
- you run analyses that mirror the textbook examples, and
- you can adapt the code to your own projects.

34.7 1.6 What you should take away

By the end of this chapter you should be able to:

- explain why psychological science cannot rely on intuition alone,
- distinguish between frequency, association, and causal claims,
- describe the basic scientific method loop (theory → hypothesis → observation → revision),
- name and briefly define the four big validities (construct, external, statistical, internal),
- see where Python and PyStatsV1 will fit into the research cycle.

When later chapters become more technical, come back here and ask:

- *What is the claim?*
- *What is the design?*
- *Which kind(s) of validity are strongest?*
- *What exactly are the numbers telling us about the psychological story?*

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 2

35.1 Ethics in Research: Why They Matter More Than Ever

Psychology studies people. That means psychology must protect people.

Every research method, every measurement, and every statistical test is built on a foundation of *ethical responsibility*. Before we learn how to analyze data, we need to understand why scientific ethics exist, where they came from, and why modern research is designed the way it is.

This chapter has five goals:

- show why historical failures shape everything psychologists do today,
- introduce the Belmont Report and the principles used in all human research,
- explain the core elements of the APA ethics code,
- describe data-specific ethics (p-hacking, HARKing, replication crisis),
- introduce Open Science and how PyStatsV1 supports ethical transparency.

35.2 2.1 Historical failures: why rules exist

Modern research ethics were not invented in a classroom—they were built after real harm occurred. Some landmark cases:

- **Tuskegee Syphilis Study (1932–1972)** Participants were *denied treatment* so researchers could “observe” the progression of disease. The study continued for decades after penicillin became the known cure.
- **Milgram Obedience Studies (1960s)** Participants believed they were delivering painful electric shocks. Although no physical harm occurred, emotional distress was severe.
- **Stanford Prison Experiment (1971)** A simulated prison environment escalated quickly toward psychological abuse. Oversight and controls were nearly absent.
- **Tearoom Trade Study (1970)** There was recorded identifiable information on participants engaging in private sexual behavior without consent.

These cases illustrate the same theme:

When research goals overshadow human well-being, ethics are violated.

The entire system of modern IRB review and ethics training exists because of these events.

35.3 2.2 The Belmont Report

After Tuskegee became public in the 1970s, the U.S. formed the National Commission for the Protection of Human Subjects. The result was the **Belmont Report**, which introduced the three principles that still govern research today.

- 1. Respect for persons** Individuals must be treated as autonomous agents. People with diminished autonomy (children, prisoners, cognitively impaired individuals) receive special protection.
- 2. Beneficence** Researchers must maximize benefits and minimize possible harms. Participants should never be exposed to unnecessary risk.
- 3. Justice** The burdens and benefits of research should be distributed fairly. Vulnerable populations should not be targeted simply because they are easy to recruit.

Every ethics review board—worldwide—uses these ideas.

35.4 2.3 APA Guidelines for psychologists

The American Psychological Association provides detailed guidance for researchers. Some core principles:

- **Informed consent** Participants must understand what the study involves, any risks, and their rights—including the right to withdraw at any time.
- **Debriefing** If deception is used, it must be justified and followed by a complete, honest explanation.
- **Protection from harm** Psychologists must prevent physical, emotional, and psychological harm. Risks must be minimized and reasonable.
- **Confidentiality and privacy** Data must be stored securely, anonymized where possible, and reported in a way that does not reveal identities.
- **Avoiding deception unless necessary** Deception must have strong justification and must not conceal risk or prevent consent.

These rules are not obstacles—they are tools that protect participants and strengthen research quality.

35.5 2.4 Data ethics: p-hacking, HARKing, and the replication crisis

Ethics does not stop at participant treatment. It also covers how researchers *handle data and report results*.

p-hacking Trying many analyses and only reporting the ones that produce a significant p-value.

HARKing (“**Hypothesizing After Results are Known**”) Presenting exploratory findings as if they were predicted in advance.

Selective reporting Only publishing studies that “work,” leaving null results hidden in file drawers.

These practices contributed to psychology’s **replication crisis**, where many high-profile results could not be reproduced by independent researchers.

The most positive way to improve the state of research is to increase honesty, transparency, and reproducibility.

35.6 2.5 Open Science: pre-registration and data sharing

Open Science is a movement dedicated to making research:

- transparent,
- reproducible,
- shareable, and

- resistant to questionable research practices.

Key tools include:

- **Pre-registration** Researchers publicly record their hypotheses, methods, and planned analyses *before* collecting data.
- **Registered reports** Journals review and accept a study's methods *before* results exist.
- **Data and code sharing** When possible, datasets and analysis scripts are shared so others can reproduce the work.
- **Reproducible workflows** Analyses are performed with code rather than unrecorded menu clicks.

This is where PyStatsV1 begins to shine.

35.7 2.6 A small reproducible example (PyStatsV1)

Here is a gentle demonstration of how a reproducible workflow starts in Python:

```
import pandas as pd

# Replace this after the lab dataset is added
data = pd.read_csv("data/study1_memory_scores.csv")

print(data.head())
print(data.describe())
```

This tiny script shows how:

- the entire analysis can be reproduced by running the same file,
- every transformation is visible and documented,
- code encourages transparency rather than hidden analytical decisions.

35.8 2.7 What you should take away

By the end of this chapter, you should be able to:

- explain why ethical rules exist in human research,
- summarize the Belmont Report principles (Respect, Beneficence, Justice),
- describe core APA guidelines like informed consent and debriefing,
- identify questionable data practices (p-hacking, HARKing, selective reporting),
- understand the motivation behind the Open Science movement,
- see how PyStatsV1 supports transparent, reproducible workflows.

Before we analyze data, we need to protect the people who provide it. Ethical research is not just a rule—it is a responsibility.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 3

36.1 Defining and Measuring Variables: How Concepts Become Data

Psychology studies things we cannot directly see: attention, emotion, memory, stress, prejudice, personality, motivation. These are *concepts*, but our statistical tools require *numbers*. Chapter 3 shows how psychologists translate ideas into measurable variables, and how good measurement determines the quality of scientific conclusions.

This chapter has five goals:

- explain conceptual vs. operational definitions,
- introduce the four scales of measurement (NOIR),
- describe reliability and why consistency matters,
- explain validity and how we know a measure works,
- connect these ideas to a short PyStatsV1 lab.

Most of the statistical mistakes students struggle with later trace back to measurement problems introduced here.

36.2 3.1 Conceptual vs. operational definitions

A **conceptual definition** is the idea you care about. A **operational definition** is how you *measure* that idea.

Examples:

- *Conceptual*: Anxiety *Operational*: Score on the GAD-7 questionnaire.
- *Conceptual*: Aggression *Operational*: Number of noise blasts delivered in a competitive reaction-time task.
- *Conceptual*: Working memory *Operational*: Number of items correctly recalled in a digit-span test.

Why operational definitions matter:

- They determine what the data actually represent.
- Different operationalizations can lead to different conclusions.
- Clarity allows replication — another researcher must be able to reproduce your measure.

A strong research question always pairs both:

Conceptual definition → Operational definition → Data

36.3 3.2 Scales of measurement (NOIR)

Not all numbers behave the same way statistically. Psychologists classify variables into four **scales of measurement**, often remembered as **NOIR**:

Nominal (names)

Categories with no numeric meaning. *Examples:* gender identity, therapy type, favorite color.

Ordinal (rank order)

Ordered categories, but distances between ranks are unknown. *Examples:* symptom severity ratings (“mild / moderate / severe”), Likert scales.

Interval (equal units)

Numeric scales with equal steps, but no true zero. *Examples:* temperature in °C or °F, many psychological test scores.

Ratio (meaningful zero)

All properties of interval scales plus a true zero. *Examples:* reaction time, number of errors, hours slept.

Why NOIR matters:

- Some statistical tests **require** interval or ratio data.
- Treating ordinal data as interval is common — but needs justification.
- Ratio scales allow multiplicative statements (“twice as fast”).

36.4 3.3 Reliability: consistency of measurement

A measure must be **reliable** to be useful. Reliability asks:

“If we measured the same thing again, would we get a similar result?”

Three major forms:

Test–retest reliability

Are scores stable across time? Important for traits (e.g., personality).

Inter-rater reliability

Do two observers agree? Important for coding behavior or scoring essays.

Internal consistency

Do items on a questionnaire measure the same underlying construct? Measured with statistics like **Cronbach’s alpha**.

Rules of thumb:

- Reliability near **0.70** is acceptable in early research.
- Low reliability puts an upper bound on validity.
- Reliability is necessary — but not sufficient — for good measurement.

36.5 3.4 Validity: accuracy of measurement

If reliability is about consistency, **validity** is about truth.

A measure is valid if it accurately captures the construct it claims to measure.

Major forms of validity:

Face validity

Does the measure *look* like it measures the construct?

Content validity

Does it cover all relevant aspects of the construct?

Criterion validity

Do scores predict something they should predict? *Example:* A depression score predicting therapist diagnoses.

Convergent validity

Does it correlate with other measures of the same construct?

Discriminant validity

Does it *not* correlate with measures of different constructs?

Key idea:

A test can be reliable but **not** valid.
A test cannot be valid unless it **is** reliable.

36.6 3.5 PyStatsV1 lab: exploring variable types

Let's look at a tiny example dataset (from a future chapter's lab) and use PyStatsV1 conventions to understand variable types.

```
import pandas as pd

# Example dataset (placeholder path)
data = pd.read_csv("data/study1_sleep_anxiety.csv")

print(data.head())
print("\nVariable types:")
print(data.dtypes)
```

This simple script begins the habit of **inspecting** variables before analyzing them — a crucial skill for psychological researchers.

In full PyStatsV1 labs, students will:

- classify variables using NOIR,
- identify operational definitions,
- check reliability (e.g., Cronbach's alpha),
- assess validity using correlations and scatterplots.

Measurement is where scientific thinking meets statistical reasoning.

36.7 3.6 What you should take away

By the end of this chapter you should be able to:

- distinguish conceptual and operational definitions,
- classify variables into nominal, ordinal, interval, or ratio,
- explain why reliability matters and name its three major forms,
- describe common types of validity and why no single type is sufficient,

- see how PyStatsV1 helps document and inspect variables before analysis.

In later chapters, we will build statistical tests on top of this foundation. Sound conclusions require sound measurement.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 4

37.1 Frequency Distributions and Visualization: Finding Patterns in Data

Before we run statistics, we need to *see* the data. Psychology students often jump straight to t-tests or correlations, but rigorous researchers begin with a simpler, essential question:

“What does the data *look* like?”

This chapter introduces **Exploratory Data Analysis (EDA)**. These tools allow us to detect patterns, spot data entry errors (e.g., a participant aged 150), and determine whether our data meets the assumptions required for complex testing later.

37.2 Why this chapter matters

Exploratory Data Analysis (EDA) is the first real step of psychological science. Before testing hypotheses, we must ensure our data are clean, plausible, and interpretable. Visualizing and tabulating data helps researchers:

- detect errors (e.g., impossible ages or reaction times),
- understand participant behavior patterns,
- diagnose assumptions required for later inferential tests,
- and communicate results clearly using APA-style figures.

This chapter builds the foundation for every analysis that follows.

37.3 4.1 Frequency tables: organizing data

A **frequency table** counts how many times each value occurs. However, raw counts aren’t always enough. In psychology, we often need two additional columns:

1. **Relative Frequency (%)** – the percentage of the total sample.
2. **Cumulative Frequency** – the running total (helps compute percentiles).

37.3.1 Relative Frequency Formula

$$\text{relative frequency} = \frac{f_i}{N}$$
$$\text{percent} = \frac{f_i}{N} \times 100$$

37.3.2 Cumulative Frequency Formula

$$\text{cum } f_i = f_1 + f_2 + \dots + f_i$$

37.3.3 Categorical Example

Imagine we asked 50 students about their preferred study method:

Study Method	Frequency (f)	Relative Freq (%)
Flashcards	18	36%
Re-reading	12	24%
Practice tests	15	30%
Other	5	10%
Total	50	100%

Psychological Insight: While cognitive science shows retrieval practice (“Practice tests”) is highly effective, only 30% of this sample uses it.

37.3.4 Continuous Example (Grouped)

Continuous variables (like reaction time or sleep duration) have too many unique values to list individually. We group them into **bins** (intervals).

Table 1: Sleep Duration (N=100)

Hours Slept	Frequency	Cumulative Frequency
4.0 – 5.9	6	6
6.0 – 6.9	21	27
7.0 – 7.9	44	71
8.0 – 8.9	23	94
9.0 – 10.0	6	100

Note

The “Real Limits” Rule: In PyStatsV1 and most statistical software, bins are treated as inclusive of the lower bound and exclusive of the upper bound (e.g., $6.0 \leq x < 7.0$).

37.4 4.2 Visualizing continuous data: histograms

Numbers are helpful, but humans are visual creatures.

A histogram looks like a bar chart, but the bars **touch**. This signifies that the variable is continuous—there is no gap between 6.99 hours and 7.00 hours.

Key interpretation checks:

- **Peaks:** Where is the data clustered?
- **Spread:** Tight vs. wide distributions.
- **Outliers:** Lone bars far from the others.

37.4.1 The Frequency Polygon

If we place a dot at the top-center of every histogram bar and connect the dots, we get a **Frequency Polygon**.

Why polygons?

- Comparing groups becomes cleaner.
- Two overlaid histograms are messy; two polygons are readable.

Note

Accessibility Reminder: Use colorblind-safe palettes (e.g., blue/orange) and never encode group differences by color alone. Add labels or line styles.

37.5 4.3 Visualizing categorical data: bar charts

For nominal variables, we use **bar charts**.

37.5.1 Rules for APA-Style Bar Charts

1. Bars **should not** touch (these are discrete categories).
2. Order bars by **frequency**, not alphabetically, unless the categories have a natural order.
3. Keep labeling clean and descriptive.

Warning

The Prohibition of Pie Charts

Pie charts are rarely used in scientific psychology.

- Humans struggle to compare angles.
- Small differences are nearly invisible.
- More than 4 categories = unreadable.

Use bar charts instead.

37.6 4.4 The shape of data: skewness and kurtosis

Understanding the *shape* helps diagnose phenomena and potential design issues.

37.6.1 Skewness: the tails

Positive Skew (Right Skew) Tail extends to the right. *Psychology Context:* **Floor Effects** Example: A very difficult memory test — most people score low, but a few score high.

Negative Skew (Left Skew) Tail extends to the left. *Psychology Context:* **Ceiling Effects** Example: An exam that was too easy — most people score high, with a small tail of low performers.

37.6.2 Kurtosis: the peak

Kurtosis describes thickness of tails vs. center.

- **Leptokurtic:** Tall, thin — very similar scores (e.g., elite athletes).
- **Platykurtic:** Flat — large variability (e.g., general population samples).
- **Mesokurtic:** Normal distribution — moderate shape.

37.7 4.5 PyStatsV1 Lab: Exploring the sleep study dataset

In this chapter's lab we will use the synthetic *sleep study* dataset you saw earlier in this mini-book plan. It lives in the PyStatsV1 repository and is generated by the helper module `scripts/sim_psych_sleep_study.py`.

Note

For instructors and maintainers

The `sleep_study` dataset used in this and later chapters is generated from a small simulation script in the PyStatsV1 repository. If you ever need to recreate the CSV from scratch (for example, after editing the data-generating assumptions), run the following command from the project root:

```
python scripts/sim_psych_sleep_study.py
```

By default this will (re)write the file `data/psych_sleep_study.csv` with the same structure and random seed that the textbook examples and tests expect. See `scripts/sim_psych_sleep_study.py` for more details and optional arguments.

The goal is to give you a repeatable pattern:

- load a realistic psychology dataset in Python,
- inspect its variables,
- and create basic plots that match the ideas in this chapter.

37.7.1 Loading the data

From the root of the PyStatsV1 repository, open a Python session or Jupyter notebook and run:

```
from scripts.sim_psych_sleep_study import load_sleep_study

df = load_sleep_study()
df.head()
```

You should see columns like:

- `id` – participant ID,
- `class_year` – `first_year`, `second_year`, `third_year`, or `fourth_year`,
- `sleep_hours` – average weeknight sleep (hours),
- `study_method` – `flashcards`, `rereading`, `practice_tests`, or `mixed`,
- `exam_score` – exam percentage (0–100).

The first time you call `load_sleep_study()`, it will create the CSV file `data/synthetic/psych_sleep_study.csv`. Later calls simply read that file so you get the *same* dataset each time.

37.7.2 Frequency tables for study methods

Let's build a frequency table for the categorical variable `study_method`:

```
# Frequency (counts)
freq = df["study_method"].value_counts().rename("f")

# Relative frequency (percentages)
rel_freq = (freq / len(df) * 100).round(1).rename("percent")

# Combine into one DataFrame
table = (
    pd.concat([freq, rel_freq], axis=1)
    .reset_index()
    .rename(columns={"index": "study_method"})
    .sort_values("f", ascending=False)
)

print(table)
```

This prints a table like:

study_method	f	percent
practice_tests	36	30.0
flashcards	34	28.3
rereading	30	25.0
mixed	20	16.7

This mirrors the frequency tables earlier in the chapter, but now the numbers come from data you could plausibly collect in a real study skills experiment.

37.7.3 Histogram of sleep hours

Now make a histogram of the continuous variable `sleep_hours`:

```
import matplotlib.pyplot as plt

plt.hist(df["sleep_hours"], bins=10, edgecolor="black")
plt.xlabel("Sleep hours (weeknight average)")
plt.ylabel("Number of students")
plt.title("Distribution of sleep duration")
plt.show()
```

When you look at the plot, ask:

- Where is the data clustered (what is the *mode*)?
- Are there students sleeping very little or a lot (potential outliers)?
- Does the shape look roughly symmetric, or skewed?

37.7.4 Bar chart for study methods

For the categorical `study_method` variable, use a bar chart:

```
freq = df["study_method"].value_counts().sort_values(ascending=False)

plt.bar(freq.index, freq.values)
plt.ylabel("Number of students")
plt.title("Preferred study method")
plt.xticks(rotation=20)
plt.show()
```

Notice that the bars are separated (this is categorical, not continuous) and we have ordered them from most common to least common.

37.8 4.6 What you should take away

By the end of this chapter and lab you should be able to:

- construct frequency tables for categorical and grouped continuous data,
- choose appropriate visualizations (histograms for continuous data, bar charts for categorical data),
- interpret the *shape* of a distribution (center, spread, skewness),
- and run a small, reproducible analysis by loading `load_sleep_study()` from the PyStatsV1 repository.

In later chapters we will reuse this same dataset when we talk about measures of central tendency, variability, and eventually correlation and regression. That way the plots and statistics you see in the text are always tied to a concrete psychological story.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 5

38.1 Central Tendency and Variability: Summarizing What We See

Chapter 4 was about *looking* at data—frequency tables and graphs that show the overall shape of a distribution. In this chapter we take the next step:

Can we describe that distribution with a few meaningful numbers?

Psychology relies heavily on this kind of summary. We say things like:

- “On average, the treatment group slept longer than the control group.”
- “There was a lot of variability in stress scores.”
- “Most participants were near the mean, but a few scored far out in the tails.”

This chapter introduces:

- measures of **central tendency** (where the distribution is centered), and
- measures of **variability** (how spread out the scores are).

We will keep connecting these ideas back to the sleep-study dataset and other realistic psychological examples.

38.2 5.1 Why central tendency and variability both matter

If you only know the *average* of a set of scores, you know almost nothing about how individuals actually behaved.

Imagine two classes that took the same exam:

- Class A: Most students scored between 78 and 82.
- Class B: Half the students scored around 50 and the other half around 100.

Both classes could have the **same mean**, but the stories these data tell are very different.

To understand a distribution, we need **both**:

- a number that summarizes “typical” or “central” performance, and
- a number that summarizes how much scores vary around that center.

38.3 5.2 Measures of central tendency: mean, median, mode

There are three classic measures of central tendency.

38.3.1 The mean (arithmetic average)

The **mean** is what most people casually call the “average”. It is defined as

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i,$$

where x_i are the individual scores and N is the number of participants.

Psychology use case: We often report the mean reaction time, mean depression score, mean hours of sleep, etc.

Strengths

- Uses *all* the data.
- Works well with many statistical models (especially those based on the Normal distribution).

Weaknesses

- Extremely sensitive to **outliers** (one person who barely slept can drag down the mean sleep hours).
- Can be misleading for heavily skewed distributions.

38.3.2 The median (the middle score)

The **median** is the value that splits the distribution in half:

- 50% of scores are at or below the median.
- 50% are at or above.

To find the median, sort the scores from smallest to largest, then pick the middle one (or average the two middle scores if there are an even number of observations).

Strengths

- Robust against outliers and skewed data.
- Often a better description of “typical” behavior when the distribution is highly skewed (e.g., income, number of social media followers).

38.3.3 The mode (most frequent score)

The **mode** is simply the most common value in a distribution.

- For continuous variables (like reaction time), the mode is often less useful.
- For categorical variables (e.g., study method, therapy type), the mode tells you which category is most popular.

In practice

In applied psychology, we usually report **mean** and **standard deviation** for roughly symmetric, continuous variables, and we may report **median** and a measure of spread (e.g., interquartile range) when the distribution is skewed.

38.4 5.3 The problem with averages: when the mean misleads

The mean can give a false sense of what is “typical”.

38.4.1 Example: Sleep and outliers

Suppose we measured hours of sleep last night for 10 participants:

$$6, 6.5, 7, 7, 7.5, 8, 8, 8.5, 9, 2$$

Most participants slept between 6 and 9 hours, but one person only slept 2. The mean is

$$\bar{x} = \frac{6 + 6.5 + \dots + 9 + 2}{10} = 7.1 \text{ hours (approx).}$$

The median, however, is 7.5 hours.

If you were designing a sleep intervention, which number better captures what is typical in this group?

This demonstrates:

- **Outliers** can pull the mean away from where most data lie.
- Reporting the median alongside the mean can help detect this problem.
- Graphs (like histograms) are essential companions to numerical summaries.

38.5 5.4 Measures of variability: range, IQR, variance, SD

Central tendency tells us *where* scores cluster. Variability tells us *how tightly* they cluster.

38.5.1 The range

The **range** is the simplest measure:

$$\text{Range} = \text{Maximum} - \text{Minimum}.$$

It shows the width of the distribution but is extremely sensitive to outliers.

38.5.2 The interquartile range (IQR)

The **interquartile range (IQR)** focuses on the middle 50% of the data:

- Q_1 (first quartile): 25th percentile.
- Q_3 (third quartile): 75th percentile.

$$\text{IQR} = Q_3 - Q_1.$$

A large IQR means scores are spread out; a small IQR means participants are relatively similar.

38.5.3 The variance and standard deviation

The **variance** and **standard deviation (SD)** go beyond extremes and quantiles by using all the data.

For a **sample** of scores x_1, \dots, x_N , the sample variance is

$$s^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2.$$

The standard deviation is the square root:

$$s = \sqrt{s^2}.$$

Interpretation:

- If scores are tightly clustered around the mean, s is small.
- If scores are widely spread out, s is large.
- Under many models, most scores fall within about 1–2 standard deviations of the mean.

In psychological research reports we almost always see something like:

“Participants slept an average of 7.2 hours ($SD = 1.1$).”

38.6 5.5 Degrees of freedom: why divide by $N - 1$?

You may have noticed that the variance formula uses $N - 1$ rather than N in the denominator. This is related to the idea of **degrees of freedom (df)**.

Informally, degrees of freedom are the number of independent pieces of information available for estimating a parameter.

For the sample variance:

- Once you know the sample mean \bar{x} , the deviations $(x_i - \bar{x})$ must sum to zero.
- That means if you know $N - 1$ of the deviations, the last one is already determined.

So there are only $N - 1$ independent deviations, and we divide by $N - 1$ to obtain an **unbiased** estimate of the population variance.

This idea of degrees of freedom will appear again in t-tests and ANOVAs later in the mini-book.

38.7 5.6 PyStatsV1 Lab: Summarizing the sleep-study data

In this lab we return to the **sleep study** dataset. We will:

- compute mean, median, and mode for hours of sleep,
- compute range, IQR, and standard deviation,
- compare summaries across study methods.

If you ever need to regenerate the underlying CSV file for this dataset, see the instructor note in *Psychological Science & Statistics – Chapter 4* about running `scripts/sim_psych_sleep_study.py`.

38.7.1 Loading the dataset

If you have cloned the PyStatsV1 repository, the CSV file will be located in the `data` folder. You can load it with pandas:

```
import pandas as pd

data = pd.read_csv("data/psych_sleep_study.csv")

print(data.head())
print(data.dtypes)
```

You should see variables such as:

- `participant_id` – unique ID per participant,
- `sleep_hours` – hours of sleep last night (continuous),
- `study_method` – preferred study method (categorical),

- chronotype – morning/evening type (categorical),
- possibly additional variables (e.g., stress score) depending on the simulation.

38.7.2 Overall summaries

First, let us compute basic summaries for the entire sample:

```
sleep = data["sleep_hours"]

mean_sleep = sleep.mean()
median_sleep = sleep.median()
mode_sleep = sleep.mode() # may return more than one value

print(f"Mean sleep: {mean_sleep:.2f} hours")
print(f"Median sleep: {median_sleep:.2f} hours")
print("Mode(s):")
print(mode_sleep.values)

# Measures of spread
sleep_range = sleep.max() - sleep.min()
iqr_sleep = sleep.quantile(0.75) - sleep.quantile(0.25)
sd_sleep = sleep.std(ddof=1)

print(f"Range: {sleep_range:.2f} hours")
print(f"IQR: {iqr_sleep:.2f} hours")
print(f"SD: {sd_sleep:.2f} hours")
```

As you run this code, ask:

- Is the mean close to the median, or are there signs of skewness?
- Does the SD seem small (participants similar) or large (participants differ widely)?
- Do the numbers match what you saw in the histogram from Chapter 4?

38.7.3 Group summaries by study method

Now let us see whether preferred study method is associated with how much students slept. We compute group-wise means and SDs:

```
grouped = (
    data
    .groupby("study_method")["sleep_hours"]
    .agg(["count", "mean", "median", "std"])
    .rename(columns={"std": "sd"})
    .sort_values("mean", ascending=False)
)

print(grouped)
```

This table tells us, for each study method:

- how many students chose it (count),
- their average hours of sleep (mean),
- the median hours of sleep (median),

- how much their sleep varies (sd).

You might find, for example, that students who use practice tests have slightly different sleep patterns than those who rely on re-reading.

38.7.4 Connecting back to research design

In a real study, we might ask:

- Is study method causing differences in sleep?
- Or are both sleep and study method being influenced by some third variable, like stress or personality?

For now, our goal is more modest: using central tendency and variability to summarize what is happening in this sample.

38.8 5.7 What you should take away

By the end of this chapter and lab you should be able to:

- explain the difference between **mean**, **median**, and **mode**, and when each is most appropriate,
- describe why averages can be misleading in the presence of **outliers** or **skewed** distributions,
- compute and interpret common measures of variability (range, IQR, variance, standard deviation),
- understand, at an intuitive level, why the sample variance uses $N - 1$ in the denominator (degrees of freedom),
- use Python and pandas to compute these summaries for a realistic psychological dataset,
- compare central tendency and spread across groups (e.g., different study methods) to generate new research questions.

In the next chapter we connect these ideas to the **Normal distribution** and **z-scores**, which provide a bridge from descriptive summaries to probabilities and, eventually, to hypothesis testing.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 6

39.1 The Normal Distribution and z-Scores

In Chapters 4 and 5 you learned how to *describe* a distribution using graphs (histograms, frequency polygons) and summary statistics (mean, median, variance, standard deviation). These tools help us understand what our data *look like*.

