Probability & Statistics Fundamentals

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Probability rules, conditional probability & Bayes' Theorem

Random variables & expected value · binomial, Poisson & normal distributions

The Central Limit Theorem · descriptive statistics

Capstone: monitoring a real feature rollout, start to finish

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Probability & Statistics Matter for Programmers

Probability & Statistics Fundamentals

Chapter 1 · Why Probability & Statistics Matter for Programmers

Every system you build eventually runs into something it can't fully control — a request that might fail, a user who might click, a server that might go down at an inconvenient hour. Probability and statistics are the two branches of math built specifically for reasoning carefully about exactly that kind of uncertainty, rather than guessing or hoping it averages out.

Probability vs. Statistics — Two Directions of Reasoning

These two names get used almost interchangeably in casual conversation, but they answer genuinely opposite questions.

	PROBABILITY	STATISTICS
Direction	Model → predicted outcomes	Observed data → inferred model
Starting point	You already know the rules (a fair coin, a known defect rate)	You only have data, and need to work out what's actually going on
Example question	"Given a fair die, what's the chance of rolling a 6?"	"Given 10,000 actual page loads, what's the real average response time?"
Core tools	Probability rules, distributions (Ch.2–8 of this course)	Descriptive statistics (Ch.9), and inferential methods — this subject's own separate future course

Both matter, and in practice they're used together constantly — you build a probability *model* of how something should behave, then use statistics to check whether real, observed data actually matches it. This particular course focuses on the probability side and the basic descriptive tools for summarizing data; the reasoning-from-data-back-to-conclusions side (confidence intervals, hypothesis testing, A/B testing, regression) is deliberately this subject's own separate next course.

Five Concrete Connections to Code Already On This Site

Every topic in this course maps onto something already relevant to real engineering work, whether or not the underlying math was ever named:

PROBABILITY/STATISTICS TOPIC	WHERE IT ACTUALLY SHOWS UP
Probability rules & conditional probability (Ch.2–3)	Retry logic and compounding failure rates, cache-hit probability, error-rate budgets
Bayes' Theorem (Ch.4)	Spam filters and fraud-detection systems that combine several weak, individually unreliable signals into one updated probability
Random variables & expected value (Ch.5)	Estimating expected cost, expected load, or expected risk before a decision is made — not just the single most likely outcome
Distributions — binomial, Poisson, normal (Ch.6–8)	Modelling conversion rates (binomial), server incident/arrival rates (Poisson — directly relevant to Technical Support's own `perfdiag1` / `incident1` material), and noisy measurement error (normal)
Descriptive statistics (Ch.9)	Every monitoring dashboard's own averages, and the well-known trap (already covered from the other direction in `perfdiag1`'s own material) that an average can quietly hide a real spike

What This Course Won't Cover

A substantial, genuinely related set of topics is deliberately left for this subject's own next course, **Statistical Inference & Applied Statistics**, rather than folded in here:

- **Sampling & confidence intervals** — how to reason honestly about uncertainty in an estimate drawn from a sample rather than an entire population
- **Hypothesis testing & A/B testing** — the formal machinery for deciding whether an observed difference is real or just noise
- **Correlation, regression & Bayesian updating** — inferring relationships between variables from real data, and formally updating a belief as new evidence arrives

WHY DRAW THE LINE HERE INSTEAD OF COVERING EVERYTHING AT ONCE

This course builds the probability vocabulary and distributions that every one of those inferential techniques is built on top of — trying to cover hypothesis testing before random variables and distributions exist yet would mean constantly stopping mid-explanation to backfill missing foundations. This course is that foundation; the next course is where it gets put to direct, data-driven use.

Where This Course Is Headed

CHAPTER	TOPIC
2	Sample Spaces, Events & Basic Probability Rules
3	Conditional Probability & Independence

CHAPTER	TOPIC
4	Bayes' Theorem
5	Random Variables & Expected Value
6	The Binomial Distribution
7	The Poisson Distribution
8	The Normal Distribution & the Central Limit Theorem
9	Descriptive Statistics — Mean, Median, Variance & Standard Deviation
10	Capstone — Probability & Statistics in Practice

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: given a clearly stated model of how something behaves, what can be said *precisely* about the range of outcomes it could produce? That's a genuinely different skill from reading data after the fact — it's reasoning forward, before the data even exists, which is exactly what's needed to set a reasonable alert threshold, size a system for expected load, or judge whether an outcome you just observed was actually surprising at all.

Hands-On Exercises

EXERCISE 1

Classify each of the following as fundamentally a probability question (model → predicted outcome) or a statistics question (data → inferred conclusion), and briefly justify each answer: (a) given a fair six-sided die, what's the chance of rolling a 6? (b) given 1,000 recorded server response times, what's their average? (c) given a known 2% manufacturing defect rate, how many defective units are expected in a batch of 500? (d) given last month's actual incident counts, estimate the true underlying incident rate.

View solution

EXERCISE 2

A colleague claims "probability and statistics are only really relevant if you're doing data science or machine learning." Using this chapter's own five connections, explain at least two places probability/statistics reasoning shows up in code that has nothing to do with ML.

View solution

EXERCISE 3

For each of the following real system-design scenarios, name which topic from this chapter's own five-connections table it most directly maps to, and explain the connection in one or two sentences: (a) a retry mechanism where each attempt independently succeeds 90% of the time, and you want to know the chance all three attempts fail; (b) a fraud-detection system combining an unusual login location, an unusual purchase amount, and a new device into one overall risk score; (c) a dashboard graph showing "average response time" over the last hour.

View solution

CHAPTER 1 QUICK REFERENCE

- **Probability** reasons from a known model forward to predicted outcomes; **statistics** reasons from observed data backward to an inferred model
- Five direct connections: probability rules → retries/error budgets, Bayes' Theorem → spam/fraud filters, expected value → risk/cost estimation, distributions → conversion rates & incident rates & measurement noise, descriptive stats → dashboard averages
- Deliberately out of scope here: sampling, confidence intervals, hypothesis testing, A/B testing, regression, and Bayesian updating — all reserved for this subject's own next course, Statistical Inference & Applied Statistics
- This course builds the probability vocabulary and distributions every inferential technique in that next course is built on top of
- **Next chapter:** Sample spaces, events, and the basic rules of probability

CHAPTER 2 OF 10

Sample Spaces, Events & Basic Probability Rules

Probability & Statistics Fundamentals

Chapter 2 · Sample Spaces, Events & Basic Probability Rules

Every probability question starts by pinning down two things precisely: every outcome that could possibly happen, and the specific outcomes you actually care about. Get those two definitions right, and most basic probability calculations turn into simple counting — genuinely just set operations wearing a different name.

Sample Space & Events

The **sample space** (usually written `S`) is the set of every possible outcome of an experiment. An **event** is any subset of that sample space — the specific outcomes that count as a "success" for whatever question is being asked.

WORKED EXAMPLE: ROLLING A SINGLE SIX-SIDED DIE

`S = {1, 2, 3, 4, 5, 6}`

Event A ("rolling an even number") = `{2, 4, 6}`

Event B ("rolling a number greater than 4") = `{5, 6}`

THIS IS EXACTLY SET THEORY, APPLIED

A sample space is a set; an event is a subset. "A and B" is the intersection `A ∩ B`; "A or B" is the union `A ∪ B`; "not A" is the complement. Every probability rule in this chapter is really a Discrete Mathematics Fundamentals Chapter 4 set operation, counted and divided.

Computing Probability for Equally Likely Outcomes

When every outcome in the sample space is equally likely (a fair die, a fair coin), probability is a straightforward ratio:

THE EQUALLY-LIKELY-OUTCOMES FORMULA

```
P(event) = |event| / |sample space|
```

For event A above: `P(A) = |{2,4,6}| / |{1,...,6}| = 3/6 = 0.5`. For event B: `P(B) = |{5,6}| / 6 = 2/6 = 1/3`.

> **THIS FORMULA ONLY WORKS BECAUSE THE OUTCOMES ARE EQUALLY LIKELY**
>
> A biased die, a loaded coin, or almost any real-world event (a server failing, a user clicking) does *not* have equally likely outcomes — this simple counting formula breaks down immediately. It's a useful starting tool, not a general-purpose one; later chapters build the machinery (random variables, distributions) that handles the general case properly.