In this chapter we take an important next step: we introduce the **normal distribution**, a mathematical model used throughout psychology and statistical inference. The normal distribution helps us:

- compare individual scores to a reference population,
- identify “unusual” observations,
- standardize variables measured on different scales, and
- compute probabilities, percentiles, and standardized effect sizes.

We will use the same synthetic sleep-study dataset introduced earlier (psych_sleep_regen_note) and learn how to:

- Fit a normal model to a single variable
- Convert raw scores to **z-scores**
- Interpret where an individual score falls in a distribution
- Visualize empirical data alongside a theoretical normal curve
- Compute simple probabilities under the normal model

39.2 A Statistical Clarification: Raw Data vs. Sampling Distributions

It is important to distinguish between:

1. **The distribution of raw scores**, which may or may not be normal (e.g., reaction times are usually positively skewed), and
2. **The distribution of sample means**, which *is* approximately normal under broad conditions, by the Central Limit Theorem (CLT).

Even when raw observations are skewed, the *sample mean* has an approximately normal distribution when the sample size is large enough. If the parent population is Normal then for small samples, the standardized (using the sample standard deviation) sample mean follows a **Student’s t distribution**, not a normal distribution.

In this chapter, we focus on **z-scores for individual observations**, a tool commonly used in psychological measurement. Later chapters (especially Chapter 7 and Chapter 9) introduce sampling distributions, the CLT, and the t distribution in more depth.

39.3 The Normal Distribution: A Workhorse in Applied Statistics

The normal distribution (also called the *Gaussian* or *bell curve*) is one of the most important models in psychological science and applied statistics. Many aggregated or psychometrically smoothed variables are approximately (“close enough”) normal or can be modeled as normal in appropriate scenarios. For example:

- IQ scores (by design)
- Composite cognitive performance scores
- Height and weight
- Many Likert-scale *sum scores*
- Measurement error under many conditions

A normal distribution is defined by only two parameters:

- The **mean** (center)
- The **standard deviation** (spread)

Once these two numbers are known, we can describe the *entire* distribution mathematically.

39.4 Standardizing: From Raw Scores to z-Scores

A **z-score** expresses a raw score in standard deviation units:

$$z = \frac{X - \mu}{\sigma}$$

When population parameters are unknown (as is typical), we substitute sample estimates.

Interpreting z-scores (where average = mean):

- $z = 0 \rightarrow$ Exactly average
- $z = +1 \rightarrow$ One standard deviation above average
- $z = -2 \rightarrow$ Two standard deviations below average
- $z = +2 \rightarrow$ Approximately the 97.5th percentile (under the normal model)

In psychology, z-scores are useful because they make variables comparable even when measured on different scales.

Examples:

- `sleep_hours_z = -1.2` The participant slept *less* than average (mean) by 1.2 SDs.
- `reaction_time_z = +0.8` The participant was *slower* than the mean (higher RTs \rightarrow slower responses).

In this chapter’s PyStatsV1 lab, you will compute z-scores using empirical summary statistics (sample mean and sample standard deviation). This is a common practice when working with real psychological data.

39.5 PyStatsV1 Lab: Normal Distribution and z-Scores

In this lab, you will:

1. Load the sleep-study dataset
2. Compute z-scores for one continuous variable (e.g., `sleep_hours`)
3. Generate a histogram overlaid with a normal density curve
4. Inspect the distribution of z-scores

5. Compute simple probabilities using a normal model

Remember: the normal curve is a *model*. The empirical data do not need to be perfectly normal in order for z-scores to be meaningful or useful.

All code for this chapter lives in:

`scripts/psych_ch6_normal_zscores.py`

and the dataset lives in:

`data/synthetic/psych_sleep_study.csv`

See `psych_sleep_regen_note` if you need to regenerate the dataset.

39.5.1 Running the Lab Script

From the project root, run:

```
python -m scripts.psych_ch6_normal_zscores
```

Or, if using the Makefile target:

```
make psych-ch06
```

This will:

- Load the sleep-study dataset
- Compute the sample mean, sample standard deviation, and corresponding z-scores
- Generate a histogram overlaid with a fitted normal curve
- Print summary information to the console
- Save a PNG of the plot (depending on your local settings)

39.5.2 Expected Console Output

Numbers may vary slightly depending on random seed and sample size:

```
Loaded dataset with 200 participants
sleep_hours mean = 6.98
sleep_hours SD   = 1.02
First five z-scores:
Participant 0: 0.13
Participant 1: -0.87
Participant 2: 1.10
Participant 3: -0.05
Participant 4: 0.44
```

39.5.3 Interpreting the Plot

The figure shows a histogram of `sleep_hours` and a fitted normal curve.

Questions to consider:

- Does the empirical distribution appear approximately normal?
- Are there signs of skewness or kurtosis?
- Would the normal model be a reasonable approximation?

Remember that many psychological variables are only *approximately* normal, and some (like reaction times) are typically skewed.

39.5.4 Your Turn: Practice Interpreting z-Scores

1. **Identify two typical sleepers.** Look for participants with z-scores near 0. What does this tell you?
2. **Identify one unusually high value.** Find a participant with $z > 2$. What percentile does this correspond to?
3. **Identify one unusually low value.** Find a participant with $z < -2$. What might explain such low sleep values?
4. **Estimate simple probabilities.** Using the normal model, estimate:
 - $P(Z > 1)$
 - $P(-1 < Z < 1)$
 - $P(Z < -2)$

How close are these to the 68–95–99.7 rule?

39.5.5 Optional Extension: Reaction Times

Try computing and visualizing z-scores for:

`reaction_time_ms`

Questions:

- Are reaction times normally distributed?
- Are they positively skewed?
- What happens if you apply a log-transform before standardizing?

39.6 Summary

In this chapter you learned how to:

- Model a psychological variable using a **normal distribution**
- Compute and interpret **z-scores**
- Overlay a theoretical normal curve on empirical data
- Estimate probabilities and percentiles under the normal model
- Distinguish between raw-score distributions and sampling distributions

These tools prepare you for:

- Sampling distributions (Chapter 7)
- Hypothesis testing and the t distribution (Chapter 8)
- The one-sample t-test (Chapter 9)

39.7 Next Steps

In Chapter 7, you will learn how probability and sampling work together to form the basis of **statistical inference**, including why the sample mean becomes approximately normal even when the raw data are not.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 7

40.1 Probability, Sampling, and the Distribution of Sample Means

Up to this point, we have focused on **describing** data:

- Chapter 4: What does the distribution *look* like? (graphs)
- Chapter 5: How can we summarize it with numbers? (center & spread)
- Chapter 6: How can we model it with a **normal distribution** and **z-scores**?

Starting in this chapter, we begin the transition from **description** to **inference**. We rarely have data for an entire population. Instead, we:

- draw a **sample**,
- compute statistics (like a mean), and
- use those statistics to say something about the population.

To understand why this works, we need three ideas:

1. Basic probability as long-run relative frequency
2. Random sampling from a population
3. The **distribution of sample means** (a sampling distribution)

This chapter provides the conceptual bridge to the inferential procedures you will encounter in later chapters.

40.2 Why Probability?

In everyday language, “probability” often means “how confident I feel.” In statistics, probability has a more precise interpretation:

Probability is the long-run relative frequency of an event under repeated conditions.

For example, if you flip a fair coin many times, the proportion of heads will stabilize around 0.5. In the long run:

$$P(\text{Heads}) = 0.5$$

We use probabilities in statistics to quantify uncertainty about:

- which sample we might observe,
- how far a sample mean might fall from the population mean,
- and how surprising our data would be if a null hypothesis were true.

40.3 Populations, Samples, and Sampling Error

A **population** is the full set of individuals or observations we care about. A **sample** is a subset of that population that we actually measure.

Key ideas:

- The **population mean** (often written μ) is usually unknown.
- The **sample mean** (often written \bar{x}) is an estimator of μ .
- Different random samples produce different sample means. This variability is called **sampling error**.

Sampling error is not a mistake. It is a built-in feature of working with samples.

40.4 Random Sampling

To keep our reasoning clean, we often assume we have a **random sample**. Informally, this means:

- Every member of the population has some chance of being included.
- The selection mechanism does not systematically favor certain individuals.

In practice, real psychological samples (e.g., volunteers from a subject pool) are rarely perfectly random. However, random sampling is a useful *ideal model* for understanding variability in sample statistics.

40.5 The Distribution of Sample Means

Imagine we could:

1. Start with a large population (or a very large synthetic dataset).
2. Draw many random samples of size n .
3. Compute the sample mean for each sample.

If we repeated this process a large number of times and graphed all the sample means, we would obtain the **sampling distribution of the mean**.

Important properties of the sampling distribution of the mean:

- It is centered near the population mean μ .
- Its spread is given by the **standard error of the mean**:

$$SE_{\bar{X}} = \frac{\sigma}{\sqrt{n}}$$

- As the sample size n increases, the sampling distribution becomes narrower (less variable).
- Under broad conditions, the sampling distribution of the mean is approximately **normal**, even when the raw data are skewed.

This last point is the heart of the **Central Limit Theorem**.

40.6 The Central Limit Theorem (Informal)

The **Central Limit Theorem** (CLT) can be stated informally as follows:

If you draw many independent random samples of size n from a population with mean μ and standard deviation σ , then for sufficiently large n , the distribution of the sample mean \bar{x} will be approximately normal, centered at μ , with standard deviation σ / \sqrt{n} .

This is remarkable because it holds even when the raw data are **not** normal. For example, reaction times are often positively skewed, but the distribution of sample means of reaction times can still be quite normal-looking.

In later chapters, the CLT justifies many inferential procedures, including confidence intervals and hypothesis tests.

40.7 PyStatsV1 Lab: Simulating Sampling Distributions

In this lab, you will use PyStatsV1 to build intuition for sampling distributions:

1. Generate a **synthetic population** of “stress scores” that is skewed (not normal).
2. Draw many random samples of size n from this population.
3. Compute the sample mean for each sample.
4. Plot:
 - the **population distribution**, and
 - the **distribution of sample means**.
5. Compare their shapes and spreads.
6. Relate the observed spread of sample means to the idea of **standard error**.

All code for this lab lives in:

- `scripts/sim_psych_ch7_sampling.py`

and it will write outputs to:

- `data/synthetic/psych_ch7_population_stress.csv` (population)
- `data/synthetic/psych_ch7_sample_means.csv` (sample means)
- `outputs/track_b/ch07_population_vs_sample_means.png` (plot)

40.7.1 Running the Lab Script

From the project root, you can run the Chapter 7 lab script directly:

```
python -m scripts.sim_psych_ch7_sampling
```

If you prefer to use `make` and your Makefile defines the convenience target, you can run:

```
make psych-ch07
```

This will:

- Generate a large synthetic population of “stress scores”
- Draw many random samples of size n
- Compute the sample mean for each sample
- Save the population and sample means to CSV files
- Produce a plot comparing the population distribution and the sampling distribution of the mean
- Print summary information to the console

40.7.2 Expected Console Output

Your exact numbers may vary slightly depending on configuration, but you should see output similar to:

```
Generated population with 50000 individuals
Population mean stress_score = 19.98
Population SD    stress_score = 9.95

Drew 1000 samples of size n = 25
Sampling distribution mean = 20.02
Sampling distribution SD    = 2.02 (theoretical SE  1.99)

Wrote population to: data/synthetic/psych_ch7_population_stress.csv
Wrote sample means to: data/synthetic/psych_ch7_sample_means.csv
Wrote plot to: outputs/track_b/ch07_population_vs_sample_means.png
```

40.7.3 Interpreting the Plot

The script produces a figure with two histograms:

1. The **population distribution** of `stress_score`: * typically skewed to the right (more low-to-moderate scores, fewer high scores).
2. The **sampling distribution of the mean** (sample means): * much less skewed, * more bell-shaped, * and noticeably narrower (less variable).

Questions to consider:

- How does the shape of the sample means compare to the shape of the population?
- Why is the distribution of sample means narrower?
- How does the observed SD of the sample means compare to the theoretical standard error σ/\sqrt{n} ?

40.7.4 Your Turn: Experiments with Sample Size

Try modifying the script (or using function arguments, if exposed) to explore:

1. **Different sample sizes** Compare $n = 5$, $n = 25$, and $n = 100$. How does the spread of the sampling distribution change?
2. **Number of replications** Increase the number of samples (e.g., from 200 to 2000). Does the sampling distribution look smoother? Does the observed SD of the sample means get closer to the theoretical standard error?
3. **Different population shapes** Experiment with different population-generating mechanisms (e.g., less skewed vs. more skewed distributions). How does this affect the sampling distribution for small vs. large sample sizes?

40.8 Summary

In this chapter you learned how:

- Probability can be interpreted as long-run relative frequency.
- Samples differ from populations due to **sampling error**.
- The **sampling distribution of the mean** is centered at the population mean and becomes less variable as the sample size increases.

- Under broad conditions, the sampling distribution of the mean is approximately normal (the **Central Limit Theorem**).
- Simulation can make these abstract ideas concrete and visual.

These ideas are the backbone of the inferential tools in later chapters:

- Hypothesis testing (Chapter 8)
- The one-sample t-test (Chapter 9)
- Confidence intervals for means
- And more advanced models in later chapters

40.9 Next Steps

In Chapter 8, we will build on these ideas to introduce the logic of **null hypothesis significance testing (NHST)**, using probability to decide whether an observed sample is “surprising” under a particular null model.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 8

41.1 Hypothesis Testing and the One-Sample t-Test

In Chapters 6 and 7 you learned:

- how to model a variable with the **normal distribution** and interpret **z-scores** (Chapter 6), and
- how repeated sampling leads to a **sampling distribution of the mean** (Chapter 7).

In this chapter we put those pieces together to introduce **null hypothesis significance testing (NHST)** for a **single mean** using the **one-sample t-test**.

Our goals are to help you:

- understand the logic of NHST as a decision procedure,
- interpret the **one-sample t-statistic**,
- see how sampling variability drives the **p-value**, and
- connect the theoretical t-test to a **simulation-based view**.

41.2 The Logic of NHST for a Single Mean

Suppose we have a quantitative variable such as `stress_score` and we want to test whether the mean in a population is equal to some reference value μ_0 (for example, a published norm or a policy target).

The one-sample t-test follows these steps:

1. **State the hypotheses.**

$$\begin{aligned} H_0 : \mu &= \mu_0 && \text{(null hypothesis)} \\ H_1 : \mu &\neq \mu_0 && \text{(two-sided alternative)} \end{aligned}$$

2. **Collect a sample** of size n from the population and compute:

- the sample mean \bar{x} ,
- the sample standard deviation s .

3. **Compute the test statistic.**

Because the population standard deviation σ is unknown, we use the **t-statistic**:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

This tells us how many **standard errors** the sample mean is from the null value μ_0 .

4. Ask: how surprising is this t if H_0 were true?

Under H_0 and certain assumptions (independent observations, approximately normal population), the t-statistic follows a **t distribution** with $n - 1$ degrees of freedom.

The **p-value** is the probability of obtaining a t-statistic as extreme as (or more extreme than) the one we observed if H_0 is true.

5. Make a decision.

- If the p-value is small (for example, $p < 0.05$), we say our observed t is **unlikely under the null**, and we **reject** H_0 .
- If the p-value is not small, we **fail to reject** H_0 . This does *not* prove that H_0 is true; it only says the data are not very inconsistent with it.

41.3 Connecting to Chapter 7: Sampling Distributions

In Chapter 7 you simulated the **sampling distribution of the mean** for a stress scale. You saw that:

- the distribution of sample means is centered near the true population mean,
- its spread shrinks as n increases (standard error idea), and
- most sample means lie close to the population mean, with a few in the tails.

The one-sample t-statistic builds directly on that idea, but with an extra wrinkle: we do not know the population standard deviation σ , so we estimate it with the sample standard deviation s . Because s varies from sample to sample, the distribution of t has **heavier tails** than the normal distribution.

41.4 Simulation-Based View of a One-Sample Test

In Chapter 7, you simulated the sampling distribution of the mean. You tracked where sample means ended up when repeatedly sampling from a fixed population.

In this chapter's lab, we use a similar idea to approximate a p-value, but with a crucial twist: instead of just tracking means, we track **t-statistics**.

1. Generate a large synthetic population of **stress scores**.
2. Specify a null hypothesis about the population mean:

$$H_0 : \mu = \mu_0$$

3. Draw a single random sample from this population and compute:
 - the sample mean \bar{x} ,
 - the sample standard deviation s ,
 - the observed t-statistic t_{obs} .
4. Construct a world where H_0 is exactly true by **recentering** the population so its mean is μ_0 . The shape and spread stay the same; only the center changes.
5. Draw many random samples from this recentered population. For **each** simulated sample, compute its own mean, its own standard deviation, and its own t-statistic t_{sim} .
6. Approximate the two-sided p-value as:

$$\hat{p} = \frac{\text{number of simulations with } |t_{\text{sim}}| \geq |t_{\text{obs}}|}{\text{number of simulations}}$$

This is a simulation-based analogue of the theoretical p-value. By checking how extreme our *t*-statistic is compared to the distribution of *simulated t*-statistics, we correctly account for the uncertainty in estimating the standard deviation.

41.5 PyStatsV1 Lab: A One-Sample Test on Stress Scores

In this lab, you will:

1. Generate a large synthetic population of stress scores.
2. State a null hypothesis about the population mean.
3. Draw one random sample of size n and compute the observed t -statistic.
4. Use simulation to generate a **null distribution of t-values**.
5. Approximate the two-sided p-value by locating your observed t in that distribution.
6. Make a decision about whether to reject the null hypothesis at $\alpha = 0.05$.

All code for this lab lives in:

- `scripts/psych_ch8_one_sample_test.py`

and it will write outputs to:

- `data/synthetic/psych_ch8_population_stress.csv` (population),
- `data/synthetic/psych_ch8_null_t_values.csv` (simulated t-values),
- optionally `outputs/track_b/ch08_null_t_distribution.png` (plot).

41.5.1 Running the Lab Script

From the project root, run:

```
python -m scripts.psych_ch8_one_sample_test
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch08
```

This will:

- Generate a synthetic `stress_score` population.
- Specify a null value μ_0 (for example, 20).
- Draw a sample of size n (for example, 25).
- Compute the observed t-statistic for $H_0 : \mu = \mu_0$.
- Simulate a null distribution by recentring the population, resampling, and computing t for every resample.
- Estimate a two-sided p-value as a long-run relative frequency.
- Print a verbal conclusion (reject vs fail to reject at $\alpha = 0.05$).
- Optionally, save a plot of the simulated t-distribution with the observed t-statistic marked.

41.5.2 Expected Console Output

Your exact numbers will vary, but the output will look similar to:

```
Generated population with 50000 individuals
Population mean stress_score = 19.98
Population SD    stress_score = 9.95

Null hypothesis: mu = 20.00
Observed sample size n = 25
Observed sample mean   = 22.13
Observed sample SD     = 10.31
t statistic            = 1.03

Using 4000 simulations under H0...
Approximate two-sided p-value = 0.31
Decision at alpha = 0.05: fail to reject H0
```

41.5.3 Interpreting the Output

Focus on the following pieces:

- The **observed t-statistic**: how many standard errors the sample mean is from the null value.
- The **p-value**: the probability of obtaining a t-statistic this extreme (or more) if the null hypothesis were true.
- The **decision**: the binary result based on your alpha threshold. Remember that “fail to reject” is not the same as “prove the null true.”

41.5.4 Your Turn: Practice with Different Scenarios

1. Change the null hypothesis

Modify the null value μ_0 in the script. How does this change the t-statistic and the resulting p-value?

2. Change the sample size

Increase the sample size n (for example, from 25 to 100). Notice how the t-statistic changes. The \sqrt{n} in the denominator makes the test more sensitive to small departures as the sample size grows.

3. Replicate the experiment

Run the script multiple times. Do you always reach the same decision? If the true mean is close to μ_0 , you may see the decision flip back and forth — this is the nature of sampling variability.

41.5.5 Optional Plot: Null t-Distribution

If enabled in the script, a plot file is saved to:

outputs/track_b/ch08_null_t_distribution.png

The figure shows:

- a histogram of simulated **t-statistics** under H_0 , and
- a vertical line marking your **observed t-statistic**.

Questions to consider:

- Does the histogram look roughly bell-shaped and centered at 0?
- Is your observed line in the main bulk of the distribution (a common result) or out in the thin tails (a rare result)?

41.6 Summary

In this chapter you learned how to:

- frame a research question as a **null hypothesis** about a mean,
- compute and interpret the **one-sample t-statistic**,
- approximate a **p-value** using a simulated null distribution of t-statistics, and
- make a decision to reject or fail to reject H_0 .

These ideas form the backbone of classical inference and set you up for the next steps:

- confidence intervals for a mean,
- comparisons of two means (independent and paired-samples t-tests), and
- more complex models such as ANOVA and regression.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 9

42.1 The One-Sample t-Test and Confidence Intervals

In Chapter 8 you learned the *logic* of hypothesis testing using a simulation-based one-sample **t-test**. You saw how to:

- state a null hypothesis,
- compute a sample t-statistic,
- build a null distribution of t-values under H_0 , and
- approximate a p-value by checking how extreme t_{obs} is.

In this chapter we take the next step: the *analytic* (formula-based) one-sample t-test, and the closely related **95% confidence interval** (CI) for a population mean.

This chapter connects the simulation-based intuition from Chapter 8 to the classical t-test formulas used throughout psychological science.

42.2 When to Use a One-Sample t-Test

Use a **one-sample t-test** when you want to compare a sample mean to a known or hypothesized population mean:

Does the population of students represented by this class have a mean stress score of 20?

Mathematically, the hypotheses are:

$$\begin{aligned}H_0 : \mu &= \mu_0 \\H_1 : \mu &\neq \mu_0\end{aligned}$$

42.3 Why We Use t (Instead of z)

When the population standard deviation σ is unknown—as is almost always the case in psychology—we use the **sample standard deviation** s . Substituting s introduces extra uncertainty, leading to a **t distribution** instead of a normal distribution.

The estimated standard error is:

$$SE = \frac{s}{\sqrt{n}}$$

The t-statistic is:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

42.4 Confidence Intervals

A **95% confidence interval** around a mean provides a range of plausible population values:

$$\bar{x} \pm t^* \frac{s}{\sqrt{n}}$$

where t^* is the critical value from the t distribution with $n - 1$ degrees of freedom.

Interpretation:

If we repeated the study many times, 95% of the resulting CIs would contain the true population mean.

42.5 Connecting Confidence Intervals and Hypothesis Tests

A powerful insight:

*If the 95% CI **does not include** μ_0 , the two-sided t-test will reject H_0 at $\alpha = 0.05$.*

*If the CI **includes** μ_0 , the t-test will fail to reject H_0 .*

42.6 PyStatsV1 Lab: A One-Sample t-Test With Confidence Intervals

In this lab, you will:

1. Load a synthetic population of stress scores (same population used in Ch. 7–8).
2. Draw a random sample of size n .
3. Compute:
 - sample mean \bar{x} ,
 - sample SD s ,
 - standard error SE ,
 - t-statistic,
 - degrees of freedom $df = n - 1$,
 - p-value (analytic),
 - 95% confidence interval.
4. Compare:
 - the analytic t-test,
 - the 95% CI,
 - and (optionally) the simulation results from Chapter 8.

All code for this chapter lives in:

`scripts/psych_ch9_one_sample_ci.py`

42.6.1 Running the Lab Script

From the project root:

```
python -m scripts.psych_ch9_one_sample_ci
```

If you have a Makefile target:

```
make psych-ch09
```

42.6.2 Expected Console Output

Your numbers will vary due to randomness, but output will look similar to:

```
Loaded synthetic population with 50000 individuals
Population mean stress_score = 19.98
Population SD    stress_score = 9.94

Drawn sample size n = 25
Sample mean = 22.10
Sample SD   = 10.44
SE          = 2.09
t statistic = 1.01
df          = 24

Analytic two-sided p-value = 0.323
95% CI = [17.80, 26.40]
```

42.6.3 Interpreting Your Output

Focus on:

- the **t-statistic**: How many standard errors your mean is from the null;
- the **p-value**: Is the result “rare” under H_0 ?
- the **CI**: Does the interval contain the hypothesized value μ_0 ?

42.6.4 Your Turn: Practice

1. **Change the null value** μ_0 and observe how the t-statistic changes.
2. **Change the sample size** n and see how the CI narrows or widens.
3. **Run the analysis multiple times** to see sampling variability.

42.7 Summary

In this chapter you learned:

- the formula-based one-sample t-test,
- how to compute a 95% confidence interval,
- the connection between CIs and hypothesis tests.

In the next chapter (Chapter 10) we extend this logic to comparing **two independent groups** using the independent-samples t-test.

PSYCHOLOGICAL SCIENCE & STATISTICS – CHAPTER 10

43.1 The Independent-Samples t-Test

In Chapter 8, you learned the **logic of hypothesis testing** using a one-sample t-test and a simulated null distribution of t-statistics.

In Chapter 9, you computed an **analytic one-sample t-test and confidence interval** for a single mean using the theoretical t distribution.

In this chapter, we extend those ideas to **comparing two independent groups**. This is the standard “between-subjects” design in experimental psychology: participants are randomly assigned to one of two conditions, and we compare the means.

Typical examples include:

- Control vs. Treatment
- Placebo vs. Drug
- No-training vs. Training

Our running example will again use a **stress_score** variable.

43.2 When to Use the Independent-Samples t-Test

Use an independent-samples t-test when:

- You have **two groups** that are **independent** of each other. (No person appears in both groups.)
- Your dependent variable (DV) is **approximately continuous** and **approximately Normal** within each group.
- You are interested in whether the **population means differ**:

$$H_0 : \mu_1 = \mu_2$$

$$H_1 : \mu_1 \neq \mu_2$$

Here, μ_1 is the population mean for group 1 and μ_2 is the population mean for group 2.

43.3 The Logic of the Independent-Samples t-Test

The basic logic mirrors the one-sample case:

1. **State the hypotheses**

$$H_0 : \mu_1 = \mu_2$$

$$H_1 : \mu_1 \neq \mu_2$$

2. Compute the observed difference in sample means

$$\bar{x}_1 - \bar{x}_2$$

3. Estimate the standard error of the difference (assuming equal variances)

When we assume the two populations have **equal variances**, we first compute a **pooled standard deviation**:

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$$

where

- n_1, n_2 are the group sample sizes
- s_1, s_2 are the sample standard deviations

Then the **standard error of the difference in means** is

$$SE_{\bar{x}_1 - \bar{x}_2} = s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}.$$

Modern Best Practice: Welch's t-test vs. Pooled t-test

The pooled-variance t-test above is the **classical Student's t-test** taught in most introductory textbooks. It assumes that the two populations have **equal variances**.

In real research, however, variances are often **not** equal, especially when group sizes differ. In those situations, the pooled test can have an inflated Type I error rate (too many false positives).

Modern statistical software therefore defaults to **Welch's t-test**, which does **not** assume equal variances and adjusts the degrees of freedom using the Welch–Satterthwaite equation.

Table 1: Classical Pooled t-test vs. Welch's t-test

Method	Variance assumption	Degrees of freedom	Typical use
Pooled t-test	Assumes $\sigma_1^2 = \sigma_2^2$	$n_1 + n_2 - 2$	Teaching; balanced designs with similar spreads
Welch's t-test	Does not assume equal variances	Approximate df (Welch–Satterthwaite)	Modern default; safer when variances or n differ

In this chapter we focus on the **pooled** test to explain the logic of comparing two means. In applied work, however, it is good practice to **report Welch's t-test as well**, especially when the variances or sample sizes are noticeably different.