The Probability Axioms

Every valid probability assignment must satisfy three basic rules, regardless of how the probabilities were derived:

AXIOM	MEANING
Non-negativity	`0 ≤ P(event) ≤ 1` for any event
Certainty	`P(S) = 1` — something in the sample space is guaranteed to happen
Impossibility	`P(∅) = 0` — the empty event (no outcomes) never happens

The Complement Rule

The **complement** of an event, written `Aᶜ` ("not A"), is every outcome *not* in `A`. Since `A` and `Aᶜ` together always cover the whole sample space exactly once:

> **COMPLEMENT RULE**
>
> `P(Aᶜ) = 1 − P(A)`

The probability of *not* rolling a 6: `P(not 6) = 1 − 1/6 = 5/6`. Often faster than counting the complement's outcomes directly, especially when "not A" covers far more cases than "A" does.

The Union (Addition) Rule

For "A or B," the naive instinct is to add `P(A) + P(B)` — but that **double-counts** any outcome that belongs to both events. The correct rule subtracts the overlap back out:

> **UNION RULE**

`P(A ∪ B) = P(A) + P(B) − P(A ∩ B)`

Continuing the dice example: `A ∩ B = {6}` (even *and* greater than 4), so `P(A ∩ B) = 1/6`.

QUANTITY	VALUE
P(A) + P(B)	0.5 + 0.333 = 0.833 (5/6) — **wrong**, double-counts 6
P(A) + P(B) − P(A ∩ B)	0.5 + 0.333 − 0.167 = 0.667 (2/3) — **correct**
Direct check: |A ∪ B| / |S|	{2,4,5,6} → 4/6 = 2/3 ✓ matches

THE DOUBLE-COUNTING MISTAKE, MADE CONCRETE

Naively adding `P(A) + P(B)` counts outcome 6 twice — once as part of "even," once as part of "greater than 4" — inflating the true answer. Whenever two events can genuinely overlap, skipping the `− P(A ∩ B)` term silently overstates the real probability.

Mutually Exclusive Events — When the Simple Sum *Is* Correct

If two events can *never* both happen at once (`A ∩ B = ∅`), they're called **mutually exclusive** — and the subtraction term simply vanishes, since `P(∅) = 0`:

UNION RULE FOR MUTUALLY EXCLUSIVE EVENTS

`P(A ∪ B) = P(A) + P(B)` — only valid when `A ∩ B = ∅`

Rolling a 1 (`C = {1}`) and rolling a 6 (`D = {6}`) can never both happen on the same roll: `P(C ∪ D) = 1/6 + 1/6 = 1/3` — directly correct here, with no overlap to subtract.

Sample Spaces & Events in Code

```
S = {1, 2, 3, 4, 5, 6}
A = {2, 4, 6}   # even
B = {5, 6}      # greater than 4

def prob(event, sample_space):
    return len(event) / len(sample_space)

P_A = prob(A, S)
P_B = prob(B, S)
P_A_and_B = prob(A & B, S)   # set intersection
P_A_or_B = prob(A | B, S)    # set union

# the union rule, verified directly against the real set union above
assert abs((P_A + P_B - P_A_and_B) - P_A_or_B) < 1e-9
print(P_A, P_B, P_A_and_B, P_A_or_B)  # 0.5 0.333... 0.1666... 0.6666...
```

Hands-On Exercises

EXERCISE 1

A standard 52-card deck. Event A = "drawing a heart" (13 cards). Event B = "drawing a face card" (Jack, Queen, or King — 12 cards total across all suits). Compute `P(A)`, `P(B)`, and `P(A ∩ B)` (hearts that are also face cards), then use the union rule to find `P(A ∪ B)`. Confirm your answer by directly counting how many of the 52 cards satisfy "heart or face card."

View solution

EXERCISE 2

Out of 500 requests handled by a service last week, 15 returned an error. Using the complement rule, compute the probability that a randomly selected request from that week did *not* return an error.

View solution

EXERCISE 3

Rolling a single die: event E = "rolling an odd number" (`{1,3,5}`), event F = "rolling a number less than 4" (`{1,2,3}`). Compute `P(E)`, `P(F)`, and `P(E ∩ F)`, then use the union rule to find `P(E ∪ F)`. Separately, compute the naive (incorrect) sum `P(E) + P(F)` and explain specifically which outcome(s) it double-counts.

View solution

CHAPTER 2 QUICK REFERENCE

- **Sample space** = every possible outcome; **event** = a subset of the sample space
- Equally-likely-outcomes formula: `P(event) = |event| / |sample space|` — only valid when every outcome is equally likely
- **Axioms:** `0 ≤ P(event) ≤ 1`, `P(S) = 1`, `P(∅) = 0`
- **Complement rule:** `P(Aᶜ) = 1 − P(A)`
- **Union rule:** `P(A ∪ B) = P(A) + P(B) − P(A ∩ B)` — the subtraction fixes double-counting the overlap
- **Mutually exclusive** events (`A ∩ B = ∅`) simplify to `P(A ∪ B) = P(A) + P(B)`, with nothing to subtract
- **Next chapter:** Conditional probability and independence

CHAPTER 3 OF 10

Conditional Probability & Independence

Probability & Statistics Fundamentals

Chapter 3 · Conditional Probability & Independence

Chapter 2's rules all assumed no extra information. Real questions are rarely that clean — "what's the chance this request fails, *given* that the last three also failed?" This chapter is about updating a probability once new information narrows down what's actually possible.

Conditional Probability — Narrowing the Sample Space

Conditional probability, written `P(A|B)` ("the probability of A, given B"), asks: once we know `B` has happened, what fraction of *that* narrowed-down world does `A` still cover?

CONDITIONAL PROBABILITY FORMULA

`P(A|B) = P(A ∩ B) / P(B)`

Reusing Chapter 2's dice example: `S = {1,...,6}`, `A` = "even" = `{2,4,6}`, `B` = "greater than 4" = `{5,6}`, `A ∩ B = {6}`.

QUANTITY	CALCULATION	RESULT
P(A|B)	(1/6) / (1/3)	1/2 — given the roll is 5 or 6, half the time it's also even
P(B|A)	(1/6) / (1/2)	1/3 — given the roll is even, only 1 in 3 times is it also >4

P(A|B) ≠ P(B|A), IN GENERAL

These two numbers answer genuinely different questions and there's no reason to expect them to match — here they don't (1/2 vs 1/3). Confusing the two is a classic, real mistake (sometimes called "the prosecutor's fallacy" in legal/forensic contexts): "the probability of this evidence given innocence" is not the same number as "the probability of innocence given this evidence." Chapter 4's Bayes' Theorem exists specifically to convert correctly between the two.

Independence

Two events are **independent** if knowing one happened tells you nothing new about the other: `P(A|B) = P(A)`. Rearranging the conditional probability formula gives an equivalent, more practical test:

THE INDEPENDENCE TEST / MULTIPLICATION RULE

`A and B are independent ⟺ P(A ∩ B) = P(A) × P(B)`

A GENUINELY NON-OBVIOUS RESULT, CHECKED DIRECTLY

Checking Chapter 2's own A and B: `P(A) × P(B) = 0.5 × (1/3) = 1/6`, which exactly equals `P(A ∩ B) = 1/6`. "Even" and "greater than 4" are, on a fair die, actually **independent** — a fact that isn't obvious just by looking at the two events, and only confirmed by checking the numbers directly. Never assume independence from intuition alone; always check.

For contrast, reusing Chapter 2's own exercise events `E` = "odd" = `{1,3,5}` and `F` = "less than 4" = `{1,2,3}`: `P(E) × P(F) = 0.5 × 0.5 = 0.25`, but `P(E ∩ F) = P({1,3}) = 1/3 ≈ 0.333`. Since `0.25 ≠ 0.333`, `E` and `F` are **dependent** — knowing a roll is odd genuinely does change the probability it's also less than 4.