The PyStatsV1 Chapter 10 script prints **both** pooled and Welch results so you can see how they compare on the same data.

4. Compute the t-statistic (pooled version)

$$t_{\text{pooled}} = \frac{\bar{x}_1 - \bar{x}_2}{SE_{\bar{x}_1 - \bar{x}_2}}$$

with degrees of freedom

$$df_{\text{pooled}} = n_1 + n_2 - 2.$$

5. Find the p-value and make a decision (pooled version)

Under H_0 , the statistic t_{pooled} follows a t distribution with $\text{df}_{\text{pooled}}$ degrees of freedom. For a two-sided test, the p-value is

$$p_{\text{pooled}} = 2 \cdot P(T_{\text{df}_{\text{pooled}}} \geq |t_{\text{obs}}|).$$

If $p_{\text{pooled}} < \alpha$ (typically 0.05), we **reject** H_0 and conclude that the group means differ.

43.4 Effect Size: Cohen's d

Statistical significance does not tell us **how large** the effect is. For independent groups with a pooled standard deviation, a common effect size is **Cohen's d** :

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s_p}.$$

Rough guidelines (Cohen, 1988):

- $d \approx 0.2$ – small effect
- $d \approx 0.5$ – medium effect
- $d \approx 0.8$ – large effect

43.5 Confidence Interval for the Mean Difference (Pooled Version)

We can also construct a **confidence interval** for the difference in means:

$$(\bar{x}_1 - \bar{x}_2) \pm t_{\text{crit}} \cdot \text{SE}_{\bar{x}_1 - \bar{x}_2},$$

where t_{crit} is the critical t value from the t distribution with $\text{df}_{\text{pooled}} = n_1 + n_2 - 2$ at your chosen α level (e.g., $\alpha = 0.05$ for a 95% CI).

43.6 PyStatsV1 Lab: Independent-Samples t-Test on Stress Scores

In this lab you will:

1. Generate a synthetic dataset of **stress scores** for two independent groups:
 - **control**
 - **treatment**
2. Compute sample means, standard deviations, and group sizes.
3. Compute the **pooled standard deviation** and **standard error**.
4. Compute the **independent-samples t-statistic (pooled version)** and its **two-sided p-value**.
5. Compute **Cohen's d** as an effect size.
6. Construct a **95% confidence interval** for the difference in means.
7. Compute **Welch's t-test** as a modern, variance-robust comparison.
8. Optionally, visualize the group means with error bars.

All code for this lab lives in:

- `scripts/psych_ch10_independent_t.py`

and the script will optionally write outputs to:

- data/synthetic/psych_ch10_independent_groups.csv
- outputs/track_b/ch10_group_means_with_ci.png

43.6.1 Running the Lab Script

From the project root, run:

```
python -m scripts.psych_ch10_independent_t
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch10
```

This will:

- Generate a synthetic dataset with two groups (e.g., 25 participants per group).
- Compute the independent-samples t-test comparing **control** vs. **treatment** using the **pooled** version.
- Compute **Welch's t-test** on the same data as a **safety check**.
- Compute Cohen's *d* and a 95% confidence interval for the mean difference.
- Print a short APA-style summary line.
- Optionally, save a bar plot of the group means with error bars.

43.6.2 Expected Console Output

Your exact numbers will vary, but the output will look similar to:

```
Generated independent groups with n = 25 per condition
Group: control    mean = 18.48  SD =  8.74  n = 25
Group: treatment  mean = 16.82  SD =  9.87  n = 25

--- Pooled-variance independent-samples t-test (classic Student's t) ---
Mean difference (control - treatment) = 1.66
Pooled SD = 9.32
SE of difference = 2.64
df (pooled) = 48
t (pooled) = 0.63
Two-sided p-value (pooled) = 0.53
95% CI (pooled) for mean difference: [-3.65, 6.97]
Cohen's d (pooled) = 0.18

--- Welch's t-test (modern default, equal_var = False) ---
df (Welch) 45.2
t (Welch) = 0.61
Two-sided p-value (Welch) = 0.54

Wrote data to: data/synthetic/psych_ch10_independent_groups.csv
Wrote plot to: outputs/track_b/ch10_group_means_with_ci.png
```

43.6.3 Interpreting the Output

Focus on the following pieces:

- **Mean difference:** How far apart are the sample means?
- **t statistic and p-value (pooled vs. Welch):** Do the methods agree about whether the difference is statistically significant?
- **Confidence interval:** Does the 95% CI for $\mu_1 - \mu_2$ include zero?
- **Cohen's d :** How large is the effect in standardized units?

43.6.4 Your Turn: Practice Scenarios

1. Change the group means

In `psych_ch10_independent_t.py`, try changing the assumed population means for the two groups. How does this affect the mean difference, t , and Cohen's d ?

2. Change the sample size

Increase n per group (e.g., from 25 to 100). Notice how the standard error shrinks and the test becomes more sensitive to small differences.

3. Make the variances very different

Use very different standard deviations for the two groups. Compare the pooled and Welch results. How do the degrees of freedom and p-values differ?

4. Practice APA-style reporting

Using the script output, practice writing a short APA-style sentence, e.g.:

“Participants in the treatment condition did not differ significantly from those in the control condition on stress scores, $t(48) = 0.63$, $p = .53$, $d = 0.18$ (pooled).”

CHAPTER 11 – WITHIN-SUBJECTS DESIGNS AND THE PAIRED-SAMPLES *T*-TEST

In Chapter 8, you learned the logic of hypothesis testing using a simulation-based one-sample *t*-test.

In Chapter 9, you connected that simulation intuition to the analytic one-sample *t*-test and 95% confidence intervals for a single mean.

In Chapter 10, you extended those ideas to comparing **two independent groups** using the independent-samples *t*-test (pooled vs. Welch). That was a classic **between-subjects** design: each person appeared in only one condition.

In this chapter, we move to **within-subjects** designs. Instead of asking whether two different groups of people differ, we ask whether the **same people change over time** or across conditions.

Our goals are to help you:

- understand when a paired-samples *t*-test is appropriate,
- see why within-subjects designs can be more statistically powerful,
- recognize design threats specific to repeated measures (practice, fatigue, carryover),
- understand basic counterbalancing strategies, and
- run and interpret a paired-samples *t*-test using PyStatsV1.

44.1 Design Logic: Repeated Measures and Matched-Subjects

Suppose a lab wants to know whether a short training module improves performance on a problem-solving test. Each participant completes the test:

- once **before** the training (pre-test), and
- once **after** the training (post-test).

We record a continuous score (for example, number of problems solved). The key design feature is:

The same people produce both the pre and post scores.

This is a **within-subjects** (or **repeated-measures**) design. A closely related idea is a **matched-subjects** design, where participants are paired on important characteristics (for example, age, IQ, baseline performance) and each member of the pair receives a different condition. Analytically, each pair is treated like a “unit”, similar to a person measured twice.

In both cases, the data naturally form **pairs**:

- (pre, post) for each participant, or
- (score in condition A, score in condition B) for each matched pair.

From Chapter 1’s perspective:

- **Construct validity:** Does the test measure the construct (skill, ability) we care about?
- **External validity:** Would the training effect generalize beyond this sample and this test?
- **Statistical validity:** Is the observed change statistically reliable? (This is where the t -test lives.)
- **Internal validity:** Are we justified in saying the **training** caused the change, or could some other factor (for example, practice, fatigue) explain the difference?

The paired-samples t -test is a tool for statistical validity. To protect internal validity, we also need to think carefully about design issues unique to repeated measures.

44.2 The Power of “Self-Control”: Reducing Individual Differences Error

Why bother with within-subjects designs?

In a between-subjects design (Chapter 10), people differ in many ways that have nothing to do with the experiment: baseline ability, motivation, prior knowledge, sleep, and so on. Those differences inflate the variability of scores within each group, which shows up as **error variance** in the denominator of the t -statistic.

In a within-subjects design, we focus on **difference scores** for each person:

$$D_i = X_{i,\text{post}} - X_{i,\text{pre}}.$$

If person i is generally high-performing, they tend to be high at both pre and post. When we subtract, that stable “true ability” component largely cancels out. What is left in the difference is:

$$\text{Difference} = \text{Condition Effect} + \text{Residual Noise}$$

By removing stable between-person differences, we:

- shrink the variability of the difference scores,
- shrink the standard error of the mean difference, and
- make it easier to detect a real effect (greater **power**).

This is sometimes called the “**self-control**” idea:

Each participant serves as their own control condition.

The paired-samples t -test is the analytic tool that formalizes this idea.

44.3 Issues: Practice Effects, Fatigue, and Carryover

Within-subjects designs are statistically powerful, but they have their own **design risks**. Three big ones are:

- **Practice effects**

Participants might improve simply because they have seen the test before. On the second attempt, they know the format, remember some items, or adopt better strategies. Improvement may reflect **familiarity with the test**, not the training.

- **Fatigue effects**

If the tasks are long or demanding, participants can become tired, bored, or less attentive over time. Performance might deteriorate at Time 2 even if the training helped, simply because participants are exhausted.

- **Carryover effects**

What happens in the first condition changes how people respond in later conditions. In drug studies, the effect of the first dose might still be active during the second. In cognitive tasks, strategies learned in one condition carry over to the next.

These problems are **not** solved by a different t -test formula. They are **design problems**, not analysis problems.

The paired-samples t -test can tell you whether the average difference is reliably different from zero. It cannot tell you *why* that difference exists.

This connects back to **internal validity**: if practice, fatigue, or carryover are plausible alternative explanations, your ability to claim a causal effect is weakened.

44.4 Counterbalancing: Controlling for Order Effects

One of the main tools psychologists use to deal with order-related threats is **counterbalancing**: systematically varying the order of conditions across participants so that order effects can be detected or averaged out.

Common strategies include:

- **Simple AB / BA counterbalancing**

Two conditions, A and B. Half the participants receive A then B, the other half B then A. Practice or fatigue effects are spread across conditions rather than confounded with one specific condition.

- **Full counterbalancing**

For three conditions A, B, C you could run all possible orders (ABC, ACB, BAC, BCA, CAB, CBA). This becomes impractical as the number of conditions grows (the number of orders grows factorially).

- **Latin square and related designs**

For many conditions, a Latin square ensures each condition appears in each position equally often and follows each other condition equally often, without using all possible orders.

How does this apply to a simple pre–post training study?

- You **cannot** undo the training to swap the order (once trained, always trained).
- Instead, researchers often:
 - Use **alternate forms** of the test (Form A at pre, Form B at post) and counterbalance which form comes first.
 - Include a **control group** that completes the same pre/post testing but receives no training (or a placebo training).
 - Add **rest breaks** or shorter test batteries to reduce fatigue.

You will see these ideas again in mixed-model designs (Chapter 17), where pre–post measurements are combined with separate groups (for example, Training vs. Control).

44.5 The Paired-Samples t -Test Formula

Mathematically, the paired-samples t -test works on the **difference scores**:

- For each participant i , compute

$$D_i = X_{i,\text{post}} - X_{i,\text{pre}}.$$

- Let there be n participants (so n differences).

Define:

- Mean difference

$$\bar{D} = \frac{1}{n} \sum_{i=1}^n D_i$$

- Standard deviation of differences

$$s_D = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (D_i - \bar{D})^2}$$

- Standard error of the mean difference

$$SE_D = \frac{s_D}{\sqrt{n}}$$

The hypotheses are:

$$H_0 : \mu_D = 0 \quad (\text{no average change})$$

$$H_1 : \mu_D \neq 0 \quad (\text{some average change})$$

The paired-samples t -statistic is:

$$t = \frac{\bar{D} - \mu_{D,0}}{s_D / \sqrt{n}},$$

where $\mu_{D,0}$ is the null value for the mean difference (usually 0), and the degrees of freedom are:

$$df = n - 1.$$

Under H_0 , if the difference scores are approximately Normal and independent across participants (even though pre and post within a participant are correlated), this t -statistic follows a t distribution with $n - 1$ degrees of freedom.

44.6 Effect Size: Cohen's d_z

Just like in independent samples, we need to report the **magnitude** of the effect, not only whether it is statistically significant.

For paired samples, a common measure is **Cohen's d_z** :

$$d_z = \frac{\bar{D}}{s_D}$$

This expresses the mean difference in terms of standard deviation units of the *difference scores*.

Note: There are other ways to calculate d for paired samples (for example, using the average standard deviation of pre and post scores), but d_z is the direct analogue to the t -statistic because it uses the same standard deviation that appears in the denominator of the paired test.

Rough guidelines (Cohen, 1988):

- $d \approx 0.2$ – small effect
- $d \approx 0.5$ – medium effect
- $d \approx 0.8$ – large effect

Reporting example:

“Participants’ scores improved significantly from pre to post, $t(39) = 3.75$, $p = .001$, $d_z = 0.59$, indicating a medium-sized effect of the training.”

44.7 Confidence Interval for the Mean Difference

A 95% confidence interval for the population mean difference μ_D is:

$$\bar{D} \pm t^* \cdot \frac{s_D}{\sqrt{n}},$$

where t^* is the critical value from the t distribution with $df = n - 1$ at $\alpha = 0.05$.

Interpretation is parallel to Chapter 9:

If we repeated the study many times, 95% of the resulting confidence intervals would contain the true mean difference μ_D .

If a 95% CI for μ_D **does not include 0**, a two-sided paired t -test will reject $H_0 : \mu_D = 0$ at $\alpha = 0.05$. If the CI **does include 0**, the test will fail to reject H_0 .

44.8 What If We (Wrongly) Treated the Data as Independent?

A common mistake is to ignore the pairing and run an independent-samples t -test on the pre and post scores as if they came from two different groups.

If we do that:

- Each person's pre and post scores are separated into different "groups".
- The analysis no longer uses the **within-person correlation**.
- Between-person variability (stable differences in ability) sits in the error term.
- The test usually has **less power** (smaller absolute t , larger p).

In the PyStatsV1 Chapter 11 code, we include a function that deliberately performs this mis-specified independent t -test so you can see the difference in practice.

Take-away: Using the wrong test wastes information and can make real effects look non-significant.

44.9 PyStatsV1 Lab: A Paired t -Test for a Training Study

In this lab, you will analyze a synthetic pre-post training study using the paired-samples t -test.

You will:

- simulate a repeated-measures dataset with pre and post scores for each participant,
- compute:
 - mean pre score and mean post score,
 - difference scores D_i ,
 - mean difference \bar{D} ,
 - s_D , standard error, t -statistic, p -value, and 95% CI,
 - Cohen's d_z as an effect size for the mean difference,
- compare the correct paired t -test to a mis-specified independent-samples t -test on the same data, to see the power advantage of using the right model.

All code for this lab lives in:

- `scripts/psych_ch11_paired_t.py`

and the script can optionally write outputs to:

- `data/synthetic/psych_ch11_paired_training.csv`

44.10 Running the Lab Script

From the project root, you can run:

```
python -m scripts.psych_ch11_paired_t
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch11
```

This will:

- simulate a pre–post training dataset (for example, 40 participants),
- compute the paired-samples t -test for the mean difference in scores,
- compute Cohen’s d_z from the mean difference and s_D ,
- print:
 - sample size and degrees of freedom,
 - mean pre and post scores,
 - mean difference,
 - t -statistic, p -value, and a 95% CI for μ_D ,
 - Cohen’s d_z as an effect size,
- optionally save the dataset as a CSV file for further exploration.

44.11 Expected Console Output

Your exact numbers will vary if you change the seed or parameters, but with the default settings, you will see something like:

```
Paired samples t-test for pre-post training study
n = 40, df = 39
Mean(pre) = 72.23
Mean(post) = 77.67
Mean diff = 5.44
t(39) = 3.75, p = 0.0006
95% CI for mean diff: [2.51, 8.37]
Cohen's d_z = 0.59
```

Focus on:

- **Mean(pre) and Mean(post):** Did scores increase, and by how much (in raw units)?
- **Mean diff:** The average post–pre difference \bar{D} .
- **t-statistic:** How many standard errors the mean difference is from 0.
- **p-value:** Is this difference “rare” under $H_0 : \mu_D = 0$?
- **95% CI:** A range of plausible values for the true mean change.

- **Cohen's d_z** : How large is the effect in standardized units of the difference scores?

44.12 Your Turn: Practice Scenarios

As in previous chapters, you can experiment by changing the simulation parameters in `psych_ch11_paired_t.py`:

- **Change the mean change**

Increase or decrease the assumed population mean difference. How does this affect \bar{D} , t , the p -value, and Cohen's d_z ?

- **Change the variability of change**

Increase `sd_change` to make individual change scores more variable. What happens to s_D , the standard error, the width of the 95% CI, and d_z ?

- **Change the sample size n**

Increase the number of participants (for example, from 20 to 80). Observe how the standard error shrinks and the t -test becomes more sensitive to the same mean difference. Does d_z change?

- **Compare paired vs. mis-specified independent tests**

Use the helper function for the mis-specified independent-samples t -test (described in the code). Compare the t -statistics and p -values. Do you see that the paired test usually produces a larger absolute t (more power) while Cohen's d_z stays tied to the actual magnitude of within-person change?

- **Design reflection**

Imagine realistic practice or fatigue effects for this training study. How might you redesign the study (alternate test forms, control group, rest breaks, counterbalancing) to protect internal validity?

44.13 Summary

In this chapter you learned:

- when to use a paired-samples t -test: whenever your data come in **pairs** from the same (or tightly matched) units,
- how within-subjects designs reduce individual differences error and increase power by treating each participant as their own control,
- design threats specific to repeated measures: practice effects, fatigue, and carryover effects,
- how counterbalancing and related design strategies help control order effects,
- how to compute and interpret the paired-samples t -statistic, p -value, 95% CI for a mean difference, and Cohen's d_z as an effect size, using PyStatsV1.

In the bigger arc:

- Chapter 8: NHST logic with a simulation-based one-sample t -test.
- Chapter 9: Analytic one-sample t -test and confidence intervals.
- Chapter 10: Independent-samples t -test for between-subjects designs.
- Chapter 11: Paired-samples t -test for within-subjects designs, with an appropriate effect size measure.

Together, these chapters give you a solid toolkit for simple experiments with one independent variable, whether the design uses independent groups or repeated measures. In later chapters, you will extend these ideas to more complex designs and models (ANOVA, regression, mixed models), but the core logic will remain the same.

For the full Python implementation, see `scripts/psych_ch11_paired_t.py` in the PyStatsV1 GitHub repository.

CHAPTER 12 – ONE-WAY ANALYSIS OF VARIANCE (ANOVA)

In Chapters 8–11 you learned how to compare **one** mean (one-sample t) and **two** means (independent and paired-samples t).

In this chapter we take the next step: comparing **three or more groups** in a single experiment. The standard tool is the **one-way analysis of variance (ANOVA)**.

Typical psychology examples include:

- three therapy conditions (for example, control, CBT, mindfulness),
- four study strategies (rereading, highlighting, practice testing, mixed), or
- multiple dose levels of a drug.

Our goals are to help you:

- understand why simply running many t -tests is a bad idea (inflated Type I error),
- see how ANOVA **partitions variance** into between-groups and within-groups components,
- interpret the F -ratio as a signal-to-noise statistic,
- understand why we need **post-hoc tests** when F is significant, and
- run a one-way ANOVA and simple Bonferroni-corrected post-hoc tests using PyStatsV1.

45.1 The Problem with Multiple t -Tests

Suppose a clinical psychologist compares stress scores across **three conditions**:

- control (no treatment),
- CBT training, and
- mindfulness training.

A naïve approach is to run a separate independent-samples t -test for each pair:

- control vs. CBT,
- control vs. mindfulness,
- CBT vs. mindfulness.

That is **three** tests. If each test uses $\alpha = 0.05$, what is the chance of making **at least one** Type I error (false positive) across all three tests, *even if all population means are equal*?

If each test has a 0.05 chance of a false positive and we (roughly) treat them as independent, then the chance of **no** Type I errors is

$$P(\text{no false positives}) \approx (1 - 0.05)^3 = 0.95^3 \approx 0.86.$$

So the chance of **at least one** false positive is about:

$$P(\text{at least one false positive}) \approx 1 - 0.86 = 0.14.$$

Instead of a 5% family-wise error rate, we are now closer to **14%**. As the number of groups grows, the number of pairwise tests grows quickly and the family-wise error rate can become unacceptably large.

The core idea of ANOVA is:

Rather than doing many separate t -tests, we perform **one overall test** of the null hypothesis that all group means are equal.

If that global test is significant, we then follow up with more targeted comparisons (post-hoc tests) using methods that control the overall error rate.

45.2 Partitioning Variance: Between-Groups vs. Within-Groups

ANOVA works by **partitioning** the total variability in the data into two parts:

- variability **between groups** (how far the group means are spread out), and
- variability **within groups** (how spread out the scores are inside each group).

Let:

- k be the number of groups,
- N be the total sample size (sum of all group sizes),
- \bar{X}_j be the mean of group j ,
- n_j be the sample size in group j , and
- \bar{X} be the **grand mean** across all participants.

45.2.1 Total Sum of Squares

The total variability around the grand mean is:

$$SS_{\text{Total}} = \sum_{j=1}^k \sum_{i=1}^{n_j} (X_{ij} - \bar{X})^2.$$

45.2.2 Between-Groups Sum of Squares

The between-groups (or treatment) sum of squares measures how far the **group means** are from the grand mean, weighted by group size:

$$SS_{\text{Between}} = \sum_{j=1}^k n_j (\bar{X}_j - \bar{X})^2.$$

45.2.3 Within-Groups Sum of Squares

The within-groups (or error) sum of squares measures how spread out the scores are **inside** each group:

$$SS_{\text{Within}} = \sum_{j=1}^k \sum_{i=1}^{n_j} (X_{ij} - \bar{X}_j)^2.$$

These pieces satisfy the key identity:

$$SS_{\text{Total}} = SS_{\text{Between}} + SS_{\text{Within}}.$$

45.2.4 Degrees of Freedom

Each sum of squares has associated degrees of freedom (df):

- Between groups:

$$df_{\text{Between}} = k - 1.$$

- Within groups:

$$df_{\text{Within}} = N - k.$$

- Total:

$$df_{\text{Total}} = N - 1.$$

45.2.5 Mean Squares

ANOVA converts sums of squares to **mean squares** by dividing by their degrees of freedom:

$$MS_{\text{Between}} = \frac{SS_{\text{Between}}}{df_{\text{Between}}},$$

$$MS_{\text{Within}} = \frac{SS_{\text{Within}}}{df_{\text{Within}}}.$$

- MS_{Within} is an estimate of the population variance based on within-group variability.
- If the null hypothesis is true, MS_{Between} is also an estimate of the same variance (plus tiny sampling noise).
- If the group means really differ, MS_{Between} becomes **larger** than MS_{Within} .

45.3 The F -Ratio: Signal-to-Noise Logic

The ANOVA test statistic is the **F-ratio**:

$$F = \frac{MS_{\text{Between}}}{MS_{\text{Within}}}.$$

This has the **same basic logic** as the t -tests you have seen:

- The **numerator** reflects systematic differences between group means (signal).
- The **denominator** reflects unsystematic variability within groups (noise).

When all population means are equal, the expected values of MS_{Between} and MS_{Within} are the same, so F tends to be near 1. As the group means separate, the numerator grows relative to the denominator and F becomes larger than 1.

Under the null hypothesis that all population means are equal,

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_k,$$

the F -statistic follows an **F distribution** with

- $df_1 = df_{\text{Between}} = k - 1,$
- $df_2 = df_{\text{Within}} = N - k.$

For a right-tailed test, the p -value is:

$$p = P(F_{df_1, df_2} \geq F_{\text{obs}}).$$

If $p < \alpha$ (for example, 0.05), we reject H_0 and conclude that **at least one** group mean differs from the others.

Professional Tip: Assumptions Matter

The standard ANOVA F-test assumes that all groups have roughly **equal variances** (homogeneity of variance).

When sample sizes are unequal and group variances differ a lot, the classical F-test can be misleading. In professional research, statisticians often check this assumption (for example, using **Levene's test**) and may switch to a robust alternative called **Welch's ANOVA** if the assumption is badly violated.

In this chapter we focus on the classical, equal-variance version for clarity. Later, when you read research articles, pay attention to how authors check (or ignore) this assumption.

45.4 Effect Size: η^2

As with t -tests, statistical significance does not tell us how large an effect is. A simple effect size for one-way ANOVA is **eta-squared**:

$$\eta^2 = \frac{SS_{\text{Between}}}{SS_{\text{Total}}}.$$

Interpretation:

- η^2 represents the **proportion of total variance** in the dependent variable that can be attributed to group membership.
- Values range from 0 to 1, with higher values indicating a stronger relationship between the grouping factor and the outcome.

Very rough guidelines:

- $\eta^2 \approx 0.01$ – small effect
- $\eta^2 \approx 0.06$ – medium effect
- $\eta^2 \approx 0.14$ – large effect

Note

η^2 is a descriptive statistic for the *sample*. It tends to slightly **overestimate** the effect size in the *population*, especially with small samples. Advanced researchers often use a corrected measure called **omega-squared** (ω^2) for less biased estimates, but η^2 is a standard and useful starting point for introductory ANOVA.

45.5 Post-Hoc Tests: Where Is the Difference?

A significant ANOVA tells you that not all means are equal, but it does **not** tell you which pairs of groups differ.

For our three-condition example, a significant F might reflect:

- CBT < control, mindfulness < control, and CBT < mindfulness,
- or CBT < mindfulness, but both similar to control,

- or some other pattern.

To answer “Where is the difference?” we use **post-hoc tests** (or planned contrasts). These tests compare specific means while controlling the overall (Type I) error rate.

Two common approaches are:

- **Tukey’s Honestly Significant Difference (HSD)**
 - Uses a special distribution (studentized range) tailored to all pairwise comparisons.
 - Controls the family-wise error rate at α across all pairs.
 - Widely used default in many statistical packages.
- **Bonferroni correction**
 - Very simple: - Decide on the number of comparisons m . - Use a per-comparison alpha of α/m , or equivalently multiply each p -value by m and compare to α .
 - Conservative but easy to explain and implement.

In the PyStatsV1 Chapter 12 lab we use **Bonferroni-corrected pairwise t -tests** so that you can see:

- the connection to the Chapter 10 independent-samples t -test, and
- how the correction keeps the family-wise error rate under control.

45.6 PyStatsV1 Lab: One-Way ANOVA on Stress Scores

In this lab, you will analyze a synthetic experiment in which students are randomly assigned to one of **three** conditions:

- `control` – no stress-management training,
- `cbt` – a brief cognitive-behavioral training module,
- `mindfulness` – a brief mindfulness training module.

The outcome variable is a continuous `stress_score` similar to earlier chapters.

You will:

- simulate a dataset with stress scores for all three groups,
- compute:
 - group means and sample sizes,
 - SS_{Between} , SS_{Within} , SS_{Total} ,
 - MS_{Between} , MS_{Within} ,
 - the F -statistic and its p -value,
 - η^2 as an effect size,
- perform Bonferroni-corrected pairwise t -tests between the three groups,
- cross-check your ANOVA results against SciPy’s `f_oneway` as a **safety check**.