The Multiplication Rule for Independent Events, in Practice

When events genuinely are independent, the multiplication rule becomes a fast, direct tool — and it generalizes cleanly to more than two events, which is exactly what reliability calculations need.

WORKED EXAMPLE: THREE INDEPENDENT RETRY ATTEMPTS

An operation is retried up to 3 times. Each attempt independently succeeds 90% of the time (a 10% failure rate). What's the probability *all three* attempts fail?

QUANTITY	CALCULATION	RESULT
P(all 3 fail)	0.1 × 0.1 × 0.1	0.001 (0.1%)
P(at least one succeeds)	1 − 0.001 (complement rule, Ch.2)	0.999 (99.9%)

This is the exact combination of two rules from this course so far: the multiplication rule for independent events, and Chapter 2's own complement rule, chained together to answer a genuinely practical reliability question.

The General Multiplication Rule — A Bayes' Theorem Forward Reference

Rearranging the conditional probability formula, without assuming independence, gives the fully general version:

GENERAL MULTIPLICATION RULE

`P(A ∩ B) = P(A|B) × P(B) = P(B|A) × P(A)`

That last equality — two different ways of writing the exact same joint probability — is the entire foundation Chapter 4's Bayes' Theorem is built from. It's what makes it possible to solve for `P(B|A)` when only `P(A|B)` is actually known, which turns out to be an extremely common real situation.

Conditional Probability & Independence in Code

```
S = {1, 2, 3, 4, 5, 6}
A = {2, 4, 6}
B = {5, 6}

def prob(event, sample_space):
    return len(event) / len(sample_space)

P_A = prob(A, S)
P_B = prob(B, S)
P_A_given_B = prob(A & B, S) / P_B

print(P_A_given_B)               # 0.5
print(P_A_given_B == P_A)       # True — independent, per this chapter's own test

# Retry reliability: independent multiplication rule
p_fail = 0.1
p_all_fail = p_fail ** 3
print(1 - p_all_fail)  # 0.999 — probability at least one of 3 attempts succeeds
```

Hands-On Exercises

EXERCISE 1

Rolling a single die: event G = "rolling a number ≤ 3" (`{1,2,3}`), event H = "rolling an even number" (`{2,4,6}`). Compute `P(G|H)` and `P(H|G)`, and determine whether G and H are independent by applying this chapter's own multiplication-rule test.

View solution

EXERCISE 2

An operation is retried up to 4 times, each attempt independently succeeding 80% of the time. Compute the probability all 4 attempts fail, and the probability at least one succeeds.

View solution

EXERCISE 3

In a product analytics dataset: P(user is on mobile) = 0.6, P(user completes checkout) = 0.1, P(user is on mobile AND completes checkout) = 0.03. Compute `P(checkout | mobile)`, compare it to the overall `P(checkout)` to say whether mobile users are more or less likely than average to complete checkout, and determine whether "on mobile" and "completes checkout" are independent events.

View solution

CHAPTER 3 QUICK REFERENCE

- **Conditional probability:** `P(A|B) = P(A ∩ B) / P(B)` — the probability of A, restricted to the world where B already happened
- **P(A|B) ≠ P(B|A)** in general — confusing the two is a classic real mistake
- **Independence:** `P(A|B) = P(A)`, equivalently `P(A ∩ B) = P(A) × P(B)` — always check the numbers, never assume from intuition
- The independent-events multiplication rule generalizes to any number of events — e.g., `P(all n fail) = p^n` for n independent attempts each with failure probability p
- **General multiplication rule:** `P(A ∩ B) = P(A|B)P(B) = P(B|A)P(A)` — the direct foundation of Chapter 4's Bayes' Theorem
- **Next chapter:** Bayes' Theorem

CHAPTER 4 OF 10

Bayes' Theorem

Probability & Statistics Fundamentals

Chapter 4 · Bayes' Theorem

Chapter 3 ended with a promise: a way to solve for `P(B|A)` when only `P(A|B)` is actually known. That tool is **Bayes' Theorem** — and it delivers one of the most consistently surprising, genuinely useful results in this entire course.

Deriving Bayes' Theorem

Chapter 3's general multiplication rule gave two equal expressions for the same joint probability:

STARTING POINT (CHAPTER 3)

`P(A ∩ B) = P(A|B) × P(B) = P(B|A) × P(A)`

Dropping the middle term and dividing both sides by `P(A)` isolates `P(B|A)` directly:

BAYES' THEOREM

`P(B|A) = [P(A|B) × P(B)] / P(A)`

Written in the notation it's usually taught with — `H` for hypothesis, `E` for evidence:

BAYES' THEOREM, STANDARD FORM

`P(H|E) = [P(E|H) × P(H)] / P(E)`

Each term has a name worth knowing: `P(H)` is the **prior** (what you believed before seeing the evidence), `P(E|H)` is the **likelihood**, `P(E)` is the evidence's overall probability, and `P(H|E)` is the **posterior** — the updated belief after accounting for the evidence.

The Law of Total Probability — Filling in the Denominator

`P(E)` is rarely given directly — it needs to be built from the two ways evidence can occur: when the hypothesis is true, and when it's false.

LAW OF TOTAL PROBABILITY

`P(E) = P(E|H) × P(H) + P(E|not H) × P(not H)`

The Classic Example: A Diagnostic Test

A test for a rare disease is genuinely quite accurate:

QUANTITY	VALUE
P(D) — disease prevalence	1% (a rare disease)
P(+|D) — sensitivity	99% — correctly flags 99% of people who have it
P(+|not D) — false positive rate	5% — incorrectly flags 5% of healthy people

Question: given a positive result, what's the actual probability of having the disease?

STEP	CALCULATION	RESULT
P(+), total probability	0.99×0.01 + 0.05×0.99	0.0594
P(D|+), Bayes' Theorem	(0.99×0.01) / 0.0594	0.1667 (≈16.7%)

A 99%-ACCURATE TEST, AND ONLY A 16.7% CHANCE OF ACTUALLY HAVING THE DISEASE

This is the correct answer, not a mistake. Because the disease is rare, the sheer number of healthy people who get a false positive (5% of the huge "healthy" group) swamps the smaller number of genuinely sick people who correctly test positive (99% of the tiny "sick" group). This is sometimes called **base-rate neglect** — ignoring how rare something actually is leads to badly overestimating how meaningful a positive result really is.

A Second Worked Example: A Spam Filter

Reusing this course's own forward reference from Chapter 1 — a spam filter reasoning about a single word:

QUANTITY	VALUE
P(spam) — prior	40% of all email is spam
P("free"|spam)	30% of spam emails contain the word "free"
P("free"|not spam)	5% of legitimate emails contain "free"

STEP	CALCULATION	RESULT
P("free"), total probability	0.30×0.40 + 0.05×0.60	0.15
P(spam|"free"), Bayes' Theorem	(0.30×0.40) / 0.15	0.80 (80%)

Seeing "free" in an email raises the belief it's spam from a 40% prior all the way to an 80% posterior — a real, quantified update, exactly the mechanism named in Chapter 1's own connections table.

Why This Matters for Real Alerting Systems

THE SAME MATH APPLIES DIRECTLY TO ANOMALY/INTRUSION DETECTION

An intrusion-detection system with a 99% true-positive rate and a 1% false-positive rate *sounds* excellent. But if genuinely malicious connections are extremely rare (say, 1 in 10,000), the exact same base-rate-neglect math applies: an alert firing is correct only a small fraction of the time — the overwhelming majority of alerts are false positives, purely because malicious events are so rare relative to the flood of ordinary traffic. This is the precise mathematical root of **alert fatigue**, a genuine operational problem this subject's sibling Technical Support courses (`secsupport1`, `incident1`) deal with directly from the response-process side; this chapter explains exactly why it happens numerically.