All code for this lab lives in:

- `scripts/psych_ch12_one_way_anova.py`

and the script can optionally write outputs to:

- `data/synthetic/psych_ch12_one_way_stress.csv`

45.7 Running the Lab Script

From the project root, you can run:

```
python -m scripts.psych_ch12_one_way_anova
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch12
```

This will:

- simulate a one-way between-subjects dataset with three groups (for example, 30 participants per group),
- compute the one-way ANOVA table:
 - SS_{Between} , df_{Between} , MS_{Between} ,
 - SS_{Within} , df_{Within} , MS_{Within} ,
 - F , p , and η^2 ,
- print the group means and sample sizes,
- run pairwise independent-samples t -tests for:
 - control vs. cbt,
 - control vs. mindfulness,
 - cbt vs. mindfulness,
 and report both uncorrected p -values and Bonferroni-adjusted p -values,
- cross-check against SciPy's built-in `f_oneway` and warn if the values disagree beyond tiny numerical differences,
- optionally save the simulated dataset as a CSV file for further exploration.

45.8 Expected Console Output

Your exact numbers will vary if you change the seed or parameters, but with the default settings you will see something like:

```
One-way ANOVA on stress scores (control vs CBT vs mindfulness)
-----
Group means (n per group = 30):
control      mean = 18.73
cbt          mean = 14.91
mindfulness  mean = 12.32

ANOVA table:
SS_between = 1235.48, df_between = 2, MS_between = 617.74
SS_within  = 6710.26, df_within  = 87, MS_within   = 77.01
SS_total   = 7945.74, df_total   = 89
F(2, 87) = 8.02, p = 0.0006
eta^2 = 0.16

Pairwise comparisons (Bonferroni-corrected p-values):
control vs cbt:          t(58) = 1.98, p_unc = 0.052, p_bonf = 0.155
```

(continues on next page)

(continued from previous page)

```
control vs mindfulness:  t(58) = 3.63, p_unc = 0.001, p_bonf = 0.004
cbt vs mindfulness:      t(58) = 1.65, p_unc = 0.104, p_bonf = 0.312

SciPy check: f_oneway F = 8.02, p = 0.0006
```

Focus on:

- **Group means:** Which condition appears best (lowest stress) in raw units?
- **F and p:** Does the overall ANOVA reject H_0 that all means are equal?
- **eta-squared:** How much of the variance in stress scores is explained by condition?
- **Post-hoc tests:** Which specific pairs of conditions differ once we control for multiple comparisons?

45.9 Your Turn: Practice Scenarios

As in earlier chapters, you can experiment by editing the parameters in `psych_ch12_one_way_anova.py`:

- **Change the group means**

Make the three conditions more or less separated. How does this affect F , p , η^2 , and the pairwise tests?

- **Change the group sizes**

Use unequal group sizes (for example, 20, 30, 40). How does this affect the sums of squares and degrees of freedom? What happens if you also make the group variances very different—does the equal-variance assumption still seem reasonable?

- **Change the within-group variability**

Increase the group standard deviations. Watch how MS_{Within} grows, making the F -ratio smaller for the same difference in means.

- **Compare Bonferroni with your intuition**

Look at uncorrected p -values vs. Bonferroni-adjusted p -values. Do some pairwise differences lose significance after correction? Why is that a *feature*, not a bug, from the perspective of Type I error control?

45.10 Summary

In this chapter you learned:

- why running many separate t -tests can inflate the family-wise Type I error rate,
- how one-way ANOVA partitions variability into between-groups and within-groups components,
- how to compute and interpret the F -ratio and its p -value under the equal-variance assumption,
- how to quantify effect size using η^2 (and why it is slightly biased upward as an estimate of the population effect),
- why post-hoc tests are needed after a significant F and how Bonferroni correction works in simple pairwise comparisons,
- how to implement a one-way ANOVA and Bonferroni-corrected post-hoc tests using PyStatsV1, with a SciPy-based safety check.

In the bigger arc:

- Chapter 10 introduced independent-samples t -tests for two groups.
- Chapter 11 introduced paired-samples t -tests for within-subjects designs.

- Chapter 12 generalizes the **between-subjects** logic to **three or more groups** using ANOVA.

In the next chapter, you will extend these ideas further to **factorial designs**, where more than one independent variable is manipulated at the same time.

For the full Python implementation, see `scripts/psych_ch12_one_way_anova.py` in the PyStatsV1 GitHub repository.

CHAPTER 13 – FACTORIAL DESIGNS AND THE TWO-WAY ANOVA

In Chapters 10–12 you learned how to compare two or more groups on **one** independent variable (IV):

- independent-samples *t*-tests for two groups (between-subjects),
- paired-samples *t*-tests for repeated measures (within-subjects),
- one-way ANOVA for three or more groups on a single factor.

In real psychological research, however, we rarely care about just one factor at a time. We ask questions like:

- Does a training program help **more** under high stress than low stress?
- Does a therapy work **better for some age groups than others**?
- Does feedback style interact with **time pressure** to affect performance?

These questions involve **more than one independent variable**. The standard tool is the **factorial design**, analyzed with a **two-way ANOVA** (for two factors).

Our goals in this chapter are to help you:

- understand the logic of factorial designs and the 2×2 notation,
- distinguish **main effects** from **interactions**,
- interpret interactions as “**differences of differences**”,
- see examples of spreading vs. crossover interactions,
- appreciate the idea of **simple main effects** as follow-up tests, and
- run and interpret a two-way ANOVA using PyStatsV1 on a balanced design.

46.1 Design Logic: Two Factors at Once

We will use a simple 2×2 example throughout.

Suppose a lab studies a stress-management training program. Participants are randomly assigned to:

- **Training** factor (Factor A) - control – no training - cbt – brief cognitive–behavioral training

and then complete a challenging task either under:

- **Context** factor (Factor B) - low_stress – quiet room, no time pressure - high_stress – noisy room, strict time pressure

The dependent variable is a continuous `stress_score` (higher = more stress).

This design has **two factors** (Training and Context), each with **two levels**, so we call it a **2×2 factorial design**.

46.2 Notation for Factorial Designs

A shorthand like 2×2 tells you:

- the number of levels of each factor, and
- how many experimental cells there are.

Examples:

- 2×2
 - Factor A has 2 levels, Factor B has 2 levels.
 - Total cells: $2 \times 2 = 4$ (control/low, control/high, cbt/low, cbt/high).
- 2×3
 - Factor A has 2 levels, Factor B has 3 levels.
 - Total cells: $2 \times 3 = 6$.
- $2 \times 2 \times 2$
 - Three factors, each with 2 levels.
 - Total cells: $2 \times 2 \times 2 = 8$.

In this chapter we focus on the **two-way** case (two factors). The ideas generalize to more complex designs, but two-way ANOVA is the workhorse for most undergraduate research projects.

46.3 Main Effects

A **main effect** is the overall effect of **one factor**, averaging over the levels of the other factor.

- The **main effect of Training** asks:
 - > On average across both Context conditions, do participants in cbt > differ from those in control?
- The **main effect of Context** asks:
 - > On average across both Training conditions, do participants in > high_stress differ from those in low_stress?

We compute **marginal means** to answer these questions. For example:

- Mean stress in control: average of (control, low_stress) and (control, high_stress).
- Mean stress in cbt: average of (cbt, low_stress) and (cbt, high_stress).

If those marginal means differ, there is evidence of a main effect for that factor.

46.4 Interactions: The “It Depends” Effect

The most important idea in factorial designs is the **interaction**.

An **interaction** occurs when **the effect of one factor depends on the level of the other factor**.

In our example, we ask:

Does the benefit of CBT training depend on whether the context is low-stress or high-stress?

Mathematically, we can think of an interaction as a **difference of differences**.

Let:

- $\bar{X}_{\text{control, low}}$ be the mean stress for control/low,
- $\bar{X}_{\text{control, high}}$ for control/high,
- $\bar{X}_{\text{cbt, low}}$ for cbt/low,
- $\bar{X}_{\text{cbt, high}}$ for cbt/high.

Compute the Training effect within each Context:

$$\text{Training effect at low stress} = \bar{X}_{\text{control, low}} - \bar{X}_{\text{cbt, low}},$$

$$\text{Training effect at high stress} = \bar{X}_{\text{control, high}} - \bar{X}_{\text{cbt, high}}.$$

The **interaction** for Training \times Context is the difference between these two effects:

$$\text{Interaction (difference of differences)} = (\bar{X}_{\text{control, high}} - \bar{X}_{\text{cbt, high}}) - (\bar{X}_{\text{control, low}} - \bar{X}_{\text{cbt, low}}).$$

If that difference of differences is zero (within random error), we say there is **no interaction**. If it is clearly non-zero, we have an interaction: the effect of training changes across contexts.

46.5 Graphical View: Non-Parallel Lines

Interactions are often easiest to see in a **line graph**:

- Put Context on the x-axis (low vs high).
- Plot mean stress for each Training condition as a separate line.

Then:

- If the lines are **parallel**, the Training effect is similar at low and high stress \rightarrow no interaction (or only a trivial one).
- If the lines **spread apart**, **converge**, or **cross**, the Training effect changes with Context \rightarrow interaction.

Two common patterns:

- **Spreading interaction**

Training helps under high stress but has little effect under low stress. The lines diverge as you move from low to high stress.

- **Crossover interaction**

Control performs better in low stress, but CBT performs better in high stress (or vice versa). The lines literally cross.

46.6 Simple Main Effects

When an interaction is present, main effects can be hard to interpret on their own.

For example, suppose CBT reduces stress **only** in the high-stress context. The overall (marginal) means for Training might still show a “modest” effect, even though the real story is:

- CBT < control under low stress,
- CBT < control under high stress.

To unpack this, researchers examine **simple main effects**:

- the effect of one factor **at a single level of the other factor**.

Examples:

- Simple effect of Training **within low_stress**: compare control vs cbt using only low-stress participants.
- Simple effect of Training **within high_stress**: compare control vs cbt using only high-stress participants.
- Simple effect of Context **within control**: compare low vs high stress among control participants only.

In practice, simple main effects are often tested with t -tests or one-way ANOVAs conducted **within** a subset of the data, sometimes combined with Bonferroni or other corrections for multiple tests.

In the PyStatsV1 lab for this chapter, the two-way ANOVA is the primary analysis. For pedagogical purposes, the script also shows how to compute a few simple main effects (for example, Training within each Context) using independent-samples t -tests when the interaction is statistically significant.

46.7 The Two-Way ANOVA: Partitioning Variance

Factorial ANOVA extends the one-way ANOVA logic from Chapter 12. We still partition the total variability in the outcome into meaningful components.

Let:

- Factor A = Training (2 levels),
- Factor B = Context (2 levels),
- Y_{ijk} be the score for person k in cell (i, j) , where i indexes levels of A and j indexes levels of B,
- n_{ij} be the number of participants in cell (i, j) ,
- \bar{Y}_{ij} be the **cell mean** for cell (i, j) ,
- $\bar{Y}_{i\cdot}$ be the marginal mean for level i of A,
- $\bar{Y}_{\cdot j}$ be the marginal mean for level j of B,
- $\bar{Y}_{\cdot\cdot}$ be the **grand mean** across all participants.

Then the total sum of squares can be written as:

$$SS_{\text{Total}} = \sum_i \sum_j \sum_k (Y_{ijk} - \bar{Y}_{\cdot\cdot})^2.$$

For a **balanced** 2×2 design (equal n_{ij} in each cell), we can decompose this into:

$$SS_{\text{Total}} = SS_A + SS_B + SS_{AB} + SS_{\text{Within}},$$

where:

- SS_A captures the main effect of Training,
- SS_B captures the main effect of Context,
- SS_{AB} captures the interaction,
- SS_{Within} is the within-cell (error) variability.

Note: For these components to be additive ($A + B + AB$), the design must be balanced.

Each component has associated degrees of freedom (df) and a mean square (MS) obtained by dividing SS by its df. The F -tests for each effect are:

$$F_A = \frac{MS_A}{MS_{\text{Within}}}, \quad F_B = \frac{MS_B}{MS_{\text{Within}}}, \quad F_{AB} = \frac{MS_{AB}}{MS_{\text{Within}}}.$$

Effect sizes (for example, η^2 for each effect) can be computed as the proportion of total variance associated with each SS component. As with one-way ANOVA, these sample-based measures tend to slightly overestimate the population effect sizes, but they are helpful descriptive summaries.

46.8 Assumptions in the Two-Way ANOVA

The classical two-way ANOVA relies on similar assumptions to the one-way case:

- **Independence**

Observations are independent within and across cells (for example, each participant appears in only one cell).

- **Normality**

The outcome scores within each cell are approximately Normally distributed.

- **Equal variances**

The population variances are roughly equal across cells (homogeneity of variance).

- **Balanced design (for our manual calculations)**

In this chapter, we assume **equal sample sizes in each cell** (for example, 25 participants per Training \times Context combination). This greatly simplifies the sums of squares and matches the PyStatsV1 implementation.

Warning

Balanced vs. Unbalanced Designs

The manual calculations and PyStatsV1 helpers in this chapter assume a **balanced design** with equal sample sizes in every cell.

If sample sizes differ (unbalanced), the factors become correlated and sums of squares must be computed using more advanced methods (for example, Type III sums of squares). Professional software (such as SPSS, SAS, or R packages) handles this automatically, but our hand-calculation formulas and simple code do **not**.

For this reason, when you experiment with the Chapter 13 script, keep the cell sizes equal. If you need to analyze an unbalanced design in real research, use dedicated statistical software and pay close attention to how it defines and reports sums of squares.

46.9 PyStatsV1 Lab: Two-Way ANOVA on Stress Scores

In this lab, you will analyze a simulated 2×2 factorial experiment with:

- Factor A: training (control vs cbt),
- Factor B: context (low_stress vs high_stress),
- Dependent variable: stress_score.

You will:

- simulate a balanced dataset with the same n in each cell,
- compute:
 - cell means and sample sizes,
 - marginal means for each level of Training and Context,
 - sums of squares SS_A , SS_B , SS_{AB} , SS_{Within} , SS_{Total} ,

- corresponding degrees of freedom and mean squares,
- F -statistics and p -values for each main effect and the interaction,
- eta-squared style effect sizes for each effect,
- visualize the interaction with a simple line plot of cell means,
- optionally compute a small set of **simple main effects** (for example, Training within each Context) when the interaction is statistically significant.

All code for this lab lives in:

- `scripts/psych_ch13_two_way_anova.py`

and the script can optionally write outputs to:

- `data/synthetic/psych_ch13_two_way_stress.csv`

46.10 Running the Lab Script

From the project root, you can run:

```
python -m scripts.psych_ch13_two_way_anova
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch13
```

This will:

- simulate a balanced 2×2 Training \times Context dataset,
- print the cell means and sample sizes,
- compute the two-way ANOVA table with F -tests for:
 - main effect of Training,
 - main effect of Context,
 - Training \times Context interaction,
- report eta-squared style effect sizes for each effect,
- draw (or save) a simple interaction plot of mean stress by Context, with separate lines for each Training condition,
- optionally compute and print simple main effects (for example, Training within low_stress and within high_stress) when the interaction is statistically significant.

46.11 Expected Console Output

Your exact numbers will vary if you change the seed or parameters, but with the default settings you might see output like:

```
Two-way ANOVA on stress scores (Training  $\times$  Context)
-----
Cell means (n per cell = 25):
  control, low_stress    mean = 17.9
  control, high_stress   mean = 23.4
  cbt,    low_stress     mean = 16.8
```

(continues on next page)

(continued from previous page)

```

cbt,      high_stress  mean = 18.9

ANOVA table:
SS_A (Training)      = 95.21, df_A = 1, MS_A = 95.21, F_A = 4.10, p_A = 0.046
SS_B (Context)       = 640.37, df_B = 1, MS_B = 640.37, F_B = 27.56, p_B < 0.001
SS_AB (Interaction)  = 118.94, df_AB = 1, MS_AB = 118.94, F_AB = 5.11, p_AB = 0.026
SS_within            = 1087.42, df_within = 96, MS_within = 11.33
SS_total              = 1941.94, df_total = 99

Effect sizes (eta-squared style):
eta^2_Training      = 0.049
eta^2_Context       = 0.330
eta^2_Interaction    = 0.061

Simple main effects (because interaction is significant):
Training within low_stress: t(48) = 0.82, p = 0.416
Training within high_stress: t(48) = 2.78, p = 0.008

Interaction plot saved to: outputs/track_b/ch13_training_by_context.png

```

Focus on:

- **Cell means and lines** in the interaction plot: are the Training lines parallel, spreading, or crossing?
- **Main effects:** Are there overall differences between Training conditions, or between Contexts, when averaging across the other factor?
- **Interaction:** Does the Training effect depend on Context? Are the differences between control and cbt larger in one context than the other?
- **Simple main effects:** If the interaction is significant, do follow-up tests show that Training matters only under high stress, or in both contexts?

46.12 Your Turn: Practice Scenarios

As in earlier chapters, you can experiment by editing parameters in `psych_ch13_two_way_anova.py`. Some ideas:

- **Create a pure main-effect scenario**
Make CBT slightly better than control in **both** contexts by the same amount. What happens to the Training main effect and the interaction?
- **Create a spreading interaction**
Make Training have little effect under low_stress but a strong effect under high_stress. How does this change the interaction plot and the F for the interaction?
- **Create a crossover interaction**
Make control slightly better under low_stress but cbt clearly better under high_stress. Can the overall Training main effect be small or even misleading, while the interaction is large?
- **Change the within-cell variability**
Increase the standard deviation of the simulated scores. Watch how MS_{Within} grows and the F -statistics shrink even if the cell means stay the same.
- **(Do not break the balance!)**

You can change the **shared n_per_cell** parameter (for example, 20 instead of 25), but resist the temptation to give different cells different sample sizes. Our manual formulas and PyStatsV1 helpers assume equal sample sizes in each cell. For unbalanced designs, you will need more advanced tools (for example, Type III sums of squares in specialized software).

46.13 Summary

In this chapter you learned:

- why psychologists often use **factorial designs** with more than one independent variable,
- how to interpret **main effects** as overall differences for each factor,
- how to interpret **interactions** as “it depends” or **difference of differences** effects, often revealed by non-parallel lines in an interaction plot,
- why simple main effects are useful follow-ups when interactions are present,
- how the two-way ANOVA partitions variance into main effects, interaction, and error for a balanced design,
- how to implement a two-way ANOVA and basic simple main-effects analyses using PyStatsV1.

In the bigger arc:

- Chapter 10 introduced independent-samples *t*-tests for two groups.
- Chapter 11 introduced paired-samples *t*-tests for within-subjects designs.
- Chapter 12 extended the between-subjects logic to **three or more groups** using one-way ANOVA.
- Chapter 13 generalizes the ANOVA framework to **factorial designs**, where more than one independent variable is manipulated at the same time.

Factorial designs are powerful tools. They let you ask richer questions about how psychological processes behave across different contexts, and they prepare you for even more complex models (mixed designs, ANCOVA, and beyond) in later chapters.

For the full Python implementation, see `scripts/psych_ch13_two_way_anova.py` in the PyStatsV1 GitHub repository.

CHAPTER 14 – REPEATED-MEASURES ANOVA

In Chapters 10–13 you learned how to compare groups when each participant contributes **one** score per condition:

- independent-samples *t*-tests for two groups (between-subjects),
- paired-samples *t*-tests for two time points or matched pairs,
- one-way ANOVA for three or more independent groups,
- two-way ANOVA for factorial designs with two between-subjects factors.

In many psychology studies, however, we repeatedly measure **the same** participants over time or across conditions. Examples include:

- stress measured **before**, **after**, and **one month after** a training program,
- memory performance under **low**, **medium**, and **high** distraction,
- mood ratings at **multiple time points** during a therapy course.

These are **repeated-measures** (within-subjects) designs with **three or more levels** of a within-subject factor (often called **time** or **condition**).

In this chapter you will:

- understand the logic of repeated-measures designs,
- see how the ANOVA is adapted to handle **correlated** observations,
- learn how the total variability is partitioned into **subjects**, **time**, and **residual** components,
- understand the **sphericity** assumption and why it matters,
- appreciate why repeated-measures designs often have **more power** than between-subjects designs, and
- run and interpret a simple repeated-measures ANOVA using PyStatsV1 on a balanced design with three time points.

47.1 Design Logic: Following the Same People Over Time

Suppose a clinical psychologist evaluates a stress-management program. Each participant completes a stress questionnaire at three time points:

- **pre** – before training,
- **post** – immediately after training,
- **followup** – one month later.

The outcome variable is a continuous `stress_score` (higher = more stress). Every participant contributes **three scores**. The research question is:

Does mean stress change over time?

A naïve approach would be to run multiple paired-samples *t*-tests:

- pre vs post,
- post vs followup,
- pre vs followup.

As in Chapter 12, this would inflate the **family-wise** Type I error rate. Instead, we use a **repeated-measures ANOVA** with a single within-subject factor `time` (three levels: pre, post, followup).

Key design features:

- Each participant is measured at **all** levels of the factor.
- Scores within a participant are **correlated** (same person, same traits).
- We can separate variability due to **stable individual differences** from variability due to **time** and random noise.

47.2 Why Not Just Treat It as a One-Way ANOVA?

If we ignored the repeated-measures nature of the design and treated the three time points as independent groups, we would:

- violate the independence assumption,
- **underestimate** the standard errors (because we pretend repeated scores are independent when they are not),
- inflate the Type I error rate.

Repeated-measures ANOVA explicitly accounts for the fact that observations within a person are correlated. The basic idea is to give each participant their **own baseline** and then focus on **how they change over time**.

47.3 Partitioning Variance in the Repeated-Measures ANOVA

In Chapter 12 (one-way ANOVA) we partitioned the total variability into:

- between-groups variability (signal), and
- within-groups variability (noise).

In the repeated-measures design with one within-subject factor (Time), the picture is slightly more complex. Let

- Y_{it} be the stress score for participant i at time t (pre, post, followup),
- $\bar{Y}_{i\cdot}$ be the mean for participant i across time,
- $\bar{Y}_{\cdot t}$ be the mean at time t across participants,
- $\bar{Y}_{\cdot\cdot}$ be the grand mean (all participants, all times).

We start with the total sum of squares:

$$SS_{\text{Total}} = \sum_i \sum_t (Y_{it} - \bar{Y}_{\cdot\cdot})^2.$$

This can be decomposed into two major parts:

$$SS_{\text{Total}} = SS_{\text{Subjects}} + SS_{\text{Within}},$$

where:

- SS_{Subjects} measures stable differences between participants (some people are generally more stressed than others),
- SS_{Within} captures variability **within** participants across time points.

The within-participants portion is then further decomposed into:

$$SS_{\text{Within}} = SS_{\text{Time}} + SS_{\text{Residual}},$$

where:

- SS_{Time} reflects systematic changes in the mean stress score over time (our effect of interest),
- SS_{Residual} captures the leftover, unsystematic variability (error) after accounting for subject and time effects.

Note the difference from Chapter 12. In a between-subjects one-way ANOVA, the *error* term includes **all** individual differences. In the repeated-measures ANOVA, we explicitly separate out those individual differences into SS_{Subjects} and remove them from the error term (SS_{Residual}). This reduction in error variance is the main reason repeated-measures designs are often **more powerful**.

47.4 Degrees of Freedom and F-Test for Time

If there are N participants and k time points (for example, $k = 3$ for pre, post, followup), then:

- Total degrees of freedom:

$$df_{\text{Total}} = Nk - 1.$$

- Subjects degrees of freedom:

$$df_{\text{Subjects}} = N - 1.$$

- Within-subjects degrees of freedom:

$$df_{\text{Within}} = (N - 1)(k - 1).$$

- Time degrees of freedom:

$$df_{\text{Time}} = k - 1.$$

- Residual (error) degrees of freedom:

$$df_{\text{Residual}} = (N - 1)(k - 1).$$

We convert sums of squares to mean squares in the usual way:

$$MS_{\text{Time}} = \frac{SS_{\text{Time}}}{df_{\text{Time}}}, \quad MS_{\text{Residual}} = \frac{SS_{\text{Residual}}}{df_{\text{Residual}}}.$$

The F-statistic for the Time effect is then:

$$F_{\text{Time}} = \frac{MS_{\text{Time}}}{MS_{\text{Residual}}}.$$

Under the null hypothesis that the population means at all time points are equal,

$$H_0 : \mu_{\text{pre}} = \mu_{\text{post}} = \mu_{\text{followup}},$$

the F-statistic follows an F distribution with

- $df_1 = df_{\text{Time}} = k - 1$ in the numerator, and
- $df_2 = df_{\text{Residual}} = (N - 1)(k - 1)$ in the denominator.

47.5 Effect Sizes

A common effect size for repeated-measures ANOVA is **eta-squared** for the Time effect:

$$\eta_{\text{Time}}^2 = \frac{SS_{\text{Time}}}{SS_{\text{Total}}}.$$

This represents the **proportion of total variance** in the outcome that can be attributed to the Time factor.

As before, η^2 tends to slightly overestimate the population effect size, especially for small samples. In professional work, researchers often report **partial eta-squared** or alternative measures, but for introductory purposes η_{Time}^2 is a useful summary.

47.6 The Sphericity Assumption

Repeated-measures ANOVA relies on all of the usual assumptions (independence, approximate Normality), plus a **new one** called **sphericity**.

Intuitively, **sphericity** means that the variability of the **difference scores** between any pair of time points is roughly the same. Formally, for three time points (pre, post, followup), sphericity requires that:

- the variance of (pre – post),
- the variance of (pre – followup), and
- the variance of (post – followup)

are approximately equal in the population.

If sphericity is badly violated, the standard F-test for Time becomes too liberal (Type I error rate is higher than advertised). To correct for this, many statistical packages apply **epsilon corrections** that reduce the effective degrees of freedom. Two common choices are:

- **Greenhouse–Geisser** correction,
- **Huynh–Feldt** correction.

Professional Tip: Sphericity vs. Compound Symmetry

Some textbooks talk about **compound symmetry** (equal variances and equal covariances across time). Compound symmetry is a **stronger** condition than sphericity. Sphericity is the true requirement for the standard repeated-measures F-test, and it can hold even when compound symmetry does not.

In practice, many researchers rely on software which automatically checks sphericity (for example, with Mauchly's test) and applies Greenhouse–Geisser or Huynh–Feldt corrections as needed.

Our Approach in PyStatsV1

In this mini-book we focus on **balanced, well-behaved designs** where sphericity is a reasonable approximation. The PyStatsV1 Chapter 14 helpers implement the *classical* repeated-measures ANOVA for a single within-subject factor with equal n at each time point.

For real research projects, you should use dedicated libraries that implement sphericity checks and corrections automatically. In Python, two especially useful tools are:

- **pingouin** – a user-friendly statistics package for psychology, including `pingouin.rm_anova()` for repeated-measures ANOVA;

- **statsmodels** – a general-purpose modeling library that supports repeated-measures and mixed models via formulas.

These tools also encourage you to think in a **data-science style**: reshaping datasets from “wide” (one column per time point) to “long” (one row per person–time combination with columns like **subject**, **time**, and **score**), which is the standard in modern statistical computing.

47.7 Advantages and Trade-Offs of Repeated-Measures Designs

Advantages:

- **Higher statistical power**

By removing stable individual differences from the error term, we often get much more precise estimates of the Time effect.

- **Fewer participants required**

Measuring each participant multiple times can achieve the same precision with a smaller sample than a comparable between-subjects design.

- **Focus on change**

Repeated-measures designs naturally answer questions about trajectories: improvement, decline, adaptation, and so on.

Trade-offs and challenges:

- **Carryover effects**

Experience in earlier conditions can influence later responses (for example, practice, fatigue, or learning).