Bayes' Theorem in Code

```
def bayes(p_evidence_given_h, p_h, p_evidence_given_not_h):
    p_not_h = 1 - p_h
    p_evidence = (p_evidence_given_h * p_h) + (p_evidence_given_not_h * p_not_h)
    return (p_evidence_given_h * p_h) / p_evidence

# The diagnostic test example
p_disease_given_positive = bayes(p_evidence_given_h=0.99, p_h=0.01, p_evidence_given_not_h=0.05)
print(p_disease_given_positive)  # 0.16666... — only ~16.7%

# The spam filter example
p_spam_given_free = bayes(p_evidence_given_h=0.30, p_h=0.40, p_evidence_given_not_h=0.05)
print(p_spam_given_free)  # 0.8 — 80%
```

Hands-On Exercises

EXERCISE 1

A fraud-detection system flags 95% of genuinely fraudulent transactions (`P(flagged|fraud) = 0.95`) and incorrectly flags 2% of legitimate transactions (`P(flagged|not fraud) = 0.02`). Only 0.1% of all transactions are actually fraudulent (`P(fraud) = 0.001`). Using Bayes' Theorem, compute `P(fraud|flagged)` — the actual probability a flagged transaction is really fraud.

View solution

EXERCISE 2

A different spam filter: 30% of email is spam (`P(spam) = 0.30`), 25% of spam emails contain the word "winner" (`P("winner"|spam) = 0.25`), and 1% of legitimate emails contain "winner" (`P("winner"|not spam) = 0.01`). Compute `P(spam|"winner")` using this chapter's own Bayes' Theorem method.

View solution

EXERCISE 3

An intrusion-detection system has a 99% true-positive rate (`P(alert|malicious) = 0.99`) and a 1% false-positive rate (`P(alert|not malicious) = 0.01`). Only 1 in 10,000 connections on the network is actually malicious. Compute `P(malicious|alert)` , and explain — using this chapter's own base-rate-neglect finding — what this result means for whether the system is genuinely "good enough" to alert a human on every trigger.

View solution

CHAPTER 4 QUICK REFERENCE

- **Bayes' Theorem:** `P(H|E) = [P(E|H) × P(H)] / P(E)` — derived directly from Chapter 3's general multiplication rule
- **Prior** P(H), **likelihood** P(E|H), **posterior** P(H|E) — the belief before and after accounting for evidence
- **Law of total probability:** `P(E) = P(E|H)P(H) + P(E|not H)P(not H)` — needed whenever P(E) isn't given directly
- A highly accurate test can still produce a low posterior when the underlying condition (the prior) is rare — **base-rate neglect**
- The same math explains why real intrusion/anomaly-detection alerts are often mostly false positives, and why alert fatigue is a real, quantifiable consequence, not just an inconvenience
- **Next chapter:** Random variables and expected value

CHAPTER 5 OF 10

Random Variables & Expected Value

Probability & Statistics Fundamentals

Chapter 5 · Random Variables & Expected Value

Every chapter so far has asked about the probability of a specific event. This chapter asks a slightly different question: given a whole range of numeric outcomes, each with its own probability, what's a single number that fairly summarizes the whole situation? That number is the **expected value**, and it's the mathematical foundation of essentially every risk, cost, and budgeting calculation built on top of uncertainty.

Random Variables & Probability Distributions

A **random variable** (conventionally written as a capital letter, like `X`) assigns a number to every outcome in a sample space. Its **probability distribution** (or probability mass function, for a discrete variable) lists every possible value it can take, together with the probability of each.

BASELINE EXAMPLE: ROLLING A FAIR DIE

`X` = the value shown. Every value `1` through `6` has probability `1/6` — a genuinely uniform distribution.

Expected Value

The **expected value** `E[X]` is the probability-weighted average of every possible outcome:

EXPECTED VALUE FORMULA

`E[X] = Σ x · P(X = x)` — sum over every possible value `x`

For the die: `E[X] = (1+2+3+4+5+6)/6 = 21/6 = 3.5`.

3.5 IS NEVER A REAL OUTCOME — AND THAT'S EXPECTED

You can never actually roll a 3.5. Expected value is a **long-run average**, not a prediction of any single result — it's the number the average of many, many rolls converges toward, not a value any one roll can produce. This distinction matters constantly in practice: an "expected cost" of $1,500 doesn't mean any given week costs exactly $1,500 — it means that's the fair average across many weeks.

Worked Example: Expected Weekly Incident Cost

A service's weekly incident cost, `X`, has this distribution:

OUTCOME (COST)	PROBABILITY	SCENARIO
$0	0.70	No incident
$500	0.20	Minor incident
$5,000	0.08	Major incident
$50,000	0.02	Catastrophic incident

(Probabilities sum to exactly 1.0, as any valid distribution must — Chapter 2's own certainty axiom.)

TERM	CALCULATION
0 × 0.70	0
500 × 0.20	100
5,000 × 0.08	400
50,000 × 0.02	1,000

`E[X] = 0 + 100 + 400 + 1,000 = $1,500` — the expected weekly incident cost, useful directly for budgeting a reserve fund or setting an insurance premium, exactly the reasoning behind Technical Support's own `backup1` / `incident1` material approached from a numeric angle.

Variance & Standard Deviation of a Random Variable

Expected value alone hides something important: *how spread out* the possible outcomes actually are. **Variance** measures that spread directly, for a known distribution — a genuinely different calculation from Chapter 9's own variance, which is computed from real, already-collected sample data rather than a theoretical distribution.

> **VARIANCE & STANDARD DEVIATION OF A RANDOM VARIABLE**
>
> `Var(X) = E[X²] − (E[X])²`, and `SD(X) = √Var(X)`

For the incident-cost example: `E[X²] = 0²(0.70) + 500²(0.20) + 5,000²(0.08) + 50,000²(0.02) = 0 + 50,000 + 2,000,000 + 50,000,000 = 52,050,000`.

QUANTITY	VALUE
Var(X)	52,050,000 − 1,500² = 49,800,000
SD(X)	√49,800,000 ≈ $7,057

A STANDARD DEVIATION FAR LARGER THAN THE MEAN — A REAL RISK SIGNAL

A standard deviation of roughly $7,057 against a mean of just $1,500 is enormous — it means the "typical" week doesn't look anything like $1,500 at all; most weeks cost $0, and the average is almost entirely dragged up by the rare, catastrophic 2% outcome. Expected value tells you what to budget on average; variance/standard deviation tells you how badly a single bad week could still blow that budget — both numbers matter, not just one.

Random Variables & Expected Value in Code

```
dist = {0: 0.70, 500: 0.20, 5000: 0.08, 50000: 0.02}

assert abs(sum(dist.values()) - 1.0) < 1e-9  # a valid distribution sums to 1

def expected_value(dist):
    return sum(x * p for x, p in dist.items())

def variance(dist):
    ex = expected_value(dist)
    ex2 = sum((x ** 2) * p for x, p in dist.items())
    return ex2 - ex ** 2

print(expected_value(dist))          # 1500.0
print(variance(dist) ** 0.5)       # 7056.91... (standard deviation)
```

Hands-On Exercises

EXERCISE 1

A company's monthly new-signups count, `X`, has the distribution: `P(X=100) = 0.5`, `P(X=200) = 0.3`, `P(X=500) = 0.2`. Confirm the probabilities sum to 1, then compute `E[X]`.

View solution

EXERCISE 2

A feature-flag rollout has three possible cost outcomes: no issues (probability 0.85, cost $0), a minor rollback (probability 0.12, cost $2,000), a major outage (probability 0.03, cost $80,000). Compute `E[X]`, `Var(X)`, and `SD(X)`. Given how large the standard deviation is relative to the mean, what does this chapter's own finding suggest about relying on the expected value alone to decide whether the rollout is "safe enough"?

View solution

EXERCISE 3

Using this chapter's own incident-cost example (`E[X] = $1,500` per week), explain in your own words why "the expected cost over the next 1,000 weeks is $1,500,000" is a reasonable statement to make, even though no single week is ever likely to cost exactly $1,500. Ground your answer in this chapter's own "long-run average, not a single prediction" distinction.

View solution

CHAPTER 5 QUICK REFERENCE

- **Random variable:** assigns a number to every outcome; its **distribution** lists every value with its probability, summing to exactly 1
- **Expected value:** `E[X] = Σ x · P(X=x)` — a probability-weighted average, and a long-run average, not a prediction of any single outcome
- **Variance of a random variable:** `Var(X) = E[X²] − (E[X])²`; **standard deviation** is its square root — distinct from Chapter 9's own sample-based variance, computed from real collected data rather than a known distribution
- A large standard deviation relative to the mean signals a risky, skewed distribution — expected value alone can badly understate real risk
- **Next chapter:** The binomial distribution

CHAPTER 6 OF 10

The Binomial Distribution

Probability & Statistics Fundamentals

Chapter 6 · The Binomial Distribution

Chapter 5 covered random variables in general — any list of outcomes and probabilities was fair game. This chapter is about the single most common specific shape that list takes in real engineering work: counting how many times something succeeds out of a fixed number of independent attempts, each with the same fixed probability.

When the Binomial Distribution Applies

The binomial distribution requires four conditions, all of which are worth checking explicitly before using it:

CONDITION	MEANING
Fixed number of trials	`n` is decided in advance, not itself random
Two outcomes per trial	Each trial is a "success" or "failure" — nothing in between
Fixed success probability	Every trial has the exact same probability `p` of success
Independent trials	One trial's outcome doesn't affect any other's — Chapter 3's own independence, required here explicitly

The Binomial Formula

`X` = the number of successes in `n` trials. The probability of getting *exactly* `k` successes is:

BINOMIAL PROBABILITY MASS FUNCTION

$$P(X = k) = C(n, k) \times p^k \times (1-p)^{n-k}$$

`C(n, k)` is "n choose k" — the number of different ways to pick which `k` of the `n` trials were the successes (the same combinations formula from Discrete Mathematics Fundamentals Chapter 9, for anyone who's taken that course; here it's just `n! / (k! (n-k)!)`). The rest of the formula is one specific sequence's own probability — p^k for the `k` successes, $(1-p)^{n-k}$ for the remaining failures — multiplied by how many different sequences produce that same count.

Worked Example: Flipping a Coin

Flip a fair coin `n = 4` times. `X` = number of heads. What's `P(X = 2)`?

TERM	VALUE
C(4, 2)	6
$p^2 = 0.5^2$	0.25
$(1-p)^2 = 0.5^2$	0.25
P(X=2) = 6 × 0.25 × 0.25	0.375

Mean & Variance — A Shortcut Built on Chapter 5

Computing `E[X]` and `Var(X)` the long way, using Chapter 5's own formulas across every possible value of `k`, works — but the binomial distribution has direct shortcuts that skip all of it:

BINOMIAL MEAN AND VARIANCE

`E[X] = np` `Var(X) = np(1-p)`

For the coin-flip example: `E[X] = 4 × 0.5 = 2` heads on average, `Var(X) = 4 × 0.5 × 0.5 = 1`.

A Practical Example: A/B Test Conversions

A new checkout flow has a 20% conversion rate (`p = 0.2`). Out of the next `n = 10` users, what's the probability *exactly* 3 convert?

TERM	VALUE
C(10, 3)	120
$p^3 = 0.2^3$	0.008
$(1-p)^7 = 0.8^7$	0.2097
P(X=3) = 120 × 0.008 × 0.2097	≈ 0.2013 (20.1%)

`E[X] = 10 × 0.2 = 2` expected conversions, `Var(X) = 10 × 0.2 × 0.8 = 1.6`, `SD(X) ≈ 1.26`. This exact framing — a fixed conversion rate applied across a fixed number of users — is precisely the model behind Probability & Statistics Fundamentals' own sibling course, Statistical Inference & Applied Statistics, and its A/B testing material.

Revisiting Chapter 1's Defect-Rate Example

Chapter 1's own opening exercise asked: given a known 2% defect rate, how many defects are expected in a batch of 500? That's exactly `E[X] = np = 500 × 0.02 = 10` — the binomial mean, finally formalized. The full distribution can answer sharper questions too:

HOW LUCKY WOULD A ZERO-DEFECT BATCH BE?

`P(X = 0) = C(500, 0) × 0.02⁰ × 0.98⁵⁰⁰ ≈ 0.000041` — about a 0.004% chance. Even though 10 defects is only "expected," an entirely defect-free batch of this size, at this defect rate, would be a genuinely remarkable outcome, not a plausible one.

REAL-WORLD CAVEAT: INDEPENDENCE ISN'T AUTOMATIC

If defects come from a single faulty machine setting rather than independent random chance, or if one server failure makes a second one more likely (a cascading outage), the fixed-probability-and-independence assumptions this chapter opened with are violated — and the binomial model will systematically underestimate how often extreme outcomes (many defects/failures at once) actually happen. Always check those four conditions before trusting the formula.

The Binomial Distribution in Code

```
import math

def binomial_pmf(n, k, p):
    return math.comb(n, k) * (p ** k) * ((1 - p) ** (n - k))

print(binomial_pmf(4, 2, 0.5))     # 0.375 — the coin-flip example
print(binomial_pmf(10, 3, 0.2))    # 0.2013... — the A/B test example
print(binomial_pmf(500, 0, 0.02))  # 4.1e-05 — the zero-defect batch

def binomial_mean_var(n, p):
    return n * p, n * p * (1 - p)

print(binomial_mean_var(10, 0.2))   # (2.0, 1.6)
```

Hands-On Exercises

EXERCISE 1

Flip a fair coin 5 times. Using this chapter's own binomial formula, compute `P(exactly 3 heads)`, showing the C(n,k), p^k, and $(1-p)^{n-k}$ terms separately before combining them.

View solution

EXERCISE 2

A new signup flow converts 15% of visitors (`p = 0.15`). Out of the next 8 visitors, compute `P(exactly 2 convert)` , and separately compute `E[X]` and `Var(X)` using this chapter's own mean/variance shortcuts.

View solution

EXERCISE 3

A batch of 20 units has a 5% defect rate. Compute `P(exactly 1 defective unit)` , and separately compute `E[X]` using the mean shortcut. Then name which of this chapter's own four required conditions would be violated if the real cause of defects were a single faulty machine setting affecting every unit in the batch identically, and briefly explain the practical consequence.

View solution

CHAPTER 6 QUICK REFERENCE

- **Requires:** a fixed number of trials `n` , two outcomes per trial, a fixed success probability `p` , and independent trials
- **PMF:** `P(X=k) = C(n,k) × p^k × (1−p)^(n−k)` — combinations × one sequence's own probability
- **Mean/variance shortcuts:** `E[X] = np` , `Var(X) = np(1−p)` — skip Chapter 5's full sum-over-every-value calculation
- Models conversion rates, defect rates, and any "how many successes out of n fixed independent attempts" question
- Always verify independence and a genuinely fixed probability before trusting the model — correlated failures break both assumptions
- **Next chapter:** The Poisson distribution

CHAPTER 7 OF 10

The Poisson Distribution

Probability & Statistics Fundamentals

Chapter 7 · The Poisson Distribution

Chapter 6's binomial distribution needed a fixed, countable number of trials — flip the coin exactly 4 times, check exactly 500 units. Plenty of real questions don't have a natural "number of trials" at all: how many incidents hit a server in a week? How many requests arrive at an endpoint in a second? There's no fixed `n` to point to — just a rate. The Poisson distribution is built specifically for exactly that shape of question.

What the Poisson Distribution Models

The Poisson distribution counts how many times a rare, independent event happens within a fixed interval (of time, space, or anything else continuous), given only the **average rate** at which it happens — a single parameter, conventionally called `λ` (lambda).

POISSON PROBABILITY MASS FUNCTION

$$P(X = k) = (\lambda^k \times e^{-\lambda}) / k!$$

`e` here is Euler's number (≈2.71828), the same constant used throughout continuous math; `k!` is a factorial, exactly Discrete Mathematics Fundamentals Chapter 9's own territory for anyone who's taken that course. `λ` is both the distribution's only parameter *and*, conveniently, its mean.

Mean, Variance — and a Distinctive Property

POISSON MEAN AND VARIANCE

$$E[X] = \lambda \qquad Var(X) = \lambda$$

MEAN EQUALS VARIANCE — GENUINELY UNUSUAL, AND WORTH REMEMBERING

Chapter 6's binomial variance, `np(1−p)`, is always strictly *less* than its mean `np`, since `(1−p) < 1`. The Poisson distribution has no such gap — its variance is always exactly equal to its mean. This is actually a useful real-world diagnostic: if a dataset that's supposedly Poisson-distributed shows a variance noticeably larger than its mean ("overdispersion"), that's a signal the independence or constant-rate assumption is probably broken — real incidents

often cluster (one root cause triggers several), which inflates variance beyond what pure Poisson randomness would predict.