- **Order effects**

The order of conditions matters. Researchers use techniques like counterbalancing or randomization to mitigate this.

- **Missing data complexity**

If some participants miss a time point, analysis becomes more complex. Modern mixed-effects models (see Chapter 17) are often better suited for heavily unbalanced longitudinal data.

47.8 PyStatsV1 Lab: Repeated-Measures ANOVA on Stress Over Time

In the Chapter 14 lab, you will analyze a simulated repeated-measures study of stress scores at three time points:

- **pre** – before a training program,
- **post** – immediately after training,
- **followup** – one month later.

The design assumes:

- N participants (for example, 40),
- all participants measured at all three time points (balanced design),
- modest decreases in stress from pre → post and post → followup.

You will:

- simulate a balanced repeated-measures dataset with columns like `subject_id`, `time` (pre/post/followup), and `stress_score`,
- compute:
 - means and standard deviations for each time point,
 - sums of squares SS_{Total} , SS_{Subjects} , SS_{Within} , SS_{Time} , SS_{Residual} ,
 - the corresponding degrees of freedom and mean squares,
 - the F-statistic and p -value for the Time effect,
 - η^2_{Time} as an effect size,
- produce a simple **line plot** of mean stress over time with error bars,
- optionally run pairwise comparisons (for example, pre vs post, post vs followup) with Bonferroni-adjusted p -values,
- optionally cross-check the ANOVA results against `pingouin.rm_anova()` if the `pingouin` package is installed.

All code for this lab lives in:

- `scripts/psych_ch14_repeated_measures_anova.py`

and the script can optionally write outputs to:

- `data/synthetic/psych_ch14_repeated_stress.csv`

47.9 Running the Lab Script

From the project root, you can run:

```
python -m scripts.psych_ch14_repeated_measures_anova
```

If your Makefile defines a convenience target, you can instead run:

```
make psych-ch14
```

This will:

- simulate a balanced repeated-measures dataset,
- print descriptive statistics for each time point,
- compute the repeated-measures ANOVA table (Time effect),
- report η^2_{Time} ,
- draw (or save) a line plot of mean stress over time with error bars,
- optionally print pairwise comparisons and, if available, a `pingouin`-based check of the ANOVA results.

47.10 Expected Console Output

Your exact numbers will vary if you change the seed or parameters, but with the default settings you might see output like:

Repeated-measures ANOVA on stress scores (Time: pre, post, followup)

Descriptive stats by time:

```
pre:      mean = 22.4, sd = 4.9,  n = 40
post:     mean = 18.5, sd = 4.6,  n = 40
followup: mean = 17.1, sd = 4.8,  n = 40
```

ANOVA table (within-subjects factor: Time)

```
SS_Total      = 4752.39, df_Total      = 119
SS_Subjects   = 3180.12, df_Subjects   = 39
SS_Within     = 1572.27, df_Within     = 80
SS_Time       = 892.54, df_Time        = 2
SS_Residual   = 679.73, df_Residual    = 78

MS_Time       = 446.27
MS_Residual   = 8.71
F_Time(2, 78) = 51.23, p < 0.001
eta^2_Time    = 0.19
```

Pairwise comparisons (Bonferroni-adjusted p-values):

```
pre vs post:    t(39) = 7.10, p_bonf < 0.001
post vs followup: t(39) = 2.60, p_bonf = 0.036
pre vs followup: t(39) = 8.40, p_bonf < 0.001
```

(Optional) Pingouin check:

```
rm_anova F = 51.25, p < 0.001, np2 = 0.19
```

Plot saved to: outputs/track_b/ch14_stress_over_time.png

Focus on:

- **Mean trends:** Do the line plot and descriptives show clear improvement over time?
- **ANOVA result:** Does the F-test for Time indicate a statistically significant change in mean stress?
- **Effect size:** Is η^2_{Time} small, medium, or large in your field's context?
- **Pairwise comparisons:** Which specific time intervals show reliable change after correcting for multiple tests?

47.11 Your Turn: Practice Scenarios

As in earlier chapters, you can experiment by editing parameters in `psych_ch14_repeated_measures_anova.py`. Some ideas:

- **Change the mean trajectory**

Make stress drop sharply from pre to post and then stay flat, or include a slight rebound at followup. How does this affect the F-test and pairwise comparisons?

- **Change the within-person variability**

Increase or decrease the standard deviation of the noise term. Watch how MS_{Residual} changes and how that affects the F-statistic.

- **Compare to multiple paired *t*-tests**

Manually run paired *t*-tests for pre vs post, post vs followup, and pre vs followup. How do their *p*-values compare to the global ANOVA test when you correct for multiple comparisons?

- **Experiment with the sample size**

Try $N = 20$ vs $N = 80$. How does the power (sensitivity) of the test change? Can you see smaller effects with larger samples?

47.12 Summary

In this chapter you learned:

- how repeated-measures designs follow the **same participants** across three or more time points or conditions,
- how repeated-measures ANOVA partitions total variability into **subjects**, **time**, and **residual** components,
- why removing stable individual differences from the error term increases **statistical power**,
- what the **sphericity** assumption is and why epsilon corrections (Greenhouse–Geisser, Huynh–Feldt) are used in professional software,
- how to implement a simple repeated-measures ANOVA in PyStatsV1 for a balanced design, and how to visualize trajectories over time,
- how modern Python tools like `pingouin` and `statsmodels` support more advanced repeated-measures and mixed-model analyses in a data-science-friendly workflow (long-format data, formula syntax).

In the bigger arc:

- Chapter 11 introduced paired-samples t -tests for two time points.
- Chapter 12 extended the logic to multiple independent groups via one-way ANOVA.
- Chapter 13 introduced factorial designs and two-way ANOVA for multiple between-subjects factors.
- Chapter 14 takes the next step to **within-subjects ANOVA** for three or more time points on the same participants.

In Chapter 17, you will see how these ideas generalize further to **mixed-model designs**, where within-subjects (repeated) factors and between-subjects factors are analyzed together in a unified framework.

For the full Python implementation, see `scripts/psych_ch14_repeated_measures_anova.py` in the PyStatsV1 GitHub repository.

CHAPTER 14 APPENDIX – PINGOUIN FOR REPEATED-MEASURES AND MIXED ANOVA

In Chapter 14 you learned how to:

- design a simple repeated-measures experiment (for example, Pre, Post, Followup),
- compute the sums of squares by hand for Time, Subjects, and Residual,
- and run a repeated-measures ANOVA using our own PyStatsV1 helper functions.

In that main chapter, the script optionally **cross-checks** the results with `pingouin` if it is installed. This appendix explains:

- what Pingouin is and why it is useful for psychology students,
- how to install it safely,
- how it connects to Chapter 14 (repeated measures) and later chapters (mixed ANOVA, ANCOVA, regression, and more),
- and a small example using the Chapter 14 synthetic dataset.

48.1 What is Pingouin?

`Pingouin` is an open-source statistical package written in Python 3 and built on top of NumPy and pandas. It is designed for users who want **simple but comprehensive** statistical functions without having to write low-level code.

Some of its capabilities include:

- **ANOVAs:** multi-way between-subjects ANOVA, repeated-measures ANOVA, mixed (split-plot) ANOVA, and ANCOVA.
- **Pairwise tests and correlations:** parametric and non-parametric post-hoc tests, as well as Pearson, Spearman, robust, partial, distance, and repeated-measures correlations.
- **Regression and mediation:** linear and logistic regression, as well as mediation analysis with bootstrap confidence intervals.
- **Effect sizes and power:** Cohen's d , partial eta-squared, confidence intervals around effects, and power analysis helpers.
- **Other tools:** reliability and consistency measures, circular statistics, chi-square tests, and convenience plotting functions (for example, Q–Q plots and Bland–Altman plots).

A key advantage for teaching is that most Pingouin functions return a **tidy pandas DataFrame** with all the usual quantities in one place: test statistic, degrees of freedom, p -value, effect size, confidence intervals, and often a Bayes factor and power estimate.

For example, while `scipy.stats.ttest_ind()` returns only a t value and p -value, `pingouin.ttest()` returns a table with t , df , p , Cohen's d , 95% confidence intervals, power, and Bayes factor.

48.2 Installation and Dependencies

Pingouin is a Python 3 package currently tested on Python 3.8–3.11.

Its core dependencies include:

- NumPy
- SciPy
- pandas
- pandas-flavor
- statsmodels
- matplotlib
- seaborn

Some additional features require:

- scikit-learn
- mpmath

The simplest way to install Pingouin in your PyStatsV1 environment is with `pip`:

```
pip install pingouin
```

If you prefer conda and the conda-forge channel:

```
conda install -c conda-forge pingouin
```

Pingouin is under active development, so it is a good idea to keep it updated:

```
pip install --upgrade pingouin
```

If you run into issues, check that your Python version and environment are compatible, and consult the Pingouin documentation and GitHub issues.

48.3 Why We Still Teach Sums of Squares

In the main body of Chapter 14 we derived the repeated-measures ANOVA from first principles:

$$SS_{\text{Total}} = SS_{\text{Subjects}} + SS_{\text{Within}}$$

and

$$SS_{\text{Within}} = SS_{\text{Time}} + SS_{\text{Residual}}.$$

This decomposition shows *why* repeated-measures designs are powerful: they remove stable individual differences (Subjects) from the error term (Residual), which often leads to larger F statistics for Time.

By contrast, a typical between-subjects ANOVA lumps all individual differences into a single error term. Here you see the contrast explicitly:

- **Between-subjects ANOVA (Chapter 12):** error includes all person-to-person variation.

- **Repeated-measures ANOVA (Chapter 14):** error is shrunk by separating out Subject variability.

Pingouin does not replace this conceptual understanding—it **implements** it in a robust, well-tested way and extends it to more complex designs.

48.4 How Chapter 14 Uses Pingouin

The Chapter 14 lab script (`psych_ch14_repeated_measures_anova.py`) implements the repeated-measures ANOVA twice:

1. Manual / PyStatsV1 implementation

- compute cell means, sums of squares, degrees of freedom, and F by hand,
- print a table summarizing SS_{Time} , SS_{Subjects} , SS_{Residual} , and SS_{Total} ,
- compute a simple eta-squared style effect size for Time.

2. Optional Pingouin cross-check

- if pingouin is available, the script reshapes the data to long format and calls `pingouin.rm_anova()` with:
 - `dv="stress"` (the dependent variable),
 - `within="time"` (the repeated factor),
 - `subject="id"` (the participant identifier),
 - `detailed=True` (to get effect size and sphericity information).
- the Pingouin output includes:
 - sum of squares, degrees of freedom, F ,
 - uncorrected p -values,
 - a generalized eta-squared style effect size (η^2),
 - and an estimate of the sphericity epsilon (ϵ) for the Time factor.

This cross-check confirms that the **hand-calculated ANOVA matches what a modern stats library reports**, at least for the balanced, well-behaved designs used in our teaching examples.

48.5 Example: Using Pingouin with the Chapter 14 Dataset

After running the Chapter 14 lab script, you will have a synthetic dataset saved in:

- `data/synthetic/psych_ch14_repeated_measures_stress.csv`

with one row per participant and separate columns for `pre`, `post`, and `followup` stress scores.

The following example shows how you could analyze this dataset directly in a Python session or Jupyter notebook using Pingouin:

```
import pandas as pd
import pingouin as pg

# Load the wide-format data simulated by the Chapter 14 lab
df_wide = pd.read_csv(
    "data/synthetic/psych_ch14_repeated_measures_stress.csv"
)
```

(continues on next page)

(continued from previous page)

```

# Wide -> long format: one row per person per time point
df_long = df_wide.melt(
    id_vars="id",
    value_vars=["pre", "post", "followup"],
    var_name="time",
    value_name="stress",
)

# Run repeated-measures ANOVA with Pingouin
aov = pg.rm_anova(
    data=df_long,
    dv="stress",
    within="time",
    subject="id",
    detailed=True,
)

print(aov)

# Optionally, pretty-print the table
pg.print_table(aov, floatfmt=".3f")

```

You should see a table with:

- a row for the Time effect (Source = "time"),
- a row for the error term,
- sum of squares (SS), degrees of freedom (DF), mean squares (MS), F , p -value (p-unc), generalized eta-squared (η^2), and the sphericity epsilon (ϵ).

If your manual ANOVA and the Pingouin ANOVA disagree substantially, check:

- that you used the same data (no filtering or different random seeds),
- that the design is still balanced,
- and that the way you computed sums of squares matches the design that Pingouin assumes.

48.6 Beyond Chapter 14: Mixed ANOVA and ANCOVA

The same ideas extend to more complex designs in later chapters.

- **Mixed (split-plot) ANOVA – Chapter 17**

If you add a between-subjects factor (for example, treatment group) in addition to the repeated Time factor, you can use:

```

aov_mixed = pg.mixed_anova(
    data=df_long,
    dv="stress",
    within="time",      # repeated factor
    between="group",    # between-subjects factor
    subject="id",
    effsize="np2",

```

(continues on next page)

(continued from previous page)

```
correction=False, # or True to request a correction
)
```

This produces a table with rows for the main effects of Group and Time and the Group \times Time interaction, using the appropriate error terms.

- **ANCOVA – Chapter 18**

Pingouin also links naturally with `statsmodels` for regression-style analyses and ANCOVA, where you add a covariate (such as a pre-test score) to control for pre-existing differences between groups.

In both cases, understanding the **logic** of sums of squares from Chapters 12–14 makes it much easier to interpret what these more advanced functions are doing under the hood.

48.7 Summary

In this appendix you:

- saw how Pingouin fits into the PyStatsV1 ecosystem as a higher-level, psychology-friendly stats package,
- learned how to install and update Pingouin in a modern Python environment,
- connected the Chapter 14 manual repeated-measures ANOVA to Pingouin's `rm_anova()`,
- and previewed how Pingouin can help with mixed ANOVA and more advanced analyses in later chapters.

The big picture:

- PyStatsV1 scripts and hand calculations teach you **why** repeated-measures designs and ANOVAs work the way they do.
- Pingouin shows you how professional data scientists and quantitative psychologists **actually run** these analyses in code.

Both perspectives are valuable. Together, they prepare you to read modern research articles, run your own studies, and move smoothly between psychology-focused tools and the broader Python data-science ecosystem.

CHAPTER 15 – CORRELATION

In the previous chapters you learned how to *compare groups* using t-tests and ANOVA. Those designs are built around **experimental** questions:

Does changing X cause a difference in Y ?

In this chapter we turn to **association** questions:

Do X and Y move together? If so, how strongly and in what direction?

Correlation is the workhorse of non-experimental psychology. It is used to study relationships between naturally occurring variables such as stress, sleep, depression, and exam performance. You will see correlation again in the next chapter as the foundation of **linear regression**.

This chapter focuses on three big ideas:

- how to quantify the direction and strength of a linear relationship,
- how to read and interpret scatterplots,
- and why **correlation does not imply causation**.

At the end, the PyStatsV1 lab shows how to compute and visualise correlations in Python using both NumPy/pandas and the `pingouin` statistics library.

49.1 15.1 What Is a Correlation?

A **correlation** is a number that describes how two variables are related. In this chapter we focus on the most common measure: the **Pearson product–moment correlation**, usually written as r .

Pearson's r tells you two things:

- **Direction** – whether high scores on one variable tend to go with high scores on the other (a *positive* correlation), or with low scores on the other (a *negative* correlation).
- **Strength** – how tightly the points cluster around a straight line.

The value of r always lies between -1 and +1:

- $r = +1.00$ – a perfect positive linear relationship
- $r = -1.00$ – a perfect negative linear relationship
- $r = 0$ – no linear relationship

In real data, r is almost never exactly -1, 0, or +1. Instead we see values like $r = .10$ (a weak relationship) or $r = .60$ (a moderately strong relationship).

49.2 15.2 Computing Pearson's r

Conceptually, Pearson's r is the **standardized covariance** between two variables:

$$r = \frac{\text{cov}(X, Y)}{s_X s_Y}$$

Here $\text{cov}(X, Y)$ is the covariance between X and Y , and s_X and s_Y are their sample standard deviations. Covariance captures whether high values of X tend to go with high (or low) values of Y . Dividing by the standard deviations rescales the covariance to the familiar -1 to +1 range.

In practice, you will almost always compute r using software. However, it is important to understand the basic ingredients:

1. Convert X and Y to **z-scores**.
2. Multiply the paired z-scores $z_X z_Y$ for each participant.
3. Average these cross-products.

The resulting average is exactly Pearson's r . When most participants have the **same sign** of z_X and z_Y , the cross-products are positive and r is positive. When participants tend to have opposite signs (high on one variable, low on the other), r is negative.

49.3 15.3 Scatterplots and Visual Intuition

Before computing any correlation, you should **plot the data**.

A **scatterplot** places one variable on the x-axis and the other on the y-axis. Each participant is one point.

Scatterplots help you answer questions that the single number r cannot:

- Is the relationship **linear** or curved?
- Are there **outliers** that might distort the correlation?
- Does the variability change across the range of X ?

For example, a strong curved relationship can produce $r \approx 0$ even though X clearly predicts Y . Likewise, a single extreme outlier can produce a large r that does not represent the pattern for most participants.

Important

Always inspect a scatterplot before interpreting a correlation. The number r is helpful, but it is not a substitute for looking at the data.

49.4 15.4 Correlation Does Not Imply Causation

Psychology students often hear the slogan: “**Correlation does not imply causation.**” It is worth unpacking why this is true.

Suppose you find a strong positive correlation between time spent on social media and self-reported anxiety. At least three causal stories are possible:

1. **Social media causes anxiety.**
2. **Anxiety causes social media use** (perhaps anxious people are more likely to scroll in bed).
3. A **third variable** (e.g., loneliness, insomnia) causes both heavy social media use and anxiety.

A correlation alone cannot distinguish between these possibilities. To make a causal claim, you need an appropriate **research design**, such as an experiment with random assignment or a carefully controlled longitudinal study.

In this book we encourage the following mindset:

Use correlation to describe and explore relationships, use experimental design to test causal claims.

49.5 15.5 Partial Correlation: Controlling for a Third Variable

Sometimes you want to know whether two variables are related **after controlling for** another variable. For example:

- Does study time predict exam score **after controlling for** prior GPA?
- Does therapy attendance predict symptom improvement **after controlling for** baseline severity?

A **partial correlation** answers questions like these. It measures the relationship between X and Y *after removing the linear effect* of a third variable (or set of variables).

One way to think about this:

1. Regress X on the control variable(s) and keep the residuals.
2. Regress Y on the control variable(s) and keep the residuals.
3. Correlate the two sets of residuals.

The resulting partial correlation tells you whether X and Y still move together once the shared influence of the control variable has been removed.

You do not need to implement these regression steps by hand. Libraries such as `pingouin` and `statsmodels` can compute partial correlations directly from a tidy data frame.

49.6 15.6 PyStatsV1 Lab – Correlation in Python

The PyStatsV1 lab for this chapter is implemented in the script `scripts.psych_ch15_correlation`. It demonstrates three key skills:

1. Simulating data with a known population correlation

We use NumPy to simulate pairs of scores from a bivariate normal distribution with a specified population correlation (for example, $\rho = .50$). This allows us to check whether our estimation procedures recover the true value.

2. Computing correlations with NumPy and Pingouin

The script shows how to compute Pearson's r in two ways:

- using NumPy / pandas:

```
import numpy as np

r_np = np.corrcoef(df["x"], df["y"])[0, 1]
```

- using `pingouin`, which also returns p-values, confidence intervals, Bayes factors, and power:

```
import pingouin as pg

corr_table = pg.corr(df["x"], df["y"], method="pearson")
r_pg = corr_table["r"].iloc[0]
```

In our automated tests we verify that `r_np` and `r_pg` are essentially identical, and that `pingouin` recovers the population correlation used to generate the data.

3. Correlation matrices, heatmaps, and partial correlations

The script also simulates a small set of psychology variables (for example, stress, sleep, anxiety, and exam scores) and then:

- computes a full correlation matrix,
- visualises the matrix as a color-coded heatmap,
- and calculates a partial correlation, such as the association between study time and exam performance after controlling for motivation.

These analyses use `pingouin` helper functions such as `pingouin.pairwise_corr()` and `pingouin.partial_corr()`.

The synthetic data are saved in the `data/synthetic/psych_ch15_correlation.csv` file, and the heatmap is written to `outputs/track_b/ch15_corr_heatmap.png` for easy inclusion in slides or lecture notes.

Note

In Chapters 15–19 we rely increasingly on `pingouin` and `statsmodels` for the actual statistical computations. `PyStatsV1` focuses on **simulation, data management, and workflow**, while these libraries provide well-tested implementations of advanced techniques (correlation, regression, mixed ANOVA, ANCOVA, and more). Our unit tests use simulated data with known answers to check that these tools behave as expected in the scenarios we teach.

CHAPTER 15 APPENDIX – PINGOUIN FOR CORRELATION AND PARTIAL CORRELATION

In *Chapter 15 – Correlation*, you learned how to:

- define and interpret Pearson’s correlation coefficient r ,
- visualize relationships with scatterplots and heatmaps,
- compute correlations using both NumPy and pingouin, and
- run a single partial correlation (exam score ~ study hours | stress).

In that main chapter, the focus was on the *concepts* of correlation and partial correlation. This appendix shifts the emphasis to the pingouin library itself. We will treat Pingouin as a compact “stats workbench” that can:

- compute all pairwise correlations (and effect sizes) for a whole set of variables at once,
- correct p-values for multiple comparisons,
- and estimate partial correlations while adjusting for one or more covariates.

As before, all examples are reproducible with PyStatsV1 and use synthetic datasets so that you can freely experiment without privacy concerns.

50.1 Why this appendix?

In introductory courses it is common to present correlation as a single number between two variables. In real research, we rarely look at just one pair. A typical psychology study might collect ten or more measures (stress, sleep, anxiety, mood, study hours, exam score, etc.). The interesting questions are then:

- which variables are most strongly related,
- whether those relationships survive correction for multiple testing,
- and whether an association remains after we statistically control for one or more third variables.

Doing all of this by hand, or with low-level functions, is tedious and error-prone. pingouin provides higher-level helpers that match how researchers actually work. In this appendix we highlight two of them:

- `pingouin.pairwise_corr()` – compute all pairwise correlations in one shot (with effect sizes, confidence intervals, p-values, and optional p-value correction); and
- `pingouin.partial_corr()` – compute partial correlations while adjusting for one or more covariates.

All examples below assume that Pingouin is installed and that you have completed the Chapter 15 lab at least once.

50.2 A quick reminder: installing Pingouin

If you are working on your own machine (rather than the course server), you can install or update Pingouin as follows:

```
pip install --upgrade pingouin
```

or, if you use Conda:

```
conda install -c conda-forge pingouin
```

For details, see the official documentation at <https://pingouin-stats.org>.

50.3 Pairwise correlations with `pingouin.pairwise_corr()`

In the Chapter 15 lab (`scripts.psych_ch15_correlation`) we created a synthetic dataset with several variables:

- `stress`
- `sleep_hours`
- `anxiety`
- `study_hours`
- `exam_score`

To compute *all* pairwise Pearson correlations among these variables using Pingouin, we can write:

```
import pingouin as pg

from scripts.psych_ch15_correlation import simulate_psych_correlation_dataset

df = simulate_psych_correlation_dataset(n=200, random_state=456)

pairwise = pg.pairwise_corr(
    data=df,
    columns=df.columns,
    method="pearson",
    padjust="none", # or "fdr_bh", "bonf", ...
)

print(pairwise.head())
```

The resulting `pandas.DataFrame` has one row per *unique* variable pair and includes:

- `X` and `Y` – the variable names,
- `r` – Pearson’s correlation coefficient,
- `CI95%` – a 95% confidence interval for `r`,
- `p-unc` – the uncorrected p-value,
- `BF10` – an optional Bayes Factor,
- and several other useful columns.

Because each pair appears only once (e.g., `stress-exam_score` but not also `exam_score-stress`), the number of rows is:

$$n_{\text{pairs}} = \frac{k(k-1)}{2}$$

where k is the number of variables.

50.4 Correcting for multiple comparisons

When you compute many correlations at once, some may look “significant” purely by chance. Pingouin helps you control the family-wise error rate or the false discovery rate by adjusting p-values.

For example, to apply the Benjamini–Hochberg false discovery rate (FDR) correction, use `padjust="fdr_bh"`:

```
pairwise_fdr = pg.pairwise_corr(
    data=df,
    columns=df.columns,
    method="pearson",
    padjust="fdr_bh",
)
```

The output now includes a `p-adjust` column with corrected p-values. In this course we mostly treat the corrected p-values as “advanced tools” for research projects, but it is important for students to see that the option exists and is easy to use.

50.5 Spearman correlations

Sometimes you may not want to assume a strictly linear relationship or you might worry about outliers. In those cases, Spearman’s rank correlation can be more robust. Switching methods is as simple as:

```
pairwise_spearman = pg.pairwise_corr(
    data=df,
    columns=df.columns,
    method="spearman",
    padjust="fdr_bh",
)
```

You can then compare Pearson and Spearman estimates for the same pair of variables to see whether outliers or non-linearity are having a large impact.

50.6 Partial correlations with `pingouin.partial_corr()`

In the main chapter we computed a single partial correlation between `study_hours` and `exam_score` while controlling for `stress`. Pingouin makes it easy to extend this idea to multiple covariates.

The basic usage is:

```
import pingouin as pg

df = simulate_psych_correlation_dataset(n=200, random_state=456)

partial = pg.partial_corr(
    data=df,
    x="study_hours",
    y="exam_score",
    covar=["stress"],      # one or more covariates
    method="pearson",
)
```

(continues on next page)

(continued from previous page)

```
print(partial)
```

The result again is a one-row `pandas.DataFrame` with columns:

- `r` – the partial correlation,
- `CI95%` – a confidence interval for `r`,
- `p-val` – the p-value,
- plus the sample size `n`.

Controlling for *multiple* covariates is just as easy:

```
partial_two = pg.partial_corr(  
    data=df,  
    x="study_hours",  
    y="exam_score",  
    covar=["stress", "anxiety"],  
    method="pearson",  
)
```

In a research context, partial correlations are especially useful when trying to decide whether a relationship is likely to be “direct” or whether it can be explained away by a third (or fourth) variable.

50.7 PyStatsV1 demo scripts for Chapter 15a

To keep the main Chapter 15 lab focused, the PyStatsV1 repository includes two small helper scripts that live in this appendix:

- `scripts.psych_ch15a_penguin_pairwise_demo`

Shows how to:

- generate the Chapter 15 synthetic dataset,
- compute all pairwise Pearson and Spearman correlations with `penguin.pairwise_corr()`,
- apply FDR correction to the p-values,
- and save the resulting tables to `outputs/track_b`.

- `scripts.psych_ch15a_penguin_partial_demo`

Shows how to:

- compare a zero-order correlation with one or more partial correlations,
- control for multiple covariates at once,
- and summarize the results in a compact table.

You can run these scripts from the command line (inside your PyStatsV1 virtual environment) using:

```
python -m scripts.psych_ch15a_penguin_pairwise_demo  
python -m scripts.psych_ch15a_penguin_partial_demo
```

or, if you prefer the Makefile shortcuts (once they have been added):

```
make psych-ch15a
```

50.8 Unit tests for Chapter 15a

To make sure that the demos behave as expected, we include two small test files:

- `tests.test_psych_ch15a_pingouin_pairwise_demo`
- `tests.test_psych_ch15a_pingouin_partial_demo`

The tests do not check every value. Instead, they verify structural and conceptual properties such as:

- `pingouin.pairwise_corr()` returns the expected number of pairs,
- the sign and approximate strength of the correlation between `stress` and `exam_score` match the design of the synthetic dataset,
- partial correlations shrink (but do not reverse) the positive association between `study_hours` and `exam_score` when we control for `stress`.