Worked Example: Weekly Server Incidents

A service experiences an average of $\lambda = 3$ incidents per week. What's the probability of *exactly* 5 incidents in a given week?

TERM	VALUE
$\lambda^5 = 3^5$	243
e^{-3}	≈ 0.0498
$5!$	120
$P(X=5) = (243 \times 0.0498) / 120$	≈ 0.1008 (10.1%)

For contrast, the probability of a completely quiet, incident-free week: $P(X=0) = (3^0 \times e^{-3}) / 0! = e^{-3} \approx 0.0498$ — about a 5% chance, despite the average being 3 incidents. This is directly Technical Support's `perfdiag1` / `incident1` territory, now with a precise number attached instead of just "incidents happen sometimes."

A Second Example: Request Arrival Rate

An API endpoint receives an average of $\lambda = 2$ requests per second. What's the probability of exactly 4 requests arriving in a given second?

TERM	VALUE
$\lambda^4 = 2^4$	16
e^{-2}	≈ 0.1353
$4!$	24
$P(X=4) = (16 \times 0.1353) / 24$	≈ 0.0902 (9.0%)

This is the exact reasoning behind capacity planning: knowing the full distribution of "requests per second," not just the average, is what lets a team provision for the realistic worst case rather than just the mean.

The Poisson Distribution as a Limit of the Binomial

The Poisson distribution isn't an unrelated new idea — it's what the binomial distribution turns into when `n` gets very large and `p` gets very small, while their product `np` stays fixed at `λ`. This makes intuitive sense: "requests per second" has effectively infinite tiny sub-instants where a request *could* arrive (huge `n`), each individually very unlikely (tiny `p`), but the overall rate `λ = np` stays meaningful.

MODEL	P(X=0), N=10,000, P=0.0002 (Λ=2)
Exact binomial	0.135308...
Poisson approximation (λ=2)	0.135335...

The two agree to four decimal places — confirming the Poisson distribution as a genuinely accurate, and far simpler, stand-in whenever `n` is huge and `p` is tiny.

The Poisson Distribution in Code

```
import math

def poisson_pmf(k, lam):
    return (lam ** k * math.exp(-lam)) / math.factorial(k)

print(poisson_pmf(5, 3))   # 0.1008... — weekly incidents example
print(poisson_pmf(0, 3))   # 0.0498... — a quiet week
print(poisson_pmf(4, 2))   # 0.0902... — request arrivals example

# Binomial-vs-Poisson agreement check, n large / p small
def binomial_pmf(n, k, p):
    return math.comb(n, k) * (p ** k) * ((1 - p) ** (n - k))

print(binomial_pmf(10000, 0, 0.0002))  # 0.135308...
print(poisson_pmf(0, 2))              # 0.135335... — matches closely
```

Hands-On Exercises

EXERCISE 1

A different service averages `λ = 2` incidents per week. Using this chapter's own Poisson formula, compute `P(exactly 4 incidents)` in a given week, showing the λ^k, $e^{-\lambda}$, and k! terms separately.

View solution

EXERCISE 2

An endpoint averages 5 requests per minute (`λ = 5`). Compute `P(exactly 3 requests in a given minute)`, and separately state `E[X]` and `Var(X)` using this chapter's own mean/variance property.

View solution

EXERCISE 3

A team monitors a Poisson-modeled metric (say, incidents per week) over several months and notices the observed variance is consistently much larger than the observed mean. Using this chapter's own "mean equals variance" property and its overdispersion finding, explain what this observation suggests is actually happening, and why a pure Poisson model might now be misleading for capacity/reserve planning.

View solution

CHAPTER 7 QUICK REFERENCE

- **Models:** the count of rare, independent events over a fixed interval, given only a rate `λ` — no fixed "number of trials" needed, unlike the binomial
- **PMF:** `P(X=k) = (λ^k × e^−λ) / k!`
- **Mean = variance = λ** — a distinctive, memorable property; real data showing variance far above the mean ("overdispersion") signals broken independence, often clustered events sharing a root cause
- The Poisson distribution is the limit of the binomial as `n → ∞` and `p → 0`, with `np = λ` held fixed — verified numerically to four decimal places in this chapter
- Directly models server incident rates and request arrival rates — Technical Support's own `perfdiag1` / `incident1` material, now with exact probabilities attached
- **Next chapter:** The normal distribution and the Central Limit Theorem

CHAPTER 8 OF 10

The Normal Distribution & the Central Limit Theorem

Probability & Statistics Fundamentals

Chapter 8 · The Normal Distribution & the Central Limit Theorem

Chapters 6 and 7 covered *discrete* distributions — counting successes, counting events. The normal distribution is *continuous* — a measurement like response time or height can take any value along a range, not just whole numbers. It's the single most common shape in all of statistics, for a reason this chapter's second half explains directly.

The Bell Curve, Defined by Two Numbers

A normal distribution is completely determined by just two parameters: its mean `μ` (where the peak sits) and its standard deviation `σ` (how wide the spread is). Its shape is always the same symmetric bell — only the center and width change.

THE EMPIRICAL RULE (68-95-99.7)

For any normal distribution: about **68%** of values fall within 1 standard deviation of the mean, about **95%** within 2 standard deviations, and about **99.7%** within 3.

Z-Scores — Standardizing Any Normal Distribution

A **z-score** converts any value from any normal distribution into "how many standard deviations from the mean" — a universal, distribution-independent scale:

Z-SCORE FORMULA

```
z = (x − μ) / σ
```

Once a value is expressed as a z-score, its probability can be looked up against the **standard normal distribution** (μ=0, σ=1) — the same lookup works for literally any normal distribution, regardless of its own original mean and standard deviation.

Worked Example: Server Response Times

Response times are normally distributed with `μ = 200ms`, `σ = 30ms`. What fraction of requests take longer than 260ms?

STEP	CALCULATION	RESULT
z-score for 260ms	(260 − 200) / 30	z = 2
Empirical rule estimate	95% within ±2σ → 5% outside, split evenly	≈ 2.5% above 260ms
Precise value	1 − standard normal CDF(2)	≈ 2.275%

The empirical rule's quick estimate (2.5%) and the precise calculation (2.275%) are close, as expected — the empirical rule is a fast approximation, useful for a sanity check even when reaching for the precise number.

GETTING THE PRECISE NUMBER WITHOUT CALCULUS

The exact normal distribution probability technically requires integrating its density curve — genuine calculus, deliberately out of scope for this course per Chapter 1. In practice, nobody derives that integral by hand: the standard tool is either a printed z-table, or (in code) the **error function**, `erf`, which Python's standard library already implements — no calculus knowledge required to use it correctly.

The Central Limit Theorem

Here's the genuinely remarkable part: it doesn't matter what the underlying distribution looks like. Take repeated samples of size `n` from *any* distribution — normal, Poisson, wildly skewed, anything with a finite mean and variance — and compute each sample's own average. As `n` grows, the distribution of *those averages* approaches a normal distribution, centered on the true population mean.

THE CENTRAL LIMIT THEOREM (CLT)

For samples of size `n` drawn from a distribution with mean `μ` and variance `σ²`, the sample mean's own distribution approaches Normal(`μ` , `σ²/n`) as `n` grows — regardless of the original distribution's shape.

That `σ²/n` term is the key: the sample mean's *own* variance shrinks as `n` grows, meaning larger samples produce averages that cluster more and more tightly around the true mean.

Illustrating with a single die roll (`X` , uniform, *not* remotely normal): `Var(X) = 35/12 ≈ 2.917`.

SAMPLE SIZE N	VAR(SAMPLE MEAN) = Σ²/N	SD(SAMPLE MEAN)
10	0.2917	≈ 0.540
100	0.02917	≈ 0.171

WHY THIS MATTERS — AND WHERE IT'S HEADED NEXT

A single die roll is as far from a bell curve as a distribution gets — flat, uniform, no peak at all. Yet the *average* of many rolls behaves almost normally, and gets more tightly clustered around 3.5 the more rolls are averaged. This exact `σ/√n` quantity — the standard deviation of a sample mean — has its own name, **standard error**, and it's the direct foundation of confidence intervals: Probability & Statistics Fundamentals' own sibling course, Statistical Inference & Applied Statistics, builds its entire second chapter on precisely this idea.

The Normal Distribution & CLT in Code

```
import math

def standard_normal_cdf(z):
    return 0.5 * (1 + math.erf(z / math.sqrt(2)))

def z_score(x, mu, sigma):
    return (x - mu) / sigma

z = z_score(260, mu=200, sigma=30)
print(1 - standard_normal_cdf(z))  # 0.02275 — matches the worked example

# CLT: standard error shrinks as sample size grows
def standard_error(sigma, n):
    return sigma / math.sqrt(n)

die_sd = math.sqrt(35 / 12)
print(standard_error(die_sd, 10))   # 0.540...
print(standard_error(die_sd, 100))  # 0.171...
```

Hands-On Exercises

EXERCISE 1

Request latency is normally distributed with `μ = 150ms`, `σ = 20ms`. Compute the z-score for 110ms, use the empirical rule to estimate the probability of a request taking *less* than 110ms, then compute the precise probability using the standard normal CDF. Compare the two estimates.

View solution

EXERCISE 2

A metric has a population standard deviation of `σ = 6`. Using this chapter's own standard error formula, compute the standard deviation of the sample mean for sample sizes `n = 9`, `n = 36`, and `n = 144`. Describe the pattern you notice in how much `n` has to grow to halve the standard error.

View solution

EXERCISE 3

Chapter 5's own weekly-incident-cost example had a highly skewed distribution (mean $1,500, standard deviation ≈ $7,057) — nothing close to a normal bell curve. Using this chapter's own Central Limit Theorem, explain why the *average* cost computed across many independent weeks would behave far more predictably (more tightly clustered, more normally shaped) than any single week's own cost, even though individual weeks are wildly unpredictable.

View solution

CHAPTER 8 QUICK REFERENCE

- **Normal distribution:** a continuous, symmetric bell curve fully defined by `μ` (mean) and `σ` (standard deviation)
- **Empirical rule:** ≈68% within 1σ, ≈95% within 2σ, ≈99.7% within 3σ
- **Z-score:** `z = (x − μ) / σ` — converts any normal value onto the universal standard normal scale
- Precise probabilities come from the standard normal CDF (via `math.erf` in Python) — no calculus knowledge required to use it
- **Central Limit Theorem:** the sample mean's distribution approaches Normal(μ, σ^2/n) as n grows, regardless of the original distribution's shape
- **Standard error** (`σ/√n`) is the sample mean's own standard deviation — shrinks as sample size grows, and is the direct foundation of confidence intervals in this subject's own next course
- **Next chapter:** Descriptive statistics — mean, median, variance & standard deviation

CHAPTER 9 OF 10

Descriptive Statistics: Mean, Median, Variance & Standard Deviation

Probability & Statistics Fundamentals

Chapter 9 · Descriptive Statistics: Mean, Median, Variance & Standard Deviation

Every chapter so far started with a known model and predicted outcomes — the probability direction from Chapter 1's own opening table. This chapter finally works the other way: given real, already-collected data, how do you summarize it fairly? This is genuinely different math from Chapter 5's variance, not just a repeat of it — Chapter 5 computed statistics from a known theoretical distribution; this chapter computes them directly from actual numbers you've measured.

Mean & Median

The **mean** is the familiar average: sum every value, divide by the count. The **median** is the middle value once the data is sorted — the average of the two middle values if there's an even count.

MEAN AND MEDIAN FORMULAS

`mean = (Σ x) / n` `median` = the middle value of the sorted data (average of the two middle values if n is even)

Worked Example: When One Slow Request Skews Everything

Seven real request response times (ms): `120, 115, 130, 125, 118, 122, 890` — six ordinary requests, and one genuinely slow one.

DATASET	MEAN	MEDIAN
All 7 values (with the 890ms outlier)	231.43ms	122ms
Just the 6 ordinary values	121.67ms	121ms

THE MEAN MOVED BY OVER 100MS; THE MEDIAN BARELY MOVED AT ALL

A single outlier dragged the mean from 121.67ms up to 231.43ms — nearly double — while the median stayed almost exactly where it was (121ms → 122ms). This is exactly the dashboard trap named back in Chapter 1: an "average response time" panel can be badly distorted by a handful of extreme values, making the service look far worse (or, in

other contexts, far better) than what most real requests actually experienced. The median is far more resistant to outliers, since it only cares about the middle position, not the extreme values' actual size.

Sample Variance & Standard Deviation

Chapter 5 defined variance for a known random variable: `Var(X) = E[X²] − (E[X])²`. For real, already-collected sample data, the formula looks similar but has one crucial, easy-to-miss difference:

SAMPLE VARIANCE FORMULA

`s² = Σ(x − x̄)² / (n − 1)` `s = √s²` (sample standard deviation)

WHY N − 1, NOT N — BESSEL'S CORRECTION

Dividing by `n` would use the *sample's own mean* (`x̄`, estimated from the same limited data) as if it were the true population mean — and a sample's mean is, by construction, the value that minimizes the sum of squared deviations for *that specific sample*, which means dividing by `n` systematically **underestimates** the true population variance. Dividing by `n − 1` instead corrects for that bias. In practice, unless you genuinely have every single data point that will ever exist (the entire population, not a sample of it), `n − 1` is almost always the correct choice.

Applying this to the same two datasets from above:

DATASET	SAMPLE VARIANCE (S^2)	SAMPLE STD DEV (S)
All 7 values (with outlier)	84,357.29	≈ 290.44ms
Just the 6 ordinary values	28.27	≈ 5.32ms

Variance and standard deviation are hit even harder by the outlier than the mean was — the single 890ms value inflates the standard deviation more than fiftyfold, from ≈5.32ms to ≈290ms. Squaring deviations, as the formula does, punishes large outliers disproportionately.

Descriptive Statistics in Code

```python
import statistics as stats

data = [120, 115, 130, 125, 118, 122, 890]

print(stats.mean(data))      # 231.4285...
print(stats.median(data))    # 122
print(stats.variance(data))  # 84357.29... — Python's stdlib already uses n-1
print(stats.stdev(data))     # 290.44...

# Implementing sample variance from scratch, to see the n-1 explicitly
def sample_variance(data):
    mean = sum(data) / len(data)
    return sum((x - mean) ** 2 for x in data) / (len(data) - 1)

print(sample_variance(data))  # matches stats.variance(data)
```

Hands-On Exercises

EXERCISE 1

CPU usage samples from six consecutive checks (%): `45, 48, 50, 47, 46, 95`. Compute the mean and the median. Which one better represents a "typical" reading from this dataset, and why, per this chapter's own outlier-resistance finding?

View solution

EXERCISE 2

Given the sample `10, 12, 14`, compute the mean, then the sample variance using this chapter's own `n - 1` formula, showing each squared deviation separately, and finally the sample standard deviation.

View solution

EXERCISE 3

Explain, in your own words, why a data engineer analyzing a sample of 10,000 user sessions out of millions that occurred should almost always use the `n - 1` sample variance formula rather than dividing by `n` — and describe the one specific circumstance (per this chapter's own Bessel's-correction explanation) where dividing by plain `n` would actually be the mathematically correct choice instead.