You can run just these tests with:

```
pytest tests/test_psych_ch15a_pingouin_pairwise_demo.py
pytest tests/test_psych_ch15a_pingouin_partial_demo.py
```

or run the full Track B test suite with:

```
pytest
```

50.9 Suggested student exercises

1. Add a new variable to the Chapter 15 synthetic dataset (for example, `social_support`) that is negatively related to `stress` and positively related to `sleep_hours` and `exam_score`. Re-run the pairwise correlation demo and interpret the changes in the correlation matrix.
2. Use `pingouin.pairwise_corr()` with `method="spearman"` and compare the results to the Pearson correlations. Are any pairs sensitive to outliers or non-linearity?
3. Choose a pair of variables where you suspect a third variable might explain part of the relationship (for example, `stress` and `exam_score` with `sleep_hours` as a covariate). Compute zero-order and partial correlations and compare the results.
4. For your own research project, design a small correlation study with at least five variables. Use PyStatsV1 and Pingouin to:
 - compute all pairwise correlations,
 - adjust for multiple comparisons,
 - and report at least one partial correlation in APA style.

This appendix is meant as a bridge between the introductory correlation chapter and more advanced courses in multivariate statistics. The goal is not to memorize every option of Pingouin, but to develop a habit of using well-tested tools to explore relationships among multiple psychological variables in a principled way.

CHAPTER 16 – LINEAR REGRESSION

In Chapter 15 you learned how to quantify relationships between two variables using correlation. Correlation answers the question:

“How strongly are two variables associated?”

Linear regression goes one step further. It answers a different question:

*“How well can we ****predict*** one variable from one or more other variables?”**

In this chapter you will:

- build the idea of a **line of best fit** for predicting an outcome,
- understand how the **least-squares** method chooses that line,
- interpret the **standard error of the estimate** as typical prediction error,
- extend to **multiple regression**, where several predictors work together,
- and use `pystatsv1` and `pingouin` to fit and interpret regression models on a synthetic psychology dataset.

51.1 16.1 Prediction: The Line of Best Fit

Imagine we want to predict a student’s exam score from their number of study hours. Each student gives us one data point: (`study_hours`, `exam_score`). If we scatter those points, a clear trend often appears: students who study more tend to score higher.

Linear regression summarizes that trend with a straight line:

$$\hat{Y} = bX + a$$

where

- \hat{Y} is the *predicted* value of the outcome,
- X is the predictor,
- b is the **slope** (how much Y changes for a one-unit change in X), and
- a is the **intercept** (the predicted value of Y when $X = 0$).

In psychology, we often use regression to predict:

- exam performance from study time,
- depressive symptoms from life stress,
- therapy outcomes from baseline severity and treatment type,
- or attention scores from sleep quality and caffeine intake.

The important idea is that regression is a **model of prediction**. We care both about *how strong* the relationship is and *how well we can forecast new data*.

51.1.1 Interpretation of the slope

Suppose our fitted line is

$$\widehat{\text{Exam Score}} = 5.0 \times \text{Study Hours} + 60.$$

The slope $b = 5.0$ means:

For every extra hour of study, exam score is predicted to increase by about 5 points, on average.

The intercept $a = 60$ means:

A student who studied 0 hours is predicted to score 60 (although we should be cautious about interpreting predictions far outside the observed range).

51.2 16.2 Least Squares: Choosing the Best Line

There are infinitely many lines we could draw through a cloud of points. Linear regression chooses the one that minimizes the **sum of squared residuals**.

A **residual** is the difference between the observed outcome and the predicted outcome from the line:

$$e_i = Y_i - \hat{Y}_i.$$

The **least-squares** solution chooses a and b to minimize

$$\sum_{i=1}^n e_i^2 = \sum_{i=1}^n (Y_i - \hat{Y}_i)^2.$$

This has several nice properties:

- it gives more weight to large errors (because of squaring),
- it has a closed-form solution (no iterative search is required),
- and it links naturally to the Pearson correlation r and the ANOVA framework you saw in Chapters 12–14.

In the Chapter 16 lab script we compute the least-squares solution using both basic NumPy functions and the higher-level `pinguin.linear_regression()` helper for cross-checking.

51.3 16.3 Standard Error of the Estimate

No regression line predicts perfectly. Different students with the same study hours will still have different exam scores. The **standard error of the estimate** summarizes typical prediction error in the original units of the outcome variable.

After we fit the line, we compute residuals e_i for each case and then:

$$S_{\text{est}} = \sqrt{\frac{\sum e_i^2}{n - 2}}.$$

This is similar to a standard deviation of the residuals. A smaller S_{est} means:

- predictions are typically closer to the observed scores,
- the regression line fits the data more tightly,

- and we have more precise forecasts for new cases.

In the lab we will:

- print S_{est} for our simple regression model,
- compare it across different models,
- and relate it back to the residual plots.

51.4 16.4 Multiple Regression and R^2

Real psychological outcomes usually depend on **many** factors at once. Multiple regression extends the linear model to several predictors:

$$\hat{Y} = b_0 + b_1X_1 + b_2X_2 + \cdots + b_kX_k.$$

For example, our synthetic dataset in Chapter 16 includes:

- `study_hours` – weekly study time,
- `sleep_hours` – average nightly sleep,
- `stress` – perceived stress score,
- and `exam_score` – exam performance.

A multiple regression model might be:

$$\widehat{\text{Exam Score}} = b_0 + b_1 \times \text{Study Hours} + b_2 \times \text{Sleep Hours} - b_3 \times \text{Stress}.$$

Key ideas:

- Each slope b_j is a **partial effect**: it tells us how much Y is expected to change when X_j increases by one unit, holding the other predictors constant.
- We can assess overall model fit with R^2 , the proportion of variance in Y explained by the predictors.
- Adjusted R^2 penalizes adding predictors that do not really help.

In the lab script we use `pingouin.linear_regression()` to fit a multiple regression model with `exam_score` as the outcome and multiple predictors. We then interpret:

- the coefficient signs (which predictors help or hurt),
- their statistical significance (p-values),
- and the overall R^2 / adjusted R^2 .

51.5 16.5 PyStatsV1 Lab: Building a Predictive Model

The Chapter 16 lab script is `scripts.psych_ch16_regression`. It demonstrates:

- simulating a psychology dataset with exam performance,
- fitting and interpreting a **simple linear regression**,
- fitting a **multiple regression** with several predictors,
- saving results for replication,
- and visualizing the line of best fit.

51.5.1 Overview of the lab script

The script is structured into a few main helper functions:

- `simulate_psych_regression_dataset()`
Creates a synthetic dataset with columns such as `stress`, `sleep_hours`, `study_hours`, and `exam_score`, using a known underlying regression model. Because we know the “true” slopes, we can check that the estimated values behave as expected.
- `fit_simple_regression()`
Fits a simple regression predicting `exam_score` from `study_hours`. Returns the slope, intercept, correlation, R^2 , and standard error of the estimate.
- `fit_multiple_regression()`
Uses `pingouin.linear_regression()` to fit a multiple regression model with several predictors. Returns the regression table along with R^2 and adjusted R^2 for quick inspection.
- `plot_regression_line()`
Generates a scatterplot of `study_hours` versus `exam_score` along with the fitted line. The figure is saved to the `outputs/track_b` folder for use in slides or assignments.

When you run the script via:

```
make psych-ch16
```

you will see printed output that includes:

- a preview of the simulated dataset,
- the simple regression slope, intercept, R^2 , and standard error of the estimate,
- the multiple regression summary from `pingouin`,
- and file paths where the data, table, and figure were saved.

51.5.2 Files written by the lab

The script saves three main artifacts:

- `data/synthetic/psych_ch16_regression.csv`
The simulated psychology dataset used for all analyses.
- `outputs/track_b/ch16_regression_summary.csv`
A CSV file containing the multiple regression summary table produced by `pingouin.linear_regression()`.
- `outputs/track_b/ch16_regression_fit.png`
A scatterplot of study hours and exam scores with the regression line superimposed.

These files make it easy to reproduce the main figures and tables for homework, lecture slides, or exam preparation.

51.5.3 PyStatsV1 and Pingouin together

As in Chapters 14 and 15, we treat `pingouin` as a **trusted reference implementation**. Our own helper functions use NumPy and pandas to compute regression quantities “from scratch”, and then we cross-check key numbers against `pingouin.linear_regression()`.

This dual approach reinforces the core philosophy of PyStatsV1:

Don't just calculate your results — engineer them.

By writing small, well-tested functions and validating them against trusted libraries, students learn both the statistical ideas and the software engineering mindset needed for reproducible science.

51.6 Checklist: What You Should Be Able to Do

By the end of Chapter 16, you should be able to:

- explain what the regression line $\hat{Y} = bX + a$ means in words,
- interpret the slope and intercept in a psychology example,
- describe how the least-squares method chooses the “best” line,
- compute and interpret the standard error of the estimate,
- explain the difference between simple and multiple regression,
- interpret R^2 and adjusted R^2 ,
- and run the Chapter 16 PyStatsV1 lab to build and evaluate a predictive model.

CHAPTER 16A APPENDIX: LINEAR REGRESSION WITH PINGOUIN

52.1 Motivation

In *Chapter 16 – Linear Regression*, you learned how to:

- Simulate a psychology dataset with several predictors
- Fit a simple linear regression by hand (using NumPy)
- Fit a multiple regression model
- Interpret the slope, intercept, R^2 , and standard error of the estimate

In this appendix, we lean more heavily on the `pingouin` library to engineer regression analyses as *re-usable, testable components*.

52.2 Why Pingouin for regression?

Pingouin is a Python 3 statistics library built on top of NumPy and pandas. For regression, `pingouin.linear_regression()` gives you, in a single DataFrame:

- Unstandardized coefficients (`coef`)
- Standard errors (`se`)
- t-statistics (`T`) and p-values (`pval`)
- Model-level R^2 and adjusted R^2
- Confidence intervals for each coefficient

This pairs naturally with the PyStatsV1 philosophy:

Don't just calculate your results — engineer them.

Instead of copying numbers from an output window into a homework sheet, we write small, well-tested scripts that can be re-run, inspected and adapted to new research projects.

52.3 Overview of the 16a lab

The 16a appendix is powered by the script:

```
python -m scripts.psych_ch16a_pingouin_regression_demo
```

It reuses the same simulated dataset generator introduced in Chapter 16:

```
from scripts.psych_ch16_regression import simulate_psych_regression_dataset
```

and then uses `pingouin` to:

1. Fit a *multiple regression* model

$$\text{exam_score} = b_0 + b_1 \times \text{study_hours} + b_2 \times \text{sleep_hours} + b_3 \times \text{stress} + b_4 \times \text{motivation} + e$$

2. Compute *standardized* regression coefficients (betas) by running the same model on z-scored variables.
3. Extract *partial effects* using `pingouin.partial_corr()`, so students can see how a predictor relates to the outcome after controlling for other variables.

The goal is not to introduce a brand-new design, but to show how the *measurement model* from Chapter 16 behaves when we add a professional regression toolbox on top.

52.4 Section 16a.1 – Recap: Why multiple regression?

In Chapter 16, we motivated multiple regression as an extension of correlation:

- Correlation: how two variables move together
- Regression: how we *predict* one variable from one (simple) or many (multiple) predictors

Multiple regression helps us answer questions like:

- “How many points of exam score do we gain for each extra hour of study, **holding sleep constant?**”
- “Is the effect of sleep on exam performance still present after controlling for stress and motivation?”

This language (“holding constant”) maps directly onto partial regression and partial correlation. Pingouin makes those quantities easy to compute.

52.5 Section 16a.2 – Pingouin’s `linear_regression`

The core workhorse in this appendix is `pingouin.linear_regression()`. Its minimal usage pattern looks like:

```
import pingouin as pg

X = df[["study_hours", "sleep_hours", "stress", "motivation"]]
y = df["exam_score"]

reg_table = pg.linear_regression(X=X, y=y)
```

The returned `reg_table` is a pandas DataFrame with one row per term (intercept and predictors). Key columns include:

- `names` – the name of the predictor (or Intercept)
- `coef` – the unstandardized regression coefficient
- `se` – standard error of the coefficient
- `T` – t-statistic (coefficient divided by its standard error)
- `pval` – p-value for a two-sided test of $H_0 : b_i = 0$
- `r2` – model R^2 (repeated on each row)
- `adj_r2` – adjusted R^2

In `scripts.psych_ch16a_penguin_regression_demo`, we wrap this logic into a helper function so that it can be imported and tested:

```
from scripts.psych_ch16_regression import simulate_psych_regression_dataset
import pingouin as pg

def build_penguin_regression_tables(df):
    X = df[["study_hours", "sleep_hours", "stress", "motivation"]]
    y = df["exam_score"]
    raw_table = pg.linear_regression(X=X, y=y)
    # (plus a standardized version, see next section)
    ...
```

This also means that future chapters (e.g., ANCOVA or mixed models) could reuse the same simulation code and regression helpers for more advanced demos.

52.6 Section 16a.3 – Standardized coefficients (betas)

Unstandardized coefficients (e.g., +4 points per extra study hour) are often the most intuitive for reporting. However, standardized coefficients (“betas”) can be helpful when predictors are on very different scales.

To obtain standardized coefficients, we simply:

1. Z-score the predictors and the outcome.
2. Run `pingouin.linear_regression()` on the standardized variables.
3. Interpret the resulting `coef` values as *change in standard deviations of the outcome per one standard deviation change in the predictor*.

In the 16a script we perform this transformation with:

```
def zscore_columns(df, columns):
    zdf = df.copy()
    for col in columns:
        col_mean = zdf[col].mean()
        col_std = zdf[col].std(ddof=0)
        zdf[col + "_z"] = (zdf[col] - col_mean) / col_std
    return zdf

zdf = zscore_columns(
    df,
    ["exam_score", "study_hours", "sleep_hours", "stress", "motivation"],
)
```

We then fit a second regression model:

```
X_z = zdf[["study_hours_z", "sleep_hours_z", "stress_z", "motivation_z"]]
y_z = zdf["exam_score_z"]

standardized_table = pg.linear_regression(X=X_z, y=y_z)
```

The resulting `standardized_table` is saved to `outputs/track_b/ch16a_regression_standardized.csv` and printed to the console so students can compare unstandardized and standardized effect sizes.

52.7 Section 16a.4 – Partial effects and partial correlation

In a multiple regression, each coefficient is a *partial effect*: it describes the association between that predictor and the outcome *after controlling for* (all else equal to) the other predictors in the model.

Pingouin also exposes these partial relationships directly via `pingouin.partial_corr()`, which computes partial correlation coefficients.

For example, to examine the relationship between exam score and study hours while controlling for stress and motivation, the 16a script uses:

```
partial = pg.partial_corr(
    data=df,
    x="study_hours",
    y="exam_score",
    covar=["stress", "motivation"],
    method="pearson",
)
```

The resulting DataFrame contains:

- `r` – the partial correlation coefficient
- `CI95%` – a confidence interval for the partial correlation
- `p-val` – a p-value testing $H_0 : \rho_{xy \cdot covar} = 0$

The key conceptual link for students is:

- The *sign* and *relative magnitude* of the partial correlation align with the regression coefficient for that predictor.
- The partial correlation can be interpreted in the same “holding other variables constant” language used to explain multiple regression.

52.8 Section 16a.5 – Running the 16a lab

The appendix demo is designed to run from the command line and to save its artifacts into the same folder structure as the other Track B labs.

From the root of the repository, with the virtual environment activated:

```
# Run the demo script (regression + partial correlations)
make psych-ch16a

# Or directly via Python
python -m scripts.psych_ch16a_pingouin_regression_demo

# Run the tests for this chapter's appendix
make test-psych-ch16a

# Inspect all outputs under:
# - data/synthetic/psych_ch16_regression.csv
# - outputs/track_b/ch16a_regression_raw.csv
# - outputs/track_b/ch16a_regression_standardized.csv
# - outputs/track_b/ch16a_partial_corr_exam_study.csv
```

The associated test module, `tests.test_psych_ch16a_pingouin_regression_demo`, checks that:

- The regression tables contain the expected columns.

- Study hours and sleep hours have positive regression coefficients.
- Stress has a negative regression coefficient.
- The partial correlation between exam score and study hours (controlling for stress and motivation) is positive and statistically significant.

These tests turn the 16a appendix into *executable documentation* for both students and instructors.

52.9 Section 16a.6 – For instructors

Some suggestions for using this appendix in teaching:

- **Compare models in class.** Run the Chapter 16 lab and the 16a Pingouin appendix side-by-side. Ask students to reconcile the manual calculations with the Pingouin output.
- **Highlight effect sizes.** Use the standardized regression table to discuss which predictors have the strongest relative influence on exam performance.
- **Discuss collinearity.** Because predictors like study hours, sleep, stress, and motivation are correlated with each other, multiple regression is a natural context to introduce collinearity and its consequences.
- **Encourage replication.** Invite students to fork the PyStatsV1 repo, modify the simulation parameters (e.g., make sleep more important), and observe how the Pingouin tables change.

In later chapters (e.g., ANCOVA, mixed-model designs), we can revisit this dataset and the Pingouin regression helpers as a familiar sandbox for more advanced modeling.

CHAPTER 16B – REGRESSION DIAGNOSTICS WITH PINGOUIN

Track B: Psychological Science & Statistics – Appendix to Chapter 16

53.1 Overview

In Chapter 16, we introduced linear regression as a tool for prediction and interpretation. We focused on

- the **line of best fit** ($Y' = bX + a$),
- the **least squares** criterion,
- the **standard error of the estimate**, and
- **multiple regression** (predicting behavior from multiple variables).

However, a good PyStatsV1 workflow does **not** stop after fitting a model. We must *engineer* our results by checking whether the model and the data behave as the assumptions require.

This appendix shows how to:

- use **pingouin** to fit a multiple regression model,
- compute standard regression diagnostics (residuals, leverage, Cook's distance),
- identify potentially influential observations, and
- illustrate the dangers of relying only on summary statistics using **Anscombe's Quartet**.

The goal is to give students a reproducible, testable set of tools they can reuse in their own projects.

53.2 Learning goals

After working through this appendix, you should be able to:

1. Explain the difference between **good fit** (e.g., high R^2) and **good model** (reasonable assumptions).
2. Interpret standard regression diagnostics:
 - residuals and standardized residuals,
 - leverage (hat values),
 - Cook's distance.
3. Use **pingouin's** regression tools together with NumPy and pandas to compute these diagnostics.
4. Explain how **Anscombe's Quartet** shows that
 - identical means, variances, and correlations can hide very different data patterns,
 - visualization and diagnostics are crucial in a PyStatsV1 workflow.

53.3 Files for this appendix

This appendix uses the following PyStatsV1 files:

- **Script:** `scripts/psych_ch16b_pingouin_regression_diagnostics.py`
 - simulates a psychology regression dataset (reusing the Chapter 16 data generator),
 - fits a multiple regression model with `pingouin.linear_regression()`,
 - computes regression diagnostics (residuals, leverage, Cook’s distance),
 - identifies the most influential observations,
 - generates diagnostic plots,
 - constructs and analyzes **Anscombe’s Quartet** to demonstrate why diagnostics and visualization matter.
- **Tests:** `tests/test_psych_ch16b_pingouin_regression_diagnostics.py`
 - verify that diagnostics have the expected shape and properties,
 - check that leverage behaves as theory predicts,
 - ensure that model R^2 is in a reasonable range,
 - run the full end-to-end pipeline in a temporary directory,
 - verify that CSV and PNG outputs are written correctly,
 - check that the Anscombe datasets have nearly identical summary statistics while having different shapes.
- **Makefile targets** (added in a separate CI branch):
 - `make psych-ch16b` – run the diagnostics demo (including Anscombe’s Quartet),
 - `make test-psych-ch16b` – run tests for this appendix only.

Note

As with previous chapters, the script is written in a way that makes it easy to import its functions into other projects or Jupyter notebooks. The tests treat regression diagnostics as *software* objects that can be checked, versioned, and reused.

53.4 Section 1 – Regression diagnostics in practice

Recall that a linear regression model makes several assumptions:

- **Linearity** – the relationship between predictors and outcome is approximately linear.
- **Homoscedasticity** – the spread (variance) of residuals is roughly constant across the range of fitted values.
- **Independence** – residuals are not systematically related to each other (e.g., no strong time trends).
- **Normality of residuals** – residuals are approximately normally distributed.

The pingouin function `pingouin.linear_regression()` focuses primarily on **estimation** and **inference**:

- regression coefficients and standard errors,
- t -tests and p -values,
- R^2 and adjusted R^2 .

To check assumptions, we need additional diagnostics. In `psych_ch16b_pinguin_regression_diagnostics.py` we therefore:

1. Simulate a dataset that extends the Chapter 16 example, with variables such as:
 - `stress`
 - `sleep_hours`
 - `study_hours`
 - `motivation`
 - `exam_score` (outcome)
2. Fit a multiple regression model predicting `exam_score` from several predictors.
3. Compute diagnostics using NumPy and pandas:
 - **fitted values** – model predictions \hat{y} ,
 - **residuals** – observed minus fitted ($y - \hat{y}$),
 - **standardized residuals** – residuals scaled by their estimated standard deviation,
 - **leverage** – hat values on the diagonal of the hat matrix, $H = X(X'X)^{-1}X'$,
 - **Cook's distance** – a measure of how much the regression coefficients would change if we removed a given observation.
4. Save the diagnostics to CSV and plot simple diagnostics:
 - **Residuals vs Fitted** plot – to check linearity and homoscedasticity.
 - **Leverage vs Cook's distance** plot – to identify high-leverage, influential observations.

53.4.1 Interpreting diagnostics (high level)

- Residuals should be roughly centered around zero. A clear curve or pattern in residuals vs fitted values suggests non-linearity.
- Leverage values near 0 indicate little influence on the model fit; values closer to 1 indicate observations that are far from the center of the predictor space.
- Cook's distance combines residual size and leverage. Points with unusually large Cook's distance are candidates for closer inspection. They are not automatically “bad” data points, but they may be influential.

53.5 Section 2 – Anscombe's Quartet

To see why diagnostics and visualization are essential, this appendix includes a second dataset: **Anscombe's Quartet**.

Anscombe (1973) constructed four small datasets (I–IV) with the following surprising property:

- Each dataset has nearly identical:
 - mean of x ,
 - mean of y ,
 - variance of x ,
 - variance of y ,
 - correlation r between x and y ,

- regression line $Y' = bX + a$.
- But when you plot them, the **shapes are completely different**:
 - One looks like a typical linear relationship.
 - One is clearly non-linear.
 - One is linear except for a single outlier.
 - One has a nearly perfect vertical line with one extreme point.

In other words, **summary statistics alone can mislead us**. Two datasets can share the same correlation and regression line but tell completely different stories once we visualize them.

53.5.1 How we use Anscombe's Quartet in PyStatsV1

The script `psych_ch16b_penguin_regression_diagnostics.py` includes:

- a helper that constructs a tidy version of **Anscombe's Quartet** with columns
 - `x`
 - `y`
 - `dataset` (I, II, III, IV)
- a function that computes, for each dataset:
 - \bar{x} ,
 \bar{y} ,
 - s_x^2 , s_y^2 ,
 - correlation r ,
 - simple regression line (a and b).
- a **2x2 grid of scatterplots** with
 - the same axis limits,
 - the fitted regression line overlaid,
 - one panel per dataset (I–IV).

The corresponding tests check that:

- all four datasets have nearly identical summary statistics, and
- the code produces the expected summary table and plot file.

53.5.2 Worked example (conceptual)

1. Run the diagnostics script (once your Makefile targets are wired):

```
make psych-ch16b
```

2. The script first runs the psychology regression diagnostics example (as described in Section 1).
3. Then the script constructs Anscombe's Quartet, computes summary statistics by dataset, and prints something like:

Anscombe summary (per dataset):

	dataset	mean_x	mean_y	var_x	var_y	r	slope	intercept
0	I	9.00	7.50	11.00	4.13	0.82	0.50	3.00
1	II	9.00	7.50	11.00	4.13	0.82	0.50	3.00
2	III	9.00	7.50	11.00	4.13	0.82	0.50	3.00
3	IV	9.00	7.50	11.00	4.13	0.82	0.50	3.00

The exact numbers may differ slightly due to floating point rounding, but the key idea is that the four datasets have almost identical summary statistics.

4. Finally, the script creates a 2x2 scatterplot figure and writes it to:

- `outputs/track_b/ch16b_anscombe_quartet.png`

When you inspect this image, you will see four very different patterns, despite having “the same” regression summary.

53.5.3 Takeaway for students and instructors

Anscombe’s Quartet makes two core points that align with the PyStatsV1 philosophy:

1. **Do not stop at statistics.**

- A single number like r or R^2 can hide very different data stories.
- Always pair numerical output with plots and diagnostics.

2. **Treat models as software artifacts.**

- In PyStatsV1, every substantial analysis step is backed by functions, tests, and CI checks.
- Adding a new diagnostic (e.g., Cook’s distance, Anscombe analysis) means adding new code *and* new tests.

53.6 Section 3 – The code: overview of key functions

You do not need to memorize the exact implementation details, but it is useful to know what the main functions do.

In `scripts/psych_ch16b_penguin_regression_diagnostics.py`:

- `compute_regression_diagnostics(df, predictors, outcome)`

- Fits a multiple regression model,

exam

score

simstudy

hours + sleep

hours + stress + motivation,

- returns a diagnostics DataFrame with

- * `fitted`,
- * `residual`,
- * `std_residual`,
- * `leverage`,
- * `cooks_distance`,

- and a `pingouin` regression summary table for cross-checking.
- `run_ch16b_demo(n, random_state)`
 - Simulates the psychology regression dataset,
 - calls `compute_regression_diagnostics()`,
 - saves diagnostics and **top influential points** to CSV,
 - generates residuals vs fitted and leverage vs Cook’s distance plots,
 - constructs and analyzes Anscombe’s Quartet,
 - saves Anscombe summary statistics and plots,
 - prints a concise narrative summary to the console.
- Anscombe helpers (internal names may differ slightly):
 - a function to construct the tidy Anscombe dataset,
 - a function to compute summary statistics by dataset,
 - a plotting function to generate the 2x2 Anscombe scatterplot figure with regression lines.

In `tests/test_psych_ch16b_pingouin_regression_diagnostics.py`:

- One test verifies that diagnostics have the expected columns and that leverage behaves as theory predicts (e.g., the average leverage is approximately p/n , where p is the number of parameters including the intercept).
- Another test runs `run_ch16b_demo()` in a temporary directory and verifies that all expected CSV and PNG files exist and are non-empty.
- A third test checks that **Anscombe’s Quartet** is implemented correctly:
 - there are four datasets with the expected number of rows,
 - group-level summary statistics are nearly identical across datasets,
 - the code produces an Anscombe summary CSV and plot image.

53.7 How this Appendix fits into the Track B narrative

- Chapter 15 and 15a introduced **correlation** and **partial correlation**, using `pingouin` as a high-level toolbox.
- Chapter 16 developed the core ideas of **linear regression**: prediction, least squares, standard error of the estimate, and multiple regression.
- Appendix 16a expanded regression with additional estimation examples.
- Appendix 16b (this chapter) emphasizes that
 - even a beautifully written model can be misleading if we ignore diagnostics,
 - the *shape* of the data always matters,
 - simple, testable diagnostics can be integrated into every analysis pipeline.

By the time students reach Chapter 17 (Mixed-Model Designs), they will have seen that a PyStatsV1-style analysis is not just about “getting significant results.” It is about building **robust, transparent, and reproducible** statistical workflows that can be trusted.

53.8 Next steps

After completing this appendix, you are ready to move into

- **Chapter 17 – Mixed-Model Designs**, where we combine between-subjects and within-subjects factors, and
- later, **Chapter 18 – ANCOVA**, where we explicitly control for covariates in more complex models.

In both chapters, the habits you practiced here—**checking assumptions, visualizing patterns, and treating models as software artifacts**—will remain central.

CHAPTER 17 – MIXED-MODEL DESIGNS

54.1 Learning goals

In this chapter you will learn how to:

- describe the logic of *mixed-model* (split-plot) designs,
- distinguish between **between-subjects** and **within-subjects** factors,
- understand why mixed designs have different error terms for different effects,
- interpret a mixed ANOVA table (Group, Time, and Group \times Time),
- and use `pystatsv1` and `pingouin` to analyze a simple treatment study.

By the end of the chapter, you should be able to read a mixed ANOVA output and explain, in plain language, what each line means for a psychology research question.

54.2 17.1 The hybrid design: between-subjects + within-subjects

So far in Track B you have seen:

- **Between-subjects** designs (Chapters 10, 12, 13), where each participant belongs to one condition only; and
- **Within-subjects** designs (Chapters 11 and 14), where the *same* participants are measured repeatedly (e.g., Pre, Post, Follow-up).

A **mixed-model design** combines these two ideas. A classic example is a treatment study where:

- people are randomly assigned to a **Group** (Treatment vs Control), and
- everyone is measured at multiple **Time** points (Pre, Post, Follow-up).

Note

In psychology, mixed designs are extremely common. Any longitudinal study that compares two or more groups over time is likely to be a mixed model.

54.2.1 Terminology

Between-subjects factor

A factor where different participants belong to different levels. In this chapter, group ("control" vs "treatment") is a between-subjects factor.

Within-subjects factor

A factor where each participant is measured at *each* level. In this chapter, time ("pre", "post", "followup") is a within-subjects factor.

Mixed design

A design that includes at least one between-subjects factor and at least one within-subjects factor. Sometimes called a *split-plot* design.

54.2.2 Why use a mixed design?

Mixed designs give you the best of both worlds:

- You can look at **group differences** (Treatment vs Control).
- You can look at **change over time** (Pre vs Post vs Follow-up).
- You can test whether the **pattern of change over time is different for each group** (the Group \times Time *interaction*).

This is usually the main scientific question:

Did the treatment group improve more over time than the control group?

54.3 17.2 The split-plot logic and error terms

In a pure between-subjects ANOVA (Chapter 13), all error comes from **differences between participants** within each condition.

In a pure repeated-measures ANOVA (Chapter 14), a lot of that individual difference error is removed because each person acts as their own control.

In a **mixed** design, we have both types of variation:

- differences **between participants** (some students are generally more anxious than others), and
- differences **within participants over time** (everyone may change from Pre to Post to Follow-up).

A mixed ANOVA therefore has different error terms for different effects:

- The **Group** effect (Treatment vs Control) uses an error term based on *between-subject* variability.
- The **Time** effect and the **Group \times Time interaction** use an error term based on *within-subject* variability.

You do **not** have to compute these error terms by hand in this chapter. Instead, we focus on:

- understanding the design,
- structuring the data correctly,
- and learning how to read the output from a trusted library (here, pingouin).

54.4 17.3 Example: Treatment vs control across three time points

We will work with a simple, synthetic example that mimics a common clinical psychology design.

54.4.1 Scenario

A clinical psychologist wants to test whether a new cognitive-behavioural program reduces anxiety compared to a waitlist control group.

- Participants are randomly assigned to **Treatment** or **Control**.
- Everyone completes an anxiety scale at three time points:

- pre – before treatment starts,
- post – immediately after treatment, and
- followup – three months later.

54.4.2 Hypotheses

- **Group main effect:**

H0: There is no overall difference in anxiety between Treatment and Control. *H1*: One group has higher average anxiety than the other.

- **Time main effect:**

H0: Average anxiety is the same at Pre, Post, and Follow-up. *H1*: Average anxiety changes over time (e.g., decreases after treatment).

- **Group × Time interaction** (the most important):

H0: The pattern of change over time is the same for both groups. *H1*: The pattern of change over time is *different* (e.g., Treatment improves more from Pre to Post and maintains gains at Follow-up).

54.4.3 Data structure

As with Chapter 14, we will work with both **wide** and **long** formats.

Wide format (one row per participant):

subject	group	anxiety_pre	anxiety_post	anxiety_followup
01	control	52.3	50.1	48.7
02	control	47.8	49.2	48.9
03	treatment	51.9	40.6	38.2
...				

Long format (one row per person per time point):

subject	group	time	anxiety
control_01	control	pre	52.3
control_01	control	post	50.1
control_01	control	followup	48.7
treat_01	treatment	pre	51.9
treat_01	treatment	post	40.6
treat_01	treatment	followup	38.2
...			

Most mixed ANOVA functions (including `pingouin.mixed_anova()`) expect the **long** format with columns that label:

- the **within-subjects factor** (time),
- the **between-subjects factor** (group), and
- the **dependent variable** (anxiety).

54.5 17.4 PyStatsV1 lab – structuring and analyzing a mixed design

The Chapter 17 lab is implemented in the script `scripts.psych_ch17_mixed_models`. It has three main responsibilities:

1. **Simulate a mixed design dataset** where Treatment improves more than Control over time.
2. **Reshape the data** into both wide and long formats.
3. **Run a mixed ANOVA with `:mod:`pingouin``** and save clean outputs for students to inspect.

54.5.1 Simulating the data

The function `simulate_mixed_design_dataset()` constructs a dataset with:

- two groups ("control" and "treatment"),
- three time points ("pre", "post", "followup"),
- and an anxiety outcome designed so that:
 - both groups start with similar anxiety at pre,
 - the treatment group shows a strong drop from pre to post, and
 - the control group changes very little.

To keep the lab deterministic and testable, the simulation uses a fixed random seed by default.

54.5.2 Running the mixed ANOVA

The core analysis uses `pingouin.mixed_anova()` applied to the long-format data:

```
import pingouin as pg

aov = pg.mixed_anova(
    data=long_df,
    dv="anxiety",
    within="time",
    between="group",
    subject="subject",
)
```

The resulting table contains separate rows for:

- the **Group** main effect,
- the **Time** main effect, and
- the **Group × Time** interaction.

For each effect, you get:

- degrees of freedom (DF1, DF2),
- F-statistic (F),
- p-value (p-unc),
- and effect size metrics such as partial eta-squared (np2).

When the simulation is working correctly, you should see:

- a modest or small **Group** main effect,

- a clear **Time** main effect (participants change over time), and
- a strong **Group × Time** interaction (Treatment improves more than Control).

54.5.3 Saved outputs

When you run the lab via the Makefile target:

```
make psych-ch17
```

the script will:

- print key results to the console,
- save the simulated long-format data to:

```
data/synthetic/psych_ch17_mixed_design_long.csv
```

- save the wide-format data to:

```
data/synthetic/psych_ch17_mixed_design_wide.csv
```

- save the mixed ANOVA table to:

```
outputs/track_b/ch17_mixed_anova.csv
```

- and create an interaction plot (group means over time) at:

```
outputs/track_b/ch17_group_by_time_means.png
```

Instructors can use these files for in-class demonstrations, and students can use them for homework or project work without having to re-run the simulation.

54.6 Connection to future chapters

Mixed-model designs sit at the intersection of several ideas you have already seen:

- **Factorial logic** from Chapter 13 (main effects and interactions),
- **Repeated-measures logic** from Chapter 14 (within-subject error terms),
- and **Regression logic** from Chapters 15–16 (predicting outcomes from multiple sources of information).

The next chapters extend these ideas further:

- Chapter 18 shows how to statistically control for a *covariate* using ANCOVA.
- Chapter 19 introduces non-parametric alternatives for situations where standard ANOVA assumptions are not met.
- Chapter 20 brings everything together in a full PyStatsV1 project.

For now, the goal is not to master every technical detail of mixed-model mathematics, but to develop a solid *conceptual* understanding and a reliable, reproducible workflow for analyzing the kinds of treatment-over-time studies that are central to modern psychological science.

CHAPTER 18 – ANALYSIS OF COVARIANCE (ANCOVA)

Chapter overview

- *Learning goals*
- *18.1 Statistical control and covariates*
- *18.2 The logic of ANCOVA*
- *18.3 Adjusted means and interpretation*
- *18.4 Assumptions of ANCOVA*
- *18.5 PyStatsV1 Lab – One-way ANCOVA with a pre-test covariate*
- *Concept check*

In Chapter 17 you learned how to analyse *mixed-model* designs with both between-subjects and within-subjects factors. In this chapter we introduce a closely related idea: using a *covariate* to statistically control for pre-existing differences between participants.

Analysis of covariance (ANCOVA) combines the logic of regression and ANOVA. It answers questions like:

- Do two treatment groups differ on a post-test **after controlling for** baseline differences?
- Does a new therapy reduce anxiety **over and above** what we can predict from initial symptom severity?

Throughout the chapter we will work with a simple psychology example and use `pystatsv1` and `pingouin` to fit and interpret an ANCOVA model.

55.1 Learning goals

After working through this chapter you should be able to:

- explain what a covariate is and why researchers include covariates in experimental designs,
- distinguish between **raw** (unadjusted) group means and **adjusted** means from an ANCOVA,
- describe the assumptions of ANCOVA (linearity, homogeneity of regression slopes, reliability of the covariate),
- run a basic one-way ANCOVA in Python using `pingouin`,
- interpret the output (F statistic, p-value, effect size, adjusted means),
- understand how ANCOVA is related to multiple regression.

55.2 18.1 Statistical control and covariates

Suppose a researcher is evaluating a new study-skills workshop designed to improve exam performance. Students volunteer and are randomly assigned to either a **control** group (no workshop) or a **treatment** group (workshop). Everyone completes a pre-test measuring current study skills and a final exam at the end of term.

In an ideal randomized experiment, random assignment ensures that the two groups are similar *on average* before the intervention. In practice, however, there will always be some pre-existing differences. In our example, some students may start with better study skills or higher motivation.

A **covariate** is a continuous variable that is:

- measured prior to the manipulation (e.g., pre-test score),
- related to the outcome (e.g., final exam score), and
- **not** directly affected by the experimental treatment.

ANCOVA uses the covariate to statistically control for pre-existing differences. Conceptually, we are asking:

*“If all students had started with the ****same*** pre-test score, would the treatment and control groups still differ on the exam?”**

55.3 18.2 The logic of ANCOVA

ANCOVA can be viewed in two equivalent ways:

- as an ANOVA that has been extended to include a continuous predictor, or
- as a multiple regression in which group membership is coded as a categorical predictor and the covariate is a continuous predictor.

The key idea is to partition the variance in the outcome into:

- variance explained by the covariate,
- variance explained by the group factor *after controlling for the covariate*, and
- residual (error) variance.

If the group factor explains a non-trivial amount of variance **over and above** the covariate, the ANCOVA will yield a significant F statistic for the group effect. The adjusted means provide a way to visualize that effect.

55.4 18.3 Adjusted means and interpretation

Because the covariate is continuous, each participant has a unique combination of covariate value and outcome value. ANCOVA uses the regression of the outcome on the covariate to compute **adjusted means** for each group at a common reference value of the covariate (often the overall mean).

In our example, imagine that we adjust all students to have the same pre-test score. The adjusted means then tell us what the average exam score *would have been* for each group **if** they had started at the same baseline.

When reporting ANCOVA results in APA style, researchers typically:

- report the F statistic, degrees of freedom, p-value, and effect size for the group effect,
- describe the direction and magnitude of the adjusted group difference,
- mention the covariate and whether it was a significant predictor of the outcome.

For example:

“Controlling for pre-test study skills, students in the workshop condition scored higher on the final exam than those in the control condition, $F(1, 77) = 8.42$, $p = .005$, partial $\eta^2 = .10$.”

55.5 18.4 Assumptions of ANCOVA

ANCOVA shares many assumptions with regression and ANOVA:

- **Linearity** – the relationship between the covariate and outcome is approximately linear within each group.
- **Homogeneity of regression slopes** – the slope relating the covariate to the outcome is similar for each group. If the slopes differ substantially, a model with an interaction between group and covariate may be more appropriate.
- **Independence of observations** – the usual assumption for between- subjects designs.
- **Normality and homogeneity of variance** – residuals are approximately normal and have similar variance across groups.
- **Reliable covariate** – the covariate should be measured with reasonable reliability; noisy covariates provide little benefit and can even reduce power.

In practice, researchers check these assumptions using plots (e.g., scatterplots and residual plots) and model diagnostics.

55.6 18.5 PyStatsV1 Lab – One-way ANCOVA with a pre-test covariate

The Chapter 18 lab shows how to run a simple one-way ANCOVA using a synthetic psychology dataset.

The script `scripts.psych_ch18_ancova`:

- simulates data for a control and treatment group,
- includes a **pre-test** covariate that is correlated with the **post-test** exam score,
- compares an ordinary one-way ANOVA on the post-test scores to a one-way ANCOVA that controls for the pre-test,
- uses `pingouin.ancova()` to fit the ANCOVA and report the F statistic, p-value, partial η^2 , and adjusted means,
- saves the synthetic dataset and ANCOVA table to the usual `data/synthetic` and `outputs/track_b` folders, and
- produces a simple plot that visualizes the group effect before and after adjusting for the covariate.

To run the lab from the command line, use the Makefile target:

```
make psych-ch18
```

or, equivalently:

```
python -m scripts.psych_ch18_ancova
```

To run the tests for this chapter only:

```
make test-psych-ch18
```

As in earlier chapters, the tests provide a lightweight “contract” for the simulation:

- the covariate must be positively correlated with the outcome,
- the ANCOVA model must show a significant treatment effect when it is present in the data-generating process, and

- the adjusted mean for the treatment group should exceed that of the control group.

55.7 Concept check

- Why might an experimenter include a pre-test covariate instead of simply comparing post-test scores with a t-test or one-way ANOVA?
- What does it mean to say that ANCOVA “controls for” a covariate?
- How are adjusted means different from raw means?
- What does the assumption of homogeneity of regression slopes require?
- How is ANCOVA related to multiple regression?

In the next chapter we will turn to non-parametric statistics – tools that relax some of the assumptions we have relied on so far and allow us to analyse ordinal and highly non-normal data.

CHAPTER 19 – NON-PARAMETRIC STATISTICS

56.1 Learning goals

By the end of this chapter you will be able to:

- Explain when non-parametric tests are preferred over traditional (parametric) procedures such as the t -test or ANOVA.
- Describe the logic of the chi-square family of tests.
- Distinguish between chi-square tests of **goodness of fit** and **independence**.
- Recognize rank-based alternatives to t -tests and one-way ANOVA (Mann–Whitney U, Wilcoxon signed-rank, Kruskal–Wallis, Friedman).
- Use PyStatsV1 and pingouin to analyze survey-style data with chi-square tests on categorical variables.

56.2 19.1 When parametric assumptions break down

In earlier chapters, we focused on *parametric* procedures:

- t -tests (Chapters 9–11)
- One-way and factorial ANOVA (Chapters 12–14)
- Regression and ANCOVA (Chapters 16–18)

These procedures make several assumptions about the data:

- **Quantitative scale** – variables are interval or ratio, not purely nominal.
- **Normality** – scores within each group are (approximately) normally distributed.
- **Homogeneity of variance** – population variances are equal across groups.
- **Linearity** – for correlation and regression, the relationship between variables is approximately linear.

When these assumptions are badly violated, parametric tests can give misleading p -values and confidence intervals. In those cases, we often turn to **non-parametric** methods.

Non-parametric tests typically:

- Work with **ranks** or **counts** rather than raw numeric values.
- Make **fewer distributional assumptions**.
- Are often slightly **less powerful** when parametric assumptions *are* met, but **more robust** when those assumptions fail.

In psychological research, non-parametric tests are especially useful when:

- The outcome is **ordinal** (e.g., Likert scales: “Strongly disagree” to “Strongly agree”).
- The data are **severely skewed** or have heavy **outliers** that cannot be reasonably transformed.
- The variable is **categorical** (e.g., therapy preference, diagnostic category, treatment response yes/no).

56.3 19.2 Chi-square tests for categorical data

The most common non-parametric tests in introductory psychology involve **frequency counts** in categories. The basic question is:

Do the observed counts differ from what we would expect by chance?

The chi-square family addresses this question in two main situations.

56.3.1 19.2.1 Goodness of fit

A **chi-square goodness-of-fit test** compares observed category counts to a theoretical or expected distribution. For example:

- A survey asks which coping strategy students use most often: *Exercise, Therapy, Mindfulness, or Social support*.
- If there was **no preference**, we would expect roughly equal counts in each category (25% each).
- The chi-square goodness-of-fit test asks whether the observed distribution differs significantly from this uniform expectation.

Statistically, we compute:

$$\chi^2 = \frac{(\text{Observed} - \text{Expected})^2}{\text{Expected}}$$

with degrees of freedom $df = k - 1$, where k is the number of categories.

If χ^2 is large relative to its degrees of freedom, the p -value will be small and we reject the null hypothesis that the observed frequencies match the expected distribution.

56.3.2 19.2.2 Test of independence

A **chi-square test of independence** asks whether two categorical variables are related. For example:

- Variable 1: Type of therapy received (*Control, CBT, Mindfulness*).
- Variable 2: Treatment outcome (*Improved vs. Did not improve*).

We arrange the counts in a **contingency table** and again compute a chi-square statistic. Here, the null hypothesis states that the variables are **independent** – knowing a person’s therapy type tells you nothing about their likelihood of improvement.

We also report an **effect size** such as **Cramér’s V**, which is based on the chi-square value but scaled to lie between 0 and 1:

- ~0.10: small association
- ~0.30: medium
- ~0.50 or higher: large

56.4 19.3 Rank-based tests

Not all non-parametric tests are based on counts. Many are based on **ranks** of the outcome variable. Instead of analyzing raw scores, we:

1. Combine all scores across groups.
2. Rank them from lowest to highest.
3. Analyze the ranks using an appropriate test statistic.

Some common rank-based tests and their parametric counterparts:

- **Mann–Whitney U**: alternative to an independent-samples t -test.
- **Wilcoxon signed-rank**: alternative to a paired-samples t -test.
- **Kruskal–Wallis H**: alternative to a one-way ANOVA with independent groups.
- **Friedman test**: alternative to a repeated-measures one-way ANOVA.

These tests are especially helpful when:

- The outcome is ordinal (e.g., 1–7 rating scales).
- The data are heavily skewed or contain extreme outliers.
- Sample sizes are small, making normality assumptions doubtful.

56.5 19.4 When to choose non-parametric methods

There is no single “magic rule,” but some practical guidelines:

- **Use chi-square tests** when both your predictor and outcome are **categorical** (nominal) and you are working with **counts**, not percentages.
- **Use rank-based tests** when:
 - * The outcome variable is **ordinal**, or
 - * You have strong violations of normality or homogeneity that cannot be fixed by transformations, and
 - You are more concerned about **validity** than about squeezing out every bit of statistical **power**.

When in doubt, you can often:

- Run the **parametric** test (e.g., t -test or ANOVA).
- Run the **non-parametric** alternative.
- Compare conclusions – if they agree, your result is probably robust.

56.6 19.5 PyStatsV1 Lab: Chi-square analysis of survey data

In this chapter’s lab you will use PyStatsV1 to analyze **simulated survey data** using chi-square tests. The code lives in:

- `scripts.psych_ch19_nonparametrics`
- `tests.test_psych_ch19_nonparametrics`

56.6.1 Running the lab

From the project root (with your virtual environment activated), run:

```
make psych-ch19
make test-psych-ch19
```

The first command will:

1. Simulate a **coping strategies** survey with four categories (e.g., Exercise, Therapy, Mindfulness, Social support).
2. Run a **chi-square goodness-of-fit** test to check whether the observed distribution differs from a uniform (no-preference) null.
3. Save the raw data and a summary table to:
 - data/synthetic/psych_ch19_survey_gof.csv
 - outputs/track_b/ch19_gof_table.csv
4. Generate a bar chart comparing **observed** versus **expected** counts:
 - outputs/track_b/ch19_gof_barplot.png

The second dataset in the script simulates a **therapy × improvement** contingency table:

1. Students are randomly assigned to *Control*, *CBT*, or *Mindfulness* conditions.
2. Each person is classified as *Improved* or *No change*.
3. The script uses:
 - `scipy.stats.chi2_contingency()` for a traditional chi-square test.
 - `pingouin.chi2_independence()` to obtain effect sizes (e.g., Cramér's V) and power estimates.
4. The script saves:
 - data/synthetic/psych_ch19_survey_independence.csv – individual-level data.
 - outputs/track_b/ch19_independence_table.csv – full chi-square summary.
 - outputs/track_b/ch19_stacked_bar.png – a stacked bar plot showing the proportion improved within each therapy type.

56.6.2 Interpreting the output

After running `make psych-ch19`, inspect the console output and figures:

- For the **goodness-of-fit** example, ask:
 - Does the chi-square test detect that some coping strategies are preferred over others?
 - Which categories contribute most to the chi-square statistic (largest observed – expected differences)?
- For the **independence** example, ask:
 - Is there evidence that treatment type and improvement are associated?
 - How large is the association (Cramér's V)?
 - Do the stacked bar plots reveal a pattern that matches the numerical results?

56.6.3 Connection to earlier chapters

This chapter ties together several themes from earlier in the book:

- Just as in Chapter 7, we rely on **sampling distributions** to interpret chi-square statistics.
- As in Chapters 9–12, we balance **Type I error** (false positives) against **power** (true positives).
- In Chapters 16–18, we extended ANOVA to regression and ANCOVA. Here, we extend the logic of hypothesis testing to **categorical outcomes** and **ordinal data**.

Non-parametric methods are not a separate universe – they are another set of tools in your scientific toolbox. When used thoughtfully, they allow you to test important psychological questions even when real-world data refuse to behave “nicely.”

CHAPTER 19A – RANK-BASED NON-PARAMETRIC ALTERNATIVES

57.1 Where this chapter fits in the story

In **Chapter 10–12** you met the classic parametric workhorses:

- Independent-samples t test (Chapter 10)
- Paired-samples t test (Chapter 11)
- One-way ANOVA (Chapter 12)

In **Chapter 19** you stepped into a different world: categorical outcomes, contingency tables, and χ^2 tests (goodness-of-fit and independence). Those procedures are designed for **counts in categories**.

This appendix, Chapter 19a, fills in the missing bridge:

What do we do when our outcome is still a continuous score (like stress, reaction time, or exam performance), but the assumptions of t and ANOVA are badly violated?

That is the home territory of **rank-based non-parametric tests**. They keep the basic research questions from Chapters 10–12 but answer them using **ranks instead of raw scores**.

57.2 When to reach for rank-based tests

Rank-based tests are especially useful when:

- Your outcome is **ordinal** or strongly **skewed** (e.g., reaction times, income, symptom counts).
- You have **outliers** that are hard to justify removing, and transformations do not fully fix the problem.
- Your sample sizes are moderate or small, and normal-theory approximations for t and F are questionable.

They are still asking the *same substantive questions* as the parametric tests:

- Is there a difference between two independent groups?
- Is there a difference between two paired conditions (e.g., pre vs. post)?
- Are there differences among three or more groups?

But instead of modeling the **mean** of a (roughly) normal distribution, they work with **ranks** and use test statistics that are robust to shape and outliers.

57.3 Mann–Whitney U: Alternative to an independent-samples t test

Recall the independent-samples t test from Chapter 10:

- Two independent groups (e.g., Control vs. Treatment)

- Approximately normal scores within each group
- Roughly equal variances

The **Mann–Whitney U** test answers a similar question using ranks. Instead of comparing group means, it asks whether scores in one group tend to be **larger** than scores in the other group.

Key ideas:

- Combine all scores from both groups, rank them from lowest to highest.
- Compute a statistic U that reflects how often one group has higher ranks than the other.
- Under the null hypothesis (no difference), rank patterns are similar between groups.

In the lab script `scripts.psych_ch19a_rank_nonparametrics`, we:

- Simulate a continuous, **skewed** outcome for Control and Treatment.
- Run `pingouin.mwu()` to obtain U , p , and an effect size (rank-biserial correlation).
- Visualize the distribution with boxplots to highlight skew and overlap.

57.4 Wilcoxon signed-rank: Alternative to a paired-samples t test

In Chapter 11, the paired-samples t test analyzed **difference scores** (e.g., Post – Pre) under the assumption that those differences were roughly normal.

The **Wilcoxon signed-rank** test keeps the same design (paired data), but:

- Works with the **ranks of absolute differences** instead of the raw differences.
- Uses the signs (+/-) to capture direction (improvement vs. decline).

It is especially useful when:

- Individual differences are large.
- Change scores are asymmetric or have outliers (e.g., a few participants improve *a lot*).

In the lab script we:

- Simulate Pre vs. Post scores with a **positive shift** plus skew and noise.
- Run `pingouin.wilcoxon()` on the paired data.
- Confirm that Wilcoxon detects the systematic shift, even when parametric assumptions are dubious.

57.5 Kruskal–Wallis: Alternative to one-way ANOVA

The one-way ANOVA from Chapter 12 extends the independent-samples t test to three or more groups, comparing **means** via an F statistic.

The **Kruskal–Wallis** test is its rank-based cousin:

- Combine all group scores and rank them.
- Compute a statistic H that reflects how far each group's **mean rank** is from the overall mean rank.
- Under the null hypothesis, all groups have similar rank distributions.

In the lab script we:

- Simulate a three-group dose example (Low, Medium, High) with positively skewed scores.
- Run `pingouin.kruskal()` to test for any group differences.

- Confirm that the Kruskal–Wallis test recovers the intended order of group effects.

57.6 PyStatsV1 Lab: Rank-based non-parametric tests in action

The Chapter 19a lab is implemented in:

- `scripts.psych_ch19a_rank_nonparametrics`
- `tests.test_psych_ch19a_rank_nonparametrics`

The script does three things:

1. Mann–Whitney U demo

- Simulates skewed scores for Control vs. Treatment.
- Runs `pingouin.mwu()` and prints the U statistic, p value, and effect size.
- Saves the simulated dataset to:
 - `data/synthetic/psych_ch19a_mannwhitney_demo.csv`
- Saves the test results to:
 - `outputs/track_b/ch19a_mannwhitney_results.csv`

2. Wilcoxon signed-rank demo

- Simulates Pre vs. Post scores for the same participants.
- Runs `pingouin.wilcoxon()` on the paired data.
- Saves the wide-format data (one row per participant) to:
 - `data/synthetic/psych_ch19a_wilcoxon_demo.csv`
- Saves the test results to:
 - `outputs/track_b/ch19a_wilcoxon_results.csv`

3. Kruskal–Wallis demo

- Simulates a three-group design (e.g., Low / Medium / High dosing).
- Runs `pingouin.kruskal()` to test for differences among groups.
- Saves the dataset to:
 - `data/synthetic/psych_ch19a_kruskal_demo.csv`
- Saves the test results to:
 - `outputs/track_b/ch19a_kruskal_results.csv`

Finally, the script creates a small figure comparing group distributions (via boxplots) and saves it as:

- `outputs/track_b/ch19a_rank_nonparam_boxplots.png`

57.7 Running the Chapter 19a lab

From the project root, you can run the full demo with:

```
make psych-ch19a
```

To run only the tests for this chapter:

```
make test-psych-ch19a
```

These targets simply wrap:

- `python -m scripts.psych_ch19a_rank_nonparametrics`
- `pytest tests/test_psych_ch19a_rank_nonparametrics.py`

57.8 Conceptual summary

- Chapter 19 (chi-square) focused on **categorical outcomes** and counts in categories.
- Chapter 19a (this appendix) focuses on **continuous / ordinal outcomes** where parametric assumptions are shaky.