View solution

CHAPTER 9 QUICK REFERENCE

- **Mean:** sum divided by count; **median:** the middle value of sorted data — far more resistant to outliers than the mean
- A single extreme outlier can drag the mean dramatically while barely moving the median — the exact mechanism behind the "average hides a real spike" dashboard trap named in Chapter 1
- **Sample variance:** `s² = Σ(x−x̄)² / (n−1)` — divides by `n−1` (Bessel's correction), not `n`, whenever the data is a sample rather than the entire population
- Squaring deviations means variance and standard deviation are even more sensitive to outliers than the mean is
- This chapter's statistics are computed from real, already-collected data — distinct from Chapter 5's variance of a known theoretical random variable
- **Next chapter:** Capstone — probability & statistics in practice

CHAPTER 10 OF 10

Capstone — Probability & Statistics in Practice

Probability & Statistics Fundamentals

Chapter 10 · Capstone — Probability & Statistics in Practice

One continuous worked project, touching every chapter of this course in the order a real engineer would actually reach for each idea: monitoring the canary rollout of a new checkout feature, from the first hour's error logs through the final response-time analysis.

A Full Worked Rollout — Monitoring a New Checkout Feature

1 — READING THE FIRST HOUR'S PROBLEM LOGS (CH.2)

Two problem types are logged per session: `A` = "UI error" (`P(A) = 0.03`) and `B` = "timeout" (`P(B) = 0.02`), with `P(A ∩ B) = 0.005` — some sessions hit both. The union rule gives the overall "any problem" rate: `P(A ∪ B) = 0.03 + 0.02 − 0.005 = 0.045`. The complement rule then gives the genuinely useful number: `P(clean session) = 1 − 0.045 = 0.955` — 95.5% of sessions have no problem at all.

2 — CHECKING WHETHER THE TWO PROBLEM TYPES ARE RELATED (CH.3)

Testing independence: `P(A) × P(B) = 0.03 × 0.02 = 0.0006` , but the actual `P(A ∩ B) = 0.005` — over eight times larger. They are clearly **not** independent. Computing `P(timeout | UI error) = 0.005 / 0.03 ≈ 16.7%` — far above the 2% baseline timeout rate — confirms the two problems cluster together, consistent with a shared root cause like server overload rather than two unrelated glitches.

3 — IS THE MONITORING ALERT TRUSTWORTHY? (CH.4)

A monitoring alert fires when the problem rate spikes. Genuine load issues happen on 5% of days (`P(load) = 0.05`); the alert catches 90% of real load issues (`P(alert|load) = 0.9`) but also false-fires on 3% of normal days (`P(alert|no load) = 0.03`). By the law of total probability, `P(alert) = (0.9)(0.05) + (0.03)(0.95) = 0.0735` . Bayes' Theorem then gives `P(load | alert) = 0.045 / 0.0735 ≈`

`61.2%` — a meaningfully informative alert, though still short of certainty, exactly the base-rate reasoning Chapter 4 built.

4 — BUDGETING FOR A POSSIBLE ROLLBACK (CH.5)

The rollout's cost, as a random variable: `P(no rollback) = 0.85` (cost $0), `P(partial rollback) = 0.12` (cost $3,000), `P(full rollback) = 0.03` (cost $40,000). Expected value: `E[X] = 0(0.85) + 3,000(0.12) + 40,000(0.03) = 360 + 1,200 = $1,560` — the number the team should actually budget for, not the (much lower) most-likely single outcome.

5 — HOW UNUSUAL WOULD A GREAT CANARY RESULT BE? (CH.6)

Reusing Chapter 6's own conversion rate, `p = 0.2`, for `n = 10` canary users: what's the probability at least half convert (`X ≥ 5`)? Summing the binomial PMF from `k = 5` to `10` gives `P(X ≥ 5) ≈ 0.033` (3.3%). If the canary group actually shows 5 or more conversions, that's a genuinely rare result under the existing 20% rate — real evidence the new feature may be improving conversion, not just random noise.

6 — HOW WORRIED SHOULD AN "INCIDENT-HEAVY" WEEK BE? (CH.7)

Reusing Chapter 7's own incident rate, `λ = 3` per week: what's the probability of *at least* 2 incidents during the rollout's first monitored week? `P(X ≥ 2) = 1 − P(X=0) − P(X=1) = 1 − 0.0498 − 0.1494 ≈ 0.801` (80.1%) — a week with two or more incidents is actually the *normal* case at this rate, not a red flag on its own.

7 — READING THE RESPONSE-TIME DASHBOARD CORRECTLY (CH.8)

Reusing Chapter 8's own response-time model, `μ = 200ms`, `σ = 30ms`: the probability a request falls between 170ms and 230ms (within 1σ either side) is `P(170 < X < 230) ≈ 0.6827` — matching the empirical rule's 68% directly. For a monitoring dashboard averaging 100 requests at a time, the Central Limit Theorem gives that average's own standard error: `SE = 30/√100 = 3ms` — the averaged metric is far more stable than any single request's own time, exactly why dashboards average in the first place.

8 — THE FINAL RESPONSE-TIME SAMPLE, AND THE OUTLIER TRAP (CH.9)

Reusing Chapter 9's own seven sampled response times — `120, 115, 130, 125, 118, 122, 890` — one request during the rollout was genuinely slow. Mean: `231.43ms`. Median: `122ms`. Reporting "average

response time: 231ms" to stakeholders would badly misrepresent what most users actually experienced — the median, far less shaken by the single outlier, is the honest number to lead with.

This is, in essence, exactly what a real feature-rollout review looks like — every step traceable to a specific chapter of this course, none of it abstract math floating free of the actual monitoring dashboard.

What This Course Doesn't Cover

In the interest of an honest accounting: **sampling and confidence intervals**, **hypothesis testing and A/B testing**, and **correlation, regression, and Bayesian updating** were all named in Chapter 1 as deliberately out of scope, reserved for this subject's own next course, Statistical Inference & Applied Statistics. This course built the probability vocabulary and distributions every one of those techniques is built on top of, not a substitute for them.

This Course's Throughline, Restated

REASONING FORWARD FROM A KNOWN MODEL

Every chapter in this course answered a version of the same question: given a clearly stated model of how something behaves, what can be said precisely about the outcomes it produces? The capstone above used exactly eight ideas — probability rules, conditional probability, Bayes' Theorem, expected value, and three named distributions plus descriptive statistics — across a single continuous scenario, without needing anything beyond them. That's the real payoff: a small, reusable toolkit for reasoning honestly about uncertainty before the data even exists.

Where This Course Connects

This course is the direct foundation under Technical Support's own diagnostic material — `perfdiag1`'s and `incident1`'s handling of incident rates and monitoring dashboards is exactly Chapters 7–9's territory, applied without the underlying math ever being named explicitly there. Within this subject's own next course, Statistical Inference & Applied Statistics builds directly on this course's Chapter 8 (the Central Limit Theorem feeds its own sampling-distribution chapter) and Chapter 4 (Bayes' Theorem feeds its own Bayesian-updating chapter) — nothing here was built in isolation from where this subject is actually headed next.

Hands-On Exercises

EXERCISE 1

A different rollout logs two problem types with `P(A) = 0.04`, `P(B) = 0.025`, and `P(A ∩ B) = 0.001`. Using this chapter's own Step 1–2 techniques, compute `P(clean session)`, then determine whether A and B are independent.

View solution

EXERCISE 2

Reusing Step 4's rollback-cost distribution, suppose the "full rollback" probability is revised upward to 0.05 (with "partial rollback" correspondingly reduced to 0.10, and "no rollback" still 0.85). Recompute `E[X]`, and state whether the team's budget should increase or decrease compared to Step 4's original $1,560 figure.

View solution

EXERCISE 3

For each of the eight steps in this chapter's own worked rollout, name the specific probability/statistics topic it relied on, without looking back at the step labels — just from the description of what each step actually does.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** union/complement rules (Ch.2) → conditional probability & independence (Ch.3) → Bayes' Theorem (Ch.4) → expected value (Ch.5) → binomial (Ch.6) → Poisson (Ch.7) → normal distribution & CLT (Ch.8) → descriptive statistics (Ch.9)
- Out of scope: sampling/confidence intervals, hypothesis testing/A-B testing, and correlation/regression/Bayesian updating — all reserved for Statistical Inference & Applied Statistics
- This course's throughline: a small, reusable toolkit for reasoning forward from a known model to precise statements about likely outcomes
- This course is the direct foundation under Technical Support's own `perfdiag1` / `incident1` material, and under this subject's own next course
- **Course complete** — Probability & Statistics Fundamentals, 10 chapters, from sample spaces to descriptive statistics