Rank-based non-parametric tests:

- Keep the *research questions* from your t-tests and ANOVAs.
- Answer them using **ranks** and robust test statistics instead of relying on normality of raw scores.
- Provide a practical toolset when data are messy, skewed, or resistant to transformation.

In more advanced courses (e.g., Robust Methods, Categorical Data Analysis, or Generalized Linear Models), you will see how these ideas generalize further: logistic regression for categorical outcomes, robust regression for outliers, and permutation tests for flexible, assumption-lean inference.

For now, Chapter 19a gives you a principled way to say:

“Even when the assumptions of t and ANOVA are not met, I can still design and analyze my study using methods that respect the structure of my data.”

CHAPTER 20 – THE RESPONSIBLE RESEARCHER (CONCLUSION)

58.1 Where this chapter fits in the story

Chapters 1–19 walked you through the **tools** of quantitative psychology:

- From z-scores, *t* tests, and ANOVA,
- Through correlation, regression, and mixed-model designs,
- All the way to ANCOVA and non-parametric alternatives.

Along the way, the PyStatsV1 labs treated each analysis as **production software**:

- Deterministic simulators with fixed random seeds.
- Re-usable helper functions and command-line entry points.
- Tests that automatically verify key properties of the results.

This final chapter zooms out from individual techniques to the broader question:

What does it mean to do responsible, cumulative, and reproducible research?

We will highlight three pillars:

1. **Power analysis** – planning sample sizes before you collect data.
2. **Meta-analysis** – combining evidence across studies.
3. **Clear communication** – writing honest, transparent summaries of what your data can (and cannot) support.

The chapter closes with a **final PyStatsV1 project** that guides you from raw data to an APA-style report, using the tools you developed in earlier labs.

58.2 20.1 Power analysis: Planning samples before you collect data

58.2.1 Why power matters

Every statistical test juggles four quantities:

- **Effect size** (how big the effect really is).
- **Sample size** :math:`N` (how much data you collect).
- **Significance level** :math:`\alpha` (your Type I error rate).
- **Power** (the probability of detecting the effect if it is real).

Once you fix any three of these, the fourth is determined. Power analysis is about *solving that equation on purpose* instead of hoping that “ $N = 30$ per group” will magically be enough.

A study with very low power is problematic because:

- True effects are often *missed* (high Type II error).
- Effects that *are* detected tend to be over-estimated (“winner’s curse”).
- Resources (time, money, participant goodwill) can be wasted on studies that were never likely to succeed.

58.2.2 A priori vs. post hoc power

- **A priori power analysis** happens *before* you collect data.
 - You specify a meaningful effect size (e.g., $d = 0.5$ for a medium standardized mean difference).
 - Choose α (often 0.05) and desired power (often 0.80).
 - Solve for N per group.
- **Post hoc (“observed”) power** is calculated *after* the fact, using the effect size observed in your sample.
 - This value is mostly a complicated re-expression of p and is rarely informative.
 - In PyStatsV1 we emphasize **a priori** planning instead.

In the Chapter 20 lab script, we use `pingouin` to compute sample sizes for simple scenarios (e.g., independent-samples t tests), and we write the results to a small **power grid** for inspection.

58.2.3 Practical considerations

Some practical rules-of-thumb you will see in the wild:

- If you expect a *large* effect ($d \approx 0.8$), smaller samples might be OK (but replication still matters).
- If you expect a *small* effect ($d \approx 0.2$), you may need hundreds of participants per group to achieve good power.
- In within-subjects designs, power is boosted by lower error variance (participants act as their own control).

Power analysis is ultimately **ethical** as well as technical. You are deciding how many people to involve, how much time to spend, and how likely your study is to make a cumulative contribution.

58.3 20.2 Meta-analysis: The study of studies

Individual experiments are noisy. Even well-planned studies will occasionally miss true effects or overstate them. **Meta-analysis** is a set of tools for combining evidence across multiple studies.

At a high level:

- Each study contributes an **effect size** (e.g., Cohen’s d , correlation r) and an **estimate of its precision** (e.g., a standard error or variance).
- More precise studies (usually those with larger N) receive **more weight**.
- A combined or **pooled effect size** is computed, along with confidence intervals, measures of heterogeneity, and often tests for moderation (do effects differ by method, sample, or context?).

58.3.1 Fixed-effect vs. random-effects models

- In a **fixed-effect** meta-analysis, we assume all studies are estimating the *same* underlying true effect.
 - Differences among studies are attributed only to sampling error.
 - The pooled estimate answers the question: “What is the best estimate of the common effect size in this set of studies?”

- In a **random-effects** meta-analysis, we allow the true effect to *vary* from study to study (e.g., different labs, populations, or protocols).
 - We estimate both the *typical* effect and the *heterogeneity* among effects.
 - The pooled estimate answers: “What is the average effect across a distribution of study contexts?”

In Chapter 20, we keep things simple with a **fixed-effect illustration**:

- We simulate several “published” effect sizes with different sample sizes.
- We compute a weighted mean effect size and its confidence interval.
- We calculate a basic heterogeneity statistic (Q and I^2) to flag when the effects are more variable than chance alone would predict.

58.3.2 How this connects back to earlier chapters

Earlier chapters focused on **within-study** inference (what can we conclude from *this* experiment?). Meta-analysis steps back and asks:

- How consistent are the effects across **many** experiments?
- When results disagree, is it due to chance, small samples, or genuine differences in context or methods?
- How can we make decisions (in policy, clinical practice, or scientific theory) that respect the *whole* body of evidence?

58.4 20.3 Communicating results responsibly

Statistical tools are only as useful as the **stories we tell** with them. Responsible communication means:

- Being **transparent** about your methods (design, sampling, analytic choices).
- Reporting **effect sizes** and **confidence intervals**, not just p values.
- Discussing **limitations** and **alternative explanations**.
- Being honest about the **uncertainty** that remains.

58.4.1 Writing the Discussion section

A good Discussion section typically:

1. **Restates the research questions** in plain language.
 - What did we want to know?
 - How does this connect to theory or prior research?
2. **Summarizes the key findings** without overstating them.
 - Focus on patterns of results, not just individual p values.
 - Tie back to effect sizes and confidence intervals.
3. **Integrates with prior work**.
 - Do your results replicate or challenge previous findings?
 - How might they fit into a broader meta-analytic picture?
4. **Acknowledges limitations**.
 - Sample characteristics (e.g., only undergraduates from one university).

- Measurement issues (e.g., self-report scales, ceiling effects).
- Design constraints (e.g., no true random assignment).

5. **Outlines future directions.**

- What follow-up studies could clarify the story?
- How might improved design, larger samples, or different populations alter the conclusions?

The aim is not to **sell** your results, but to **situate** them – as one piece of a collaborative, cumulative effort.

58.5 20.4 PyStatsV1 Lab: A final project from raw data to APA report

The last PyStatsV1 lab is different from earlier chapters. Instead of a single, tightly scripted analysis, you will:

1. Choose a **research question**.
2. Select or import a **dataset**.
3. Design an analysis pipeline using the tools you have already implemented.
4. Generate a short **APA-style report** with transparent, reproducible code.

The Chapter 20 lab script provides a lightweight scaffold for this process.

58.5.1 What the Chapter 20 lab script does

The module `scripts.psych_ch20_responsible_researcher` includes three main components:

1. **Power planning helper**

- Uses `pingouin.power_ttest()` to compute required per-group sample sizes for different effect sizes and power levels.
- Writes a small CSV grid, `outputs/track_b/ch20_power_grid.csv`, that you can inspect or modify as you plan your own study.

2. **Toy meta-analysis simulator**

- Simulates several “studies” with varying sample sizes and effect sizes.
- Computes a **fixed-effect pooled effect**, confidence interval, and basic heterogeneity statistics Q and I^2 .
- Saves both the per-study table and a one-row summary to:
 - `outputs/track_b/ch20_meta_studies.csv`
 - `outputs/track_b/ch20_meta_summary.csv`

This is **not** meant to replace real meta-analysis software, but to demystify the core ideas using familiar PyStatsV1 tools.

3. **Final project report template**

- Creates a Markdown template at `outputs/track_b/ch20_final_project_template.md`.
- The template contains section headings and bullet prompts for:
 - Introduction & research questions
 - Methods (design, participants, measures, procedure)
 - Results (with placeholders for tables and figures generated by your PyStatsV1 scripts)
 - Discussion (including limitations and future directions)

- Reproducibility notes (Git commit hash, random seeds, and CLI commands used)

You can open this file in any text editor or import it into a reference manager / writing tool.

58.5.2 Suggested project workflow

Here is a possible end-to-end workflow for your final project:

1. Pick a question

- Example: “Does a brief mindfulness exercise reduce stress scores relative to a control condition?”
- Example: “Is there an association between sleep quality and exam performance?”

2. Choose a dataset

- Start with one of the PyStatsV1 synthetic datasets (e.g., the sleep study or exam performance data), or
- Import a small real dataset of your own – but keep it simple enough to analyze reproducibly in a single notebook or script.

3. Plan your analysis

- Identify the appropriate model (t-test, ANOVA, mixed-model, regression, non-parametric alternative, etc.).
- Use the power helper in Chapter 20 to think about how many participants would be needed to replicate or extend your findings.

4. Run the analysis with PyStatsV1 tools

- Reuse simulation and analysis helpers from earlier chapters.
- Save any intermediate tables or figures in the `consistent data` and `outputs` directories.

5. Write the report using the template

- Copy `ch20_final_project_template.md` to a new location (or new filename) and gradually fill in each section.
- Whenever you report a result, note which script or function produced it.

6. Record reproducibility details

- Save your final notebook or script under version control.
- Record the Git commit hash and any command-line calls (e.g., `make psych-ch16`) that reproduce your figures and tables.
- Share both the **code** and **narrative** with collaborators or instructors.

58.6 Running the Chapter 20 lab

To run the Chapter 20 lab script from the project root:

```
make psych-ch20
```

This target runs:

```
python -m scripts.psych_ch20_responsible_researcher
```

To run only the tests for this chapter:

```
make test-psych-ch20
```

which wraps:

```
pytest tests/test_psych_ch20_responsible_researcher.py
```

58.7 Conceptual summary

- Responsible research begins **before** data collection with thoughtful design and power analysis.
- Meta-analysis helps synthesize evidence across studies, revealing both typical effects and meaningful differences across contexts.
- Clear, honest communication – especially around uncertainty and limitations – is as important as any statistical computation.
- The PyStatsV1 ecosystem encourages you to treat your analyses like **production software**:
 - Deterministic, version-controlled, and reproducible.
 - Easy to rerun, extend, and audit.
 - Ready to support cumulative, collaborative science.

As you move on to more advanced courses or independent research, you can treat this mini-book (and its code) as a **launch pad**. The goal is not to memorize every formula, but to internalize a way of working:

Don't just calculate your results — engineer them. We treat statistical analysis like production software.
— PyStatsV1 Motto

TRACK C – PROBLEM SETS & WORKED SOLUTIONS

Track C is where you make PyStatsV1 *work for you*.

Track A shows how to port a classical applied statistics text into Python. Track B tells a story for psychology majors – from scientific inquiry to advanced designs – with one worked PyStatsV1 lab per chapter.

Track C adds:

- **Problem sets** tightly linked to the Track B chapters.
- **Worked, tested solutions** implemented as plain Python scripts.
- **Executable, reproducible workflows** that you can copy, modify, and extend for your own research projects.

59.1 How to use Track C

Each problem-set chapter in Track C has three parts:

1. A narrative `.rst` page describing the problems, just like a textbook or assignment sheet.
2. A solution script in `scripts` (for example, `scripts.psych_ch10_problem_set`) that shows one clean way to solve the problems using PyStatsV1 and standard scientific Python tools.
3. A small `pytest` file in `tests` that locks in the numerical answers and guarantees that the solution script keeps working as the project evolves.

59.2 Typical workflow

For a chapter `k` in the Psychology track you can usually do:

```
# Run the worked solutions for the problem set
make psych-ch10-problems

# Run only the tests for this problem set
make test-psych-ch10-problems
```

This will:

- Run the corresponding solution script (for example, `python -m scripts.psych_ch10_problem_set`).
- Print human-readable summaries to the console.
- Save any simulated data, results tables, and plots into the standard `data/` and `outputs/` folders.
- Run the tests that verify the key numbers (t statistics, p -values, effect sizes, and so on).

59.3 Philosophy

Track C is designed with the PyStatsV1 motto in mind:

“Don’t just calculate your results – engineer them. We treat statistical analysis like production software.”

That means:

- Every solution script is **deterministic** (fixed random seeds), **version-controlled**, and **tested**.
- You are encouraged to **fork, copy, and adapt** the solution scripts to your own datasets.
- Instructors can assign the problem sets as-is, or use the scripts and tests as templates for their own assignments.

59.4 Available problem sets

Currently implemented:

- *Chapter 10 Problem Set – Independent-Samples t Test* – Independent-samples t -test problem set.

More chapters will be added in future versions, following the same pattern:

- docs/psych_chk_problem_set.rst
- scripts/psych_chk_problem_set.py
- tests/test_psych_chk_problem_set.py
- Makefile targets: psych-chk-problems and test-psych-chk-problems.

CHAPTER 10 PROBLEM SET – INDEPENDENT-SAMPLES T TEST

60.1 Where this problem set fits in the story

This problem set extends *Psychological Science & Statistics – Chapter 10*, which introduced between-subjects designs and the independent-samples t test.

In Track B you learned how to:

- Design a basic treatment-versus-control experiment.
- Compute and interpret an independent-samples t test.
- Report results in APA style, including effect size (Cohen's d).

In **Track C** you will:

- Work through several realistic research scenarios.
- See how each analysis is implemented as reproducible Python code.
- Use the provided solution script as a template for your own studies.

60.2 Learning goals

After completing this problem set, you should be able to:

- Choose an appropriate independent-samples t test for a simple between-subjects design.
- Run the test in Python (using PyStatsV1 + Pingouin) and interpret the results.
- Extract and report the key quantities: group means, mean difference, t , degrees of freedom, p , and Cohen's d .
- Understand how sample size and effect size jointly influence statistical significance.

60.3 How to run the worked solutions

From the repository root, run:

```
# Run the solution script for this problem set
make psych-ch10-problems

# Run only the tests for this problem set
make test-psych-ch10-problems
```

These targets simply wrap:

```
python -m scripts.psych_ch10_problem_set
pytest tests/test_psych_ch10_problem_set.py
```

The solution script will:

- Simulate data for each exercise with a fixed random seed.
- Run the independent-samples t tests.
- Print concise summaries to the console.
- Save data and results tables under:
 - data/synthetic/psych_ch10_*.csv
 - outputs/track_c/ch10_problem_set_results.csv
 - outputs/track_c/ch10_problem_set_group_means.png

60.4 Conceptual warm-up

Before touching the code, think through these questions:

1. In an independent-samples design, why do we care about **homogeneity of variance** between groups?
2. How does increasing sample size affect the standard error of the mean difference and the resulting t statistic?
3. Why might a result be **statistically significant but not practically important**?
4. Give an example of a research question in psychology that is naturally answered with an independent-samples t test.

60.5 Applied exercises

60.5.1 Exercise 1 – Stress-reduction workshop vs. waitlist

A counseling center wants to evaluate a **stress-reduction workshop**. Students are randomly assigned to:

- **control** – waitlist (no workshop yet).
- **treatment** – immediate participation in the workshop.

After two weeks, all students complete the same **stress scale** (higher scores = **more** stress). The question:

“Does the workshop reduce average stress relative to the waitlist?”

Tasks:

1. State appropriate H_0 and H_1 .
2. Run an independent-samples t test.
3. Report:
 - Group means and mean difference
 - t , df , p
 - Cohen’s d

In code, the worked solution uses:

- `scripts.psych_ch10_problem_set.simulate_independent_t_dataset()` to generate the data, and
- `scripts.psych_ch10_problem_set.run_independent_t()` to compute the t test.

You can inspect and adapt that code to analyze your own treatment-versus-control experiments.

60.5.2 Exercise 2 – The curse of small n

Now imagine the same workshop, but with a **much smaller sample size**. The effect of the workshop on stress is similar in magnitude to Exercise 1, but because n is smaller, the test has **lower power**.

Tasks:

1. Use the provided code to simulate a smaller sample and run the independent-samples t test.
2. Compare the resulting t , p , and Cohen's d with Exercise 1.
3. Explain in words how the **same underlying effect** can fail to reach significance when you have too few participants.

In code, see:

- `scripts.psych_ch10_problem_set.exercise_1_large_sample()`
- `scripts.psych_ch10_problem_set.exercise_2_small_sample()`

60.5.3 Exercise 3 – Strong treatment effect

Finally, consider a scenario where the treatment has a **very strong effect** on the outcome. This might correspond to a highly effective clinical intervention or an artificially “clean” lab manipulation.

Tasks:

1. Simulate a dataset with a large effect size and moderate sample size.
2. Run the independent-samples t test.
3. Examine:
 - The size of t and how close p is to zero.
 - The magnitude of Cohen's d .

In code, see:

- `scripts.psych_ch10_problem_set.exercise_3_large_effect()`.

60.6 Running the Chapter 10 problem set lab

To re-run all exercises and regenerate the outputs for this problem set:

```
make psych-ch10-problems
```

Then inspect:

- `data/synthetic/psych_ch10_exercise1.csv` – Stress reduction, large sample
- `data/synthetic/psych_ch10_exercise2.csv` – Stress reduction, small sample
- `data/synthetic/psych_ch10_exercise3.csv` – Strong effect scenario
- `outputs/track_c/ch10_problem_set_results.csv` – Summary table of the three t tests.
- `outputs/track_c/ch10_problem_set_group_means.png` – Group means plot.

60.7 Conceptual summary

- Independent-samples t tests compare **mean differences** between two unrelated groups.
- **Effect size** and **sample size** jointly determine whether an effect is likely to be detected as statistically significant.
- PyStatsV1 solution scripts give you a **reusable template**: swap in your own dataset, re-run the analysis, and verify the results using the tests.

CHAPTER 11 PROBLEM SET – PAIRED-SAMPLES T TEST

61.1 Where this problem set fits in the story

This problem set extends the `psych_ch11_paired_t` chapter on **within-subjects designs** and the paired-samples t test.

Chapter 11 introduces designs where each participant serves as their own control (for example, pre–post designs). Track C adds a set of worked, fully reproducible examples that show how to:

- Simulate pre–post data for different research scenarios.
- Run paired-samples t tests using the PyStatsV1 helpers.
- Interpret the resulting means, t values, p values, and effect sizes.

61.2 Learning goals

By the end of this problem set, you should be able to:

- Recognize when a **paired-samples t test** is appropriate.
- Explain how the test is based on **difference scores** (post – pre).
- Describe how **sample size** and **effect size** jointly determine power.
- Use the PyStatsV1 solution code as a **template** for your own pre–post data.

61.3 How to run the worked solutions

From the project root, run:

```
make psych-ch11-problems
```

This wraps:

```
python -m scripts.psych_ch11_problem_set
```

and regenerates all synthetic datasets, the summary CSV, and the group means plot.

61.4 Conceptual warm-up

- In a paired design, each participant is measured **twice** (or more), so we compare them to themselves.
- This reduces error by controlling for stable individual differences.
- The paired-samples t test is equivalent to running a one-sample t test on the **difference scores** (post – pre).

- Effect sizes (Cohen's d) are typically calculated using the variability of the difference scores.

61.5 Applied exercises

Each exercise in this problem set corresponds to a realistic research scenario:

- **Exercise 1 – Moderate improvement ($n = 40$)**

A typical lab-based intervention with a medium-sized effect. The paired t test should be clearly significant.

- **Exercise 2 – Small / ambiguous effect ($n = 30$)**

A small effect with modest sample size. The t test will often be non-significant, highlighting how hard it is to detect small effects without sufficient power.

- **Exercise 3 – Strong improvement ($n = 25$)**

A large effect with a smaller sample. Despite the lower n , the effect is strong enough to be detected with high confidence.

61.6 PyStatsV1 Lab: Paired-samples t problem set in action

The solution script `scripts.psych_ch11_problem_set` shows how to:

- Generate pre–post data for each scenario (exercise label, group means, effect size, etc.).
- Run the paired t tests using `scripts.psych_ch11_paired_t.run_paired_t()`.
- Save one CSV per exercise plus a **summary CSV** with the key statistics side-by-side.
- Produce a simple bar plot comparing pre and post means for each exercise.

61.7 Running the Chapter 11 problem set lab

After running:

```
make psych-ch11-problems
```

you should see the following outputs:

- `data/synthetic/psych_ch11_exercise1.csv` – Moderate-effect pre–post study ($n = 40$).
- `data/synthetic/psych_ch11_exercise2.csv` – Small-effect pre–post study ($n = 30$).
- `data/synthetic/psych_ch11_exercise3.csv` – Strong-effect pre–post study ($n = 25$).
- `outputs/track_c/ch11_problem_set_results.csv` – Summary table of the three paired t tests.
- `outputs/track_c/ch11_problem_set_means.png` – Group means plot (pre vs post for each exercise).

61.8 Conceptual summary

- Paired-samples t tests compare **mean differences within participants** rather than between independent groups.
- They are often **more powerful** than independent-samples designs, because each person serves as their own control.
- Power depends on the magnitude of the true effect, the variability of the difference scores, and the sample size.
- PyStatsV1 solution scripts give you a **reusable pre–post template**: drop in your own dataset, rerun the analysis, and verify the results using transparent, version-controlled code.

CHAPTER 12 PROBLEM SET – ONE-WAY ANOVA

62.1 Where this problem set fits in the story

This Track C problem set is linked to **Chapter 12: One-Way ANOVA** in the Psychological Science & Statistics mini-book (Track B).

- Track B Chapter 12 introduces the *logic* of one-way ANOVA: partitioning variance, interpreting the F-ratio, and understanding how post-hoc tests localize differences between means.
- Track B Chapter 10 problem set showed how to work with *two groups* using independent-samples t-tests.
- This Track C Chapter 12 problem set extends that workflow to *three or more groups* using one-way ANOVA.

The PyStatsV1 scripts here turn the conceptual ideas from Chapter 12 into a *reusable analysis template* that you can adapt for your own studies.

62.2 Learning goals

By working through this problem set and inspecting the solution scripts, you should be able to:

- Recognize when a one-way ANOVA is appropriate (one categorical predictor, three or more groups, continuous outcome).
- Simulate and analyze between-subjects designs with multiple levels of an independent variable (e.g., **low**, **medium**, **high** intervention).
- Interpret the ANOVA table: sums of squares, degrees of freedom, mean squares, F-statistic, and p-value.
- Inspect group means and effect sizes to distinguish statistical significance from practical significance.
- Use PyStatsV1's solution scripts as a template for running your *own* one-way ANOVAs.

62.3 How to run the worked solutions

From the project root (with your virtual environment activated), the Chapter 12 problem-set lab can be re-run with:

```
make psych-ch12-problems
```

Behind the scenes this target calls:

```
python -m scripts.psych_ch12_problem_set
```

This will:

- Simulate three different one-way ANOVA scenarios.

- Run ANOVA for each scenario.
- Save the simulated datasets and a summary CSV of the results.
- Produce a simple group-means plot for quick visual comparison.

62.4 Conceptual warm-up

Before looking at the code, think through these questions:

- Why do we prefer ANOVA instead of running multiple t-tests when comparing three or more groups?
- What does it mean, conceptually, to “partition” total variability into between-groups and within-groups components?
- How does increasing the number of groups (but holding sample size fixed) affect the F-ratio and statistical power?
- How would you explain to a collaborator the difference between a statistically significant F and a large, practically meaningful effect?

62.5 Applied exercises

The script `scripts.psych_ch12_problem_set` contains three exercises that mirror common classroom and research scenarios.

62.5.1 Exercise 1 – Classic three-group ANOVA (moderate effect)

- Design: Three independent groups (`control`, `low_dose`, `high_dose`), equal sample sizes.
- Outcome: Continuous score (e.g., stress reduction, performance, symptom change).
- Effect: Moderate treatment effect – the means are separated enough that the omnibus F-test is usually significant.

Questions to consider:

- What does the ANOVA table tell you about the overall effect of Group?
- How large is the effect (e.g., partial eta-squared)?
- Which pairwise differences would you expect to be significant in a post-hoc analysis?

62.5.2 Exercise 2 – Small effect, borderline significance

- Design: Same three groups as Exercise 1, but with smaller differences between group means.
- Effect: Small effect size – the F-test may be non-significant or only weakly significant depending on the simulated sample.

Questions to consider:

- How do F and p change compared to Exercise 1?
- Does the conclusion change if you focus on effect size rather than $p < .05$?
- How would you report these results honestly in an APA-style write-up?

62.5.3 Exercise 3 – Unequal n and strong effect

- Design: Three groups with unequal sample sizes (e.g., more participants in the control group than in one of the treatment groups).
- Effect: Strong effect size – the F-test is clearly significant, and the pattern of means is obvious in the plot.

Questions to consider:

- Does unequal n change your interpretation of the ANOVA table?
- How might unequal n arise in a real study (dropout, recruitment issues)?
- What would you recommend to a collaborator planning a follow-up study?

62.6 PyStatsV1 Lab: One-way ANOVA solution scripts

The solution code for these exercises lives in:

```
scripts/psych_ch12_problem_set.py
```

Key pieces to inspect in that module:

- **Data generation helpers** that simulate one-way ANOVA designs with tunable effect sizes and sample sizes.
- A `run_one_way_anova` function that wraps *pingouin.anova* to produce a tidy ANOVA table suitable for teaching.
- A small `ProblemResult` dataclass that bundles the simulated data, ANOVA table, and metadata (exercise label, group means, effect size, etc.).
- A main routine that writes all datasets and a **summary CSV** so you can see the three scenarios side-by-side.

62.7 Running the Chapter 12 problem set lab

To re-run all exercises and regenerate the outputs for this problem set, use:

```
make psych-ch12-problems
```

Then inspect:

- `data/synthetic/psych_ch12_exercise1.csv` – Three-group design with a moderate treatment effect.
- `data/synthetic/psych_ch12_exercise2.csv` – Three-group design with a small effect.
- `data/synthetic/psych_ch12_exercise3.csv` – Three-group design with unequal n and a strong effect.
- `outputs/track_c/ch12_problem_set_results.csv` – Summary ANOVA table (one row per exercise).
- `outputs/track_c/ch12_problem_set_group_means.png` – Group means plot for the three exercises.

62.8 Conceptual summary

- One-way ANOVA compares **mean differences across three or more independent groups** using a single F-test.
- The F-ratio expresses the *signal-to-noise* logic of variance partitioning: how much variability is explained by group membership relative to residual variability within groups.
- Sample size, group spacing (effect size), and error variance jointly determine whether an effect is likely to be detected as statistically significant.
- PyStatsV1 solution scripts give you a **reusable one-way ANOVA template**: swap in your own dataset, re-run the analysis, and verify the results using the tests.